sgcWebSockets .NET 2026.4

April 2026

Documentation for .NET

info@esegece.com
www.esegece.com

Contents

Introduction

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

.NET Features

- Fully functional **multithreaded WebSocket server** implementing **RFC 6455**.
- Assemblies for **.NET Framework (2.0+)**, **.NET Standard (1.6+)**, **.NET Core (1.0+)**.
- Supports **Windows 32/64**, **macOS 64**, and **Linux 64**.
- **IOCP** threading model for high-performance servers.
- **Message Compression** using PerMessage_Deflate extension (RFC 7692).
- **Text** and **Binary** message support.
- **Server** and **Client Authentication**.
- WebSocket and HTTP connections through the **same port**.
- **Server-Sent Events** (push notifications) over HTTP.
- **WatchDog** and **HeartBeat** built-in support.
- **Socket.IO** client connections.
- **Telegram** client.
- **Binance** Spot and Futures (WebSocket, User Stream, and REST APIs).
- **STUN** and **TURN** client and server components.
- Client connections through **HTTP Proxy** and **SOCKS Proxy** servers.
- Events: **OnConnect**, **OnDisconnect**, **OnMessage**, **OnError**, **OnHandshake**.
- Built-in **JavaScript libraries** for browser clients.
- **SSL/TLS** for server and client components. **OpenSSL 1.1.1** and **3.0** supported. Client supports **SChannel** on Windows.

The following components are included in the sgcWebSockets library:

1. **sgcWebSockets**
   - **TsgcWebSocketClient:** WebSocket Client based on Indy Library.
   - **TsgcWebSocketServer:** WebSocket Server based on Indy Library
   - **TsgcWebSocketHTTPServer:** WebSocket + HTTP Server based on Indy Library.
   - **TsgcWebSocketServer_HTTPAPI:** Fast Performance WebSocket + HTTP Server based on HTTP.SYS Microsoft HTTP API.
2. **sgcWebSocket APIs**
   - **TsgcWSAPI_Binance:** Binance Spot Client, supports WebSocket + REST APIs.
   - **TsgcWSAPI_Binance_Futures:** Binance Futures Client, supports WebSocket + REST APIs.
   - **TsgcWSAPI_SocketIO:** Socket.IO Client.
3. **sgcWebSocket Libs**
   - **TsgcTDLib_Telegram:** Telegram API Client.
4. **sgcWebSocket Protocols**
   - **TsgcWSPClient_MQTT:** MQTT (3.1.1 and 5.0) Client. Supports WebSocket and Plain TCP Connections.
   - **TsgcWSPServer_AppRTC:** WebRTC Server based on AppRTC Google Project.
   - **TsgcWSPServer_WebRTC:** WebRTC Server Protocol.
   - **TsgcWSPClient_Files:** WebSocket File Transfer Client Protocol.

- **TsgcWSPServer_Files:** WebSocket File Transfer Server Protocol.

5. **sgcWebSockets HTTP**
   - **TsgcHTTP_JWT_Client:** JWT (JSON WEB TOKEN) Client.
   - **TsgcHTTP_JWT_Server:** JWT (JSON WEB TOKEN) Server.
   - **TsgcHTTP_OAuth2_Client:** OAuth 2.0 Client.
   - **TsgcHTTP_OAuth2_Server:** OAuth 2.0 Server.
6. **sgcWebSockets P2P**
   - **TsgcSTUNClient:** STUN Client.
   - **TsgcSTUNServer:** STUN Server.
   - **TsgcTURNClient:** STUN / TURN Client.
   - **TsgcTURNServer:** STUN / TURN Server.
7. **sgcWebSockets AI**

Versions Support

.NET Supported Versions

- .NET Framework 2.0+
- .NET Standard 1.6+
- .NET Core 1.0+
- VSIX Package requires .NET Framework 4.5
- Supports Windows 32 / Windows 64 / OSX64

Installation

Nuget Package

You can install sgcWebSockets .NET Community edition from the following nuget url:

https://www.nuget.org/packages/esegece.sgcWebSockets/

Check the following videos which show how to install the NuGet package and use sgcWebSockets components

Visual Studio Windows
Visual Studio Mac OS

Assembly Reference

You can work with the sgcWebSockets .NET package without using the NuGet package. Just open your project, select **Add Reference** from the context menu on the project, and select the assembly you want to add as a reference from the Assemblies folder.
Available Assemblies:

- .NET Framework 2.0
- .NET Framework 3.5
- .NET Framework 4.0
- .NET Framework 4.5
- .NET Framework 5.0
- .NET 6.0
- .NET 7.0
- .NET 8.0
- .NET Standard 1.6
- .NET Standard 2.0
- .NET Core 1.1
- .NET Core 2.0
- .NET Core 3.0

Remember to copy sgcWebSockets.dll (under Windows) or libsgcWebSockets.dylib (under OSX64)

QuickStart

WebSockets Components

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

QuickStart WebSockets

HTTP Components

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

QuickStart HTTP

Threading Flow

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

How to Configure NotifyEvents

How to Build Applications

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

Build

OpenSSL

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

Configure OpenSSL

ASP.NET

You can use sgcWebSockets in your ASP.NET, only keep in mind that sgcWebSockets requires some unmanaged dll (like sgcWebSockets.dll) and by default your ASP.NET projects won't find these libraries. In order to set the path of your bin project, set the PATH with a code like this in your Global.asax file

```
        protected void Application_Start()
        {
            var path = string.Concat(Environment.GetEnvironmentVariable("PATH"), ";",
AppDomain.CurrentDomain.RelativeSearchPath);
            Environment.SetEnvironmentVariable("PATH",                          path,
EnvironmentVariableTarget.Process);
            ......
        }
```

QuickStart | WebSockets

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

WebSocket Server

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

```
private void OnMessage(TsgcWSConnection Connection, const string Text)
{
  MessageBox.Show("Message Received From Client: " + Text);
}
```

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

```
TsgcWebSocketServer1.Active = True;
```

WebSocket Client

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

```
TsgcWebSocketClient1.Active = true;
```

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

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

Web Browser Client

1. Create a new HTML file

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

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

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

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

How To Use

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

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

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

ASP.NET

You can use sgcWebSockets in your ASP.NET, only keep in mind that sgcWebSockets requires some unmanaged dll (like sgcWebSockets.dll) and by default your ASP.NET projects won't find these libraries. In order to set the path of your bin project, set the PATH with a code like this in your Global.asax file

```
        protected void Application_Start()
        {
            var path = string.Concat(Environment.GetEnvironmentVariable("PATH"), ";",
AppDomain.CurrentDomain.RelativeSearchPath);
            Environment.SetEnvironmentVariable("PATH",                          path,
EnvironmentVariableTarget.Process);
            ......
        }
```

QuickStart | HTTP

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

HTTP/2 Server

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

```
void OnCommandGet(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo RequestInfo,
  ref TsgcWSHTTPResponseInfo ResponseInfo)
{
  if (RequestInfo.Document == "/")
  {
    ResponseInfo.ContentText = "<html><head><title>Test Page</title></head><body></body></html>";
    ResponseInfo.ContentType = "text/html";
    ResponseInfo.ResponseNo = 200;
  }
}
```

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

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

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

```
TsgcWebSocketHTTPServer1.Active = True;
```

HTTP/1 Client

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

```
TsgcHTTP1Client oHTTP1 = new TsgcHTTP1Client();
MessageBox.Show(oHTTP1.Get("https://127.0.0.1"));
```

HTTP/2 Client

1. Create a new Window Forms Application

2. Drop a TButton in a Form, Double Click and type this code:

```
TsgcHTTP2Client oHTTP2 = new TsgcHTTP2Client();
MessageBox.Show(oHTTP2.Get("https://127.0.0.1"));
```

QuickStart | Threading Flow

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

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

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

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

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

NotifyEvents = neAsynchronous

Threading Flow Best Performance (NotifyEvents = neNoSync)

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

NotifyEvents = neNoSync

```
void OnClientMessage(TsgcWSConnection Connection, string Text)
{
  Invoke((MethodInvoker)delegate
  {
    memo1.AppendText(Text + Environment.NewLine);
  });
}
```

QuickStart | Build

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

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

Fast Performance Servers

Servers based on Indy Library

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

Use the following tips to increase server performance.

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

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

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

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

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

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

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

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

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

    {$DEFINE UseReleaseStack}

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

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

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

Indy Server Windows

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

IOHandlerOptions.IOHandlerType = iohIOCP

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

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

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

Indy Server Linux

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

IOHandlerOptions.IOHandlerType = iohEPOLL

The property IOHandlerOptions.EPOLL allows customizing the EPOLL properties.

- **EPOLLThreads:** these are the threads used to handle the connections, by default the value is zero which means the threads will be calculated automatically using the number of processors.
- **WorkOpThreads:** set a value greater than zero if you want that the requests for every connection are handled always by the same thread. By default, EPOLL requests are handled by random threads, if you want that the connections are handled by always the same thread, set a value greater than zero. Example: if you

set WorkOpThreads = 32, the server will create 32 threads and every time there is a new request, if the connection was already processed previously it will be queued in the same thread.

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

Server Based on HTTP.SYS

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

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

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

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

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

OpenSSL

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

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

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

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

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

- Windows
- OSX

openSSL Configurations

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

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

API Version

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

LibPath

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

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

Load Additional OpenSSL Functions

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

Ciphers

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

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

And set the following cipher list.

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

Self-Signed Certificates

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

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

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

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

Common Errors

SSL_GET_RECORD: wrong version number

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

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

SSL3_GET_RECORD: decryption failed or bad record mac

Usually this error is raised when:

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

OpenSSL | Windows

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

API 1.0

Requires the following libraries:

- libeay32.dll
- ssleay32.dll

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

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

API 1.1

Requires the following libraries:

Windows 32

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

Windows 64

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

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

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

API 3.*

Requires the following libraries:

Windows 32

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

Windows 64

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

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

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

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

OpenSSL | OSX

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

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

API 1.0

Requires the following libraries:

- libcrypto.dylib
- libssl.dylib

You can download latest libraries from your account.

API 1.1

Requires the following libraries:

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

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

API 3.0

Requires the following libraries:

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

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

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

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

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

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

Errors

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

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

OpenSSL | Own CA Certificates

Github post

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

1. Prepare the configuration files for creating certificates without prompts

CA.cnf

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

localhost.cnf

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

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

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

3. Generate web server secret key and CSR

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

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

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

5. Output files will be:

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

OpenSSL | P12 Certificates

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

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

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

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

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

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

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

$ rm cert-decrypted.tmp
```

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

Check below the configuration for sgcWebSockets and sgcIndy packages:

sgcWebSockets

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

sgcIndy

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

OpenSSL | Verify Certificate

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

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

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

```
X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY
```

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

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

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

OpenSSL | Load Additional Functions

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

IdOpenSSLSetLoadFuncsCallback

```
delegate void TIdLoadSSLFuncsCallback(TIdLibHandle hIdSSL, TIdLibHandle hIdCrypto, TStringList FailedLoadList);
```

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

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

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

IdOpenSSLSetUnLoadFuncsCallback

```
delegate void TIdUnLoadSSLFuncsCallback();
```

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

How to load custom function

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

```
// Note: Direct function pointer loading is not applicable in .NET/C#.
// Use P/Invoke or the managed OpenSSL wrapper instead.
// The following is a conceptual equivalent:

IntPtr EVP_PKEY_CTX_set_rsa_padding = IntPtr.Zero;

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

void DoOpenSSLUnLoadFuncsCallback()
{
  EVP_PKEY_CTX_set_rsa_padding = IntPtr.Zero;
}

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

WebSocket Events

WebSocket connections have the following events:

OnConnect
The event raised when a new connection is established.

OnDisconnect
The event raised when a connection is closed.

OnError
The event raised when a connection has any error.

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

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

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

neNoSync is recommended when:

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

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

WebSocket Parameters Connection

Supported by

TsgcWebSocketClient
Java script

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

Options / Parameters

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

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

```
public void WSServerConnect(TsgcWSConnection Connection)
{
  if (Connection.URL == "/?id=1")
  {
    HandleThisParameter;
  }
}
```

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

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

Using inside a DLL

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

WebBrowser Test

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

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

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

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

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

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

Open
Close
Status
Send

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

Authentication

Supported by

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

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

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

Authentication/ Enabled

sgcWebSockets implements 3 different types of WebSocket authentication:

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

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

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

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

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

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

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

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

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

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

admin=admin
user=1234
....

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

```
private void OnAuthenticationEvent(TsgcWSConnection Connection, string User, string Password, ref bool Authentica
{
  if ((User == "user") && (Password == "1234"))
```

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

Secure Connections

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
Web Browsers

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

Server Side

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

SSL/ Enable

There are other properties that you need to define:

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

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

SSLOptions/ Port: port used on SSL connections.

Client Side

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

TLS/ Enable

OpenSSL

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

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

Microsoft SChannel

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

HeartBeat

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWebSocketClient

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

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

HeartBeat has the following properties:

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

Customize HeartBeat

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

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

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

WatchDog

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWebSocketClient

Server

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

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

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

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

Client

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

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

Logs

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

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

LogFile/ Enabled

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

Example:

```
127.0.0.1:49854 Stat Connected.

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

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

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

WebSocket Messages

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

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

HTTP

Supported by

TsgcWebSocketHTTPServer

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

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

http://localhost/test.html

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

```
private void OnCommandGetEvent(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo RequestInfo,
  ref TsgcWSHTTPResponseInfo ResponseInfo)
{
  if (RequestInfo.Document == "/myfile.js")
  {
    ResponseInfo.ContentText = "<script type='text/javascript'>alert('Hello!');</script>";
    ResponseInfo.ContentType = "text/javascript";
    ResponseInfo.ResponseNo = 200;
  }
}
```

Broadcast and Channels

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI

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

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

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

```
void OnServerConnect(TsgcWSConnection Connection)
{
  if (desktop == true)
  {
    (TsgcWSConnectionServer)(Connection).Subscribe("desktop");
  }
}
```

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

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

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

Bindings

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

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

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

```
WSServer.Bindings = "127.0.0.1:80,80.254.21.11:80";
```

Post Big Files

Supported by

TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI

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

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

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

Sample Code

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

```
TsgcWebSocketHTTPServer oServer = new TsgcWebSocketHTTPServer();
oServer.Port = 5555;
oServer.HTTPUploadFiles.StreamType = TwsPostStreamType.pstFileStream;
oServer.Active = true;
```

Then create a new html file with the following configuration

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

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

Events

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

OnHTTPUploadBeforeSaveFile

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

```
private void OnHTTPUploadBeforeSaveFileEvent(TObject Sender, ref string aFileName, ref string aFilePath)
{
  if (aFileName == "test.jpg")
  {
    aFileName = "custom_test.jpg";
  }
}
```

OnHTTPUploadAfterSaveFile

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

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

}
```

OnHTTPUploadReadInput

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

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

}
```

Compression

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
Web Browsers like Chrome

This is a feature that works very well when you need to send a lot of data, usually using a binary message, because it compresses WebSocket message using protocol "PerMessage_Deflate" which is supported by some browsers like Chrome.

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

Extensions/ PerMessage_Deflate / Enabled

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

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

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

Flash

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

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

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

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

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

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

```
swf.example.com
```

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

```
123,456-458
```

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

Groups

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI

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

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

Adding and removing users

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

The following example shows the Groups.Add method.

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

Sending Messages to a Group

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

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

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

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

Events

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

OnClientAdded
OnClientRemoved

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

```
TsgcWebSocketServer1.Groups.OnClientRemoved() = OnClientRemovedEvent();

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

IOCP

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer

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

```
Server.IOHandlerOptions.IOHandlerType = iohIOCP;
Server.IOHandlerOptions.IOCP.IOCPThreads = 0;
Server.IOHandlerOptions.IOCP.WorkOpThreads = 0;
```

IOCPThreads are the threads used for IOCP asynchronous requests (overlapped operations), by default the value is zero which means the number of threads are calculated using the number of processors (except for Delphi 7 and 2007 where the number of threads is set to 32 because the function cpucount is not supported).
WorkOpThreads should only be enabled if you want connections to always be processed in the same thread. When using IOCP, the requests are processed by a pool of threads, and every request (for the same connection) can be processed in different threads. If you want to handle every connection in the same thread set in WorkOpThreads the number of threads used to handle these requests. This impacts the performance of the server and it is only recommended to set a value greater than zero if you require this feature.

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

OnDisconnect event not fired

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

IOCP works differently, there is a thread pool which handles all connections, instead of 1 thread = 1 connection like indy does by default. For IOCP, the only way to detect if a connection is still alive is to try writing to the socket. If there is any error, it means that the connection is closed. There are 2 options to detect disconnections:

1. If you use **TsgcWebSocketClient**, you can enable it in Options property, **CleanDisconnect := True** (by default is disabled). If it's enabled, before the client disconnects it sends a message informing the server about disconnection, so the server will receive this message and the OnDisconnect event will be raised.
2. You can enable **heartbeat** on the **server** side, for example every 60 seconds, so it will try to send a ping to all clients connected and if there is any client disconnected, OnDisconnect will be called.

ALPN

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

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

Client

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

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

Server

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

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

```
// Client
void OnClientConnect(TsgcWSConnection Connection)
{
  string vProtocol = ((TsgcWSConnectionClient)Connection).ALPNProtocol;
}

// Server
void OnSSLALPNSelect(string Protocols, ref string Protocol)
{
  if (Array.IndexOf(Protocols.Split(','), "h2") >= 0)
  {
    Protocol = "h2";
  }
}
```

Forward HTTP Requests

Supported by

TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
TsgcWSHTTPWebBrokerBridgeServer
TsgcWSHTTP2WebBrokerBridgeServer
TsgcWSServer_HTTPAPI_WebBrokerBridge

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

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

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

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

```
void OnBeforeForwardHTTP(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo ARequestInfo,
  ref TsgcWSServerForwardHTTP aForward)
{
  if (ARequestInfo.Document == "/internal")
  {
    aForward.Enabled = true;
    aForward.URL = "http://localhost:8080";
  }
}
```

Other Options

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

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

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

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

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

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

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

Queues

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

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

Level 0: Messages are not queued on Server

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

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

Level 0

The message is not queued by Server

The table below shows the Queue level 0 protocol flow.

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

Level 1

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

The table below shows the Queue level 1 protocol flow.

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

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

Level 2

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

The table below shows the Queue level 2 protocol flow.

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

Transactions

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

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

StartTransaction

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

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

Commit

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

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

RollBack

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

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

TCP Connections

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

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

Client Component
Disable WebSocket protocol.

```
Client.Specifications.RFC6455 = false;
```

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

```
void OnUnknownProtocol(TsgcWSConnection Connection, ref bool Accept)
{
  Accept = true;
}
```

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

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

```
void OnWSClientConnect(TsgcWSConnection Connection)
{
  Connection.SetTCPEndOfFrameScanBuffer(TtcpEOFScanBuffer.eofScanAllBytes);
  Connection.AddTCPEndOfFrame(0);
  Connection.AddTCPEndOfFrame(10);
}
```

SubProtocol

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketServer_HTTPAPI

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

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

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

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

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

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

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

Throttle

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

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

Server-sent Events (Push Notifications)

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
Java script

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

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

Events

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

JavaScript API

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

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

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

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

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

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

Fields

The following field names are defined by the specification:

event

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

data

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

id

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

retry

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

All other field names are ignored.

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

Examples of use:

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

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

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

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

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

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

javascript code to listen "notifications" channel:

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

LoadBalancing

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketLoadBalancerServer

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

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

To enable this feature:

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

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

The component allows load balancing of **WebSocket** and **HTTP** protocols.

Files

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc

This protocol allows sending files from client to server and from server to client in an easy way. You can send from really small files to big files using a low memory usage. You can set:

1. Packet size in bytes.
2. Use custom channels to send files to only subscribed clients.
3. The progress of file send and received.
4. Authorization of files received.
5. Acknowledgement of packets sent.

Proxy

Supported by

TsgcWebSocketClient

Client WebSocket components support WebSocket connections through HTTP proxies. To enable proxy connection, you need to activate the following properties:

Proxy / Enabled

Once set to True, you can set up:

Host: Proxy server address
Port: Proxy server port
UserName/Password: Authentication to connect to proxy, only if required.
ProxyType: the following proxies are supported:

- HTTP
- Socks4
- Socks4A
- Socks5

You can configure SOCKS proxies by accessing the SOCKS property and setting Enabled to True.

Fragmented Messages

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketServer_HTTPAPI

By default, when a stream is sent using sgcWebSockets library, it sends all data in a single packet or buffers all packets and when the latest packet is received, OnBinary message event is called.
This behavior can be customized by the **Options.FragmentedMessages** property, which accepts following values:

1. frgOnlyBuffer: this is the default value. It means that packet messages will be buffered and only when the entire stream is received will the OnBinary message event be called.
2. frgOnlyFragmented: this means that OnFragmented event only will be called for every packet received.
3. frgAll: this means that OnFragmented event will be called for every packet received and when the full stream is received.

OnFragmented event is useful when you must send large streams and the receiver must show progress of the transfer.

Example: the client must send a stream of size 1.000.000 bytes to server and the server wants to show progress for every 1000 bytes received.

The client will send a stream using writedata method with a size for a packet of 1000

```
Client.WriteData(stream, 1000);
```

The server will set in Options.FragmentedMessages := frgAll and will handle OnFragmented event to receive progress of streams

```
void OnFragmented(TsgcWSConnection Connection, TMemoryStream Data, TOpCode OpCode, boolean Continuation)
{
  ShowProgress(Data.Size);
  if (Continuation == false)
  {
    SaveStream(Data);
  }
}
```

TsgcWebSocketClient

TsgcWebSocketClient implements Client WebSocket Component and can connect to a WebSocket Server. Follow the steps below to configure this component:

1. Drop a **TsgcWebSocketClient** component onto the form

2. Set **Host** and **Port** (default is 80) to connect to an available WebSocket Server. You can set **URL** property and Host, Port, Parameters... will be updated from URL. **Example:** wss://127.0.0.1:8080/ws/ will result in:

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

3. You can select if you require **TLS** (secure connection) or not, by default, it is not activated.

4. You can connect through an HTTP Proxy Server, you need to define proxy properties:

Host: proxy server hostname.
Port: proxy server port number.
Username: username for authentication, leave blank for anonymous.
Password: password for authentication, leave blank for anonymous.

5. If the server supports **compression**, you can enable compression to compress messages that are sent.

6. Set **Specifications** allowed, by default, all specifications are enabled.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: always is false

7. If you want, you can handle events

OnConnect: when a WebSocket connection is established, this event is triggered

OnDisconnect: when a WebSocket connection is dropped, this event is triggered

OnError: every time a WebSocket error occurs (like mal-formed handshake), this event is triggered

OnMessage: every time the server sends a text message, this event is triggered

OnBinary: every time the server sends a binary message, this event is triggered

OnFragmented: when a fragment from a message is received (only fired when Options.FragmentedMessages = frgAll or frgOnlyFragmented).

OnHandshake: this event is triggered when the handshake is evaluated on the client side.

OnException: whenever an exception occurs, this event is triggered.

OnSSLVerifyPeer: if verify certificate is enabled, in this event you can verify and decide whether to accept the server certificate.

OnBeforeHeartBeat: if HeartBeat is enabled, allows implementing a custom HeartBeat setting Handled parameter to True (this means, standard websocket ping won't be sent).

OnBeforeConnect: before the client tries to connect to server, this event is called.

OnBeforeWatchDog: if WatchDog is enabled, allows implementing a custom WatchDog setting Handled parameter to True (this means, won't try to connect to server). You can change the Server Connection properties too before try to reconnect, example: connect to a fallback server if first fails.

OnLoadBalancerError: if LoadBalancer is enabled and an error occurs communicating with the Load Balancer Server, this event is triggered.

8. Set the property Active to true to start a new websocket connection

Most common uses

- **Connection**
  - **How to Connect to a WebSocket Server**
  - **Open a Client Connection**
  - **Close a Client Connection**
  - **Keep Connection active**
  - **Dropped Disconnections**
  - **Connect TCP Server**
  - **WebSocket Redirections**
- **Secure Servers**
  - **Connect Secure Server**
  - **Certificates OpenSSL**
  - **Certificates SChannel**
- **Send Messages**
  - **Send Text Message**
  - **Send Binary Message**
- **Receive Messages**
  - Receive Text Messages
  - Receive Binary Messages
- **Authentication**
  - **Client Authentication**
- **Other**
  - **Client Exceptions**
  - **Client WebSocket HandShake**
  - **Client Register Protocol**
  - **Client Proxies**

Methods

WriteData: sends a message to a WebSocket Server. Could be a String or MemoryStream. If "size" is set, the packet will be split if the size of the message is greater of size.

Ping: sends a ping to a Server. If a time-out is specified, it waits for a response until a time-out is exceeded, if no response, then closes the connection.

Start: uses a secondary thread to connect to the server, this prevents your application from freezing while trying to connect.

Stop: uses a secondary thread to disconnect from the server, this prevents your application from freezing while trying to disconnect.

Connect: tries to connect to the server and wait till the connection is successful or there is an error.

Disconnect: tries to disconnect from the server and wait till disconnection is successful or there is an error.

Properties

Authentication: if enabled, WebSocket connection will try to authenticate passing a username and password.

Implements 4 types of WebSocket Authentication / Authorization methods

- **Session:** client needs to do a HTTP GET passing username and password, and if authenticated, server response a Session ID. With this Session ID, client open WebSocket connection passing as a parameter.
- **URL:** client open WebSocket connection passing username and password as a parameter.
- **Basic: uses basic authentication where user and password as sent as HTTP Header.**
- **Token:** sends a token as HTTP Header. Usually used for bearer tokens where token must be set in AuthToken property.
    - **OAuth:** if a OAuth2 component is attached, before client connects to server, it requests a new Access Token to Authorization server. OAuth2 Component.

Host: IP or DNS name of the server.
Port: the listening port of the server.

HeartBeat: if enabled tries to keep the WebSocket connection alive by sending a ping every x seconds.

Interval: number of seconds between each ping.

Timeout: max number of seconds between a ping and pong.

HeartBeatType: allows customizing how the HeartBeat works

- **hbtAlways:** sends a ping every x seconds defined in the Interval.
- **hbtOnlyIfNoMsgRcvInterval:** sends a ping every x seconds only if no messages has been received during the latest x seconds defined in the Interval property.

TCPKeepAlive: if enabled, uses keep-alive at TCP socket level, in Windows will enable SIO_KEEPALIVE_VALS if supported and if not will use keepalive. By default is disabled. Read about Dropped Disconnections.

Time: if after X time socket doesn't sends anything, it will send a packet to keep-alive connection (value in milliseconds).

Interval: after sends a keep-alive packet, if not received a response after interval, it will send another packet (value in milliseconds).

ConnectTimeout: max time in milliseconds before a connection is ready.

LoadBalancer: it's a client which connects to Load Balancer Server to broadcast messages and get information about servers.

Enabled: if enabled, it will connect to Load Balancer Server.

Host: Load Balancer Server Host.

Port: Load Balancer Server Port.

Servers: here you can set manual WebSocket Servers to connect (if you don't make use of Load Balancer Server get server connection methods), example:

```
http://127.0.0.1:80
http://127.0.0.2:8888
```

Connected: returns true if the connection is active. Use this property carefully, because uses internal "connected" Indy method, and this method may lock the thread and/or increment the use of cpu. If you want to know if the client is connected, just use the Active property, which is safer.

ReadTimeout: max time in milliseconds to read messages.

WriteTimeOut: maximum duration in milliseconds for sending data to other peer, 0 by default (only works under Windows OS).

BoundPortMin: minimum local port used by client, by default zero (means there aren't limits).

BoundPortMax: max local port used by client, by default zero (means there aren't limits).

Port: Port used to connect to the host.

LogFile: if enabled, saves socket messages to a specified log file, useful for debugging. The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

UnMaskFrames: by default True, means that saves the websocket messages are sent unmasked.

Raw: by default False, if enabled it will save the messages in hex format.

NotifyEvents: defines which mode to notify WebSocket events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access to controls that are not thread-safe, you need to implement your own synchronization methods.

Options: allows customizing headers sent on the handshake.

FragmentedMessages: allows handling fragmented messages

frgOnlyBuffer: the message is buffered until all data is received, it raises OnBinary or OnMessage event (option by default)
frgOnlyFragmented: every time a new fragment is received, it raises OnFragmented Event.
frgAll: every time a new fragment is received, it raises OnFragmented Event with All data received from the first packet. When all data is received, it raises OnBinary or OnMessage event.

Parameters: define parameters used on GET.

Origin: customize connection origin.

RaiseDisconnectExceptions: enabled by default; raises an exception whenever a protocol error causes a disconnection.

ValidateUTF8: if enabled, validates if the message contains UTF8 valid characters, by default, it is disabled.

CleanDisconnect: if enabled, every time client disconnects from server, first sends a message to inform server connection will be closed.

QueueOptions: this property allows queuing the messages in an internal queue (instead of send directly) and send the messages in the context of the connection thread, this prevents locks when several threads try to send a message. For every message type: Text, Binary or Ping a queue can be configured, by default the value set is **qmNone** which means the messages are not queued. The other types, means different queue levels and the difference between them are just the order where are processed (first are processed **qmLevel1**, then **qmLevel2** and finally **qmLevel3**).
Example: if Text and Binary messages have the property set to qmLevel2 and Ping to qmLevel1. The client will process first the Ping messages (so the ping message is sent first than Text or Binary if they are queued at the same time), and then process the Text and Binary messages in the same queue.

Extensions: you can enable compression on messages are sent.

Protocol: if it exists, shows the current protocol used

Proxy: here you can define if you want to connect through a Proxy Server, you can connect to the following proxy servers:

pxyHTTP: HTTP Proxy Server.
pxySocks4: SOCKS4 Proxy Server.
pxySocks4A: SOCKS4A Proxy Server.
pxySocks5: SOCKS5 Proxy Server.

WatchDog: if enabled, when an unexpected disconnection is detected, tries to reconnect to the server automatically.

Interval: seconds before reconnection attempts.

Attempts: maximum number of reconnection attempts; zero means unlimited.

Throttle: used to limit bits per second sent or received.

TLS: enables a secure connection.

TLSOptions: if TLS enabled, here you can customize some TLS properties.

ALPNProtocols: list of the ALPN protocols which will be sent to server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if certificate is secured with a password, set here.
VerifyCertificate: if certificate must be verified, enable this property.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default negotiates all possible TLS versions from newer to lower. A specific TLS version can be selected.

tlsUndefined: this is the default value, the client attempts to negotiate all available TLS versions (starting from newest to oldest), till connects successfully.
tls1_0: implements TLS 1.0
tls1_1:implements TLS 1.1
tls1_2: implements TLS 1.2
tls1_3: implements TLS 1.3

IOHandler: select which library you will use to connection using TLS.

iohOpenSSL: uses OpenSSL library and is the default for Indy components. Requires to deploy openssl libraries (can be download from the private account of registered customers).
iohSChannel: uses Secure Channel which is a security protocol implemented by Microsoft for Windows, doesn't require to deploy openssl libraries. Only works in Windows 32/64 bits.

OpenSSL_Options: configuration of the openSSL libraries.

APIVersion: allows defining which OpenSSL API will be used.

oslAPI_1_0: uses API 1.0 OpenSSL, it's latest supported by Indy
oslAPI_1_1: uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
oslAPI_3_0: uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).

LibPath: here you can configure where are located the openSSL libraries

oslpNone: this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
oslpDefaultFolder: sets automatically the openSSL path where the libraries should be located for all IDE personalities.
oslpCustomFolder: if this is the option selected, define the full path in the property LibPathCustom.

LibPathCustom: when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.
UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default, it is enabled, except under OSX64):

oslsSymLinksDefault: by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).

oslsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.
oslsSymLinksLoad: Load SymLinks after trying to load the version libraries.
oslsSymLinksDontLoad: don't load the SymLinks.

MinVersion: set here the minimum version that will use the client to connect to a secure server. By default, the value is tlsUndefined which means the minimum version is the same which has been set in the Version property. Example: if you want to set the Client to only connect using TLS 1.2 or TLS 1.3 set the following values.

```
SSLOptions.Version := tls1_3;
SSLOptions.OpenSSL_Options.MinVersion := tls1_2;
```

X509Checks: use this property to enable additional X509 certificate validations:
Mode: select which options will be validated
oslx509chHostName: verifies the hostname certificate.
oslx509chIPAddress: verifies the ip address of the certificate.
HostName: set the hostname if it's different from the request.
IPAddress: set the ip address if it's different from the request.

SChannel_Options: allows you to use a certificate from Windows Certificate Store.
CertHash: is the certificate Hash. You can find the certificate Hash running a dir command in power-shell.
CipherList: here you can set which Ciphers will be used (separated by ":"). Example: CALG_AES_256:CALG_AES_128
CertStoreName: the store name where is stored the certificate. Select one of below:
scsnMY (the default)
scsnCA
scsnRoot
scsnTrust
CertStorePath: the store path where is stored the certificate. Select one of below:
scspStoreCurrentUser (the default)
scspStoreLocalMachine

IPVersion: allows selecting the IP version used for the connection.

ivIP4: uses IPv4.
ivIP6: uses IPv6.

Specifications: allows setting which WebSocket specifications are enabled.

RFC6455: standard and recommended WebSocket specification.
Hixie76: draft specification for old browser compatibility.

Version: shows the current version of the component library.

TsgcWebSocketClient | Connect WebSocket Server

URL Property

The easiest way to connect to a WebSocket server is to use the **URL** property and set **Active = true**.

Example: connect to www.esegece.com using secure connection.

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

Host, Port and Parameters

You can connect to a WebSocket server using Host and port properties.

Example: connect to www.esegece.com using secure connections

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

TsgcWebSocketClient | Client Open Connection

Once your client is configured to connect to a server, there are 3 different options to open a new connection.

Active Property

The easiest way to open a new connection is to set the Active property to true. This will attempt to connect to the server using the component configuration.
If you set the Active property to false, it will close the connection if active.
This method is executed in the same thread as the caller. So if you call it from the Main Thread, the method will be executed in the Main Thread of the application.

Open Connection

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

When you call Active = true, **you still cannot send any data to the server** because the client may still be connecting. You must first wait until the OnConnect event is fired, and then you can start sending messages to the server.

Close Connection

```
oClient.Active = false;
```

When you call Active = false, **you cannot be sure that connection is already closed** just after this code, so you must wait until the OnDisconnect event is fired.

Start/Stop methods

When you call Start() or Stop() to connect/disconnect from the server, the call is executed in a secondary thread, so it does not block the thread where it is called. Use this method if you want to connect to a server and let your code below continue.

Open Connection

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

When you call Start(), **you still cannot send any data to the server** because the client may still be connecting. You must first wait until the OnConnect event is fired, and then you can start sending messages to the server.

Close Connection

```
oClient.Stop();
```

When you call Stop(), **you cannot be sure that connection is already closed** just after this code, so you must wait until the OnDisconnect event is fired.

Connect/Disconnect methods

When you call Connect() or Disconnect() to open/close a connection to the server, the call is executed in the same thread where it is called, but it waits until the process is finished. You must set a Timeout to define the maximum time to wait until the process is finished (by default 10 seconds).

Example: connect to server and wait up to 5 seconds

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

If the Connect() method returns a successful result, you can already send a message to the server because the connection is alive.

Example: disconnect from server and wait up to 10 seconds

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

If the Disconnect() method returns a successful result, this means that the connection is already closed.

OnBeforeConnect event can be used to customize the server connection properties before the client tries to connect to it.

TsgcWebSocketClient | Client Close Connection

Connection can be closed using Active property, Stop or Disconnect methods, read more from Client Open Connection.

CleanDisconnect

When a connection is closed, you can notify the other peer that the connection is being closed by sending a close message. To enable this feature, set the Options.CleanDisconnect property to true.
If this property is enabled, before the connection is closed, a Close message will be sent to the server to notify that the client is closing the connection.

Disconnect

TsgcWSConnection has a method called Disconnect(), that allows you to disconnect the connection at the socket level. If you call this method, the socket will be disconnected directly without waiting for any response from the server. You can send a Close Code with this method.

Close

TsgcWSConnection has a method called Close(), which allows you to send a message to the server requesting to close the connection. If the server receives this message, it must close the connection and the client will receive a notification that the connection is closed. You can send a Close Code with this method.

TsgcWebSocketClient | Client Keep Connection Open

Once your client has connected to a server, sometimes the connection can be closed due to poor signal, connection errors, etc. There are 2 properties that help keep the connection active.

HeartBeat

HeartBeat property allows you to **send a Ping** every X **seconds** to **keep the connection alive**. Some servers close TCP connections if there is no data exchanged between peers. HeartBeat solves this problem by sending a ping at a specific interval. Usually this is enough to maintain a connection active, but you can set a TimeOut interval if you want to close the connection when a response from the server is not received after X seconds.

Example: send a ping every 30 seconds

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

There is an event called **OnBeforeHeartBeat** which allows customizing HeartBeat behavior. By default, if HeartBeat is enabled, the client will send a WebSocket ping every X seconds as set by the HeartBeat.Interval property.
OnBeforeHeartBeat has a parameter called Handled, by default is false, which means the flow is controlled by TsgcWebSocketClient component. If you set the value to True, then ping won't be sent, and you can send your custom message using Connection class.

WatchDog

If WatchDog is enabled, when the client detects a disconnection, WatchDog tries to reconnect every X seconds until the connection is active again.

Example: reconnect every 10 seconds after a disconnection with unlimited attempts.

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

You can use **OnBeforeWatchDog** event to change the server where the client will attempt to connect. **Example:** after 3 retries, if the client cannot connect to a server, it will attempt to connect to a secondary server.
The **Handled** property, if set to True, means that the client won't try to reconnect.

TsgcWebSocketClient | Dropped Disconnections

Once the connection has been established, if no peer sends any data, then no packets are sent over the net. TCP is an idle protocol, so it assumes the connection is still active.

Disconnection reasons

- **Application closes:** when a process is finished, usually sends a FIN packet which acknowledges to the other peer that the connection has been closed. But if a process crashes, there is no guarantee that this packet will be sent to the other peer.
- **Device Closes:** if the device closes, most probably there will be no notification about this.
- **Network cable unplugged:** if the network cable is unplugged it is the same as a router closing; there is no data being transferred so the connection is not closed.
- **Loss of signal from router:** if the application loses signal from the router, the connection will still be alive.

Detect Half-Open Disconnections

You can try to detect disconnections using the following methods

Second Connection

You can try to open a second connection and attempt to connect, but this has some disadvantages: you are consuming more resources, creating new threads, etc. Also, if the other peer has rebooted, the second connection will work but the first will not.

Ping other peer

If you try to send a ping or whatever message with a half-open connection, you will see that you don't get any error.

Enable KeepAlive at TCP Socket level

A TCP keep-alive packet is simply an ACK with the sequence number set to one less than the current sequence number for the connection. A host receiving one of these ACKs responds with an ACK for the current sequence number. Keep-alives can be used to verify that the computer at the remote end of a connection is still available. TCP keep-alives can be sent once every TCPKeepAlive.Time (defaults to 7,200,000 milliseconds or two hours) if no other data or higher-level keep-alives have been carried over the TCP connection. If there is no response to a keep-alive, it is repeated once every TCPKeepAlive.Interval seconds. KeepAliveInterval defaults to 1000 milliseconds.

You can enable per-connection KeepAlive and allow the TCP protocol to check if the connection is active or not. This is the preferred method if you want to detect dropped disconnections (for example: when you unplug a network cable).

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient.TCPKeepAlive.Enabled = true;
oClient.TCPKeepAlive.Time = 5000;
oClient.TCPKeepAlive.Interval = 1000;
```

TsgcWebSocketClient | Connect TCP Server

TsgcWebSocketClient can connect to WebSocket servers, but it can also connect to plain TCP Servers.

URL Property

The easiest way to connect to a TCP server is to use the **URL** property and set **Active = true**.

Example: connect to 127.0.0.1 port 5555

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

Host, Port and Parameters

You can connect to a TCP server using Host and port properties.

Example: connect to 127.0.0.1 port 5555

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

TsgcWebSocketClient | Connections TIME_WAIT

When a client initiates a disconnection from the server, there is an exchange between client and server to communicate the state of the disconnection. When the process is finished, the client socket connection enters the TIME_WAIT state for a variable duration. This is normal behavior; in Windows operating systems, this time defaults to about 4 minutes.

You can reduce or eliminate this behavior, but do so with caution, using the following alternatives.

REGEDIT

You can reduce the TIME_WAIT value using the Windows Regedit

1. Open Regedit and access to HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TCPIP\Parameters registry subkeys.
2. Create a new REG_DWORD value named **TcpTimedWaitDelay**
3. Set the value in seconds. Example: if you set a value of 5, it means that TIME_WAIT will wait for a maximum of 5 seconds.
4. Save and restart the system.

LINGER

Another option to avoid the TIME_WAIT state is to use the socket option SO_LINGER. If enabled, instead of closing the connection gracefully, the client resets the connection so the TIME_WAIT state is avoided.
You can enable this option using the **LingerState** property, which by default has a value of -1. If you set a value of zero, the connection will be reset when disconnecting from the socket without a timeout.
This option is probably the least recommended and should only be used as a last resort.

TsgcWebSocketClient | WebSocket Redirections

When the client connects to a WebSocket server, the server can return an HTTP Response Code 30x. If the response code is 301, it means that the location has been moved permanently, and the new URL is provided in the Location HTTP Header.

The WebSocket client handles redirections automatically, so if it detects that the server response contains a redirection, it will disconnect the current connection and attempt to connect to the new Location URL.

Example

1. Client first tries to connect to url ws://127.0.0.1:5000
2. Server returns a Response Code of 301 and contains a Header Location with the value ws://80.50.1.2:3000
3. Client reads the Response from server, detects that it is a redirection and reads the Location
    1. First disconnects the current connection.
    2. Updates the URL property with the value of the Location Header (ws://80.50.1.2:3000)
    3. Connects to the new server.

TsgcWebSocketClient | Connect Secure Server

TsgcWebSocketClient can connect to WebSocket servers using secure and non-secure connections.

You can configure a secure connection, using URL property or Host / Port properties, see Connect to WebSocket Server.

TLSOptions

In **TLSOptions** property there are the properties to **customize a secure connection**. The most important property is **version**, which specifies the **version of TLS protocol**. Usually setting **TLS property to true** and **TLSOptions.Version to tlsUndefined** is enough for the wide majority of WebSocket Servers.

TLSOptions.Version allows you to set the TLS version used to connect to server or let the client negotiate the TLS version from all available (this is the default when value is **tlsUndefined**).

If you get an **error trying to connect to a server** about TLS protocol, **most probably** the server r**equires a newer TLS version** than what you have set.

If **TLSOptions.IOHandler** is set to **iohOpenSSL**, you need to **deploy OpenSSL libraries** (which are the libraries that handle all TLS stuff), check the following article about OpenSSL.
If **TLSOptions.IOHandler** is set to **iohSChannel**, then there is **no need to deploy** any library (only Windows is supported).

TsgcWebSocketClient | Certificates OpenSSL

When the server requires that the client connects using an SSL Certificate, use the TLSOptions property of TsgcWebSocketClient to set the certificate files. The certificate must be in PEM format, so if the certificate has a different format, it must first be converted to PEM.

Connection through OpenSSL libraries requires that TLSOptions.IOHandler = iohOpenSSL.

Configure the following properties:

- **CertFile:** is the path to the certificate in PEM format.
- **KeyFile:** is the path to the private key of the certificate.
- **RootCertFile:** is the path to the root of the certificate.
- **Password:** if the certificate is protected by a password, set the secret here.

TsgcWebSocketClient | Certificates SChannel

When the server requires that the client connects using an SSL Certificate, use the TLSOptions property of TsgcWebSocketClient to set the certificate files.

Connection through SChannel requires that TLSOptions.IOHandler = iohSChannel.

SChannel supports 2 types of certificate authentication:

1. Using a **PFX certificate**
2. Setting the **Hash Certificate** of an already installed certificate in the windows system.

PFX Certificate

PFX Certificate is a file that contains the certificate and private key, sometimes you have a certificate in PEM format, so before using it you must convert it to PFX.
Use the following openssl command to convert a PEM certificate to PFX

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

Once the certificate is in PFX format, you only need to deploy the certificate and set the **TLSOptions.CertFile** property to its path.

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

Hash Certificate

If the certificate is already installed in the Windows certificate store, you only need to know the certificate thumbprint and set it in the TLSOptions.SChannel_Options property.

Finding the hash of a certificate is as easy in **powershell** as running a **dir** command on the certificates container.

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

The hash is the hexadecimal **Thumbprint** value.

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

Once you have the Thumbprint value, you must set the hash and the certificate location in the **TLSOptions.SChannel_Options** property.

```
TLSOptions.IOHandler = iohSChannel
TLSOptions.SChannel_Options.CertHash = <certificate thumbprint>
TLSOptions.SChannel_Options.CertStoreName = <certificate store name>
```

```
TLSOptions.SChannel_Options.CertStorePath = <certificate store path>
TLSOptions.Password = <certificate optional password>
```

TsgcWebSocketClient | Client Send Text Message

Once the client has connected to the server, it can send text messages. To send a text message, just call the Write-Data() method.

Send a Text Message

Call the **WriteData()** method to send a text message. This method is executed on the **same thread** from which it is called.

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

If **QueueOptions.Text** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

Send a Text Message and Wait for the Response

Call the WriteAndWaitData() method to send a text message and wait for a response from the server. The function returns the text message received.

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

TsgcWebSocketClient | Client Send Binary Message

Once the client has connected to the server, it can send binary messages. To send a binary message, just call the WriteData() method.

Send a Binary Message

Call the **WriteData()** method to send a binary message. This method is executed on the **same thread** from which it is called.

```
byte[] bytes;
...
TsgcWebSocketClient1.WriteData(bytes);
```

If **QueueOptions.Binary** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

Send a Binary Message and Wait for the Response

Call the WriteAndWaitData() method to send a binary message and wait for a response from the server. The function returns the binary message received.

```
TsgcWebSocketClient1.WriteAndWaitDataData(bytes);
```

TsgcWebSocketClient | Client Send a Text and Binary Message

WebSocket protocol only allows two types of messages: Text or Binary. However, you cannot send binary data along with text in the same message.
One way to solve this is to add a header to the binary message before it is sent and decode this binary message when it is received.

There are 2 functions in sgcWebSocket_Helpers which can be used to set a short description of a binary packet. This basically adds a header to the stream which is used to identify the binary packet.

Before sending a binary message, call the method to encode the stream.

```
sgcWSBytesWrite("00001", oBytes);
TsgcWebSocketClient1.WriteData(oBytes);
```

When a binary message is received, call the method to decode the stream.

```
sgcWSBytesRead(oBytes, vID);
```

The only limitation is that the text used to identify the binary message has a maximum length of 10 characters (this can be modified if you have access to source code).

TsgcWebSocketClient | Receive Text Messages

When the client receives a Text Message, the **OnMessage** event is fired. Read the Text parameter to retrieve the string of the message received.

```
void OnMessage(TsgcWSConnection Connection, string Text)
{
  MessageBox.Show("Message Received from Server: " + Text);
}
```

By default, client uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your client receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketClient | Receive Binary Messages

When the client receives a Binary Message, the **OnBinary** event is fired. Read the Data parameter to retrieve the binary message received.

```
private void OnBinary(TsgcWSConnection Connection, byte[] Bytes)
{
  MemoryStream stream = new MemoryStream(Bytes);
  pictureBox1.Image = new Bitmap(stream);
}
```

By default, client uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your client receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketClient | Client Authentication

TsgcWebSocket client supports 4 types of Authentications:

- **Basic:** sends an HTTP Header during WebSocket HandShake with User and Password encoded as Basic Authorization.
- **Token:** sends a Token as an HTTP Header during the WebSocket HandShake. Set the required token in Authentication.Token.AuthToken as required by the server.
- **Session:** first the client requests an HTTP session from the server and if the server returns a session, it is passed in the GET HTTP Header of the WebSocket HandShake. (* own authorization method for sgcWebSockets library).
- **URL:** the client requests authorization using the GET HTTP Header of the WebSocket HandShake. (* own authorization method for sgcWebSockets library).

Authorization Basic

This is a simple authorization method where user and password are encoded and passed as an HTTP Header. Just set the User and Password and enable only the Basic Authorization type to use this method.

```
oClient = new TsgcWebSocketClient();
oClient.Authorization.Enabled = true;
oClient.Authorization.Basic.Enabled = true;
oClient.Authorization.User = "your user";
oClient.Authorization.Password = "your password";
oClient.Authorization.Token.Enabled = false;
oClient.Authorization.URL.Enabled = false;
oClient.Authorization.Session.Enabled = false;
oClient.Active = true;
```

Authorization Token

Allows you to authorize using JWT. This requires you to obtain a token using an external tool (for example: an HTTP connection, OAuth2, etc.).
If you attach an OAuth2 component, you can obtain this token automatically. Read more about OAuth2.
You must set your AuthToken and enable Token Authentication.

```
oClient = new TsgcWebSocketClient();
oClient.Authorization.Enabled = true;
oClient.Authorization.Token.Enabled = true;
oClient.Authorization.Token.AuthToken = "your token";
oClient.Authorization.Basic.Enabled = false;
oClient.Authorization.URL.Enabled = false;
oClient.Authorization.Session.Enabled = false;
oClient.Active = true;
```

Authorization Session

First the client connects to the server using an HTTP connection requesting a new Session. If successful, the server returns a SessionId and the client sends this SessionId in the GET HTTP Header of the WebSocket HandShake. Requires setting the UserName and Password and enabling Session Authentication.

```
oClient = new TsgcWebSocketClient();
oClient.Authorization.Enabled = true;
oClient.Authorization.Session.Enabled = true;
oClient.Authorization.User = "your user";
```

```
oClient.Authorization.Password = "your password";
oClient.Authorization.Basic.Enabled = false;
oClient.Authorization.URL.Enabled = false;
oClient.Authorization.Token.Enabled = false;
oClient.Active = true;
```

Authorization URL

This authentication method passes the username and password in the GET HTTP Header of the WebSocket Hand-Shake.

```
oClient = new TsgcWebSocketClient();
oClient.Authorization.Enabled = true;
oClient.Authorization.URL.Enabled = true;
oClient.Authorization.User = "your user";
oClient.Authorization.Password = "your password";
oClient.Authorization.Basic.Enabled = false;
oClient.Authorization.Session.Enabled = false;
oClient.Authorization.Token.Enabled = false;
oClient.Active = true;
```

TsgcWebSocketClient | Client Exceptions

Sometimes there are errors in communications: the server can disconnect a connection because it is not authorized or a message does not have the correct format. There are 2 events where errors are captured.

OnError

This event is fired every time there is an error in WebSocket protocol, like invalid message type, invalid utf8 string...

```
private void OnError(TsgcWSConnection Connection, string aError)
{
  Console.WriteLine("#error: " + aError);
}
```

OnException

This event is fired every time there is an exception, such as writing to a socket that is not active or accessing an object that does not exist.

```
private void OnException(TsgcWSConnection Connection, Exception E)
{
  Console.WriteLine("#exception: " + E.Message);
}
```

By default, when a **connection is closed by the server**, an **exception will be fired**. If you do not want these exceptions to be fired, disable **Options.RaiseDisconnectExceptions**.

TsgcWebSocketClient | WebSocket Hand-Shake

The WebSocket protocol uses an HTTP HandShake to upgrade from the HTTP protocol to the WebSocket protocol. This handshake is handled internally by the TsgcWebSocket Client component, but you can add your own custom HTTP headers if the server requires additional HTTP header information.

Example: if you need to add this HTTP Header "Client: sgcWebSockets"

```
void OnHandshake(TsgcWSConnection Connection, ref string Headers)
{
  Headers = Headers + Environment.NewLine + "Client: sgcWebSockets";
}
```

You can also check the HandShake string before it is sent to the server using the OnHandShake event.

TsgcWebSocketClient | Client Register Protocol

By default, TsgcWebSocketClient does not use any SubProtocol. WebSocket sub-protocols are built on top of the WebSocket protocol and define a custom message protocol. Examples of WebSocket sub-protocols include MQTT, STOMP, etc.

The WebSocket SubProtocol name is sent as an HTTP Header in the WebSocket HandShake. This header is processed by the server, and if the server supports this subprotocol, it will accept the connection. If it is not supported, the connection will be closed automatically.

Example: connect to a websocket server with SubProtocol name 'myprotocol'

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

TsgcWebSocketClient | Client Proxies

TsgcWebSocket client supports connections through proxies. To configure a proxy connection, just fill in the **Proxy** properties of the TsgcWebSocket client.

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

TsgcWebSocketServer

TsgcWebSocketServer implements Server WebSocket Component and can handle multiple threaded client connections. Follow the steps below to configure this component:

1. Drop a TsgcWebSocketServer component onto the form

2. Set Port (default is 80). If you are behind a firewall probably you will need to configure it.

3. Set Specifications allowed, by default, all specifications are allowed.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: it's a draft and it's only recommended to establish Hixie76 connections if you want to provide support to old browsers like Safari 4.2

4. The following events are available:

OnConnect: every time a WebSocket connection is established, this event is triggered.

OnDisconnect: every time a WebSocket connection is dropped, this event is triggered.

OnError: whenever a WebSocket error occurs (like mal-formed handshake), this event is triggered.

OnMessage: every time a client sends a text message and it's received by server, this event is triggered.

OnBinary: every time a client sends a binary message and it's received by server, this event is triggered.

OnHandshake: this event is triggered after the handshake is evaluated on the server side.

OnException: whenever an exception occurs, this event is triggered.

OnAuthentication: if authentication is enabled, this event is triggered. You can check user and password passed by the client and enable/disable Authenticated Variable.

OnUnknownProtocol: if WebSocket protocol is not detected (because the client is using plain TCP protocol for example), in this event connection can be accepted or rejected.

OnStartup: raised after the server has started.

OnShutdown: raised after the server has stopped.

OnTCPConnect: public event, is called AFTER the TCP connection and BEFORE Websocket handshake. Is useful when your server accepts plain TCP connections. By default the **OnConnect** event is only fired after first message sent by client, if you want to change this behaviour when using plain TCP connections, handle this event and set the connection transport to trpTCP.

```
void OnTCPConnectEvent(SgcWSConnection aConnection, ref bool Accept)
{
    aConnection.Transport = trpTcp;
    Accept = true;
}
```

OnBeforeHeartBeat: if HeartBeat is enabled, allows implementing a custom HeartBeat setting Handled parameter to True (this means, standard websocket ping won't be sent).

OnFragmented: when a fragment from a message is received (only fired when Options.FragmentedMessages = frgAll or frgOnlyFragmented).

OnLoadBalancerConnect: raised when the server connects to the Load Balancer Server.

OnLoadBalancerDisconnect: raised when the server disconnects from the Load Balancer Server.

OnLoadBalancerError: raised when an error occurs communicating with the Load Balancer Server.

OnUnknownAuthentication: if authentication is enabled and the authentication method is not recognized, this event is triggered.

5. Create a procedure and set property Active = True.

Most common uses

- **Start**
  - Server Start
  - Server Bindings
  - Server Startup - Shutdown
  - Server Keep Active
- **Connections**
  - Server Keep Connections Alive
  - Server Plain TCP
  - Server Close Connection
- **Authentication**
  - Server Authentication
- **Send Messages**
  - Server Send Text Message
  - Server Send Binary Message
- **Receive Messages**
  - Server Receive Text Message
  - Server Receive Binary Message
- **SSL**
  - Server SSL
  - Server SSL SChannel

Methods

Broadcast: sends a message to all connected clients.

Message / Stream: message or stream to send to all clients.

Channel: if you specify a channel, the message will be sent only to subscribers.

Protocol: if defined, the message will be sent only to a specific protocol.

Exclude: if defined, list of connection guid excluded (separated by comma).

Include: if defined, list of connection guid included (separated by comma).

WriteData: sends a message to a single or multiple clients. Every time a Client establishes a WebSocket connection, this connection is identified by a Guid, you can use this Guid to send a message to a client.

Ping: sends a ping to all connected clients. If a time-out is specified, it waits a response until a time-out is exceeded, if no response, then closes the connection.

DisconnectAll: disconnects all active connections.

Start: uses a secondary thread to connect to the server, this prevents your application from freezing while trying to connect.

Stop: uses a secondary thread to disconnect from the server, this prevents your application from freezing while trying to disconnect.

Properties

Authentication: if enabled, you can authenticate WebSocket connections against a username and password.

Authusers: is a list of authenticated users, following spec:

user=password

Implements 3 types of WebSocket Authentication

Session: client needs to do an HTTP GET passing username and password, and if authenticated, server response a Session ID. With this Session ID, client open WebSocket connection passing as a parameter.

URL: client open Websocket connection passing username and password as a parameter.

Basic: implements Basic Access Authentication, only applies to VCL Websockets (Server and Client) and HTTP Requests (client web browsers don't implement this type of authentication).

- **CustomHeaders:** here you can add the custom headers that will be sent if there si any authentication error.

Bindings: used to manage IP and Ports.

Count: Connections number count.

LogFile: if enabled, saves socket messages to a specified log file, useful for debugging.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

UnMaskFrames: by default True, means that saves the websocket messages received unmasked.

Extensions: you can enable message compression (if client don't support compression, messages will be exchanged automatically without compression).

FallBack: if WebSockets protocol it's not supported natively by the browser, you can enable the following fallbacks:

Flash: if enabled, if the browser hasn't native WebSocket implementation and has flash enabled, it uses Flash as a Transport.

ServerSentEvents: if enabled, allows you to send push notifications from the server to browser clients.

Retry: interval in seconds to try to reconnect to server (3 by default).

HeartBeat: if enabled, attempts to keep alive WebSocket client connections by sending a ping every x seconds.

Interval: number of seconds between each ping.

Timeout: max number of seconds between a ping and pong.

HeartBeatType: allows customizing how the HeartBeat works

- **hbtAlways:** sends a ping every x seconds defined in the Interval.
- **hbtOnlyIfNoMsgRcvInterval:** sends a ping every x seconds only if no messages has been received during the latest x seconds defined in the Interval property. When using IOHandler = iohDefault, the ping is sent in the context of the connection thread instead of using a separate thread to send a ping to all connected clients.

TCPKeepAlive: if enabled, uses keep-alive at TCP socket level; in Windows, it will enable SIO_KEEPALIVE_VALS if supported, otherwise it will use keepalive. By default is disabled.

Interval: in milliseconds.

Timeout: in milliseconds.

HTTP2Options: by default HTTP/2 protocol is not enabled, it uses HTTP 1.1 to handle HTTP requests. Enable this property if you want to use the HTTP/2 protocol if the client supports it.

Enabled: if true, HTTP/2 protocol is supported. If client doesn't supports HTTP/2, HTTP 1.1 will be used as fallback.

Settings: Specifies the header values to send to the HTTP/2 server.

EnablePush: by default enabled, this setting can be used to avoid server push content to client.

HeaderTableSize: Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block. The initial value is 4,096 octets.

InitialWindowSize: Indicates the sender's initial window size (in octets) for stream-level flow control. The initial value is 65,535 octets. This setting affects the window size of all streams.

MaxConcurrentStreams: Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially, there is no limit to this value.

MaxFrameSize: Indicates the size of the largest frame payload that the sender is willing to receive, in octets. The initial value is 16,384 octets.

MaxHeaderListSize: This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

IOHandlerOptions: by default uses normal Indy Handler (every connection runs in his own thread)

iohDefault: default indy IOHandler, every new connection creates a new thread.

iohIOCP: only for windows and requires sgcWebSockets Enterprise Edition, a thread pool handles all connections. Read more about IOCP.

LoadBalancer: it's a client which connects to Load Balancer Server to broadcast messages and send information about the server.

AutoRegisterBindings: if enabled, sends automatically server bindings to load balancer server.

AutoRestart: time to wait in seconds after a load balancer server connection has been dropped and tries to reconnect; zero means no restart (by default);

Bindings: here you can set manual bindings to be sent to Load Balancer Server, example:

```
WS://127.0.0.1:80
WSS://127.0.0.2:8888
```

Enabled: if enabled, it will connect to Load Balancer Server.

Guid: used to identify server on Load Balancer Server side.

Host: Load Balancer Server Host.

Port: Load Balancer Server Port.

MaxConnections: max connections allowed (if zero there is no limit).

NotifyEvents: defines which mode to notify WebSocket events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access to controls that are not thread-safe, you need to implement your own synchronization methods.

Options:

FragmentedMessages: allows handling fragmented messages

frgOnlyBuffer: the message is buffered until all data is received, it raises OnBinary or OnMessage event (option by default)
frgOnlyFragmented: every time a new fragment is received, it raises OnFragmented Event.
frgAll: every time a new fragment is received, it raises OnFragmented Event with All data received from the first packet. When all data is received, it raises OnBinary or OnMessage event.

HTMLFiles: if enabled, allows you to request Web Browser tests, enabled by default.

JavascriptFiles: if enabled, allows requesting built-in JavaScript libraries, enabled by default.

RaiseDisconnectExceptions: enabled by default; raises an exception every time there's a disconnection due to a protocol error.

ReadTimeOut: time in milliseconds to check if there is data in socket connection, 10 by default.

WriteTimeOut: max time in milliseconds sending data to other peer, 0 by default (only works under Windows OS).

ValidateUTF8: if enabled, validates whether the message contains valid UTF8 characters; by default, it's disabled.

Software: contains the value of the HTTP Server header; the default value is the library name and version.

QueueOptions: this property allows queuing the messages in an internal queue (instead of send directly) and send the messages in the context of the connection thread (QueueOptions only works on Indy based servers where every connection runs in his own thread), this prevents locks when several threads try to send a message using the same connection. For every message type: Text, Binary or Ping a queue can be configured, by default the value set is **qmNone** which means the messages are not queued. The other types, means different queue levels and the difference between them are just the order where are processed (first are processed **qmLevel1**, then **qmLevel2** and finally **qmLevel3**).
Example: if Text and Binary messages have the property set to qmLevel2 and Ping to qmLevel1. The server will process first the Ping messages (so the ping message is sent first than Text or Binary if they are queued at the same time), and then process the Text and Binary messages in the same queue. QueueOptions is not supported when IOHandlerOptions = iohIOCP

ReadEmptySource: max number of times an HTTP Connection is read and there is no data received, 0 by default (means no limit). If the limit is reached, the connection is closed.

SecurityOptions:

OriginsAllowed: define here which origins are allowed (by default accepts connections from all origins), if the origin is not in the list closes the connection. Examples:

- Allow all connections to IP 127.0.0.1 and port 5555. OriginsAllowed = "http://127.0.0.1:5555"
- Allow all connections to IP 127.0.0.1 and all ports. OriginsAllowed = "http://127.0.0.1:*"
- Allow all connections from any IP. OriginsAllowed = ""

SSL: enables secure connections.

SSLOptions: used to define SSL properties: certificates filenames, password...

RootCertFile: path to root certificate file.
CertFile: path to certificate file in PEM format.
KeyFile: path to certificate key file in PEM format.
Password: if certificate is secured with a password, set here.
VerifyCertificate: if certificate must be verified, enable this property.
VerifyCertificate_Options:
FailIfIfNoCertificate: if the client did not return a certificate, the TLS/SSL handshake is immediately terminated with a "handshake failure" alert.
VerifyClientOnce: only request a client certificate on the initial TLS/SSL handshake. Do not ask for a client certificate again in case of a renegotiation.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default negotiates all possible TLS versions from newer to lower. A specific TLS version can be selected.
tlsUndefined: this is the default value, the client attempts to negotiate all available TLS versions (starting from newest to oldest), till connects successfully.
tls1_0: implements TLS 1.0
tls1_1:implements TLS 1.1
tls1_2: implements TLS 1.2
tls1_3: implements TLS 1.3

OpenSSL_Options:

APIVersion: allows defining which OpenSSL API will be used.
osIAPI_1_0: uses API 1.0 OpenSSL, it's latest supported by Indy
osIAPI_1_1: uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
osIAPI_3_0: uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
LibPath: here you can configure where are located the openSSL libraries
osIpNone: this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
osIpDefaultFolder: sets automatically the openSSL path where the libraries should be located for all IDE personalities.
osIpCustomFolder: if this is the option selected, define the full path in the property LibPathCustom.
LibPathCustom: when LibPath = osIpCustomFolder define here the full path where are located the openSSL libraries.
UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
osIsSymLinksDefault: by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
osIsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.
osIsSymLinksLoad: Load SymLinks after trying to load the version libraries.
osIsSymLinksDontLoad: don't load the SymLinks.
ECDHE: if enabled, uses ECDHE instead of RSA as key exchange. Recommended to enable ECDHE if you use OpenSSL 1.0.2.

CipherList: leave blank to use the default ciphers, if you want to customize the cipher list, set the value in this property. Example: ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256
CurveList: leave blank to use the default curves. You can set your own curve list names, for example: P-521:P-384:P-256:brainpoolP256r1
MinVersion: set here the minimum version accepted by the Server. By default, the value is tlsUndefined which means the minimum version is the same which has been set in the Version property. Example: if you want to set the Server to only accept TLS 1.2 and TLS 1.3 set the following values.

```
SSLOptions.Version := tls1_3;
SSLOptions.OpenSSL_Options.MinVersion := tls1_2;
```

X509Checks: use this property to enable additional X509 certificate validations:
Mode: select which options will be validated
oslx509chHostName: verifies the hostname certificate.
oslx509chIPAddress: verifies the ip address of the certificate.
HostName: set the hostname if it's different from the request.
IPAddress: set the ip address if it's different from the request.

ThreadPool: if enabled, when a thread is no longer needed this is put into a pool and marked as inactive (do not consume CPU cycles), it's useful if there are a lot of short-lived connections. The ThreadPool is not compatible with IOCP, so please don't enable it when IOCP is enabled.

MaxThreads: max number of threads to be created, by default is 0 meaning no limit. If max number is reached then the connection is refused.

PoolSize: size of ThreadPool, by default is 32.

WatchDog: if enabled, restarts the server after an unexpected disconnection.

Interval: seconds before reconnecting.

Attempts: maximum number of reconnection attempts; if zero, unlimited.

Throttle: used to limit the number of bits per second sent or received.

Specifications: allows setting which WebSocket specifications are enabled.

RFC6455: standard and recommended WebSocket specification.
Hixie76: draft specification for old browser compatibility.

Firewall: allows configuring a firewall component to filter and protect incoming connections. Assign a TsgcWebSocketFirewall component to enable firewall protection.

ThreadPoolOptions: allows configuring thread pool options for server connections.

TsgcWebSocketServer | Start Server

The first thing you must set when you want to start a server is the listening port. By default, this is set to port 80 but you can change it to any port.
Once the port is set, there are 2 methods to start a server.

Active Property

If you set the Active property to true, the server will start listening for all incoming connections on the configured port.

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

If you set the Active property to false, the server will stop and close all active connections.

```
oServer.Active = false;
```

Start / Stop methods

While setting the Active property starts/stops the server in the same thread, the Start and Stop methods are executed in a secondary thread.

```
oServer = new TsgcWebSocketServer();
oServer.Port = 80;
oServer.Start();
```

If you call the Stop() method, the server will stop and close all active connections.

```
oServer.Stop();
```

You can use the method **ReStart**, to Stop and Start server in a secondary thread.

TsgcWebSocketServer | Server Bindings

By default, if you only fill **Port property**, the server **binds the listening port on ALL IPs**, so if for example you have 3 IPs: 127.0.0.1, 80.54.11.22 and 12.55.41.17, your server will bind this port on all 3 IPs.
It is usually recommended to bind only to the needed IPs. This is where you can use the Bindings property.
Instead of using the Port property, just use the Bindings property and fill in the required IP and Port.

Example: bind Port 5555 to IP 127.0.0.1 and IP 80.58.25.40

```
oServer = new TsgcWebSocketServer();
oServer.Bindings = "127.0.0.1:5555,80.58.25.40:5555";
oServer.Active = true;
```

TsgcWebSocketServer | Server Startup Shutdown

Once you have set all required configurations of your server, there are 2 useful events to know when the server has started and when it has stopped.

OnStartup

This event is fired when the server has started and can process new connections.

```
void OnStartup()
{
  Console.WriteLine("#server started");
}
```

OnShutdown

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

```
void OnShutdown()
{
  Console.WriteLine("#server stopped");
}
```

TsgcWebSocketServer | Server Keep Active

Once the server is started, sometimes it can stop for any reason. If you want to restart the server after an unexpected shutdown, you can use the WatchDog property.

WatchDog

If WatchDog is enabled, when the server detects a shutdown, WatchDog tries to restart every X seconds until the server is active again.

Example: restart every 10 seconds after an unexpected stop with unlimited attempts.

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

TsgcWebSocketServer | Server SSL

The server can be configured to use **SSL Certificates**. In order to get a production server with a server certificate, you must **purchase** a certificate from a **well-known provider**: Namecheap, GoDaddy, Thawte, etc. For **testing purposes** you can use a **self-signed certificate** (check the Demos/Chat example which uses a self-signed certificate).

Certificate must be in **PEM format**, PEM (from Privacy Enhanced Mail) is defined in RFCs 1421 through 1424, this is a container format that may include just the public certificate (such as with Apache installs, and CA certificate files /etc/ssl/certs), or may include an entire certificate chain including public key, private key, and root certificates. To create a single pem certificate, just open your private key file, copy the contents and paste on certificate file.

Example:

certificate.crt

```
-----BEGIN CERTIFICATE-----
.....
-----END CERTIFICATE-----
```

certificate.key

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

certificate.pem

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

To enable SSL, just **enable SSL property** and configure the paths to **CertFile**, **KeyFile** and **RootFile**. If the certificate contains the entire certificate chain (public key, private key, etc.), just set all paths to the same certificate.

Another property you must set is **SSLOptions.Port**, this is the port used for secure connections.

Simple SSL Configuration

Example: configure SSL in IP 127.0.0.1 and Port 443

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

SSL and Non-SSL

You can configure the server to listen on more than one IP and port; check the Binding article which explains how it works. The server can be configured to allow SSL connections and non-SSL connections at the same time (of course, listening on different ports). You only need to bind to two different ports and configure one port for SSL connections and another port for non-SSL connections.

Example: configure server in IP 127.0.0.1, port 80 (non-encrypted) and 443 (SSL)

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

Server SSL | SChannel for Indy Servers

Indy-based server components (**TsgcWebSocketServer**, **TsgcWebSocketHTTPServer**) can use **Windows SChannel** (Secure Channel) as the TLS provider instead of OpenSSL. SChannel is the native Windows TLS implementation, so it does not require any external DLLs.

How it works

When a server component has SSL enabled and the IOHandler is set to **iohSChannel**, the server creates a **TsgcIdServerIOHandlerSSLSChannel** instance that handles all TLS operations using the Windows SChannel API. For every incoming client connection the server performs the TLS handshake through the SChannel provider, negotiating the protocol version, cipher suite, and binding the configured certificate.
SChannel reads certificates from the **Windows Certificate Store** or from a **PFX file** (.pfx / .p12). No PEM files are needed and no OpenSSL libraries need to be deployed.

Configuration

To enable SChannel on an Indy server, configure the following properties:
1. Set the **SSL** property to **True**.
2. Set **SSLOptions.IOHandler** to **iohSChannel**.
3. Set **SSLOptions.Version** to the desired TLS version (tls1_2, tls1_3, ...).
4. Set **SSLOptions.Port** to the port used for secure connections.
5. Configure the certificate using one of the two methods described below.

SChannel_Options properties

The **SSLOptions.SChannel_Options** sub-property exposes the SChannel-specific settings:

Property	Description
CertHash	The thumbprint (hexadecimal hash) of a certificate installed in the Windows Certificate Store.
CertStoreName	Which certificate store to search: **scsnMY** (Personal), scsnRoot, scsnTrust, scsnCA.
CertStorePath	Store location: **scspStoreLocalMachine** or scspStoreCurrentUser.
CipherList	Optional colon-separated list of cipher algorithms (e.g. CALG_AES_256:CALG_AES_128). Empty means use system defaults.
UseLegacyCredentials	When True, uses the legacy SCHANNEL_CRED structure instead of SCH_CREDENTIALS. Enable this for older Windows versions that do not support the newer API.

Additionally, the general **SSLOptions** properties **CertFile** and **Password** are used when loading a certificate from a PFX file.

Certificate from Windows Store

If the certificate is already installed in the Windows Certificate Store, provide the certificate **thumbprint** and indicate where it is located.
To find the thumbprint, open **PowerShell** and run:

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

The **Thumbprint** column shows the hexadecimal hash you need.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
```

```
----------                                  -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

```
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
oServer.SSL = true;
oServer.SSLOptions.IOHandler = TsgcSSLIOHandler.iohSChannel;
oServer.SSLOptions.Version = TsgcTLSVersion.tls1_2;
oServer.SSLOptions.Port = 443;
oServer.Port = 443;
oServer.SSLOptions.SChannel_Options.CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer.SSLOptions.SChannel_Options.CertStoreName = TsgcSChannelCertStoreName.scsnMY;
oServer.SSLOptions.SChannel_Options.CertStorePath = TsgcSChannelCertStorePath.scspStoreLocalMachine;
oServer.Active = true;
```

Certificate from PFX file

If you have a PFX (.pfx or .p12) certificate file, set the **CertFile** and **Password** properties on SSLOptions. SChannel will import the certificate at startup.

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

If you have a PEM certificate and private key, convert them to PFX format first using OpenSSL:

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

TLS Version

Use the **SSLOptions.Version** property to control which TLS version the server accepts:

Value	Description
tls1_0	TLS 1.0 (not recommended)
tls1_1	TLS 1.1
tls1_2	TLS 1.2 (recommended)
tls1_3	TLS 1.3
tlsUndefined	Accept TLS 1.0, 1.1 and 1.2

Cipher List

By default SChannel uses the system cipher configuration. You can restrict the allowed ciphers by setting **SChannel_Options.CipherList** to a colon-separated list of algorithm names, for example:

```
CALG_AES_256:CALG_AES_128
```

Leave this property empty to use the Windows defaults.

Legacy Credentials

Windows Server 2019 and earlier may not support the newer **SCH_CREDENTIALS** API. If the server fails to start on an older Windows version, set **SChannel_Options.UseLegacyCredentials** to **True** to use the legacy **SCHANNEL_CRED** structure instead.
The component detects the Windows version automatically in most cases, but you can force legacy mode if needed.

Advantages over OpenSSL

- **No external DLLs:** SChannel is built into Windows, so you do not need to deploy libeay32.dll / ssleay32.dll or libcrypto / libssl.
- **Windows Certificate Store integration:** use certificates already installed and managed by the operating system.
- **Automatic updates:** TLS improvements and security patches are applied through Windows Update.

Notes

- SChannel is available on **Windows only**. For cross-platform servers, use OpenSSL (iohOpenSSL).
- The server must have the **private key** associated with the certificate. When using the Windows Store method, the certificate must have been imported with its private key.
- For production servers using the Certificate Store method, the **Local Machine** store (scspStoreLocalMachine) is recommended so the certificate is available regardless of which user account runs the service.
- Only **PFX** (.pfx / .p12) certificate files are supported. If you have PEM files, convert them to PFX format first.

TsgcWebSocketServer | Server Verify Certificate

By default, the server does not verify peer certificates. To configure the server to verify client certificates, implement the following steps:

1. Set the property SSLOptions.VerifyCertificate = true

Handle the event OnSSLVerifyPeer and implement the following code to be notified every time a client connects with a certificate.

```
private void OnSSLVerifyPeerEvent(object Sender, TIdX509 Certificate, ref bool Accept)
{
    // ... validate the certificate
    if (Certificate_OK)
        Accept = true;
    else
        Accept = false;
}
```

Note that the event **OnSSLVerifyPeer** is **only called if the client provides a certificate**, if a client doesn't provide a certificate, the event is not fired.

You can configure the **server to only allow SSL connections that use a certificate**. To do this, set the following property:

- SSLOptions.VerifyCertificate_Options.FailIfNoCertificate = true

If the client doesn't provide a certificate, the connection will be closed in the SSL Handshake.

TsgcWebSocketServer | Server Keep Connections Alive

Once a client has connected to the server, sometimes the connection can be closed due to poor signal, connection errors, etc. Use HeartBeat to keep the connection alive.

HeartBeat

The **HeartBeat** property allows you to **send a Ping** every X **seconds** to **maintain the connection alive**. Some clients close TCP connections if there is no data exchanged between peers. HeartBeat solves this problem by sending a ping at a specific interval. Usually this is enough to maintain a connection active, but you can set a Time-Out interval if you want to close the connection when a response from the client is not received after X seconds.

Example: send a ping to all connected clients every 30 seconds

```
oServer = new TsgcWebSocketServer();
oServer.HeartBeat.Interval = 30;
oServer.HeartBeat.Timeout = 0;
oServer.HeartBeat.Enabled = true;
oServer.Active = true;
```

TsgcWebSocketServer | Server Plain TCP

The WebSocket server accepts WebSocket, HTTP, SSE, and other protocols, but can also work with plain TCP connections. Read more about TCP Connections.

There are 2 events that can be used to handle TCP connections.

OnTCPConnect

This event is called after a client connects to the server and before any handshake between client and server. The OnConnect event is only fired after the client sends a message (to allow the server to detect which protocol is being used).

This event allows you to know that a new client is trying to connect to the server, and the server can accept or reject the connection. By default, the server always accepts the connection.

OnUnknownProtocol

This event is called when the server receives the first message from a client but cannot detect whether it is any of the known protocols. In this event, the server can accept or reject the protocol.

OnConnect

This event is fired after a successful and complete connection. If the connection is plain TCP, it is fired after the protocol is accepted in the OnUnknownProtocol event.

TsgcWebSocketServer | Server Close Connection

A single Connection can be closed using Close or Disconnect methods.

Disconnect

TsgcWSConnection has a method called Disconnect(), that allows you to disconnect the connection at the socket level. If you call this method, the socket will be disconnected directly without waiting for any response from the client. You can send a Close Code with this method.

Close

TsgcWSConnection has a method called Close(), which allows you to send a message to the client requesting to close the connection. If the client receives this message, it must close the connection and the server will receive a notification that the connection is closed. You can send a Close Code with this method.

DisconnectAll

Disconnects all active connections. This method is called automatically before the server stops listening, but you can call this method at any time.

TsgcWebSocketServer | Client Connections

To access the active client connections, you can use the Connections property to iterate through the list and access the client connection class. The Connections property accesses a threaded list, so first lock the list and when you are finished, unlock the list.

```
void DoClientIPAddresses()
{
  TList oList = TsgcWebSocketHTTPServer1.LockList();
  try
  {
    for (int i = 0; i < oList.Count; i++)
    {
      TsgcWSConnectionServer oConnection = (TsgcWSConnectionServer)((TIdContext)oList[i]).Data;
      MessageBox.Show(oConnection.IP + ":" + oConnection.Port.ToString());
    }
  }
  finally
  {
    TsgcWebSocketHTTPServer1.UnLockList();
  }
}
```

TsgcWebSocketServer | Server Authentication

TsgcWebSocket server supports 3 types of Authentications:

- **Basic:** reads an HTTP Header during the WebSocket HandShake with User and Password encoded as Basic Authorization.
- **Session:** first the client requests an HTTP session from the server and if the server returns a session, it is passed in the GET HTTP Header of the WebSocket HandShake. (* own authorization method for sgcWebSockets library).
- **URL:** reads the request authorization using the GET HTTP Header of the WebSocket HandShake. (* own authorization method for sgcWebSockets library).

You can set a list of Authenticated users, using **AuthUsers** property, just set your users with the following format: user=password

OnAuthentication

Every time the server receives an Authentication Request from a client, this event is called to indicate whether the user is authenticated or not.
Use Authenticated parameter to accept or not the connection.

```
void OnAuthentication(TsgcWSConnection Connection, string aUser, string aPassword,
  ref bool Authenticated)
{
  if ((aUser == "user") && (aPassword == "secret"))
  {
    Authenticated = true;
  }
  else
  {
    Authenticated = false;
  }
}
```

OnUnknownAuthentication

If the authentication type is not supported by default, such as JWT, you can still use this event to accept or reject the connection. Just read the parameters and decide whether to accept the connection.

```
void OnUnknownAuthenticationEvent(TsgcWSConnection Connection, string AuthType, string AuthData,
  ref string User, ref string Password, ref bool Authenticated)
{
  if (AuthType == "Bearer")
  {
    if (AuthData == "jwt_token")
    {
      Authenticated = true;
    }
    else
    {
      Authenticated = false;
    }
  }
  else
  {
    Authenticated = false;
  }
}
```

TsgcWebSocketServer | Server Send Text Message

Once the client has connected to the server, the server can send text messages. To send a Text Message, call the WriteData() method to send a message to a single client, or use Broadcast to send a message to all clients.

Send a Text Message

Call the **WriteData()** method to send a text message.

```
TsgcWebSocketServer1.WriteData("guid", "My First sgcWebSockets Message!.");
```

If **QueueOptions.Text** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

QueueOptions doesn't work if the property IOHandlerOptions.IOHandlerType = iohIOCP (due to the IOCP architecture, this feature is not supported).

You can also call the WriteData() method from **TsgcWSConnection** too, **example:** send a message to client when connects to server.

```
void OnConnect(TsgcWSConnection Connection)
{
  Connection.WriteData("Hello From Server");
}
```

Send a message to ALL connected clients

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

```
TsgcWebSocketServer1.Broadcast("Hello From Server");
```

TsgcWebSocketServer | Server Send Binary Message

Once the client has connected to the server, the server can send binary messages. To send a Binary Message, call the WriteData() method to send a message to a single client, or use Broadcast to send a message to all clients.

Send a Binary Message

Call the **WriteData()** method to send a binary message.

```
TsgcWebSocketServer1.WriteData("guid", new MemoryStream());
```

If **QueueOptions.Binary** has a **different value from qmNone**, instead of being processed on the same thread that is called, it will be processed on a secondary thread. By default this option is disabled.

QueueOptions doesn't work if the property IOHandlerOptions.IOHandlerType = iohIOCP (due to the IOCP architecture, this feature is not supported).

You can also call the WriteData() method from **TsgcWSConnection** too, **example:** send a message to client when connects to server.

```
void OnConnect(TsgcWSConnection Connection)
{
  Connection.WriteData(new MemoryStream());
}
```

Send a message to ALL connected clients

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

```
TsgcWebSocketServer1.Broadcast(new MemoryStream());
```

TsgcWebSocketServer | Server Receive Text Message

When the server receives a Text Message, the **OnMessage** event is fired. Read the Text parameter to retrieve the string of the message received.

```
void OnMessage(TsgcWSConnection Connection, string Text)
{
  MessageBox.Show("Message Received from Client: " + Text);
}
```

By default, server uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your server receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketServer | Server Receive Binary Message

When the server receives a Binary Message, the **OnBinary** event is fired. Read the Data parameter to retrieve the binary message received.

```
private void OnBinary(TsgcWSConnection Connection, byte[] Bytes)
{
  MemoryStream stream = new MemoryStream(Bytes);
  pictureBox1.Image = new Bitmap(stream);
}
```

By default, server uses **neAsynchronous** method to dispatch OnMessage event, this means that **this event is executed on the context of Main Thread**, so it's thread-safe to update any control of a form for example.

If your server receives lots of messages or you need to control the synchronization with other threads, set NotifyEvents property to **neNoSync**, this means that OnMessage event will be e**xecuted on the context of connection thread**, so if you require to update any control of a form or access shared objects, you must implement your own synchronization methods.

TsgcWebSocketServer | Server Read Headers from Client

When a **client connects** to the WebSocket server, it sends a list of **headers** with information about the client connection. To read these client headers, you can use the **OnHandshake** event of the server component, which is called when the server receives the headers from the client and before it sends a response.
Client headers are stored in **HeadersRequest** property of **TsgcWSConnectionServer**.

```
void OnServerHandshake(TsgcWSConnection Connection, TStringList Headers)
{
  MessageBox.Show(Headers.Text);
}
```

TsgcWebSocketHTTPServer

TsgcWebSocketHTTPServer implements Server WebSocket Component and can handle multiple threaded client connections as TsgcWebSocketServer, and allows you to serve HTML pages using a built-in HTTP Server, sharing the same port for WebSocket connections and HTTP requests.

Follow the steps below to configure this component:

1. Drop a TsgcWebSocketHTTPServer component in the form

2. Set Port (default is 80). If you are behind a firewall probably you will need to configure it.

3. Set Specifications allowed, by default, all specifications are allowed.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: it's a draft and it's only recommended to establish Hixie76 connections if you want to provide support to old browsers like Safari 4.2

4. The following events are available:

OnConnect: every time a WebSocket connection is established, this event is triggered.

OnDisconnect: every time a WebSocket connection is dropped, this event is triggered.

OnError: whenever a WebSocket error occurs (like mal-formed handshake), this event is triggered.

OnMessage: every time a client sends a text message and it's received by server, this event is triggered.

OnBinary: every time a client sends a binary message and it's received by server, this event is triggered.

OnHandshake: this event is triggered after handshake is evaluated on the server side.

OnCommandGet: this event is triggered when HTTP Server receives a GET, POST or HEAD command requesting a HTML page, an image... Example:

```
AResponseInfo.ContentText := '<HTML><HEADER>TEST</HEAD><BODY>Hello!</BODY></
HTML>';
```

OnCommandOther: this event is triggered when HTTP Server receives a command different of GET, POST or HEAD.

OnCreateSession: this event is triggered when HTTP Server creates a new session.

OnInvalidSession: this event is triggered when an HTTP request is using an invalid/expiring session.

OnSessionStart: this event is triggered when HTTP Server starts a new session.

OnSessionEnd: this event is triggered when HTTP Server closes a session.

OnException: this event is triggered when HTTP Server throws an exception.

OnAuthentication: if authentication is enabled, this event if fired. You can check user and password passed by the client and enable/disable Authenticated Variable.

OnUnknownProtocol: if WebSocket protocol is not detected (because the client is using plain TCP protocol for example), in this event connection can be accepted or rejected.

OnBeforeHeartBeat: if HeartBeat is enabled, allows implementing a custom HeartBeat setting Handled parameter to True (this means, standard websocket ping won't be sent).

OnBeforeForwardHTTP: allows you to forward a HTTP request to another HTTP server. Use forward property to enable this and set the destination URL.

OnHTTPUploadBeforeCreatePostStream: this event is called after the headers have been read and before the post stream is created.

OnHTTPUploadBeforeSaveFile: the event is fired when a new file has been uploaded and before is saved to disk file, allows you to modify the filename where will be saved.

OnHTTPUploadAfterSaveFile: the event is fired after a new file has been uploaded and saved to disk file.

OnHTTPUploadReadInput: the event is fired when the form post reads an input variable different from the file.

OnStartup: raised after the server has started.

OnShutdown: raised after the server has stopped.

OnTCPConnect: public event, is called AFTER the TCP connection and BEFORE WebSocket handshake.

OnFragmented: when a fragment from a message is received (only fired when Options.FragmentedMessages = frgAll or frgOnlyFragmented).

OnAfterForwardHTTP: allows you to know the result of the forwarded HTTP request.

OnLoadBalancerConnect: raised when the server connects to the Load Balancer Server.

OnLoadBalancerDisconnect: raised when the server disconnects from the Load Balancer Server.

OnLoadBalancerError: raised when an error occurs communicating with the Load Balancer Server.

OnUnknownAuthentication: if authentication is enabled and the authentication method is not recognized, this event is triggered.

OnHTTP2BeforeAsyncRequest: raised before an HTTP/2 asynchronous request is processed.

* In some cases, you may get a high consume of cpu due to unsolicited connections, in these cases, just return an error 500 if it's a HTTP request or close connection for Unknown Protocol requests.

5. Create a procedure and set property Active = true.

Most common uses

- **HTTP**
  - HTTP Server Requests
  - HTTP Dispatch Files
  - HTTP/2 Server
  - HTTP/2 Server Push
  - HTTP/2 Alternate Service
  - HTTP/2 Server Threads
  - HTTP Post Big Files
- **SSL**
  - Server SSL SChannel

Methods

Broadcast: sends a message to all connected clients.

Message / Stream: message or stream to send to all clients.

Channel: if you specify a channel, the message will be sent only to subscribers.

Protocol: if defined, the message will be sent only to a specific protocol.

Exclude: if defined, list of connection guid excluded (separated by comma).

Include: if defined, list of connection guid included (separated by comma).

WriteData: sends a message to a single or multiple clients. Every time a Client establishes a WebSocket connection, this connection is identified by a Guid, you can use this Guid to send a message to a client.

Ping: sends a ping to all connected clients.

DisconnectAll: disconnects all active connections.

Properties

Connections: contains a list of all clients connections.

Bindings: used to manage IP and Ports.

DocumentRoot: here you can define a directory where you can put all html files (javascript, HTML, CSS...) if a client sends a request, the server automatically will search this file on this directory, if it finds, it will be served.

Extensions: you can enable message compression (if client don't support compression, messages will be exchanged automatically without compression).

MaxConnections: max connections allowed (if zero there is no limit).

Count: Connections number count.

AutoStartSession: if SessionState is active, when the server gets a new HTTP request, creates a new session.

SessionState: if active, enables HTTP sessions.

KeepAlive: if enabled, connection will stay alive after the response has been sent.

ReadStartSSL: max. number of times an HTTPS connection tries to start.

SessionList: read-only property used as a container for TIdHTTPSession instances created for the HTTP server.

SessionTimeOut: timeout of sessions.

HTTP2Options: by default HTTP/2 protocol is not enabled, it uses HTTP 1.1 to handle HTTP requests. Enable this property if you want to use the HTTP/2 protocol if the client supports it.

Enabled: if true, HTTP/2 protocol is supported. If client doesn't supports HTTP/2, HTTP 1.1 will be used as fallback.

FragmentedData: this property allows you to configure how handle the fragments received.

- **h2fdOnlyBuffer:** it's the default option, the response is dispatched only when has been received the latest packet.
- **h2fdAll:** the response is dispatched for every packet received (one or more) on the event OnHTTP2ResponseFragment and on the event OnHTTP2Response when the latest packet has been received.
- **h2fdOnlyFragmented:**: the response is only dispatched in the event OnHTTP2ResponseFragment for every packet received (one response can be compound of 1 or multiple packets).

Settings: Specifies the header values to send to the HTTP/2 server.

EnablePush: by default enabled, this setting can be used to avoid server push content to client.

HeaderTableSize: Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block. The initial value is 4,096 octets.

InitialWindowSize: Indicates the sender's initial window size (in octets) for stream-level flow control. The initial value is 65,535 octets. This setting affects the window size of all streams.

MaxConcurrentStreams: Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially, there is no limit to this value.

MaxFrameSize: Indicates the size of the largest frame payload that the sender is willing to receive, in octets. The initial value is 16,384 octets.

MaxHeaderListSize: This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

Events: here you can configure if you want be notified when there is a new HTTP/2 connection or not.

OnConnect: if enabled when there is a new HTTP/2 connection, OnConnect event will be called (by default is disabled).

OnDisconnect: if enabled when there is a new HTTP/2 disconnection, OnDisconnect event will be called (by default is disabled).

HTTPUploadFiles: by default when a client sends a file using a POST stream, the file is saved in memory. If you want to save these streams directly as files to avoid memory problems, you set the StreamType to pstFileStream and the files will be saved in the hard disk. Read more about Post Big Files.

MinSize: Minimum size in bytes of the stream to be saved as a file stream. By default is zero, which means all streams will be saved as FileStreams (if StreamType = pstFileStream).

RemoveBoundaries: the files uploaded using POST multipart/form-data, are encapsulated in boundaries, if this property is enabled, the files will be extracted from boundaries and saved in the hard disk.

SaveDirectory: the folder where the files will be saved. If empty, will be saved in the same folder where is the application.

StreamType: the type of the stream where the stream will be saved, by default memory.

pstMemoryStream: as memory stream.
pstFileStream: as file stream.

SSL: enables secure connections.

SSLOptions: used to define SSL properties: certificates filenames, password, TLS version, and related settings.

ThreadPool: if enabled, when a thread is no longer needed it is put into a pool and marked as inactive (does not consume CPU cycles). Useful if there are a lot of short-lived connections.

ThreadPoolOptions: allows configuring thread pool options for server connections.

FallBack: if WebSockets protocol is not supported natively by the browser, you can enable fallbacks such as Flash or ServerSentEvents.

Options: allows customizing server options such as FragmentedMessages, HTMLFiles, JavascriptFiles, ReadTimeOut, WriteTimeOut, ValidateUTF8, and RaiseDisconnectExceptions.

QueueOptions: allows queuing messages in an internal queue and sending them in the context of the connection thread, preventing locks when several threads try to send a message.

SecurityOptions: allows defining which origins are allowed for connections.

Specifications: allows setting which WebSocket specifications are enabled (RFC6455, Hixie76).

NotifyEvents: defines which mode to notify WebSocket events (neAsynchronous, neSynchronous, neNoSync).

LogFile: if enabled, saves socket messages to a specified log file, useful for debugging.

Throttle: used to limit the number of bits per second sent or received.

WatchDog: if enabled, restarts the server after an unexpected shutdown.

IOHandlerOptions: allows selecting the IO handler type (iohDefault, iohIOCP, iohEPOLL).

LoadBalancer: allows connecting the server to a Load Balancer Server to broadcast messages and send information about the server.

Authentication: if enabled, you can authenticate WebSocket connections against a username and password.

Firewall: allows configuring a firewall component to filter and protect incoming connections. Assign a TsgcWebSocketFirewall component to enable firewall protection.

Version: shows the current version of the component library.

TsgcWebSocketHTTPServer | HTTP Server Requests

Use OnCommandGet to handle HTTP client requests. Use the following parameters:

- **RequestInfo:** contains HTTP request information.
- **ResponseInfo:** is the HTTP response to HTTP Request.
  - **ContentText:** is the response in text format.
  - **ContentType:** is the type of Content-Type.
  - **ResponseNo:** number of HTTP response, example: 200.

```
void OnCommandGet(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo RequestInfo,
  ref TsgcWSHTTPResponseInfo ResponseInfo)
{
  if (RequestInfo.Document == "/")
  {
    ResponseInfo.ContentText = "<html><head><title>Test Page</title></head><body></body></html>";
    ResponseInfo.ContentType = "text/html";
    ResponseInfo.ResponseNo = 200;
  }
}
```

OnBeforeCommand

Use this event to customize the HTTP response. For example, if you want some endpoints to use an authorization scheme while others can be accessed without authorization, use the options parameter to allow or disable it. Below is an example where Authorization Basic is enabled, but when a user requests the endpoint /public, authorization is not required.

```
public void OnBeforeCommand(TsgcWSConnection aConnection, TIdHTTPRequestInfo ARequestInfo, TIdHTTPResponseInfo AR
{
    if (ARequestInfo.Document == "/public")
        aOptions = TsgcHTTPCommandOptions.hcoAuthorizedBasic;
}
```

TsgcWebSocketHTTPServer | HTTP Dispatch Files

When a client requests a file, the **OnCommandGet** event is fired, but you can use the **DocumentRoot** property to dispatch files automatically.

Example: if you set **DocumentRoot** to **c:/www/files**. Every time a new file is requested, the server will search in this folder. If the file exists, it will be dispatched automatically.

TsgcWebSocketHTTPServer | HTTP/2 Server

sgcWebSockets HTTP Server allows you to handle HTTP/1.1 and HTTP/2.0 requests, you can enable HTTP/2 protocol using HTTP2Options of Server.

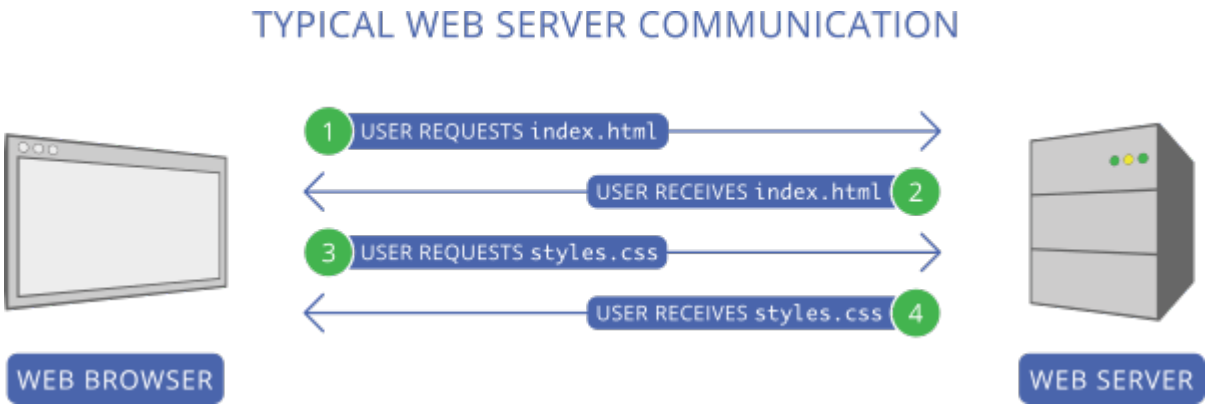
Set **HTTP2Options.Enabled = true** to allow the server to accept HTTP/2 protocol requests. The requests can be processed by the user exactly the same as with the HTTP/1.1 protocol, read more.

When the HTTP/2 protocol is enabled, the server will still support HTTP/1.1 requests.

By default, OnConnect and OnDisconnect events won't be called when there is a new HTTP/2 connection, but this can be modified by accessing the HTTP2Options.Events properties, where you can customize whether you want to be notified every time there is a new HTTP/2 connection and/or disconnection.

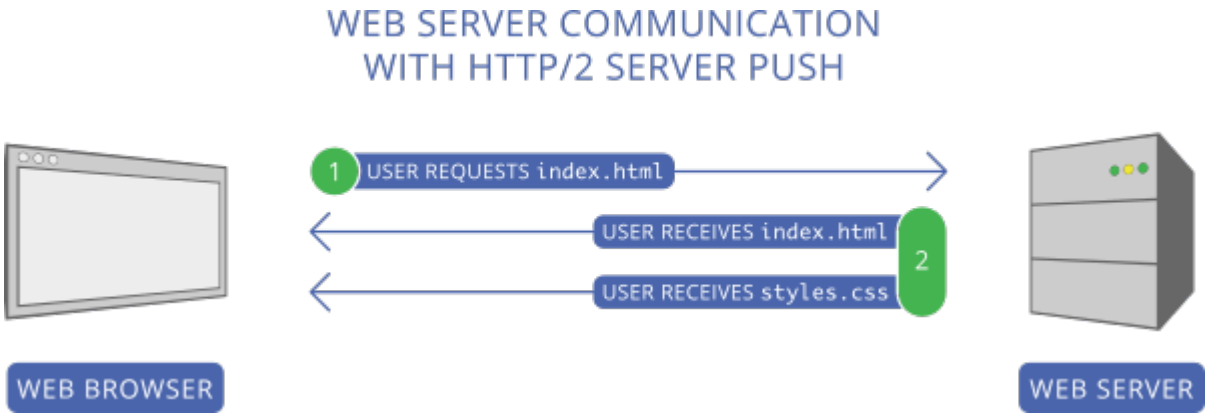
TsgcWebSocketHTTPServer | HTTP/2 Server Push

HTTP usually works with a Request/Response pattern, where the client sends a REQUEST for a resource to the SERVER, and the SERVER sends a RESPONSE with the resource requested or an error. Usually the client, such as a browser, makes a series of requests for assets that are provided by the server.



The main problem with this approach is that the client must first send a request to get the resource (for example, index.html), wait until the server sends the response, read the content, and then make all other requests (for example, styles.css).

HTTP/2 server push tries to solve this problem. When the client requests a file, if the server determines that this file requires additional files, those files will be PUSHED to the client automatically.



In the screenshot above, the client first requests index.html. The server reads this request and sends 2 files as the response: index.html and styles.css, thus avoiding a second request to get styles.css.

Configure Server Push

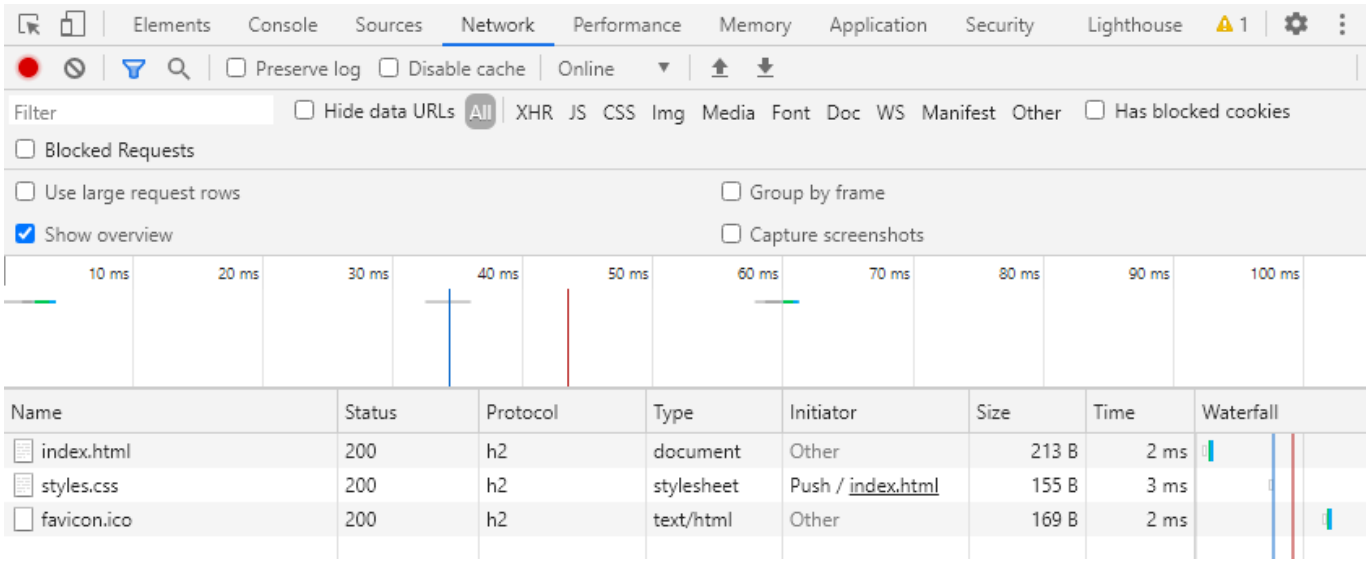
Following the screenshots above, you can configure your server so that every time there is a new request for the / index.html file, the server will send index.html and styles.css.

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

```
TsgcWebSocketHTTPServer server = new TsgcWebSocketHTTPServer(this);
server.PushPromiseAddPreLoadLinks("/index.html", "/styles.css");
void OnCommandGet(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo RequestInfo, ref TsgcWSHTTPResponseInfo Resp
{
  if (RequestInfo.Document == "/index.html")
  {
```

```
    ResponseInfo.ContentText = "";
    ResponseInfo.ContentType = "text/html";
    ResponseInfo.ResponseNo = 200;
  }
  else if (RequestInfo.Document == "/styles.css")
  {
    ResponseInfo.ContentText = "";
    ResponseInfo.ContentType = "text/css";
    ResponseInfo.ResponseNo = 200;
  }
}
```

Using the Chrome developer tool, you can view how the styles.css file is pushed to the client.



TsgcWebSocketHTTPServer | HTTP/2 Alternate Service

The **Alt-Svc HTTP header** is used to **inform** the clients that the **same resource can be reached from another service or protocol**, this is useful if you want to inform the HTTP clients that your server supports HTTP/2 for example.

Example: if your server is running on a local IP 127.0.0.1 and is listening on 2 ports: 80 (non encrypted) and 443 (encrypted). You can inform the clients, that HTTP/2 is supported on port 443 using the following HTTP header

```
Alt-Svc: h2=":443"
```

When HTTP/2 is enabled, this header is automatically added if the connection is not running on the HTTP/2 protocol.
You can enable or disable this feature using the property **HTTP2Options.AltSvc**.

TsgcWebSocketHTTPServer | HTTP/2 Server Threads

See below the differences between HTTP 1.1 and HTTP 2.0:

HTTP 1.1

In traditional HTTP behavior, when making multiple requests over the same connection, the client has to wait for the response of each request before sending the next one. This sequential approach significantly increases the load time of a website's resources. To address this issue, HTTP/1.1 introduced a feature called pipelining, allowing a client to send multiple requests without waiting for the server's responses. The server, in turn, responds to the client in the same order as it received the requests.

While pipelining appeared to be a solution, it faced challenges:

- **Server Ignorance or Response Corruption:** Some servers either ignored pipelined requests or corrupted the responses, leading to unreliable communication.
- **Head-of-Line Blocking:** The first request in the pipeline could block subsequent requests, causing a delay in the processing of other requests. This phenomenon, known as head-of-line blocking, resulted in slower page loading times.

In an effort to optimize page loading from servers supporting HTTP/1.1, web browsers implemented a workaround. They open six to eight parallel connections to the server, enabling the simultaneous transmission of multiple requests. This parallelism aims to mitigate the issues associated with pipelining and improve overall page load times.

The choice of six to eight parallel connections by web browsers is based on optimization considerations. The specific reasons behind selecting this number may involve a trade-off between resource utilization, network efficiency, and avoiding potential bottlenecks.

HTTP 2.0

In response to the constraints encountered in pipelining, HTTP/2 introduced a feature called multiplexing. **Multiplexing** allows for **more efficient communication** between the client and server by enabling the **concurrent transmission** of **multiple requests** and responses **over a single connection**.

HTTP/2 utilizes a binary framing mechanism, which means that HTTP messages are broken down into smaller, independent units called frames. These frames can be interleaved and sent over the connection independently of one another. At the receiving end, the frames are reassembled to reconstruct the original HTTP message.

This binary framing mechanism is fundamental to achieving multiplexing in HTTP/2. It enables the browser to send multiple requests over the same connection without encountering blocking issues. As a result, browsers like Chrome utilize the same connection ID for HTTP/2 requests, allowing for efficient and uninterrupted communication between the client and server.
In essence, HTTP/2's multiplexing feature, enabled by the binary framing mechanism, enhances the efficiency and speed of data exchange between clients and servers by facilitating concurrent transmission of multiple requests and responses over a single connection.

TsgcWebSocketHTTPServer

To improve the performance of the HTTP/2 protocol, the requests are dispatched by default in a pool of threads (by default 32) every time a new HTTP/2 request is received by the server. This avoids waits when a single connection

sends many concurrent requests that would require sequential processing (in the context of the connection thread) in the absence of this pool of threads.

The behavior of the pool of threads can be configured with the following properties.

- **HTTP2Options.PoolOfThreads.Enabled:** (by default false) enable this to dispatch HTTP/2 requests in the pool of threads instead of the connection thread.
- **HTTP2Options.Threads:** (by default 32) the number of threads used to handle HTTP/2 requests. Set a number according to the number of processors on your server.

To **fine-tune the requests**, selecting which requests must be processed in the pool of threads (because they are time-consuming) while others can be processed in the connection thread, you can use the event **OnHttp2BeforeAsyncRequest**. This event is raised before queuing the request in the pool of threads. Use the parameter **Async** to set whether the request is threaded or not.

```
void OnHTTP2BeforeAsyncRequest(object Sender, TsgcWSConnection Connection, TIdHTTPRequestInfo ARequestInfo, ref b
{
  if (ARequestInfo.Document == "/fast-request")
    Async = false;
}
```

TsgcWebSocketHTTPServer | 404 Error without Response Body

By default, the Indy library adds a content body to HTTP responses if there is no ContentText or ContentStream assigned. If you want to return an empty response body (for a 404 error or similar), you can use the following approach.

Create a new TStringStream without content and assign it to the ContentStream property of the HTTP Response. This way the HTTP response will be sent without the default HTML tags.

Example

TsgcWebSocketHTTPServer | Sessions

HTTP is a stateless protocol (at least up to HTTP 1.1), so the client requests a file, the server sends a response, and the connection is closed (you can enable keep-alive so the connection is not closed immediately, but that is beyond the scope of this article). Sessions allow you to store information about the client, which can be used during a client login for example. You can use any unique session ID, search the list of sessions to see if one already exists, and if not, create a new session. A session can be destroyed after a period of inactivity or manually after client logout.

Configuration

There are some properties in TsgcWebSocketHTTPServer that enable/disable sessions in the server component. The most important are:

Property	Description
SessionState	This is the first property that must be enabled in order to use Sessions. Without this property sessions will not work
SessionTimeout	Here you must set a value greater than zero (in milliseconds) for the maximum time a session active.
AutoStartSession	Sessions can be created automatically (AutoStartSession = true) or manually (AutoStartSess false). If sessions are created automatically, the server will use RemoteIP as a unique identifi check if there is an active session stored.

```
TsgcWebSocketHTTPServer1.SessionState = true;
TsgcWebSocketHTTPServer1.SessionTimeout = 600000;
AutoStartSession = false;
```

Create Session

To create a new session, you must create a new **session ID** that is **unique**. You can use any value. **Example:** if the client is authenticating, you can use user + password + remoteip as the session ID.
Then, search the session list to check if it already exists. If it does not exist, create a new one.

When a new session is created **OnSessionStart** event is called and when session is closed, **OnSessionEnd** event is raised.

```
void OnCommandGet(object sender, TsgcHTTPRequestEventArgs e)
{
  if (e.RequestInfo.Document == "/")
  {
    e.ResponseInfo.ServeFile("yourpathhere\\index.html");
  }
  else
  {
    // check if user is valid
    if (!((e.RequestInfo.AuthUsername == "user") && (e.RequestInfo.AuthPassword == "pass")))
    {
      e.ResponseInfo.AuthRealm = "Authenticate";
    }
    else
    {
      // create a new session id with authentication data
      string vID = e.RequestInfo.AuthUsername + "_" + e.RequestInfo.AuthPassword + "_" + e.RequestInfo.RemoteIP;
      // search session
      var oSession = TsgcWebSocketHTTPServer1.SessionList.GetSession(vID, e.RequestInfo.RemoteIP);
```

```
      // create new session if not exists
      if (oSession == null)
        oSession = TsgcWebSocketHTTPServer1.SessionList.CreateSession(e.RequestInfo.RemoteIP, vID);
      e.ResponseInfo.ContentText = "<html><head></head><body>Authenticated</body></html>";
      e.ResponseInfo.ResponseNo = 200;
    }
  }
}
```

TsgcWebSocketHTTPServer | Stream Video

If you want to stream a video file using the server, you can use the function IndyStreamFileVideo to stream the file using chunked transfer encoding.

The function takes the following parameters:

- **AContext:** it's taken from OnCommandGet event.
- **AResponseInfo:** it's taken from OnCommandGet event.
- **aFileName:** it's the full filename of the video to stream.
- **ContentType:** by default is "video/mpeg".
- **aBufferSize:** by default is 1024 bytes.

Example: stream a video when a user goes to the document /video.mp4 and this video is in the folder c:\videos\video.mp4

```
void OnServerCommandGet(TIdContext AContext, TIdHTTPRequestInfo ARequestInfo, TIdHTTPResponseInfo AResponseInfo)
{
  if (ARequestInfo.Document == "/video.mp4")
  {
    IndyStreamFileVideo(AContext, AResponseInfo, "C:\\videos\\video.mp4");
  }
  else
  {
    AResponseInfo.ResponseNo = 404;
  }
}
```

Server SSL | SChannel for Indy Servers

Indy-based server components (**TsgcWebSocketServer**, **TsgcWebSocketHTTPServer**) can use **Windows SChannel** (Secure Channel) as the TLS provider instead of OpenSSL. SChannel is the native Windows TLS implementation, so it does not require any external DLLs.

How it works

When a server component has SSL enabled and the IOHandler is set to **iohSChannel**, the server creates a **TsgcIdServerIOHandlerSSLSChannel** instance that handles all TLS operations using the Windows SChannel API. For every incoming client connection the server performs the TLS handshake through the SChannel provider, negotiating the protocol version, cipher suite, and binding the configured certificate.
SChannel reads certificates from the **Windows Certificate Store** or from a **PFX file** (.pfx / .p12). No PEM files are needed and no OpenSSL libraries need to be deployed.

Configuration

To enable SChannel on an Indy server, configure the following properties:
1. Set the **SSL** property to **True**.
2. Set **SSLOptions.IOHandler** to **iohSChannel**.
3. Set **SSLOptions.Version** to the desired TLS version (tls1_2, tls1_3, ...).
4. Set **SSLOptions.Port** to the port used for secure connections.
5. Configure the certificate using one of the two methods described below.

SChannel_Options properties

The **SSLOptions.SChannel_Options** sub-property exposes the SChannel-specific settings:

Property	Description
CertHash	The thumbprint (hexadecimal hash) of a certificate installed in the Windows Certificate Store.
CertStoreName	Which certificate store to search: **scsnMY** (Personal), scsnRoot, scsnTrust, scsnCA.
CertStorePath	Store location: **scspStoreLocalMachine** or scspStoreCurrentUser.
CipherList	Optional colon-separated list of cipher algorithms (e.g. CALG_AES_256:CALG_AES_128). Empty means use system defaults.
UseLegacyCredentials	When True, uses the legacy SCHANNEL_CRED structure instead of SCH_CREDENTIALS. Enable this for older Windows versions that do not support the newer API.

Additionally, the general **SSLOptions** properties **CertFile** and **Password** are used when loading a certificate from a PFX file.

Certificate from Windows Store

If the certificate is already installed in the Windows Certificate Store, provide the certificate **thumbprint** and indicate where it is located.
To find the thumbprint, open **PowerShell** and run:

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

The **Thumbprint** column shows the hexadecimal hash you need.

```
Directory: Microsoft.PowerShell.Security\Certificate::localmachine\my
Thumbprint                                Subject
```

```
----------                                  -------
C12A8FC8AE668F866B48F23E753C93D357E9BE10  CN=*.mydomain.com
```

```
TsgcWebSocketHTTPServer oServer = new TsgcWebSocketHTTPServer();
oServer.SSL = true;
oServer.SSLOptions.IOHandler = TsgcSSLIOHandler.iohSChannel;
oServer.SSLOptions.Version = TsgcTLSVersion.tls1_2;
oServer.SSLOptions.Port = 443;
oServer.Port = 443;
oServer.SSLOptions.SChannel_Options.CertHash = "C12A8FC8AE668F866B48F23E753C93D357E9BE10";
oServer.SSLOptions.SChannel_Options.CertStoreName = TsgcSChannelCertStoreName.scsnMY;
oServer.SSLOptions.SChannel_Options.CertStorePath = TsgcSChannelCertStorePath.scspStoreLocalMachine;
oServer.Active = true;
```

Certificate from PFX file

If you have a PFX (.pfx or .p12) certificate file, set the **CertFile** and **Password** properties on SSLOptions. SChannel will import the certificate at startup.

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

If you have a PEM certificate and private key, convert them to PFX format first using OpenSSL:

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

TLS Version

Use the **SSLOptions.Version** property to control which TLS version the server accepts:

Value	Description
tls1_0	TLS 1.0 (not recommended)
tls1_1	TLS 1.1
tls1_2	TLS 1.2 (recommended)
tls1_3	TLS 1.3
tlsUndefined	Accept TLS 1.0, 1.1 and 1.2

Cipher List

By default SChannel uses the system cipher configuration. You can restrict the allowed ciphers by setting **SChannel_Options.CipherList** to a colon-separated list of algorithm names, for example:

```
CALG_AES_256:CALG_AES_128
```

Leave this property empty to use the Windows defaults.

Legacy Credentials

Windows Server 2019 and earlier may not support the newer **SCH_CREDENTIALS** API. If the server fails to start on an older Windows version, set **SChannel_Options.UseLegacyCredentials** to **True** to use the legacy **SCHANNEL_CRED** structure instead.
The component detects the Windows version automatically in most cases, but you can force legacy mode if needed.

Advantages over OpenSSL

- **No external DLLs:** SChannel is built into Windows, so you do not need to deploy libeay32.dll / ssleay32.dll or libcrypto / libssl.
- **Windows Certificate Store integration:** use certificates already installed and managed by the operating system.
- **Automatic updates:** TLS improvements and security patches are applied through Windows Update.

Notes

- SChannel is available on **Windows only**. For cross-platform servers, use OpenSSL (iohOpenSSL).
- The server must have the **private key** associated with the certificate. When using the Windows Store method, the certificate must have been imported with its private key.
- For production servers using the Certificate Store method, the **Local Machine** store (scspStoreLocalMachine) is recommended so the certificate is available regardless of which user account runs the service.
- Only **PFX** (.pfx / .p12) certificate files are supported. If you have PEM files, convert them to PFX format first.

TsgcWebSocketServer_HTTPAPI

The HTTP Server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS). Applications can register to receive HTTP requests for particular URLs, receive WebSocket requests, and send WebSocket responses. The HTTP Server API includes SSL support so that applications can exchange data over secure HTTP connections without IIS. It is also designed to work with I/O completion ports.

The server supports the following protocols:

- WebSockets (Requires Windows 8 or later)
- HTTP 1.1
- HTTP/2 (Requires Windows 2016+ or Windows 10+).

By default, this component requires that your application run as Administrator mode, for URL registration. If the URL has already been registered using an external tool like netsh, you can run without Admin rights, disable the property BindingOptions.ConfigureSSLCertificate to allow starting the application without admin rights.
Set FastMM4/FastMM5 as the first unit of your project.

Follow the steps below to configure this component:

1. Drop a TsgcWebSocketServer_HTTPAPI component in the form

2. Define the listening address and port:

```
Server.Host = "127.0.0.1";
Server.Port = 80;
```

3. Set Specifications allowed, by default, all specifications are allowed.

RFC6455: is standard and recommended WebSocket specification.

Hixie76: it's a draft and it's only recommended to establish Hixie76 connections if you want to provide support to old browsers like Safari 4.2

4. If you want, you can handle events:

OnConnect: every time a WebSocket connection is established, this event is triggered.

OnDisconnect: every time a WebSocket connection is dropped, this event is triggered.

OnError: whenever a WebSocket error occurs (like mal-formed handshake), this event is triggered.

OnMessage: every time a client sends a text message and it's received by server, this event is triggered.

OnBinary: every time a client sends a binary message and it's received by server, this event is triggered.

OnHandshake: this event is triggered after the handshake is evaluated on the server side.

OnException: this event is triggered when HTTP Server throws an exception.

OnAuthentication: if authentication is enabled, this event is triggered. You can check user and password passed by the client and enable/disable Authenticated Variable.

OnUnknownProtocol: this event is not currently supported by the HTTP API server.

OnBeforeHeartBeat: if HeartBeat is enabled, allows implementing a custom HeartBeat setting Handled parameter to True (this means, standard websocket ping won't be sent).

OnAsynchronous: every time an asynchronous event has been completed, this event is called.

OnBeforeForwardHTTP: allows you to forward a HTTP request to another HTTP server. Use forward property to enable this and set the destination URL.

OnAfterForwardHTTP: allows you to know the result of the forwarded request.

OnTCPConnect: public event, is called AFTER the TCP connection and BEFORE Websocket handshake.

OnStartup: raised after the server has started.

OnShutdown: raised after the server has stopped.

OnBeforeBinding: raised before the server binds to the configured URL. Allows customizing the binding before it is registered.

OnFragmented: when a fragment from a message is received (only fired when Options.FragmentedMessages = frgAll or frgOnlyFragmented).

OnHTTPRequest: raised when an HTTP request is received by the server.

OnHTTPUploadBeforeCreatePostStream: this event is called after the headers have been read and before the post stream is created.

OnHTTPUploadBeforeSaveFile: the event is fired when a new file has been uploaded and before it is saved to disk, allows you to modify the filename.

OnHTTPUploadAfterSaveFile: the event is fired after a new file has been uploaded and saved to disk.

OnHTTPUploadReadInput: the event is fired when the form post reads an input variable different from the file.

5. Create a procedure and set property Active = true

URL Reservation

The HTTP.SYS server uses URL reservation to assign which URL endpoints will be used by the HTTP.SYS server.

Basic URL Reservation

This is the most easy simple mode to configure the Server, basically you only set the Host and Port that the HTTP.SYS server will handle.
Example: if your server runs on the IP 127.0.0.1 and Port 80, just set the following properties

```
Server.Host = "127.0.0.1";
Server.Port = 80;
```

If the server runs in more than one IP and you want bind to multiple IPS, use the **NewBinding** Method. First clear the Host and Bindings property and then use the NewBinding method to define all Server Bindings.

```
Server.Host = "";
Server.Bindings.Clear;
Server.Bindings.NewBinding("127.0.0.1", 80, '');
Server.Bindings.NewBinding("80.50.55.11", 80, '');
```

If the server requires SSL connections, do the following to define the Host and Port which will be used to handle SSL connections.

```
Server.Host := '127.0.0.1";
```

```
Server.Port := 443;
Server.SSL := true;
Server.SSLOptions.Hash := "CERTIFICATE_HASH";
```

If the server requires SSL connections with multiple IP Addresses, first clear the Host and Bindings property and the register the new Bindings.

```
Server.Host = "";
Server.Bindings.Clear;
Server.Bindings.NewBinding("127.0.0.1", 443, '', true, "CERTIFICATE_HASH1");
Server.Bindings.NewBinding("80.50.55.11", 443, '', true, "CERTIFICATE_HASH2");
```

Most common uses

- **Configuration**
  - URL Reservation
- **Connection**
  - OnDisconnect not fired
- **SSL**
  - HTTPAPI Server SSL
  - Self-Signed Certiifcates
- **HTTP**
  - Custom Headers
  - Send Text Response
  - Send File Response
  - Post Big Files
- **HTTP/2**
  - Disable HTTP/2

Properties

Host: if the property has a value, it will be used to register the URL. If you use the Bindings property to define the server bindings, clear the value of this property.

Port: the default listening port, if the Host property has a value, the Host + Port will be used to register the URL.

Timeouts: allows overriding default timeouts of HTTP API Server.

EntityBody: the time, in seconds, allowed for the request entity body to arrive.
DrainEntityBody: The time, in seconds, allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection.
RequestQueue: The time, in seconds, allowed for the request to remain in the request queue before the application picks it up.
IdleConnection: The time, in seconds, allowed for an idle connection.
HeaderWait: The time, in seconds, allowed for the HTTP Server API to parse the request header.
MinSendRate: The minimum sends rate, in bytes-per-second, for the response. The default response sends rate is 150 bytes-per-second.

MaxConnections: maximum number of connections (zero means unlimited, value by default).

MaxBandwidth: maximum allowed bandwidth rate in bytes per second (zero means unlimited, value by default).

ThreadPoolSize: by default 32 (max recommended value 64), allows setting number of threads of HTTP API Server.

ReadBufferSize: by default 16384, allows you to modify the size of the buffer size when read socket data.

WriteTimeOut: only applies when Asynchronous = False, the value is measured in milliseconds. When this property is greater than zero, if the time to send a message is greater than the value set in the property, the

request is cancelled and the connection is closed. By default, is zero, so there is no timeout writing a message. The internal thread that handles the timeouts, by default uses an interval of 10 seconds, so it means that every 10 seconds checks if there is any message that have exceeded the timeout. You can modify the value of the interval setting the value in the property WriteTimeoutInterval (in seconds, the value must be greater or equal to 5 seconds).

Asynchronous: by default is disabled, if enabled, messages sent don't wait till completed. You can check when asynchronous is completed **OnAsynchronous** event.

SSLOptions: here you can customize ssl properties.

> **CertStoreName:** (optional) allows you to set the name of certificate store where is certificate. If no value is set, 'MY' is assumed as default name.
> **Hash:** this is the hexadecimal thumbprint value of certificate and is required by server to retrieve certificate. You can find hash of certificate using powershell, running a "dir" comand on the certificates store, example: dir cert:\localmachine\my.

HeartBeat: if enabled, attempts to keep alive WebSocket client connections by sending a ping every x seconds.

Extensions: you can enable message compression (if client does not support compression, messages will be exchanged automatically without compression).

Options: allows customizing server options such as FragmentedMessages, ReadTimeOut, WriteTimeOut, ValidateUTF8, and RaiseDisconnectExceptions.

QueueOptions: allows queuing messages in an internal queue and sending them in the context of the connection thread, preventing locks when several threads try to send a message.

SecurityOptions: allows defining which origins are allowed for connections.

Specifications: allows setting which WebSocket specifications are enabled (RFC6455, Hixie76).

LogFile: if enabled, saves socket messages to a specified log file, useful for debugging.

WatchDog: if enabled, restarts the server after an unexpected shutdown.

BindingOptions: allows configuring URL binding options. Set ConfigureSSLCertificate to False to run without Admin rights if the URL has already been registered externally.

Authentication: if enabled, you can authenticate WebSocket connections against a username and password.

Firewall: allows configuring a firewall component to filter and protect incoming connections. Assign a TsgcWebSocketFirewall component to enable firewall protection.

Methods

Broadcast: sends a message to all connected clients.

Message / Stream: message or stream to send to all clients.

Channel: if you specify a channel, the message will be sent only to subscribers.

Protocol: if defined, the message will be sent only to a specific protocol.

Exclude: if defined, list of connection guid excluded (separated by comma).

Include: if defined, list of connection guid included (separated by comma).

WriteData: sends a message to a single or multiple clients. Every time a Client establishes a WebSocket connection, this connection is identified by a Guid, you can use this Guid to send a message to a client.

Ping: sends a ping to all connected clients.

DisconnectAll: disconnects all active connections.

HTTPUploadFiles: by default when a client sends a file using a POST stream, the file is saved in memory. If you want to save these streams directly as files to avoid memory problems, you set the StreamType to pstFileStream and the files will be saved in the hard disk. Read more about Post Big Files.

MinSize: Minimum size in bytes of the stream to be saved as a file stream. By default is zero, which means all streams will be saved as FileStreams (if StreamType = pstFileStream).

RemoveBoundaries: the files uploaded using POST multipart/form-data, are encapsulated in boundaries, if this property is enabled, the files will be extracted from boundaries and saved in the hard disk.

SaveDirectory: the folder where the files will be saved. If empty, will be saved in the same folder where is the application.

StreamType: the type of the stream where the stream will be saved, by default memory.

pstMemoryStream: as memory stream.
pstFileStream: as file stream.

HTTPAPI | URL Reservation

HTTP.SYS URL reservation is a feature in the Windows operating system that allows a user to reserve a specific Uniform Resource Locator (URL) for their application or service. When a URL is reserved using HTTP.SYS, the operating system will intercept any incoming HTTP requests for that URL and route them to the specified application or service.

To reserve a URL using HTTP.SYS, an application or service must first register the URL with the HTTP.SYS driver by making a call to the HTTP API. The application or service specifies the URL, the HTTP method (e.g., GET, POST), and any additional settings such as authentication requirements.

Once the URL is registered, HTTP.SYS will intercept any incoming HTTP requests for that URL and look up the registered application or service based on the URL and method. If a matching application or service is found, the HTTP.SYS driver will pass the request to that application or service for processing.

NETSH Commands

Register an URL
In this example, the URL http://example.com:80/ is being registered for the user DOMAIN\user. You can replace this with your desired URL and user.

```
netsh http add urlacl url=http://example.com:80/ user=DOMAIN\user
```

Delete an URL
In this example, the URL http://example.com:80/ is being deleted. You can replace this with the URL you want to delete.

```
netsh http delete urlacl url=http://example.com:80/
```

Show All URLs
This command will display a list of all registered URL reservations on the system.

```
netsh http show urlacl
```

TsgcWebSocketServer_HTTPAPI

The HTTP.SYS server registers the URLs automatically when it is started. This is done using the following parameters and methods.

- **Host and Port:** if Host not empty and the Port is different from zero, the server will try to register the URL. Example: the URL https://127.0.0.1:5000 will be registered using the following properties
  - Host = '127.0.0.1';
  - Port = 5000
  - SSL = True
- **NewBinding:** use this method to register one or multiple URLs.
  - Register the url https://127.0.0.1:5000 --> NewBinding('127.0.0.1', 5000, '/', True)
  - Register the url http://+:5000/ws/ --> NewBinding('+', 5000, '/ws/')

The URL registration requires admin privileges in the following cases:

- Port Number is below 1024
- The host is a wildcard "+", instead of an ip address.

If you want to register the port 443 for all IP Addresses of the server and listen only on the endpoint "/ws/" but you don't want to run the server with admin rights, do the following steps:

- Register the URL using netsh
  - netsh http add urlacl url=https://+:443/ws/ user=DOMAIN\user
- Configure the server with the following binding
  - NewBinding('+', 443, '/ws/', True);
- Disable the property ConfigureSSLCertificate
  - TsgcWebSocketServer_HTTPAPI.BindingOptions.ConfigureSSLCertificate = false;
- Configure the SSL Certificate
  - HTTPAPI Server SSL

TsgcWebSocketServer_HTTPAPI | HTTPAPI Server SSL

The server can be configured to use **SSL Certificates**. In order to get a production server with a server certificate, you must **purchase** a certificate from a **well-known provider**: Namecheap, GoDaddy, Thawte, etc. For **testing purposes** you can use a **self-signed certificate** (check the Demos/Chat example which uses a self-signed certificate). Read the following article How Create a Self-signed certificate.

Once you have your certificate, you must configure the server to specify which certificate to use for encrypting connections.

Certificate Hash

First you need to know the hash of your certificate. Finding the hash of a certificate is as easy in **powershell** as running a **dir** command on the certificates container.

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

The hash is the hexadecimal **Thumbprint** value.

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

Once you have the Thumbprint value, just set in **TsgcWebSocketServer_HTTPAPI.TLSOptions.Hash** property.

Once you have set the hash, just set TsgcWebSocketServer_HTTPAPI.SSL = true and your server is ready to start.

If you want to register the certificate manually using netsh, use the following command:

```
netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
```

<IP>: Specifies the local IP address for the binding. Do not use a wildcard binding. Use a valid IP address.
<PORT>: Specifies the port for the binding.
<THUMBPRINT>: The X.509 certificate thumbprint.
<GUID>: A developer-generated GUID to represent the app for informational purposes.

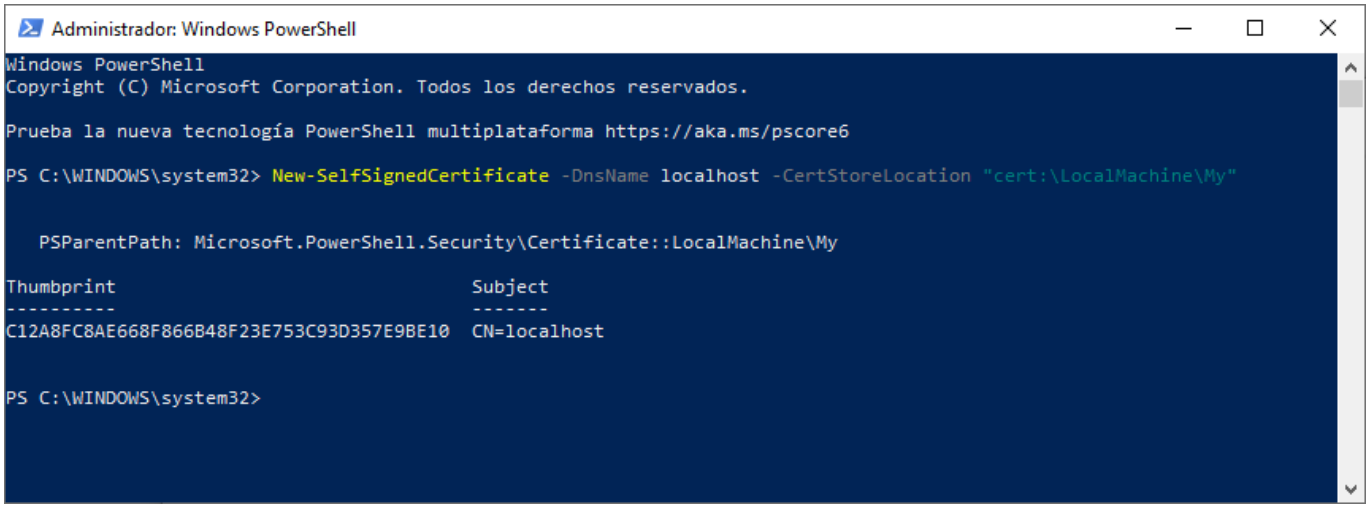
TsgcWebSocketServer_HTTPAPI | Self-Signed Certificates

If you require a certificate for your own testing, you can create a self-signed certificate on your testing machine. Follow these steps:

1. Run **Powershell** as **Administrator**.
2. Run the following command to create the certificate:

```
New-SelfSignedCertificate -DnsName localhost -CertStoreLocation "cert:\LocalMachine\My"
```

If successful, you will get a confirmation about the new certificate created. Just copy Thumbprint and paste on **TsgcWebSocketServer_HTTPAPI.TLSOptions.Hash** property.



3. **Optional**, you can add your self-signed certificate as a **trusted certificate authority**

```
Run MMC -32 as administrator
```

3.1. Select **File / Add or Remove Snap-in**

3.2. Select Certificate and then click Add

3.3. Select computer account and press Next.

3.4. Select Local computer and press Ok. You will now see your Certificates.

4.5. Select your certificate from **Personal / Certificates** and Paste on **Trusted Root Certificates Authorities / Certificates**.

TsgcWebSocketServer_HTTPAPI | Disable HTTP/2

HTTP/2 protocol is enabled by default in **Server 2016+** and **Windows 10+** OS. In some older browsers or HTTP clients, you might encounter an error because the protocol is not fully supported. You can prevent these errors by disabling HTTP/2 protocol.

How to Disable HTTP/2

- Open the Windows Registry Editor
- Go to the following registry key:

  HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\HTTP\Parameters

- Add the following DWORD values and set both values to zero.

  - EnableHttp2Tls
  - EnableHttp2Cleartext

- Reboot the computer.

TsgcWebSocketServer_HTTPAPI | Custom Headers

You can customize the response of HTTP.SYS server using the **CustomHeaders** property of response object.

Set the value of CustomHeaders with the Header Name and Header Value separated by newline characters.

Example: if you want to add the following headers, find below a sample code

```
Access-Control-Allow-Origin: *
Acces-Control-Allow-Methods: GET, POST, OPTIONS, PUT, PATCH, DELETE
```

```
private void OnHTTPRequest(TsgcWSConnection_HTTPAPI aConnection, const THttpServerRequest aRequestInfo,
  ref THttpServerResponse aResponseInfo)
{
  aResponseInfo.ResponseNo = 200;
  aResponseInfo.CustomHeaders = "Access-Control-Allow-Origin: *" + Environment.NewLine + "Acces-Control-Allow-Met
    "GET, POST, OPTIONS, PUT, PATCH, DELETE";
}
```

TsgcWebSocketServer_HTTPAPI | Send Text Response

Use the event **OnHTTPRequest** to handle the HTTP Requests.

The class **THttpServerRequest** contains the HTTP Request Data.

- **Document:** the Document the peer is trying to access.
- **Method:** the HTTP Method ('GET', 'POST', etc.)
- **Headers:** the Headers of HTTP request.
- **AcceptEncoding:** accept encoding variable, example: "gzip, deflate, br".
- **ContentType:** example: "text/html"
- **Content:** content of request if exists.
- **QueryParams:** the query parameters.
- **Cookies:** the cookies if exists.
- **ContentLength:** size of the content.
- **AuthExists,** AuthUsername, AuthPassword: authentication request data.
- **Stream:** if the http request has a body, this is the stream of the body.

The class **THttpServerResponse** contains the HTTP response Data.

- **ContentText:** is the response as text.
- **ContentType:** example: "text/html". If you want encode the ContentText with UTF8, set the charset='utf-8'. Example: text/html; charset=utf-8
- **CustomHeaders:** if you need to send your own headers use this variable
- **AuthRealm:** if the server requires authentication, set this variable.
- **ResponseNo:** the HTTP response number. Example: 200 means the response is successful.
- **ContentStream:** if the response contains a stream, set here (don't free the stream, it will be freed automatically).
- **FileName:** if the response is a filename, set here the full path to the filename.
- **Date, Expires, LastModified:** datetime variables of the response.
- **CacheControl:** allows customizing the cache behavior.

Example: if the server receives a GET request for the document "/test.html", send an OK response; otherwise send a 404 if it is a GET request for another document, or error 500 if it is a different method.

```
void OnHTTPRequest(TsgcWSConnection_HTTPAPI aConnection,
        THttpServerRequest aRequestInfo,
        ref THttpServerResponse aResponseInfo)
{
  if (aRequestInfo.Method == "GET")
  {
    if (aRequestInfo.Document == "/test.html")
        {
          aResponseInfo.ResponseNo = 200;
          aResponseInfo.ContentText = "OK";
          aResponseInfo.ContentType = "text/html; charset=UTF-8";
        }
        else
        {
          aResponseInfo.ResponseNo = 404;
        }
  }
  else
  {
    aResponseInfo.ResponseNo = 500;
  }
}
```

TsgcWebSocketServer_HTTPAPI | Send File Response

Use the FileName property of **THttpServerResponse** object if you want to send a file as a response to an HTTP request.

```
void OnHTTPRequest(TsgcWSConnection_HTTPAPI aConnection,
        THttpServerRequest aRequestInfo,
        ref THttpServerResponse aResponseInfo)
{
  if (aRequestInfo.Method == "GET")
  {
    if (aRequestInfo.Document == "/test.zip")
        {
          aResponseInfo.ResponseNo = 200;
          aResponseInfo.FileName = "c:\download\test.zip";
          aResponseInfo.ContentType = "application/zip";
        }
        else
        {
          aResponseInfo.ResponseNo = 404;
        }
  }
  else
  {
    aResponseInfo.ResponseNo = 500;
  }
}
```

Resumable Downloads

An HTTP 206 Partial Content response is used when a server is fulfilling a request for a specific portion (range) of a resource, instead of sending the entire file. This is commonly used for resumable downloads, media streaming, and large file transfers.
How it works:

Client Requests a Partial Resource: The client (browser, downloader, or media player) sends a `Range` header specifying the byte range it wants. Example request:

```
GET /video.mp4 HTTP/1.1
Host: example.com
Range: bytes=1000-5000
```

This requests bytes 1000 to 5000 of video.mp4.

Server Responds with HTTP 206: If the server supports range requests, it responds with 206 Partial Content and includes a Content-Range header. Example response:

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1000-5000/1000000
Content-Length: 4001
Content-Type: video/mp4
```

The Content-Range header shows:

The range served (1000-5000)
The total size of the file (1000000 bytes).
The Content-Length header is the size of the returned portion (4001 bytes).

Client Can Request More Chunks:
The client can send multiple requests for different parts.
This enables resumable downloads and efficient streaming.

```
[HttpGet("download")]
public async Task<IActionResult> DownloadFile()
{
    string filePath = "test.pdf";
    if (!System.IO.File.Exists(filePath))
    {
        return NotFound();
    }
    FileInfo fileInfo = new FileInfo(filePath);
    long fileSize = fileInfo.Length;
    string contentType = "application/pdf";
    HttpContext.Request.Headers.TryGetValue("Range", out var rangeHeader);

    if (!string.IsNullOrEmpty(rangeHeader))
    {
        long start, end;
        string[] range = rangeHeader.ToString().Replace("bytes=", "").Split('-');
        start = string.IsNullOrEmpty(range[0]) ? 0 : long.Parse(range[0]);
        end = string.IsNullOrEmpty(range[1]) ? fileSize - 1 : long.Parse(range[1]);
        if (end >= fileSize)
            end = fileSize - 1;
        long contentLength = end - start + 1;

        Response.StatusCode = (int)HttpStatusCode.PartialContent;
        Response.Headers["Content-Range"] = $"bytes {start}-{end}/{fileSize}";
        Response.Headers["Accept-Ranges"] = "bytes";
        byte[] buffer = new byte[contentLength];
        using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read, FileShare.Read))
        {
            fs.Seek(start, SeekOrigin.Begin);
            await fs.ReadAsync(buffer, 0, buffer.Length);
        }
        return File(buffer, contentType);
    }
    return File(System.IO.File.OpenRead(filePath), contentType);
}
```

TsgcWebSocketServer_HTTPAPI | OnDisconnect not fired

When first working with the HTTPAPI Server, it is very common to see that the OnDisconnect event is not fired immediately when a client closes the connection. The reason is that HTTPAPI Server works a bit differently than other servers like Indy. In the **Indy server** there is **a thread for every connection** and this thread checks every x milliseconds whether the **connection is active**. The **HTTPAPI Server** uses a **thread-pool** that handles all connections and **does not check** for every connection whether it is active or not.

In order to get notified when client closes connection, do the following configuration:

1. If you use a TsgcWebSocketClient, set **Options.CleanDisconnect := True**. This means that before the connection is closed, the client will try to send a notification to the server that the connection will be closed. If the server receives this message, the OnDisconnect event will be called.

2. For other disconnections, the only solution is to write something to the socket; if it fails, the connection is disconnected. **Enable HeartBeat** on HTTPAPI server, and set an interval of 60 seconds for example and a timeout of 0. This configuration means that every 60 seconds all connections will be pinged, and if any are disconnected, **OnDisconnect** event will be fired. You can put a lower value of HeartBeat.Interval, but do not set it too low (1 second for example is too low) because the performance of the server will be affected.

TsgcWebSocketFirewall

TsgcWebSocketFirewall implements a comprehensive firewall component that protects WebSocket servers against a wide range of security threats. It provides fifteen protection modules including IP blacklist/whitelist filtering, brute force protection with automatic IP banning, SQL injection detection, XSS (Cross-Site Scripting) detection, connection rate limiting, message flood protection, GeoIP country-based filtering, dynamic threat scoring, path traversal detection, command injection detection, payload size limiting, WebSocket-specific protections (origin validation, frame size limits, subprotocol filtering), progressive ban escalation, a custom rules engine, and real-time statistics.

The firewall integrates automatically with server components through the **Firewall** property available on TsgcWebSocketServer, TsgcWebSocketHTTPServer, and TsgcWebSocketServer_HTTPAPI. Once assigned and configured, it intercepts connections and messages without requiring any manual event wiring.

Quick Start

1. Drop a **TsgcWebSocketFirewall** component on the form.

2. Configure the desired protection modules:

Blacklist: Set Enabled to True and add IPs or CIDR ranges to the IPs list (e.g., "10.0.0.0/8", "192.168.1.100").

Whitelist: Set Enabled to True and add trusted IPs. Whitelisted IPs bypass all other checks.

BruteForce: Set Enabled to True. Configure MaxAttempts (default 5), TimeWindowSec (default 60), and BanDurationSec (default 300).

SQLInjection: Set Enabled to True. Detects common SQL injection patterns in messages.

XSS: Set Enabled to True. Detects cross-site scripting patterns in messages.

RateLimit: Set Enabled to True. Limits concurrent connections per IP (default 10 per 60 seconds).

FloodProtection: Set Enabled to True. Limits messages per second per IP (default 100).

PayloadLimit: Set Enabled to True. Blocks messages exceeding MaxSizeBytes (default 1MB).

PathTraversal: Set Enabled to True. Detects directory traversal patterns.

CommandInjection: Set Enabled to True. Detects shell command injection patterns.

ThreatScore: Set Enabled to True. Assigns dynamic scores to IPs based on violations.

BanEscalation: Set Enabled to True. Escalates ban durations for repeat offenders.

GeoIP: Set Enabled to True. Blocks or allows connections by country code.

WebSocketProtection: Set Enabled to True. Validates origins, frame sizes, and subprotocols.

CustomRules: Set Enabled to True. Configurable rule engine for complex filtering.

3. Assign the firewall to a server component:

```
sgcWebSocketHTTPServer1.Firewall := sgcWebSocketFirewall1;
```

4. Start the server. The firewall automatically protects all connections and messages.

Properties

Property	Description
Enabled	Enables or disables the firewall. Default: True.
Blacklist	IP blacklist configuration. Contains Enabled (Boolean) and IPs (TStringList) properties. Supports exact IPs and CIDR notation (e.g., "192.168.0.0/16").
Whitelist	IP whitelist configuration. Whitelisted IPs bypass all security checks. Same structure as Blacklist.
BruteForce	Brute force protection. Properties: Enabled, MaxAttempts (default 5), TimeWindowSec (default 60), BanDurationSec (default 300). Automatically bans IPs that exceed the attempt limit.
SQLInjection	SQL injection detection. Properties: Enabled, Action (faDeny/faAllow/faLog), CustomPatterns (TStringList for additional patterns).
XSS	Cross-site scripting detection. Properties: Enabled, Action (faDeny/faAllow/faLog).
RateLimit	Connection rate limiting. Properties: Enabled, MaxConnectionsPerIP (default 10), TimeWindowSec (default 60).
FloodProtection	Message flood protection. Properties: Enabled, MaxMessagesPerSec (default 100), Action (faDeny/faAllow/faLog).
PayloadLimit	Message size limiting. Properties: Enabled, MaxSizeBytes (default 1048576), Action (faDeny/faAllow/faLog).
PathTraversal	Path traversal detection. Properties: Enabled, Action (faDeny/faAllow/faLog).
CommandInjection	Command injection detection. Properties: Enabled, Action (faDeny/faAllow/faLog), CustomPatterns (TStringList for additional patterns).
ThreatScore	Dynamic threat scoring. Properties: Enabled, AutoBanThreshold (default 80), DecayPerHour (default 5), plus per-violation-type weights for fine-tuning score calculations.
BanEscalation	Progressive ban escalation. Properties: Enabled, Levels (TStringList of durations in seconds), ResetAfterSec (default 86400). Each subsequent ban uses the next level duration. A level value of 0 means permanent ban.
GeoIP	Geographic IP filtering. Properties: Enabled, Mode (gmBlockList/gmAllowList), Countries (TStringList of ISO 3166-1 alpha-2 country codes), DatabaseFile (path to a GeoIP CSV database).
WebSocketProtection	WebSocket-specific protections. Properties: Enabled, AllowedOrigins (TStringList), MaxFrameSize (Integer), AllowedSubprotocols (TStringList).
CustomRules	Custom rules engine. Properties: Enabled, Rules (TCollection of TsgcFirewallRuleItem). Each rule defines conditions and actions for complex filtering logic.

Events

OnFiltered: Fired when a connection or message is blocked. The *Allow* parameter (var Boolean) lets you override the firewall decision.

OnViolation: Fired when a security violation is detected. Provides the IP address, violation type, and details for logging purposes.

OnResolveCountry: Fired when GeoIP needs to resolve an IP to a country code. Assign a handler to provide custom country resolution, or load a GeoIP database file for automatic lookups.

OnThreatScoreChanged: Fired when an IP's threat score changes. Provides the IP address together with the old and new score values.

Public Methods

Method	Description
IsConnectionAllowed(IP)	Checks if a connection from the given IP is allowed. Called automatically when integrated with a server.
IsMessageAllowed(IP, Message)	Checks if a message from the given IP passes all message filters. Called automatically when integrated with a server.
RegisterConnection(IP)	Registers a new connection for rate limiting tracking. Called automatically.
UnregisterConnection(IP)	Removes a connection from tracking. Called automatically on disconnect.
RegisterFailedAttempt(IP)	Records a failed authentication attempt for brute force detection. Call this from your OnAuthentication event handler when authentication fails.
BanIP(IP, DurationSec)	Manually bans an IP address. DurationSec = 0 means permanent ban.
UnbanIP(IP)	Removes a ban from the specified IP address.
IsBanned(IP)	Returns True if the specified IP is currently banned.
ClearBans	Removes all active bans.
ClearTracking	Resets all internal tracking data (connection counts, attempt logs, message counts).
SaveBansToFile(FileName)	Saves all active bans to a file for persistence across restarts.
LoadBansFromFile(FileName)	Restores bans from a previously saved file.
LoadGeoIPDatabase(FileName)	Loads a GeoIP CSV database. Expected format: start_ip,end_ip,country_code.
LookupCountry(IP)	Returns the 2-letter ISO country code for an IP address.
GetThreatScore(IP)	Returns the current threat score (0-100) for an IP address.
ResetThreatScore(IP)	Resets an IP's threat score to 0.
GetTopThreats(Count)	Returns a list of IPs with the highest threat scores.

IsOriginAllowed(Origin)	Checks if a WebSocket Origin header is allowed by the WebSocketProtection configuration.
IsFrameSizeAllowed(Size)	Checks if a message size is within the limits defined by WebSocketProtection.MaxFrameSize.
IsSubprotocolAllowed(Subprotocol)	Checks if a WebSocket subprotocol is allowed by the WebSocketProtection configuration.
Stats	Public read-only TsgcFirewallStats object with real-time counters for connections blocked, messages filtered, bans issued, threat scores, and per-module hit counts.

Automatic Integration

When assigned to a server's Firewall property, the component automatically:

- **On new connection:** Checks blacklist, whitelist, bans, rate limits, GeoIP country, and evaluates custom rules. Rejects the connection before the OnTCPConnect event if blocked.
- **On message received:** Checks SQL injection, XSS patterns, payload size, path traversal, command injection, flood limits, and evaluates custom rules. Updates threat scores. Disconnects the client if a violation is detected.
- **On disconnect:** Unregisters the connection from internal tracking.

CIDR Support

The blacklist and whitelist support CIDR notation for IP ranges:

- **192.168.1.0/24** — matches 192.168.1.0 to 192.168.1.255
- **10.0.0.0/8** — matches all 10.x.x.x addresses
- **172.16.0.0/16** — matches 172.16.x.x addresses

Example

```
// Configure firewall on server
server.Firewall.Enabled = true;
server.Firewall.Blacklist.Enabled = true;
server.Firewall.Blacklist.IPs.Add("10.0.0.0/8");
server.Firewall.Whitelist.Enabled = true;
server.Firewall.Whitelist.IPs.Add("192.168.1.1");
server.Firewall.BruteForce.Enabled = true;
server.Firewall.BruteForce.MaxAttempts = 3;
server.Firewall.BruteForce.BanDurationSec = 600;
server.Firewall.SQLInjection.Enabled = true;
server.Firewall.XSS.Enabled = true;
server.Firewall.RateLimit.Enabled = true;
server.Firewall.RateLimit.MaxConnectionsPerIP = 5;
server.Firewall.FloodProtection.Enabled = true;

// GeoIP: block connections from specific countries
server.Firewall.GeoIP.Enabled = true;
server.Firewall.GeoIP.Mode = GeoIPMode.BlockList;
server.Firewall.GeoIP.Countries.Add("CN");
server.Firewall.GeoIP.Countries.Add("RU");
server.Firewall.LoadGeoIPDatabase("geoip.csv");

// Threat scoring with auto-ban
server.Firewall.ThreatScore.Enabled = true;
server.Firewall.ThreatScore.AutoBanThreshold = 80;
```

```
// Progressive ban escalation (5min, 30min, 2hr, 24hr, permanent)
server.Firewall.BanEscalation.Enabled = true;
server.Firewall.BanEscalation.Levels.Add("300");
server.Firewall.BanEscalation.Levels.Add("1800");
server.Firewall.BanEscalation.Levels.Add("7200");
server.Firewall.BanEscalation.Levels.Add("86400");
server.Firewall.BanEscalation.Levels.Add("0");

// Additional content protection
server.Firewall.PayloadLimit.Enabled = true;
server.Firewall.PayloadLimit.MaxSizeBytes = 65536;
server.Firewall.PathTraversal.Enabled = true;
server.Firewall.CommandInjection.Enabled = true;

// Persistent bans
server.Firewall.SaveBansToFile("bans.dat");
server.Firewall.LoadBansFromFile("bans.dat");

// Optional: Track failed login attempts
server.OnAuthentication += (sender, e) =>
{
    e.Authenticated = (e.User == "admin") && (e.Password == "secret");
    if (!e.Authenticated)
        server.Firewall.RegisterFailedAttempt(e.Connection.IP);
};

// Optional: Handle firewall events for logging
server.Firewall.OnViolation += (sender, e) =>
{
    Log("Firewall violation from " + e.IP + ": " + e.Details);
};
```

SQL Injection Patterns Detected

The built-in SQL injection detector checks for these patterns (case-insensitive):

- ' OR ', ' AND ' — Boolean injection
- UNION SELECT — Union-based injection
- '; DROP, '; DELETE, '; INSERT, '; UPDATE — Statement injection
- -- (SQL comments)
- EXEC(, EXECUTE(— Command execution
- xp_cmdshell — Extended stored procedures
- CAST(, CONVERT(— Type conversion abuse
- 1=1, 1'='1 — Tautology injection

Additional custom patterns can be added via the SQLInjection.CustomPatterns property.

XSS Patterns Detected

- <script — Script injection
- javascript: — Protocol-based XSS
- onerror=, onload=, onclick=, onmouseover= — Event handler injection
- eval(— JavaScript evaluation
- document.cookie — Cookie theft
- <iframe, <object, <embed — Element injection
- <svg onload — SVG-based XSS
- expression(— CSS expression injection

Path Traversal Patterns Detected

The built-in path traversal detector checks for these patterns:

- ../ and ..\ — Relative directory traversal
- %2e%2e%2f, %2e%2e/, ..%2f — URL-encoded variants

- %00 — Null byte injection
- /etc/passwd — Unix sensitive file access
- c:\windows — Windows system directory access

Command Injection Patterns Detected

The built-in command injection detector checks for these patterns:

- ; | && || — Shell operators (command chaining)
- `command` — Backtick execution
- $(command) — Subshell execution
- rm -rf — Destructive file operations
- wget, curl — Remote file download
- /bin/sh, /bin/bash — Unix shell invocation
- cmd.exe — Windows command interpreter
- powershell — PowerShell invocation

Additional custom patterns can be added via the CommandInjection.CustomPatterns property.

Thread Safety

The firewall component is fully thread-safe. All public methods use internal critical sections to protect concurrent access to tracking data. It has been stress-tested with 20 concurrent threads performing 100,000 operations with zero errors and zero memory leaks.

The component can safely be shared across multiple server instances and accessed from any thread (server event handlers, timer threads, etc.) without external synchronization.

Firewall: Blacklist and Whitelist

TsgcWebSocketFirewall provides IP-based access control through two complementary mechanisms: a blacklist that blocks specific IPs and a whitelist that grants unconditional access to trusted IPs.

Blacklist

The blacklist prevents connections from specified IP addresses or IP ranges. When enabled, any incoming connection from a blacklisted IP is rejected before reaching the server's connection events.

Property	Description
Blacklist.Enabled	Enables or disables blacklist checking. Default: False.
Blacklist.IPs	TStringList containing blocked IP addresses or CIDR ranges.

Adding IPs at Design Time

Click the IPs property in the Object Inspector to open the String List editor. Add one IP or CIDR range per line:

```
192.168.1.100
10.0.0.0/8
172.16.0.0/12
```

Adding IPs at Runtime

```
server.Firewall.Blacklist.Enabled = true;
server.Firewall.Blacklist.IPs.Add("192.168.1.100");
server.Firewall.Blacklist.IPs.Add("10.0.0.0/8");
```

Whitelist

The whitelist grants unconditional access to specified IP addresses. **Whitelisted IPs bypass all other firewall checks**, including blacklist, brute force bans, rate limits, and message filtering.

Property	Description
Whitelist.Enabled	Enables or disables whitelist checking. Default: False.
Whitelist.IPs	TStringList containing trusted IP addresses or CIDR ranges.

Example

```
// Allow internal network unconditionally
server.Firewall.Whitelist.Enabled = true;
server.Firewall.Whitelist.IPs.Add("192.168.1.0/24");
server.Firewall.Whitelist.IPs.Add("127.0.0.1");
```

Priority Order

When both blacklist and whitelist are enabled, the firewall evaluates them in this order:
1. If the IP is whitelisted, the connection is **allowed immediately**. No further checks are performed.
2. If the IP is blacklisted, the connection is **denied**.
3. If the IP is in neither list, the connection proceeds to other checks (brute force, rate limiting, etc.).
This means a whitelist entry always takes priority over a blacklist entry for the same IP.

CIDR Notation

Both blacklist and whitelist support CIDR (Classless Inter-Domain Routing) notation for specifying IP ranges:

CIDR	Range	Addresses
192.168.1.0/24	192.168.1.0 - 192.168.1.255	256
192.168.0.0/16	192.168.0.0 - 192.168.255.255	65,536
10.0.0.0/8	10.0.0.0 - 10.255.255.255	16,777,216
172.16.0.0/12	172.16.0.0 - 172.31.255.255	1,048,576

You can mix exact IPs and CIDR ranges in the same list:

```
server.Firewall.Blacklist.IPs.Add("203.0.113.50");     // single IP
server.Firewall.Blacklist.IPs.Add("198.51.100.0/24");  // entire subnet
```

Combining Blacklist and Whitelist

A common pattern is to block a broad range but allow specific IPs within that range:

```
// Block the entire 10.x.x.x range
server.Firewall.Blacklist.Enabled = true;
server.Firewall.Blacklist.IPs.Add("10.0.0.0/8");

// But allow the monitoring server
server.Firewall.Whitelist.Enabled = true;
server.Firewall.Whitelist.IPs.Add("10.1.1.50");
```

In this example, all IPs in the 10.x.x.x range are blocked except 10.1.1.50, which is whitelisted and bypasses all checks.

Firewall: Brute Force Protection

The brute force protection module detects repeated failed authentication attempts from the same IP address and automatically bans the offending IP for a configurable duration.

How It Works

The firewall tracks failed attempts per IP address within a sliding time window. When the number of failures exceeds the configured threshold, the IP is automatically banned. Banned IPs are rejected at the connection level, before any server events fire.

1. A client connects and attempts authentication.
2. If authentication fails, your code calls **RegisterFailedAttempt(IP)** to notify the firewall.
3. The firewall increments the failure counter for that IP within the current time window.
4. If the counter reaches **MaxAttempts**, the IP is banned for **BanDurationSec** seconds.
5. Any new connection from a banned IP is rejected immediately.
6. After the ban duration expires, the IP is automatically unbanned and the failure counter resets.

Properties

Property	Default	Description
BruteForce.Enabled	False	Enables or disables brute force protection.
BruteForce.MaxAttempts	5	Maximum number of failed attempts allowed within the time window before the IP is banned.
BruteForce.TimeWindowSec	60	Duration of the sliding time window in seconds. Failed attempts older than this are discarded.
BruteForce.BanDurationSec	300	Duration of the automatic ban in seconds. Set to 0 for a permanent ban (until manually removed or server restart).

RegisterFailedAttempt

The firewall does not know which connections represent failed logins. You must call **RegisterFailedAttempt** from your authentication event handler whenever a login attempt fails:

```
server.OnAuthentication += (sender, e) =>
{
    e.Authenticated = CheckCredentials(e.User, e.Password);
    if (!e.Authenticated)
        server.Firewall.RegisterFailedAttempt(e.Connection.IP);
};
```

Manual Ban Management

In addition to automatic banning, you can manually manage bans:

Method	Description
BanIP(IP, DurationSec)	Manually bans an IP. Use DurationSec = 0 for a permanent ban.

UnbanIP(IP)	Removes a ban from the specified IP.
IsBanned(IP)	Returns True if the IP is currently banned.
ClearBans	Removes all active bans (both automatic and manual).

Example: Manual Ban

```
// Ban an IP for 1 hour
server.Firewall.BanIP("203.0.113.50", 3600);

// Permanent ban
server.Firewall.BanIP("198.51.100.25", 0);

// Check and unban
if (server.Firewall.IsBanned("203.0.113.50"))
    server.Firewall.UnbanIP("203.0.113.50");
```

Configuration Example

```
// Strict configuration: 3 attempts in 30 seconds, ban for 10 minutes
server.Firewall.BruteForce.Enabled = true;
server.Firewall.BruteForce.MaxAttempts = 3;
server.Firewall.BruteForce.TimeWindowSec = 30;
server.Firewall.BruteForce.BanDurationSec = 600;
```

```
// Lenient configuration: 10 attempts in 5 minutes, ban for 1 minute
server.Firewall.BruteForce.Enabled = true;
server.Firewall.BruteForce.MaxAttempts = 10;
server.Firewall.BruteForce.TimeWindowSec = 300;
server.Firewall.BruteForce.BanDurationSec = 60;
```

Notes

- Whitelisted IPs are never banned, even if RegisterFailedAttempt is called for them.
- Bans are stored in memory by default. Use **SaveBansToFile** and **LoadBansFromFile** to persist bans across server restarts.
- The time window is sliding, not fixed. Each failed attempt is tracked individually and expires after TimeWindowSec seconds.

Firewall: SQL Injection and XSS Detection

TsgcWebSocketFirewall includes built-in pattern-based detection for SQL injection and Cross-Site Scripting (XSS) attacks in WebSocket messages. These modules scan incoming message content and take action when a threat pattern is detected.

SQL Injection Detection

The SQL injection module scans all incoming WebSocket messages for common SQL injection patterns. Detection is case-insensitive.

Properties

Property	Description
SQLInjection.Enabled	Enables or disables SQL injection detection. Default: False.
SQLInjection.Action	Action to take when a pattern is detected: **faDeny** (block the message and disconnect, default), **faLog** (fire OnViolation event but allow the message), **faAllow** (no action, detection disabled).
SQLInjection.CustomPatterns	TStringList of additional regex patterns to check. These are evaluated in addition to the built-in patterns.

Built-in Patterns

The following patterns are checked by default (case-insensitive):

Pattern	Attack Type
' OR ', ' AND '	Boolean-based injection
UNION SELECT	Union-based injection
'; DROP, '; DELETE, '; INSERT, '; UPDATE	Statement injection / data manipulation
-- (double dash)	SQL comment injection
EXEC(, EXECUTE(	Command execution
xp_cmdshell	Extended stored procedure abuse
CAST(, CONVERT(	Type conversion abuse
1=1, 1'='1	Tautology injection

Custom Patterns

You can add application-specific patterns using the CustomPatterns property:

```
server.Firewall.SQLInjection.Enabled = true;
server.Firewall.SQLInjection.CustomPatterns.Add("WAITFOR DELAY");
server.Firewall.SQLInjection.CustomPatterns.Add("BENCHMARK\\(");
server.Firewall.SQLInjection.CustomPatterns.Add("SLEEP\\(");
```

XSS Detection

The XSS module detects Cross-Site Scripting patterns in WebSocket messages, preventing injection of malicious scripts that could be rendered by other connected clients.

Properties

Property	Description
XSS.Enabled	Enables or disables XSS detection. Default: False.
XSS.Action	Action to take when a pattern is detected: **faDeny** (block and disconnect, default), **faLog** (fire OnViolation but allow), **faAllow** (no action).

Built-in Patterns

The following XSS patterns are detected (case-insensitive):

Pattern	Attack Type
<script	Script tag injection
javascript:	Protocol-based XSS
onerror=, onload=, onclick=, onmouseover=	Event handler injection
eval(	JavaScript code evaluation
document.cookie	Cookie theft
<iframe, <object, <embed	Element injection
<svg onload	SVG-based XSS
expression(	CSS expression injection

Action Property

Both SQLInjection and XSS modules support an Action property of type TsgcFirewallAction:

Value	Behavior
faDeny	The message is blocked and the client is disconnected. The OnViolation event fires. This is the default.
faLog	The message is allowed through, but the OnViolation event fires for logging purposes.
faAllow	No action is taken. Effectively disables detection while keeping the Enabled flag True.

OnViolation Event

When a SQL injection or XSS pattern is detected (with Action set to faDeny or faLog), the OnViolation event fires with details about the violation:

```
server.Firewall.OnViolation += (sender, e) =>
{
    switch (e.ViolationType)
    {
        case TsgcFirewallViolationType.fvSQLInjection:
            LogWarning("SQL injection attempt from " + e.IP + ": " + e.Details);
            break;
        case TsgcFirewallViolationType.fvXSS:
            LogWarning("XSS attempt from " + e.IP + ": " + e.Details);
            break;
    }
};
```

Configuration Example

```
// Enable SQL injection detection with deny action
server.Firewall.SQLInjection.Enabled = true;
server.Firewall.SQLInjection.Action = TsgcFirewallAction.faDeny;

// Add custom patterns for time-based SQL injection
server.Firewall.SQLInjection.CustomPatterns.Add("WAITFOR DELAY");
server.Firewall.SQLInjection.CustomPatterns.Add("SLEEP\\(");

// Enable XSS detection in log-only mode (monitor without blocking)
server.Firewall.XSS.Enabled = true;
server.Firewall.XSS.Action = TsgcFirewallAction.faLog;

// Assign violation handler
server.Firewall.OnViolation += FirewallViolation;
```

Notes

- Pattern detection is applied to all incoming text messages. Binary messages are not scanned.
- Whitelisted IPs bypass message filtering entirely.
- The built-in patterns cover the most common attack vectors. For application-specific threats, add custom patterns via SQLInjection.CustomPatterns.
- Use faLog mode during initial deployment to monitor detections without impacting clients, then switch to faDeny once you have validated the patterns for your application.

Firewall: Rate Limiting and Flood Protection

TsgcWebSocketFirewall provides two complementary mechanisms for controlling traffic: connection rate limiting (controlling how many connections an IP can open) and message flood protection (controlling how many messages a connected client can send per second).

Connection Rate Limiting

Rate limiting restricts the number of connections a single IP address can establish within a time window. This prevents a single client from consuming all available server connections.

Properties

Property	Default	Description
RateLimit.Enabled	False	Enables or disables connection rate limiting.
RateLimit.MaxConnectionsPerIP	10	Maximum number of concurrent connections allowed from a single IP address.
RateLimit.TimeWindowSec	60	Time window in seconds for tracking connections. Connections older than this are removed from the count.

How It Works

1. When a new connection arrives, the firewall counts the number of active connections from that IP.
2. If the count is at or above MaxConnectionsPerIP, the new connection is rejected.
3. When a connection disconnects, it is removed from the tracking count.
4. Connections older than TimeWindowSec are automatically cleaned from tracking.

Example

```
// Allow maximum 5 connections per IP
server.Firewall.RateLimit.Enabled = true;
server.Firewall.RateLimit.MaxConnectionsPerIP = 5;
server.Firewall.RateLimit.TimeWindowSec = 60;
```

Message Flood Protection

Flood protection limits the number of messages a single IP can send per second. This prevents abusive clients from overwhelming the server with rapid message bursts.

Properties

Property	Default	Description
FloodProtection.Enabled	False	Enables or disables message flood protection.
FloodProtection.MaxMessagesPerSec	100	Maximum number of messages allowed per second from a single IP address.

FloodProtection.Action	faDeny	Action when the limit is exceeded: **faDeny** (disconnect the client), **faLog** (fire OnViolation but allow), **faAllow** (no action).

How It Works

1. The firewall maintains a per-IP message counter that resets every second.
2. Each incoming message increments the counter for the sender's IP.
3. When the counter exceeds MaxMessagesPerSec, the configured Action is taken.
4. With faDeny action, the client is disconnected immediately. With faLog, the OnViolation event fires but the message is processed normally.

Example

```
// Limit to 50 messages per second per IP
server.Firewall.FloodProtection.Enabled = true;
server.Firewall.FloodProtection.MaxMessagesPerSec = 50;
server.Firewall.FloodProtection.Action = TsgcFirewallAction.faDeny;
```

Combining Both Protections

Rate limiting and flood protection work together to provide comprehensive traffic control:

```
// Connection limiting: max 5 connections per IP
server.Firewall.RateLimit.Enabled = true;
server.Firewall.RateLimit.MaxConnectionsPerIP = 5;

// Message limiting: max 50 messages/sec per IP
server.Firewall.FloodProtection.Enabled = true;
server.Firewall.FloodProtection.MaxMessagesPerSec = 50;
```

With this configuration, each IP can open at most 5 concurrent connections, and across all those connections, it can send a combined maximum of 50 messages per second.

Tuning Guidelines

Scenario	MaxConnectionsPerIP	MaxMessagesPerSec
Chat application	3-5	10-30
Real-time data feed	2-3	100-500
Gaming server	1-2	50-100
API gateway	10-20	200-1000

Start with conservative limits and increase them based on monitoring. Use faLog mode initially to observe traffic patterns before enabling faDeny.

Notes

- Whitelisted IPs bypass both rate limiting and flood protection.
- Rate limit tracking data is cleared automatically when connections close. Use ClearTracking to reset all counters manually.
- Behind a reverse proxy, all connections may appear to come from the proxy IP. Configure your proxy to pass the real client IP (e.g., X-Forwarded-For header) and ensure the server resolves it correctly.

TsgcWebSocketLoadBalancerServer

The component **TsgcWebSocketLoadBalancerServer** allows you to load-balance **WebSocket** and **HTTP** protocols. For the WebSocket protocol, it distributes messages across a group of servers and distributes client connections using a random sequence or fewest-connections algorithm.

The Load Balancer Server inherits all methods and properties from TsgcWebSocketHTTPServer.

Load Balancer Configuration

The Load Balancer server is a descendant of TsgcWebSocketHTTPServer, so read the documentation about the TsgcWebSocketHTTPServer to know how to configure it.

Additionally, the Load Balancer has the property LoadBalancer, which has the following properties:

- **LoadBalancing**: configure here how to distribute the connections
  - **lbRandom:** (default) every time a new client requests a connection, it will return a random server.
  - **lbConnections:** every time a new client requests a connection, it will return the server with the fewest connected clients.
- Protocols: configure which protocols are enabled
  - **WebSocket:** if true, the websocket connections will be handled by the Load Balancer Server.
  - **HTTP:** if true, the http connections will be handled by the Load Balancer Server.

Backup Server Configuration

The Backup Servers (the servers behind the load balancer) can be a TsgcWebSocketServer, TsgcWebSocketHTTPServer or a Datasnap Server.
Those servers have a property called **LoadBalancer** where you can configure the connection between the LoadBalancer Server and the Backup Servers.

- **Enabled:** set to true if you want to use as a backup server.
- **Host:** the host where the LoadBalancer is located.
- **Port:** the listening port of the LoadBalancer.
- **Guid:** unique id that identifies this server.
- **Bindings:** the public addresses where the connections will be forwarded. Example: if the Backup WebSocket server is listening on port 8000 and the ip address is 1.1.1.1, use the following: ws://1.1.1.1:8000;
- **AutoRegisterBindings:** if enabled, the LoadBalancer Server will use the Bindings property of the backup server to configure the public bindings.
- **AutoRestart:** in seconds, if greater than zero, the load balancer client of the backup server will enable an internal watchdog that every x seconds, will check if the connection is alive, if it's closed, it will try to reconnect.

Events

- **OnBeforeSendServerBinding:** raised before binding is sent to a new client connection.
- **OnClientConnect:** every time a client connection is established, this event is triggered.
- **OnClientDisconnect:** every time a client connection is dropped, this event is triggered.
- **OnClientMessage:** raised when a new text message is received from the server.
- **OnClientBinary:** raised when a new binary message is received from the server.

- **OnClientFragmented:** raised when a new fragmented message is received from the server.
- **OnServerConnect:** raised when a new server connects to LoadBalancerServer.
- **OnServerDisconnect:** raised when a server disconnects from LoadBalancerServer.
- **OnServerReady:** raised when a server is ready to accept messages.
- **OnLoadBalancerHTTPRequest:** the event is called when there is a new HTTP Request and before it's forwarded to a backup server.
- **OnLoadBalancerHTTPResponse:** the event is called with the HTTP Response sent by the backup server.

TsgcWebSocketProxyServer

TsgcWebSocketProxyServer implements a WebSocket Server Component that listens for client WebSocket connections and forwards data connections to a normal TCP/IP server. This is especially useful for browser connections because it allows a browser to virtually connect to any server.

TsgcWSConnection

TsgcWSConnection is a wrapper for client WebSocket connections. You can access this object in Server or Client Events.

Methods

WriteData: sends a message to the client.

Close: sends a close message to other peer. A "CloseCode" can be specified optionally. By default, the value sent is NORMAL close code. If you send a negative close code, the reason for closing will not be sent.

Disconnect: close client connection from the server side. A "CloseCode" can be specified optionally.

Ping: sends a ping to the client.

AddTCPEndOfFrame: if connection is plain TCP, allows you to set which byte/s define the end of message. Message is buffered till is received completely.

Subscribed: returns if the connection is subscribed to a custom channel.

Subscribe: subscribe this connection to a channel. Later you can Broadcast a message from server component to all connections subscribed to this channel.

UnSubscribe: unsubscribe this connection from a channel.

Properties

Protocol: returns sub-protocol used on this connection.

IP: returns Peer IP Address.

Port: returns Peer Port.

LocalIP: returns Host IP Address.

LocalPort: returns Host Port.

URL: returns URL requested by the client.

Guid: returns connection ID.

HeadersRequest: returns a list of Headers received on Request.

HeadersResponse: returns a list of Headers sent as Response.

RecBytes: number of bytes received.

SendBytes: number of bytes sent.

Transport: returns the transport type of connection:

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

trpSSE: a Server-Sent Events connection.

trpTCP: plain TCP connection.

TCPEndOfFrameScanBuffer: allows defining which method use to find end of message (if using trpTCP as transport).

eofScanNone: every time a new packet arrives, the OnBinary event is called.

eofScanLatestBytes: if latest bytes are equal to bytes added with AddTCPEndOfFrame method, OnBinary message is called, otherwise this packet is buffered

eofScanAllBytes: searches in the entire packet for bytes equal to bytes added with the AddTCPEndOfFrame method. If found, the OnBinary event is called, otherwise this packet is buffered

Data: user session data object, here you can pass an object and access this every time you need, for example: you can pass a connection to a database, user session properties...

Protocols

With WebSockets, you can implement sub-protocols, allowing you to create customized communications. **For example,** you can implement a sub-protocol over the WebSocket protocol to communicate with a customized application using JSON messages, and you can implement another sub-protocol using XML messages.

When a connection is opened on the Server side, it will validate if the sub-protocol sent by the client is supported by the server; if not, it will close the connection. A server can implement several sub-protocols, but only one can be used on a single connection.

Sub-protocols are very useful for creating customized applications and ensuring that all clients support the same communication interface.

Although the protocol name is arbitrary, it's recommended to use unique names like "dataset.esegece.com"

With sgcWebSockets package, you can build your own protocols and you can use built-in sub-protocols provided:

1. Protocol MQTT: MQTT is a Client Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement.

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

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

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

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

7. Protocol Presence: allows you to know who is subscribed to a channel, example: chat rooms, collaborators on a document, people viewing the same web page, competitors in a game...

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

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

10. Protocol STOMP: STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

10.1 STOMP for RabbitMQ: client for RabbitMQ Broker.

10.2 STOMP for ActiveMQ: client for ActiveMQ Broker.

11. Protocol AMQP: Advanced Message Queuing Protocol (AMQP 0.9.1) is created as an open standard protocol that allows messaging interoperability between systems, regardless of message broker vendor or platform used.

12. Protocol AMQP1: Advanced Message Queuing Protocol (AMQP 1.0.0) is created as an open standard protocol that allows messaging interoperability between systems, regardless of message broker vendor or platform used.

13. Protocol E2EE: Messages sent over a WebSocket connection are encrypted end-to-end at the application level, so only the communicating clients can read them even though the WebSocket server relays the data.

Protocols can be registered at **runtime**, just call Method **RegisterProtocol** and pass protocol component as a parameter.

Javascript Reference

Here you can get more information about the common JavaScript library used in sgcWebSockets.

Protocols Javascript

Default Javascript sgcWebSockets uses **sgcWebSocket.js** file.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure your access to sgcWebSocket.js file as:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
</script>
```

sgcWebSocket has 3 parameters, only first is required:

```
sgcWebSocket(url, protocol, transport)
```

- **URL:** WebSocket server location, you can use "ws:" for normal WebSocket connections and "wss:" for secured WebSocket connections.

  ```
  sgcWebSocket('ws://127.0.0.1')

  sgcWebSocket('wss://127.0.0.1')
  ```

- **Protocol:** if the server accepts one or more protocol, you can define which protocol you want to use.

  ```
  sgcWebSocket('ws://127.0.0.1', 'esegece.com')
  ```

- **Transport:** by default, first tries to connect using WebSocket connection and if not implemented by Browser, then tries Server Sent Events as Transport.

  Use WebSocket if implemented, if not, then use Server Sent Events:

  ```
  sgcWebSocket('ws://127.0.0.1')
  ```

  Only use WebSocket as transport:

  ```
  sgcWebSocket('ws://127.0.0.1', '', ['websocket'])
  ```

  Only use Server Sent as transport:

  ```
  sgcWebSocket('ws://127.0.0.1', '', ['sse'])
  ```

Open Connection With Authentication

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket({"host":"ws://{%host%}:{%port%}","user":"admin","password":"1234"});
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.on('message', function(event)
  {
    alert(event.message);
  }
</script>
```

Binary Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.on('stream', function(event)
  {
    document.getElementById('image').src = URL.createObjectURL(event.stream);
    event.stream = "";
  }
</script>
```

Binary (Header + Image) Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
  socket.on('stream', function(event)
  {
    sgcWSStreamRead(evt.stream, function(header, stream) {
      document.getElementById('text').innerHTML = header;
      document.getElementById('image').src = URL.createObjectURL(event.stream);
      event.stream = "";
    }
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  var socket = new sgcWebSocket('ws://{%host%}:{%port%}');
```

```
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script>
  socket.state();
</script>
```

Protocol MQTT

MQTT is a Client-Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and the Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium.

The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. Its features include:

· Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.

· A messaging transport that is agnostic to the content of the payload.

· Three qualities of service for message delivery:
  · "At most once", where messages are delivered according to the best efforts of the operating environment. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  · "At least once", where messages are assured to arrive but duplicates can occur.
  · "Exactly once", where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

· A small transport overhead and protocol exchanges minimized to reduce network traffic.

· A mechanism to notify interested parties when an abnormal disconnection occurs.

Features

- Supports **3.1.1** and **5.0** MQTT versions.
- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications.
- **Acknowledgment** of messages sent.
- Implements **QoS** (Quality of Service) for message delivery (all levels: At most once, At least once and Exactly once)
- **Last Will Testament**.
- **Secure** connections.
- **HeartBeat** and **Watchdog**.
- **Authentication** to server.

Components

TsgcWSPClient_MQTT: MQTT Client Component.

Most common uses

- **Connection**
  - Client MQTT Connect
  - Connect Mosquitto MQTT Servers
  - Client MQTT Sessions
  - Client MQTT Version

- **Publish & Subscribe**
  - MQTT Publish Subscribe
  - MQTT Topics
  - MQTT Subscribe
  - MQTT Publish Message

- MQTT Receive Messages
- MQTT Publish and Wait Response

- **Other**
  - MQTT Clear Retained Messages

TsgcWSPClient_MQTT

The MQTT component provides a lightweight, fully-featured MQTT client implementation with support for versions 3.1.1 and 5.0. The component supports plaintext and secure connections over both standard TCP and WebSockets.

Connection to an MQTT server is simple, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property. Set host and port in TsgcWebSocketClient and set Active := True to connect.

MQTT v5.0 is not backward compatible (like v3.1.1). Obviously too many new things are introduced so existing implementations have to be revisited.

According to the specification, MQTT v5.0 adds a significant number of new features to MQTT while keeping much of the core in place.

- The Clean Session flag functionality is divided into 2 properties to allow for finer control over session state data: the CleanStart parameter and the new SessionExpInterval.
- Server disconnect: Allow DISCONNECT to be sent by the Server to indicate the reason the connection is closed.
- All response packets (CONNACK, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBACK, UNSUBACK, DISCONNECT) now contain a reason code and reason string describing why operations succeeded or failed.
- Enhanced authentication: Provide a mechanism to enable challenge/response style authentication including mutual authentication. This allows SASL style authentication to be used if supported by both Client and Server, and includes the ability for a Client to re-authenticate within a connection.
- The Request / Response pattern is formalized by the addition of the ResponseTopic.
- Shared Subscriptions: Add shared subscription support allowing for load balanced consumers of a subscription.
- Topic Aliases can be sent by both client and server to refer to topic filters by shorter numerical identifiers in order to save bandwidth.
- Servers can communicate what features they support in ConnectionProperties.
- Server reference: Allow the Server to specify an alternate Server to use on CONNACK or DISCONNECT. This can be used as a redirect or to do provisioning.
- More: message expiration, Receive Maximums and Maximum Packet Sizes, and a Will Delay interval are all supported.

Methods

Connect: this method is called automatically after a successful WebSocket connection.

Ping: Sends a ping to the server, usually to keep the connection alive. If you enable HeartBeat property, ping will be sent automatically by a defined interval.

Subscribe: subscribe client to a custom channel. If the client is subscribed, OnMQTTSubscribe event will be fired.

SubscribeProperties: (New in MQTT 5.0)

- **SubscriptionIdentifier:** MQTT 5 allows clients to specify a numeric subscription identifier which will be returned with messages delivered for that subscription. To verify that a server supports subscription identifiers, check the "SubscriptionIdentifiersAvailable"
- **UserProperties:** This property is intended to provide a means of transferring application layer name-value tags whose meaning and interpretation are known only by the application programs responsible for sending and receiving them.

Example:

```
TsgcWSMQTTSubscribe_Properties oProperties = new TsgcWSMQTTSubscribe_Properties();
```

```
oProperties.SubscriptionIdentifier = 16385;
mqtt.Subscribe("myChannel", TmqttQoS.mtqsAtMostOnce, oProperties);
```

Unsubscribe: unsubscribe client from a custom channel. If the client is unsubscribed, OnMQTTUnsubscribe event will be fired.

UnsubscribeProperties: (New in MQTT 5.0)

- **UserProperties:** This property is intended to provide a means of transferring application layer name-value tags whose meaning and interpretation are known only by the application programs responsible for sending and receiving them.

Example:

```
TsgcWSMQTTUnsubscribe_Properties oProperties = new TsgcWSMQTTUnsubscribe_Properties();
oProperties.UserProperties = "Temp=21,Humidity=55";
mqtt.UnSubscribe("myChannel", TmqttQoS.mtqsAtMostOnce, oProperties);
```

Publish: sends a message to all subscribed clients. There are the following parameters:

Topic: is the channel where the message will be published.

Text: is the text of the message.

QoS: is the Quality Of Service of published message. There are 3 possibilities:

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

mtqsAtLeastOnce: the receipt of a message by the server is acknowledged by an ACKNOWLEDGMENT message. If there is an identified failure of either the communications link or the sending device or the acknowledgement message is not received after a specified period of time, the sender resends the message. The message arrives at the server at least once. A message with QoS level 1 has an ID param in the message.

mtqsExactlyOnce: where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied. If there is an identified failure of either the communications link or the sending device, or the acknowledgement message is not received after a specified period of time, the sender resends the message.

Retain: if True, Server MUST store the Application Message and its QoS, so that it can be delivered to future subscribers whose subscriptions match its topic name. By default is False.

PublishProperties: (New in MQTT 5.0)

- **PayloadFormat:** select payload format from: mqpfUnspecified (which is equivalent to not sending a Payload Format Indicator) or mqpfUTF8 (Message is UTF-8 Encoded Character Data).
- **MessageExpiryInterval:** Length of time after which the server must stop delivery of the publish message to a subscriber if not yet processed.
- **TopicAlias:** is an integer value that is used to identify the Topic instead of using the Topic Name. This reduces the size of the PUBLISH packet, and is useful when the Topic Names are long and the same Topic Names are used repetitively within a Network Connection.
- **ResponseTopic:** is used as the Topic Name for a response message.
- **CorrelationData:** The Correlation Data is used by the sender of the Request Message to identify which request the Response Message is for when it is received.
- **UserProperties:** This property is intended to provide a means of transferring application layer name-value tags whose meaning and interpretation are known only by the application programs responsible for sending and receiving them.
- **SubscriptionIdentifier:** A numeric subscription identifier included in SUBSCRIBE packet which will be returned with messages delivered for that subscription.
- **ContentType:** String describing content of message to be sent to all subscribers receiving the message.

PublishAndWait: is the same method as Publish, but in this case, if QoS is [mtqsAtLeastOnce, mtqsExactlyOnce] waits till server processes the message, this way, if you get a positive result, means that message has been received by server. There is a timeout of 10 seconds by default, if after the timeout there is no response from server, the response will be false.

Disconnect: disconnects from MQTT server.
ReasonCode: code identifies reason why disconnects.(New in MQTT 5.0)
DisconnectProperties (New in MQTT 5.0)

- **SessionExpiryInterval:** Session Expiry Interval in seconds.
- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.
- **ServerReference:** can be used by the Client to identify another Server to use.

Auth: is sent from Client to Server or Server to Client as part of an extended authentication exchange, such as challenge / response authentication. (New in MQTT 5.0)
ReAuthenticate: if True Initiate a re-authentication, otherwise continue the authentication with another step.
AuthProperties

- **AuthenticationMethod:** contains the name of the authentication method.
- **AuthenticationData:** contains authentication data.
- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.

Events

OnMQTTBeforeConnect: this event is triggered before a new connection is established. There are 2 parameters:
CleanSession: if True (by default), the server must discard any previous session and start a new session. If false, the server must resume communication.
ClientIdentifier: every new connection needs a client identifier, this is set automatically by component, but can be modified if needed.

QoS: configures the Quality of Service retry behavior for messages published with QoS level 1 or 2.
- **Level:** the QoS level (mtqsAtMostOnce, mtqsAtLeastOnce, mtqsExactlyOnce).
- **Interval:** the retry interval in milliseconds for unacknowledged QoS 1 and QoS 2 messages.
- **Timeout:** the timeout in milliseconds after which unacknowledged messages are discarded.

OnMQTTConnect: this event is triggered when the client is connected to MQTT server. There are 2 parameters:
Session:
1. If client sends a connection with CleanSession = True, then Server Must respond with Session = False.
2. If client sends a connection with CleanSession = False:

- If the Server has stored Session state, Session = True.
- If the Server does not have stored Session state, Session = False

ReasonCode: returns code with the result of connection.(New in MQTT 5.0)
ReasonName: text description of ReturnCode.(New in MQTT 5.0)
ConnectProperties: (New in MQTT 5.0)

- **SessionExpiryInterval:** Session Expiry Interval in seconds.
- **ReceiveMaximum:** number of QoS 1 and QoS 2 publish messages, the server will process concurrently for the client.
- **MaximumQoS:** maximum accepted QoS of PUBLISH messages to be received by the server.
- **RetainAvailable:** indicates whether the client may send PUBLISH packets with Retain set to True.
- **MaximumPacketSize:** maximum packet size in bytes the server is willing to accept.
- **AssignedClientIdentifier:** the Client Identifier which was assigned by the Server when client didn't send any.
- **TopicAliasMaximum:** indicates the highest value that the server will accept as a Topic Alias sent by the client.
- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.
- **WildcardSubscriptionAvailable:** indicates whether the server supports wildcard subscriptions.
- **SubscriptionIdentifiersAvailable:** indicates whether the server supports subscription identifiers.

- **SharedSubscriptionAvailable:** indicates whether the server supports shared subscriptions.
- **ResponseInformation:** used as the basis for creating a Response Topic.
- **ServerReference:** can be used by the Client to identify another Server to use.
- **AuthenticationMethod:** identifier of the Authentication Method.
- **AuthenticationData:** string containing authentication data.

OnQTTDisconnect: this event is triggered when the client is disconnected from MQTT server. Parameters:

ReasonCode: returns code with the result of connection.(New in MQTT 5.0)
ReasonName: text description of ReturnCode.(New in MQTT 5.0)
DisconnectProperties: (New in MQTT 5.0)

- **SessionExpiryInterval:** Session Expiry Interval in seconds.
- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.
- **ServerReference:** can be used by the Client to identify another Server to use.

OnMQTTPing: this event is triggered when the client receives an acknowledgment from a ping previously sent.

OnMQTTPubAck: this event is triggered when the client receives the response to a Publish Packet with QoS level 1. There is one parameter:

PacketIdentifier: is packet identifier sent initially.
ReasonCode: returns code with the result of connection.(New in MQTT 5.0)
ReasonName: text description of ReturnCode.(New in MQTT 5.0)
PubAckProperties: (New in MQTT 5.0)

- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.

OnMQTTPubComp: this event is triggered when the client receives the response to a PubRel Packet. It is the fourth and final packet of the QoS 2 protocol exchange. There are the following parameters:

PacketIdentifier: is packet identifier sent initially.
ReasonCode: returns code with the result of connection.(New in MQTT 5.0)
ReasonName: text description of ReturnCode.(New in MQTT 5.0)
PubCompProperties: (New in MQTT 5.0)

- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.

OnMQTTPublish: this event is triggered when the client receives a message from the server. There are 2 parameters:

Topic: is the topic name of the published message.
Text: is the text of the published message.
PublishProperties: (New in MQTT 5.0)

- **PayloadFormat:** select payload format from: mqpfUnspecified (which is equivalent to not sending a Payload Format Indicator) or mqpfUTF8 (Message is UTF-8 Encoded Character Data).
- **MessageExpiryInterval:** Length of time after which the server must stop delivery of the publish message to a subscriber if not yet processed.
- **TopicAlias:** is an integer value that is used to identify the Topic instead of using the Topic Name. This reduces the size of the PUBLISH packet, and is useful when the Topic Names are long and the same Topic Names are used repetitively within a Network Connection.
- **ResponseTopic:** is used as the Topic Name for a response message.
- **CorrelationData:** The Correlation Data is used by the sender of the Request Message to identify which request the Response Message is for when it is received.
- **UserProperties:** This property is intended to provide a means of transferring application layer name-value tags whose meaning and interpretation are known only by the application programs responsible for sending and receiving them.
- **SubscriptionIdentifier:** A numeric subscription identifier included in SUBSCRIBE packet which will be returned with messages delivered for that subscription.
- **ContentType:** String describing content of message to be sent to all subscribers receiving the message.

OnMQTTPublishEx: this event is triggered when the client receives a message from the server. It provides the payload through a TsgcWSMQTTPublishData object with Value (string), Bytes (TBytes), and Stream (TMemoryStream) properties. There are the following parameters:

Topic: is the topic name of the published message.
Data: contains the payload of the published message as a TsgcWSMQTTPublishData object (Value, Bytes, Stream).
PublishProperties: (New in MQTT 5.0) same properties as OnMQTTPublish.

OnMQTTPubRel: this event is triggered when the client receives a PUBREL Packet. It is the third packet of the QoS 2 protocol exchange. There are the following parameters:

PacketIdentifier: is packet identifier sent initially.
ReasonCode: returns code with the result of connection.(New in MQTT 5.0)
ReasonName: text description of ReturnCode.(New in MQTT 5.0)
PubRelProperties: (New in MQTT 5.0)

- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.

OnMQTTPubRec: this event is triggered when receives the response to a Publish Packet with QoS 2. It is the second packet of the QoS 2 protocol exchange. There are the following parameters:

PacketIdentifier: is packet identifier sent initially.
ReasonCode: returns code with the result of connection.(New in MQTT 5.0)
ReasonName: text description of ReturnCode.(New in MQTT 5.0)
PubRecProperties: (New in MQTT 5.0)

- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.

OnMQTTSubscribe: this event is triggered as a response to subscribe method. There are the following parameters:

PacketIdentifier: is packet identifier sent initially.
Codes: codes with the result of a subscription.
SubscribeProperties: (New in MQTT 5.0)

- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client about subscription.

OnMQTTUnSubscribe: this event is triggered as a response to the unsubscribe method. There are the following parameters:

PacketIdentifier: is packet identifier sent initially.
Codes: codes with the result of an unsubscription.
UnsubscribeProperties: (New in MQTT 5.0)

- **UserProperties:** provide additional information to the Client about subscription.

OnMQTTAuth: this event is triggered as a response to Äuth method. There is one parameter: (New in MQTT 5.0)
ReasonCode: returns code with the result of connection.
ReasonName: text description of ReturnCode.
AuthProperties:

- **AuthenticationMethod:** contains the name of the authentication method used for extended authentication.
- **AuthenticationData:** data associated to authentication.
- **ReasonString:** represents the reason associated with this response. This Reason String is a human readable string designed for diagnostic.
- **UserProperties:** provide additional information to the Client including diagnostic information.

Enhanced Authentication (New in MQTT 5.0)

To begin an enhanced authentication, the Client includes an Authentication Method in the ConnectProperties. This specifies the authentication method to use. If the Server does not support the Authentication Method supplied by the Client, it may send a Reason Code "Bad authentication method" or Not Authorized.

Example:

- Client to Server: CONNECT Authentication Method="SCRAM-SHA-1" Authentication Data=client-first-data
- Server to Client: AUTH ReasonCode="Continue authentication" Authentication Method="SCRAM-SHA-1" Authentication Data=server-first-data
- Client to Server AUTH ReasonCode="Continue authentication" Authentication Method="SCRAM-SHA-1" Authentication Data=client-final-data
- Server to Client CONNACK ReasonCode=0 Authentication Method="SCRAM-SHA-1" Authentication Data=server-final-data

Properties

MQTTVersion: select which MQTT version (3.1.1 or 5.0) will use to connect to server.
Broker: references a TsgcWSMQTTBroker component. When set, the client connects to the MQTT broker using raw TCP instead of WebSockets.

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: enabled by default, if True, send a ping every X seconds (set by Interval property) to keep alive connection. You can set a Timeout too, so if after X seconds, the client doesn't receive a response to a ping, the connection will be closed automatically.

LastWillTestament: if there is a disconnection and is enabled, a message is sent to all connected clients to inform that connection has been closed.

- **Enabled:** enable if you want activate last will testament.
- **Text:** is the message that the server will publish in the event of an ungraceful disconnection.
- **Topic:** is the topic that the server will publish the message to in the event of an ungraceful disconnection. **Is mandatory if LastWillTestament is enabled.**
- **Retain:** enable if the server must retain the message after publishing it.
- **WillProperties:** (New in MQTT 5.0)
    - **WillDelayInterval:** The Server delays publishing the Client's Will Message until the Will Delay Interval has passed or the Session ends, whichever happens first.
    - **PayloadFormat:** select payload format from: mqpfUnspecified (which is equivalent to not sending a Payload Format Indicator) or mqpfUTF8 (Message is UTF-8 Encoded Character Data).
    - **MessageExpiryInterval:** Length of time after which the server must stop delivery of the will message to a subscriber if not yet processed.
    - **ContentType:** string describing content of will message.
    - **ResponseTopic:** Used as a topic name for a response message.
    - **CorrelationData:** binary string used by client to identify which request the response message is for when received.
    - **UserProperties:** can be used to send will related properties from the Client to the Server. The meaning of these properties is not defined by MQTT specification.

ConnectProperties: (New in MQTT 5.0) are connection properties sent with packet connect.

- **Enabled:** if True, connect properties will be sent to server.
- **SessionExpiryInterval:** if value is zero, session will end when network connection is closed.
- **ReceiveMaximum:** the Client uses this value to limit the number of QoS 1 and QoS 2 publications that it is willing to process concurrently.
- **MaximumPacketSize:** the Client uses the Maximum Packet Size to inform the Server that it will not process packets exceeding this limit.
- **TopicAliasMaximum:** the Client uses this value to limit the number of Topic Aliases that it is willing to hold on this Connection.
- **RequestResponseInformation:** the Client uses this value to request the Server to return Response Information in the CONNACK. If False indicates that the Server MUST NOT return Response Information, If True the Server MAY return Response Information in the CONNACK packet.
- **RequestProblemInformation:** the Client uses this value to indicate whether the Reason String or User Properties are sent in the case of failures. If the value of Request Problem Information is False, the Server MAY return a Reason String or User Properties on a CONNACK or DISCONNECT packet but MUST NOT send a Reason String or User Properties on any packet other than PUBLISH, CONNACK, or DISCONNECT.

- **UserProperties:** can be used to send connection related properties from the Client to the Server. The meaning of these properties is not defined by MQTT specification.
- **AuthenticationMethod:** contains the name of the authentication method used for extended authentication.

Guid: unique identifier for the protocol instance. Used internally to match protocols with connections.

TsgcWSPClient_MQTT | Client MQTT Connect

In order to connect to a MQTT Server, you must create first a TsgcWebSocketClient and a TsgcWSPClient_MQTT. Then you must attach MQTT Component to WebSocket Client.

Basic Usage

Connect to Mosquitto MQTT server using websocket protocol. Subscribe to topic: "topic1" after connect.

```
oClient = new TsgcWebSocketClient();
oClient.Host = "test.mosquitto.org";
oClient.Port = 8080;
oMQTT = TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client = oClient;
oClient.Active = true;

void OnMQTTConnect(TsgcWSConnection Connection, bool Session, int ReasonCode,
  string ReasonName, TsgcWSMQTTCONNACKProperties ConnectProperties);
{
  oMQTT.Subscribe("topic1");
}
```

Client Identifier

MQTT requires a **Client Identifier** to identify client connection. Component sets a **random value** automatically but you can set your own Client Identifier if required, to do this, just handle **OnBeforeConnect** event and set your value on aClientIdentifier parameter.

```
void OnMQTTBeforeConnect(TsgcWSConnection Connection, ref bool aCleanSession,
  ref string aClientIdentifier)
{
  aClientIdentifier = "your client id";
}
```

Authentication

Some servers require a user and password to a**uthorize MQTT connections**. Use **Authentication** property to set the value for username and password before connect to server.

```
oMQTT = new TsgcWSPClient_MQTT();
oMQTT.Authentication.Enabled = true;
oMQTT.Authentication.UserName = "your user";
oMQTT.Authentication.Password = "your passwrd";
```

TsgcWSPClient_MQTT | Connect MQTT Mosquitto

Use the following sample configurations to connect to a Mosquitto MQTT Server.

MOSQUITTO MQTT WebSockets

```
oClient = new TsgcWebSocketClient();
oClient.Host = "test.mosquitto.org";
oClient.Port = 8080;
oMQTT = TsgcWSPClient_MQTT.Create(nil);
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 = TwsTLSVersions.tls1_2;
oMQTT = TsgcWSPClient_MQTT.Create(nil);
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 = TsgcWSPClient_MQTT.Create(nil);
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 = TwsTLSVersions.tls1_2;
oMQTT = TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client = oClient;
oClient.Active = true;
```

TsgcWSPClient_MQTT | Client MQTT Sessions

Clean Start

OnMQTTBeforeConnect event, there is a parameter called **aCleanSession**. If the value of this parameter is **True**, means that the client **wants to start a new session**, so if server has any session stored, it must discard it. So, when OnMQTTConnect event is fired, aSession parameter will be false. If the value of this parameter is **False** and there is a session associated to this client identifier, the server must resume communications with the client on state with the existing session.

So, if client has an **unexpected disconnection**, and you want to **recover the session** where was disconnected, in OnMQTTBeforeConnect set **aCleanSession = True** and **aClientIdentifier = Client ID of Session**.

Session

Once successful connection, check **OnMQTTConnect** event, the value of Session parameter.

Session = true, means session has been resumed.
Session = false, means it's a new session.

```
void OnMQTTBeforeConnect(TsgcWSConnection Connection, ref bool aCleanSession,
  ref string aClientIdentifier)
{
  aCleanSession = false;
  aClientIdentifier = "previous client id";
}

void OnMQTTConnect(TsgcWSConnection Connection, bool Session, int ReasonCode,
  string ReasonName, TsgcWSMQTTCONNACKProperties ConnectProperties);
{
  if (Session == true)
  {
    Console.WriteLine("Session resumed");
  }
  else
  {
    Console.WriteLine("New Session");
  }
}
```

TsgcWSPClient_MQTT | Client MQTT Version

Currently, MQTT Client supports the following specifications:

- **MQTT 3.1.1:** http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
- **MQTT 5.0:** https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

You can select which is the version which will use the MQTT Client component using MQTTVersion property.

MQTT 3.1.1: TsgcWSPClient_MQTT.Version = mqtt311
MQTT 5.0: sgcWSPClient_MQTT.Version = mqtt5

TsgcWSPClient_MQTT | MQTT Publish Subscribe

The publish/subscribe pattern (also known as pub/sub) provides an alternative to traditional client-server architecture. In the client-sever model, a client communicates directly with an endpoint.The pub/sub model **decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers)**. The publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists. **The connection between them is handled by a third component (the broker)**. The job of the broker is to filter all incoming messages and distribute them correctly to subscribers.

With **TsgcWSPClient_MQTT** you can **Publish messages** and **Subscribe to Topics**.

Subscribe Topic

Subscribe to Topic "topic1" after a successful connection.

```
oClient = new TsgcWebSocketClient();
oClient.Host = "test.mosquitto.org";
oClient.Port = 8080;
oMQTT = TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client = oClient;
oClient.Active = true;

void OnMQTTConnect(TsgcWSConnection Connection, bool Session, int ReasonCode, string ReasonName,
  TsgcWSMQTTCONNACKProperties ConnectProperties);
{
  oMQTT->Subscribe("topic1");
}
```

Publish Message

Publish a message to all subscribers of "topic1"

```
oClient = new TsgcWebSocketClient();
oClient.Host = "test.mosquitto.org";
oClient.Port = 8080;
oMQTT = TsgcWSPClient_MQTT.Create(nil);
oMQTT.Client = oClient;
oClient.Active = true;

void OnMQTTConnect(TsgcWSConnection Connection, bool Session, int ReasonCode, string ReasonName,
  TsgcWSMQTTCONNACKProperties ConnectProperties);
{
  oMQTT.Publish("topic1", "Hello Subscribers topic1");
}
```

TsgcWSPClient_MQTT | MQTT Topics

Topics

In MQTT, the word topic refers to an UTF-8 string that the broker uses to filter messages for each connected client. The topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator)

myHome / groundfloor / livingroom / temperature

In comparison to a message queue, MQTT topics are very lightweight. The client does not need to create the desired topic before they publish or subscribe to it. The broker accepts each valid topic without any prior initialization. Note that each topic must contain at least 1 character and that the topic string permits empty spaces. Topics are case-sensitive.

WildCards

When a client subscribes to a topic, it can subscribe to the exact topic of a published message or it can use wildcards to subscribe to multiple topics simultaneously. A wildcard can only be used to subscribe to topics, not to publish a message. There are two different kinds of wildcards: _single-level and _multi-level.

Single Level: +

As the name suggests, a single-level wildcard replaces one topic level. The plus symbol represents a single-level wildcard in a topic.

myHome / groundfloor / + / temperature

Any topic matches a topic with single-level wildcard if it contains an arbitrary string instead of the wildcard. For example a subscription to _myhome/groundfloor/+/temperature can produce the following results:

YES => myHome / groundfloor / livingroom / temperature
YES => myHome / groundfloor / kitchen / temperature
NO => myHome / groundfloor / livingroom / brightness
NO => myHome / firstfloor / livingroom / temperature
NO => myHome / groundfloor / kitchen / fridge / temperature

Multi Level:

The multi-level wildcard covers many topic levels. The hash symbol represents the multi-level wild card in the topic. For the broker to determine which topics match, the multi-level wildcard must be placed as the last character in the topic and preceded by a forward slash.

myHome / groundfloor / #

YES => myHome / groundfloor / livingroom / temperature
YES => myHome / groundfloor / kitchen / temperature
YES => myHome / groundfloor / kitchen / brightness
NO => myHome / firstfloor / kitchen / temperature

When a client subscribes to a topic with a multi-level wildcard, it receives all messages of a topic that begins with the pattern before the wildcard character, no matter how long or deep the topic is. If you specify only the multi-level wildcard as a topic (_#), you receive all messages that are sent to the MQTT broker.

TsgcWSPClient_MQTT | MQTT Subscribe

You can Subscribe to a Topic using method Subscribe from TsgcWSPClient_MQTT. This method has the following parameters:

Topic: is the name of the topic to be subscribed.
QoS: one of the 3 QoS levels (not all brokers support all 3 levels). If not specified, uses mtqsAtMostOnce. Read more about QoS Levels.
SubscribeProperties: if MQTT 5.0, are additional properties about subscriptions.

Subscribe QoS = At Least Once

```
MQTT.Subscribe("topic1", TmqttQoS.mtqsAtLeastOnce);
```

Subscribe MQTT 5.0

```
oProperties = new TsgcWSMQTTSubscribe_Properties();
oProperties.SubscriptionIdentifier = 1234;
oProperties.UserProperties = "name=value";

MQTT->Subscribe("topic1", TmqttQoS.mtqsAtMostOnce, oProperties);
```

TsgcWSPClient_MQTT | MQTT Publish Message

You can publish messages to all subscribers of a Topic using **Publish** method, which has the following parameters:

Topic: is the name of the topic where the message will be published.
Text: is the text of the message.
QoS: one of the 3 QoS levels (not all brokers support all 3 levels). If not specified, uses mtqsAtMostOnce. Read more about QoS Levels.
Retain: if true, this message will be retained. And every time a new client subscribes to this topic, this message will be sent to this client.
PublishProperties: if MQTT 5.0, these are the properties of the message.

Publish a simple message

```
MQTT.Publish("topic1", "Hello Subscribers topic1");
```

Publish QoS = At Least Once

```
MQTT.Publish("topic1", "Hello Subscribers topic1", TmqttQoS.mtqsAtLeastOnce);
```

Publish Retained message

```
MQTT.Publish("topic1", "Hello Subscribers topic1", TmqttQoS.mtqsAtMostOnce, true);
```

TsgcWSPClient_MQTT | MQTT Receive Messages

Messages sent by the server are received in the **OnMQTTPublish** event. This event has the following parameters:

Topic: is the name of the topic associated to this message.
Text: is the text of the message.
PublishProperties: if MQTT 5.0, these are the properties of the published message.

Read published Messages

```
void OnMQTTPublish(TsgcWSConnection Connection, string aTopic, string aText,
  TsgcWSMQTTPublishProperties PublishProperties)
{
  WriteLn("Topic: " + aTopic + ". Message: " + aText);
}
```

TsgcWSPClient_MQTT | MQTT Receive Messages (Extended)

The **OnMQTTPublishEx** event provides the published message payload in multiple formats through a **TsgcWSMQTTPublishData** object. This event has the following parameters:

Topic: is the name of the topic associated to this message.
Data: contains the payload of the published message. It has the following properties:

- **Value:** the payload as a string.
- **Bytes:** the raw payload as TBytes.
- **Stream:** the raw payload as a TMemoryStream.

PublishProperties: if MQTT 5.0, these are the properties of the published message.

Read published Messages (Extended)

```
void OnMQTTPublishEx(TsgcWSConnection Connection, string aTopic,
  TsgcWSMQTTPublishData aData, TsgcWSMQTTPublishProperties PublishProperties)
{
  // read as string
  WriteLn("Topic: " + aTopic + ". Message: " + aData.Value);
  // read as bytes
  WriteLn("Bytes Length: " + aData.Bytes.Length.ToString());
  // read as stream
  WriteLn("Stream Size: " + aData.Stream.Size.ToString());
}
```

TsgcWSPClient_MQTT | Publish and Wait Response

MQTT client allows the use of some type of QoS levels, any of those levels works in a different level to be sure that messages have been processed as expected.

There are the following QoS levels:

- **mtqsAtMostOnce:** (by default) the message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the server either once or not at all.
- **mtqsAtLeastOnce:** the receipt of a message by the server is acknowledged by an ACKNOWLEDGMENT message. If there is an identified failure of either the communications link or the sending device or the acknowledgement message is not received after a specified period of time, the sender resends the message. The message arrives at the server at least once. A message with QoS level 1 has an ID param in the message.
- **mtqsExactlyOnce:** where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied. If there is an identified failure of either the communications link or the sending device, or the acknowledgement message is not received after a specified period of time, the sender resends the message.

You can handle the events OnPubAck or OnPubComp to know if the message has been processed by server or you can use the method **PublishAndWait** to know if the message has been processed by the server.
The use of **PublishAndWait** is the same as the normal Publish method, but now you have a new parameter called Timeout, where the method will return false if after a certain period of time, there is no response from server. By default this value is 10 seconds.

```
if mqtt->PublishAndWait("topic", "text")
{
  MessageBox.Show("Message processed")
}
else
{
  MessageBox.Show("Message error");
}
```

TsgcWSPClient_MQTT | MQTT Clear Retained Messages

By default, every MQTT topic can have a retained message. The standard MQTT mechanism to clean up retained messages is sending a retained message with an empty payload to a topic. This will remove the retained message.

```
MQTT.Publish("topic1", "", TmqttQoS.mtqsAtMostOnce, true);
```

Protocol AMQP 0.9.1

The **Advanced Message Queuing Protocol** (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

Features

AMQP can be used in any situation if there is a need for high-quality and secure message delivery between client and broker.

AMQP provides the following features:

- Monitoring and sharing updates.
- Ensuring quick response of the server to requests and transmission of time-consuming tasks for further processing.
- Distribute messages to multiple recipients.
- Connection offline clients for further data retrieval.
- Increase the reliability and smooth operation of applications.
- Reliability of message delivery.
- High speed message delivery.
- Message Acceptance.

Components

TsgcWSPClient_AMQP: it's the client component that implements **AMQP 0.9.1** protocol.

Most common uses

- **Connection**
  - Client AMQP Connect
  - Client AMQP Disconnect
- **Commands**
  - AMQP Channels
  - AMQP Exchanges
  - AMQP Queues
  - AMQP Publish Messages
  - AMQP Consume Messages (Asynchronous)
  - AMQP Get Messages (Synchronous)
  - AMQP QoS
  - AMQP Transactions

TsgcWSPClient_AMQP

The **TsgcWSClient_AMQP** client implements the full **AMQP 0.9.1** protocol following the OASIS specification. The client supports Plain TCP and WebSocket connections, TLS (secure) connections are supported too.

Connection

AMQP 0.9.1 protocol defines the concept of channels, which allows you to share a single socket connection with several virtual channels, the client implements an internal thread which reads the bytes received and dispatch every message to the correct channel (which already runs in its own thread), so, if you are running an AMQP connection with 5 channels, the client will run 6 threads (5 threads which handle the data of every channel and 1 thread which handles the data of the connection).

Before connecting to an AMQP server, configure the following properties of the AMQP protocol

- **AMQPOptions.Locale:** it's the message locale to use, it's a negotiated value, so can change when compared with the supported locales supported by the server. The default value is "en_US".
- **AMQPOptions.MaxChannels:** it's the maximum number of channels which can be opened, it's a negotiated value, so can change when compared with the server configuration. The default value is 65535.
- **AMQPOptions.MaxFrameSize:** it's the maximum size in bytes of the AMQP frame, it's a negotiated value, so can change when compared with the server configuration. The default value is 2147483647.
- **AMQPOptions.VirtualHost:** it's the name of the virtual host. The default value is "/".

The AMQP HeartBeat can be configured too before connecting to the server, you can enable or disable the use of heartbeats.

- **HeartBeat.Enabled:** set to true if the client supports HeartBeats.
- **HeartBeat.Interval:** the desired interval in seconds.

Once the AMQP client has been configured, attach to a TsgcWebSocketClient and now you can configure the server connection properties to connect to the AMQP Server.
Set the property value **Specifications.RFC6455** to false if using Plain TCP connection instead of WebSocket connection.

```
oAMQP = 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;
```

Channels

Once the AMQP client has connected, it can open the first channel.

```
oAMQP.OpenChannel("channel_name");
```

Exchanges

When a Channel is opened, the client can declare new exchanges, verify that they exist... use the method **Declare-Exchange** to declare a new exchange.

```
oAMQP.DeclareExchange("channel_name", "exchange_name");
```

Queues

When a Channel is opened, the client can declare new queues, verify that they exist... use the method Declare-Queue to declare a new Queue. The queues are not provided by default by the server (unlike the exchanges), so it's always required to declare a new queue (unless a queue has been already created by another client).

```
oAMQP.DeclareQueue("channel_name", "queue_name");
```

Binding Queues

Once the Exchanges and Queues are configured, you may need to bind queues to exchanges, this way the exchanges can know which messages will be dispatched to the queues.

AMQP Servers automatically bind the queues to "direct" exchange using the queue name as routing key. This allows you to send a message to a specific queue without the need to declare a binding (just calling PublishMessage method and passing the Exchange argument as empty value and the name of the queue in the RoutingKey argument).

```
oAMQP.BindQueue("channel_name", "queue_name", "exchange_name", "routing_key");
```

Send Messages

Call the method **PublishMessage** to publish a new AMQP message. The method allows you to publish a **String** or **TStream** message.

```
oAMQP.PublishMessage("channel_name", "exchange_name", "routing_key", "Hello from sgcWebSockets!!!");
```

Receive Messages

AMQP allows you to receive the messages in 2 modes:

- **Request by Client:** using the **GetMessage** method. If there aren't messages in the queue, the event **OnAMQPBasicGetEmpty** will be called.
- **Pushed by Server:** using the Consume method.

Request By Client

```
oAMQP.GetMessage("channel_name", "queue_name");

private void OnAMQPGetOk(TObject Sender, string aChannel,
  TsgcAMQPFramePayload_Method_BasicGetOk aGetOk, TsgcAMQPMessageContent aContent)
{
  DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk.MessageCount) + " " + aContent.Body.AsString);
}
```

Pushed By Server

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

private void OnAMQPGetOk(TObject Sender, string aChannel,
  TsgcAMQPFramePayload_Method_BasicGetOk aGetOk, TsgcAMQPMessageContent aContent)
{
  DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk.MessageCount) + " " + aContent.Body.AsString);
}
```

Connection | Client AMQP Connect

In order to connect to an AMQP Server, you must create first a TsgcWebSocketClient and a TsgcWSPClient_AMQP. Then you must attach AMQP Component to WebSocket Client.

Basic Usage

Connect to AMQP server without authentication. Define the AMQPOptions property values, virtual host and then set in the TsgcWebSocketClient the Host and Port of the server.
If you are using a TCP Plain connection, set the TsgcWebSocketClient property Specifications.RFC6455 to false.

```
oAMQP = new TsgcWSPClient_AMQP();
oAMQP.AMQPOptions.Locale = "en_US";
oAMQP.AMQPOptions.MaxChannels = 100;
oAMQP.AMQPOptions.MaxFrameSize = 16384;
oAMQP.AMQPOptions.VirtualHost = "/";
oAMQP.HeartBeat.Enabled = true;
oAMQP.HeartBeat.Interval = 60;

oClient = new TsgcWebSocketClient();
oAMQP.Client = oClient;
oClient.Specifications.RFC6455 = false;
oClient.Host = "www.esegece.com";
oClient.Port = 5672;
oClient.Active = true;
```

Authentication

If the server requires authentication, use the event **OnAMQPAuthentication** to select the Authentication mechanism (if required) and set the User / Password.

```
oAMQP = new TsgcWSPClient_AMQP();
oAMQP.AMQPOptions.Locale = "en_US";
oAMQP.AMQPOptions.MaxChannels = 100;
oAMQP.AMQPOptions.MaxFrameSize = 16384;
oAMQP.AMQPOptions.VirtualHost = "/";
oAMQP.HeartBeat.Enabled = true;
oAMQP.HeartBeat.Interval = 60;

oClient = new TsgcWebSocketClient();
oAMQP.Client = oClient;
oClient.Specifications.RFC6455 = false;
oClient.Host = "www.esegece.com";
oClient.Port = 5672;
oClient.Active = true;

private void OnAMQPAuthentication(object Sender, TsgcAMQPAuthentications aMechanisms, ref TsgcAMQPAuthentication
  ref string User, ref string Password)
{
  User = "user_value";
  Password = "password_value";
}
```

Connection | Client AMQP Disconnect

The client can disconnect a current active connection, using the following methods:

Sending a Close Reason

The AMQP client can inform the server that the connection will be closed and provide information about the reason why it is closing the connection. Use the method Close to request a connection close to the server.

```
oAMQP.Close(541, "Internal Error");
```

Closing Socket Connection

Just set the property Active of TsgcWebSocketClient to False. You can read more about closing connections.

Commands | AMQP Channels

AMQP is a multi-channelled protocol. Channels provide a way to multiplex a heavyweight TCP/IP connection into several light weight connections. This makes the protocol more “firewall friendly” since port usage is predictable. It also means that traffic shaping and other network QoS features can be easily employed.

Every channel runs in its own thread, so every time a new message is received, first the client identifies the channel and queues the message in a queue which is processed by the channel thread.

The channel life-cycle is this:

1. The client opens a new channel (Open).
2. The server confirms that the new channel is ready (Open-Ok).
3. The client and server use the channel as desired.
4. One peer (client or server) closes the channel (Close).
5. The other peer hand-shakes the channel close (Close-Ok).

Open Channel

To create a new channel just call the method **OpenChannel** and pass the channel name as argument. The event **OnAMQPChannelOpen** is raised as a confirmation sent by the server that the channel has been opened.

```
AMQP.OpenChannel("channel_name");

private void OnAMQPChannelOpen(TObject Sender, const string aChannel)
{
  DoLog("#AMQP_channel_open: " + aChannel);
}
```

A synchronous call can also be done by calling the method **OpenChannelEx**, this method returns true if the channel has been opened and false if no confirmation from server has arrived.

```
if AMQP.OpenChannelEx('channel_name')
{
  DoLog("#AMQP_channel_open channel_name");
}
else
{
  DoLog("#AMQP_channel_open_error");
}
```

Close Channel

To close an existing channel, call the method **CloseChannel** and pass the channel name as argument. The event **OnAMQPChannelClose** will be called when the client receives a confirmation that the channel has been closed.

A Synchronous call can be done calling the method **CloseChannelEx**, this method returns true if the channel has been closed and false if no confirmation from server has arrived.

Channel Flow

Flow control is an emergency procedure used to halt the flow of messages from a peer. It works in the same way between client and server and is implemented by the **EnableChannel / DisableChannel** commands. Flow control is the only mechanism that can stop an over-producing publisher.

To Disable the Flow of a channel, call the method **DisableChannel**, the event **OnAMQPChannelFlow** will be called when the client receives a confirmation that the channel flow has been disabled.
The same applies when enabling the flow of a channel, call the method **EnableChannel**, the event **OnAMQPChannelFlow** will be called when the client receives a confirmation that the channel flow has been enabled.

Synchronous requests are available through the functions **EnableChannelEx** and **DisableChannelEx**.

Commands | AMQP Exchanges

The exchange class lets an application manage exchanges on the server. This class lets the application script its own wiring (rather than relying on some configuration interface). Note: Most applications do not need this level of sophistication, and legacy middleware is unlikely to be able to support this semantic.

The exchange life-cycle is:

1. The client asks the server to make sure the exchange exists (Declare). The client can refine this into, "create the exchange if it does not exist", or "warn me but do not create it, if it does not exist".
2. The client publishes messages to the exchange.
3. The client may choose to delete the exchange (Delete).

Declare Exchange

This method creates a new exchange or verifies that an Exchange already exists. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **ExchangeType:** it's the exchange type, all AMQP servers support "direct" and "fanout" exchanges. Check the server documentation to know which exchanges types are supported.
- **Passive:** if passive is true, the server only verifies that the exchange is already declared. If passive is false, and the exchange does not exist, the server will create a new one.
- **Durable:** if true, the exchange will be re-created when the server starts. If false, the exchange will be deleted when the server stops.
- **AutoDelete:** if true, the exchange will be deleted when all queues have been unbound.
- **Internal:** always false.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

To Declare a new Exchange just call the method **DeclareExchange** and pass the channel name, exchange name and exchange type as arguments. The event **OnAMQPExchangeDeclare** is raised as a confirmation sent by the server that the exchange has been declared.

```
AMQP.DeclareExchange("channel_name", "exchange_name", "direct");

private void OnAMQPExchangeDeclare(TObject Sender, const string aChannel, const string aExchange)
{
  DoLog("#AMQP_exchange_declare: [" + aChannel + "] " + aExchange);
}
```

A Synchronous call can be done too calling the method **DeclareExchangeEx**, this method returns true if the Exchange has been Declared and false if no confirmation from server has arrived.

```
if AMQP.DeclareExchangeEx("channel_name", "exchange_name", "direct")
{
  DoLog("#AMQP_exchange_declare: [" + aChannel + "] " + aExchange);
}
else
{
  DoLog("#AMQP_exchange_declare_error");
}
```

Delete Exchange

This method is used to delete an existing Exchange. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **IfUnused:** the server only deletes the exchange if there aren't any queues bound to it.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.

To Delete an existing Exchange call the method **DeleteExchange** and pass the channel name and exchange name as arguments. The event **OnAMQPExchangeDelete** is raised as a confirmation sent by the server that the exchange has been deleted.

A Synchronous call can be done too calling the method **DeleteExchangeEx**, this method returns true if the Exchange has been Deleted and false if no confirmation from server has arrived.

Commands | AMQP Queues

The queue class lets an application manage message queues on the server. This is a basic step in almost all applications that consume messages, at least to verify that an expected message queue is actually present.

The life-cycle for a durable message queue is fairly simple:

1. The client asserts that the message queue exists (Declare, with the "passive" argument).
2. The server confirms that the message queue exists (Declare-Ok).
3. The client reads messages off the message queue.

The life-cycle for a temporary message queue is more interesting:

1. The client creates the message queue (Declare, often with no message queue name so the server will assign a name). The server confirms (Declare-Ok).
2. The client starts a consumer on the message queue. The precise functionality of a consumer is defined by the Basic class.
3. The client cancels the consumer, either explicitly or by closing the channel and/or connection.
4. When the last consumer disappears from the message queue, and after a polite time-out, the server deletes the message queue.

AMQP implements the delivery mechanism for topic subscriptions as message queues. This enables interesting structures where a subscription can be load balanced among a pool of cooperating subscriber applications.

The life-cycle for a subscription involves an extra bind stage:

1. The client creates the message queue (Declare), and the server confirms (Declare-Ok).
2. The client binds the message queue to a topic exchange (Bind) and the server confirms (Bind-Ok).
3. The client uses the message queue as in the previous examples.

Declare Queue

This method creates a new queue or verifies that a Queue already exists. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **Passive:** if passive is true, the server only verifies that the queue is already declared. If passive is false, and the queue does not exist, the server will create a new one.
- **Durable:** if true, the queue will be re-created when the server starts. If false, the queue will be deleted when the server stops.
- **Exclusive:** if true means the queue is only accessed by the current connection.
- **AutoDelete:** if true, the queue will be deleted when all consumers no longer use the queue.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

To Declare a new Queue just call the method **DeclareQueue** and pass the channel name and queue name as arguments. The event **OnAMQPQueueDeclare** is raised as a confirmation sent by the server that the exchange has been declared.

```
AMQP.DeclareQueue("channel_name", "queue_name");

private void OnAMQPExchangeDeclare(TObject Sender, const string aChannel, const string aQueue,
  int aMessageCount, int aConsumerCount)
{
  DoLog("#AMQP_queue_declare: [" + aChannel + "] " + aQueue));
}
```

A Synchronous call can be done too calling the method **DeclareQueueEx**, this method returns true if the Queue has been Declared and false if no confirmation from server has arrived.

```
if AMQP.DeclareQueueEx("channel_name", "queue_name")
{
  DoLog("#AMQP_queue_declare: [" + aChannel + "] " + aQueue);
}
else
{
  DoLog("#AMQP_queue_declare_error");
}
```

Delete Queue

This method is used to delete an existing Queue. The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **IfUnused:** the server only deletes the queue if there aren't any consumers attached to it.
- **IfEmpty:** the server only deletes the queue if there are no messages.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.

To Delete an existing Queue call the method **DeleteQueue** and pass the channel name and queue name as arguments. The event **OnAMQPQueueDelete** is raised as a confirmation sent by the server that the queue has been deleted.

A Synchronous call can be done too calling the method **DeleteQueueEx**, this method returns true if the Queue has been Deleted and false if no confirmation from server has arrived.

Bind Queue

This method is used to bind a Queue to a Exchange. The Exchanges use the bindings to know which queues will be used to route the messages.

All AMQP Servers bind automatically all the queues to the default exchange (it's a "direct" exchange without name) using the Queue Name as the binding routing key. This allows you to send a message to a specific queue without declaring a binding. Just call the method **PublishMessage**, pass an empty value as Exchange Name and set the **RoutingKey** with the value of the **Queue Name**.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **RoutingKey:** it's the binding's routing key.
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

To bind a Queue to an Exchange call the method **BindQueue** and pass the channel name, queue name, exchange and routing key as arguments. The event **OnAMQPQueueBind** is raised as a confirmation sent by the server that the queue has been bound.

```
AMQP.BindQueueEx("channel_name", "queue_name", "exchange_name", "routing_key");

private void OnAMQPQueueBind(TObject *Sender, const string aChannel, const string aQueue,
  const string aExchange)
{
```

```
  DoLog("#AMQP_queue_bind: [" + aChannel + "] " + aQueue + " -->-- " + aExchange);
}
```

A Synchronous call can be done too calling the method **BindQueueEx**, this method returns true if the Queue has been Bound and false if no confirmation from server has arrived.

```
if AMQP.BindQueueEx("channel_name", "queue_name", "exchange_name", "routing_key")
{
  DoLog("#AMQP_queue_bind: [" + aChannel + "] " + aQueue + " --><-- " + aExchange);
}
else
{
  DoLog("#AMQP_queue_bind_error");
}
```

UnBind Queue

This method deletes an existing queue binding.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **RoutingKey:** it's the binding's routing key.

To UnBind a Queue just call the method **UnBindQueue** and pass the channel name, queue name, exchange and routing key as arguments. The event **OnAMQPQueueUnBind** is raised as a confirmation sent by the server that the queue has been unbound.

A Synchronous call can be done too calling the method **UnBindQueueEx**, this method returns true if the Queue has been Unbound and false if no confirmation from server has arrived.

Purge Queue

This method purges all messages of a queue. All the messages that have been sent but are awaiting acknowledgment are not affected.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **NoWait:** if true, the server doesn't send an acknowledgment to the client.

To Purge a Queue just call the method **PurgeQueue** and pass the channel name and queue name as arguments. The event **OnAMQPQueuePurge** is raised as a confirmation sent by the server that the queue has been Purged.

A Synchronous call can be done too calling the method **PurgeQueueEx**, this method returns true if the Queue has been Purged and false if no confirmation from server has arrived.

Commands | AMQP Publish Messages

Publish Messages

The method PublishMessages is used to send a message to the AMQP server.

AMQP Servers automatically bind the queues to "direct" exchange using the queue name as routing key. This allows you to send a message to a specific queue without the need to declare a binding (just calling PublishMessage method and passing the Exchange argument as empty value and the name of the queue in the RoutingKey argument).

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ExchangeName:** it's the name of the exchange, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **RoutingKey:** it's the binding's routing key name.
- **Mandatory:** if true and the message cannot be routed to any queue, the message is returned by the server, the event OnAMQPBasicReturn is fired.
- **Immediate:** if true and the message cannot be routed to any queue, the message is returned by the server, the event OnAMQPBasicReturn is fired.

```
AMQP.PublishMessage("channel_name", "exchange_name", "routing_key", "Hello from sgcWebSockets!!!");

private void OnAMQPBasicReturn(TObject Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicReturn aReturn,
  const TsgcAMQPMessageContent aContent)
{
  DoLog("#AMQP_basic_return: " + aChannel + " " + aReturn.ReplyCode.ToString() + " " + aReturn.ReplyText + " " +
}
```

Publish Confirmations

The network can fail while publishing a message, the only way to guarantee that a message isn't lost is by using transactions, then for each message/s **select transaction**, **send the message** and **commit**. The confirmation of a successful transaction is received when the event **OnAMQPTransactionOk** is fired.

AMQP Consume Messages

Consumers consume from queues. In order to consume messages there has to be a queue. When a new consumer is added, assuming there are already messages ready in the queue, deliveries will start immediately. The target queue can be empty at the time of consumer registration. In that case first deliveries will happen when new messages are enqueued.

Consuming messages is an **asynchronous** task, which means that every time a new message can be delivered to the consumer queue, it's pushed by the server to the client automatically. You can read an alternative method to Receive Message Synchronously.

Consume

The method **Consume** creates a new consumer in the queue, and every time there is a new message this will be delivered automatically to the consumer client.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer of 255 characters and not begin with "amq." (except if passive parameter is true).
- **ConsumerTag:** it's the name of the consumer and must be unique. If it's not set, then the server creates a new one.
- **NoLocal:** if true means the consumer never consumes messages published on the same channel.
- **NoAck:** if true means the server doesn't expect an acknowledgment for every message delivered.
- **Exclusive:** if true prevents that other consumers consume messages from this queue.
- **NoWait:** if true, the server won't send an acknowledgment to the client.
- **Arguments:** string which contains custom arguments, the values must be passed as a json string, example: {"x-dead-letter-exchange":"my-dlx"}.

The messages are delivered **OnAMQPBasicDeliver** event.

```
AMQP.Consume("channel_name", "queue_name", "consumer_tag");

private void OnAMQPBasicGetOk(TObject Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicDeliver aDeliver,
  const TsgcAMQPMessageContent aContent)
{
  DoLog("#AMQP_basic_deliver: " + aChannel + " " + aDeliver.ConsumerTag + " " + aContent.Body.AsString);
}
```

A Synchronous call can be done just calling the method **ConsumeEx**, this method returns true if the Consumer has been created and false if no confirmation from server has arrived.

Cancel Consume

This method is used to Cancel an existing consumer queue.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **ConsumerTag:** it's the name of the consumer.
- **NoWait:** if true, the server won't send an acknowledgment to the client.

```
AMQP.CancelConsume("channel_name", "consumer_tag");

private void OnAMQPBasicCancelConsume(TObject Sender, const string aChannel, const string aConsumerTag)
```

```
{
  DoLog("#AMQP_basic_cancel_consume: " + aChannel + " " + aConsumerTag);
}
```

A Synchronous call can be done just calling the method **CancelConsumeEx**, this method returns true if the Consumer has been cancelled and false if no confirmation from server has arrived.

Commands | AMQP Get Messages

Getting messages is a **Synchronous** task, which means that it is the client that asks the server if there are messages in the queue. You can read an alternative method to Receive Message Asynchronously.

Get Message

The method **GetMessage** sends a request to the AMQP server asking if there are messages available in a queue. If there are messages these will be dispatched **OnAMQPBasicGetOk** event and if the queue is empty, the event **OnAMQPBasicGetEmpty** will be called.

The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **QueueName:** it's the name of the queue, must be no longer than 255 characters and not begin with "amq." (except if passive parameter is true).
- **NoWait:** if true, the server won't send an acknowledgment to the client.

```
AMQP.GetMessage("channel_name", "queue_name");

private void OnAMQPBasicGetOk(TObject Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicGetOk aGetOk,
  const TsgcAMQPMessageContent aContent)
{
  DoLog("#AMQP_basic_GetOk: " + aChannel + " " + IntToStr(aGetOk.MessageCount) + " " + aContent.Body.AsString);
}

private void OnAMQPBasicGetEmpty(TObject Sender, const string aChannel)
{
  DoLog("#AMQP_basic_GetEmpty: " + aChannel);
}
```

A Synchronous call can be done just calling the method **GetMessageEx**, this method returns true if the queue has messages available, otherwise the result will be false.

Commands | AMQP QoS

AMQP allows you to set a QoS level to limit the number of messages the server sends to the client before wait to get the acknowledgment of the messages.

Set QoS

The method **SetQoS** is used to limit the number messages the server sends to the AMQP client.
The method has the following arguments:

- **ChannelName:** it's the name of the channel (must be open before calling this method).
- **PrefetchSize:** it's the windows size in bytes, the server doesn't send messages to the client if the total size of all currently unacknowledged messages already sent plus the next message to be sent it's greater than **PrefetchSize** argument. If the value is zero, means no limit.
- **PrefetchCount:** is the maximum number of unacknowledged messages already sent and not acknowledged, if the number is greater, the server stops sending messages to the client.
- **Global:** if true the QoS applies to all existing and new consumers of the connection. If false, the QoS applies to all existing and new consumers of the channel.

The response from the server is received **OnAMQPBasicQoS** event.

```
AMQP.SetQoS("channel_name", 1024000, 100, false);

private void OnAMQPBasicQoS(TObject Sender, const string aChannel,
  const TsgcAMQPFramePayload_Method_BasicQoS aQoS)
{
  DoLog("#AMQP_basic_qos: " + aChannel + " " + aQoS.PrefetchSize.ToString() + " "
    + aQoS.PrefetchCount.ToString() + " " + aQoS.Global.ToString());
}
```

A Synchronous call can be done just calling the method **SetQoSEx**, this method returns true if the request has been processed, otherwise the result will be false.

Commands | AMQP Transactions

AMQP supports two kinds of transactions:

1. Automatic transactions, in which every published message and acknowledgement is processed as a stand-alone transaction.
2. Server local transactions, in which the server will buffer published messages and acknowledgements and commit them on demand from the client.

The Transaction class (“tx”) gives applications access to the second type, namely server transactions. The semantics of this class are:

1. The application asks for server transactions in each channel where it wants these transactions (Select).
2. The application does work (Publish, Ack).
3. The application commits or rolls-back the work (Commit, Roll-back).
4. The application does work, ad infinitum.

Transactions cover published contents and acknowledgements, not deliveries. Thus, a rollback does not requeue or redeliver any messages, and a client is entitled to acknowledge these messages in a following transaction.

The Transaction methods allows publish and ack operations to be batched into atomic units of work. The intention is that all publish and ack requests issued within a transaction will complete successfully or none of them will.

Start Transaction

The method **StartTransaction** starts a new transaction in the server, the client uses this method at least once on a channel before using the Commit or Rollback methods. The event **OnAMQPTransactionOk** is raised when the server acknowledges the use of transactions.

```
AMQP.StartTransaction("channel_name");
```

A Synchronous call can be done just calling the method **StartTransactionEx**, this method returns true if the request has been processed, otherwise the result will be false.

Commit Transaction

This method commits all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a commit. The event **OnAMQPTransactionOk** is raised when the server acknowledges the use of transactions.

```
AMQP.CommitTransaction("channel_name");
```

A Synchronous call can be done just calling the method **CommitTransactionEx**, this method returns true if the request has been processed, otherwise the result will be false.

Rollback Transaction

This method abandons all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a rollback. Note that unacked messages will not be automatically redelivered by rollback; if that is required an explicit recover call should be issued. The event **OnAMQPTransactionOk** is raised when the server acknowledges the use of transactions.

```
AMQP.RollbackTransaction("channel_name");
```

A Synchronous call can be done just calling the method **RollbackTransactionEx**, this method returns true if the request has been processed, otherwise the result will be false.

Protocol AMQP 1.0.0

AMQP (Advanced Message Queuing Protocol) 1.0.0 is a messaging protocol designed for reliable, asynchronous communication between distributed systems. It facilitates the exchange of messages between applications or components in a decoupled manner, allowing them to communicate without direct dependencies. Here's a technical breakdown of some key aspects of AMQP 1.0.0:

- **Message-oriented communication:** AMQP 1.0.0 is centered around the concept of messages. Messages can carry data, instructions, or commands and are the fundamental units of communication.
- **Message Brokers:** The protocol operates on a brokered messaging model. Brokers, which can be servers or intermediary entities, manage the routing and delivery of messages between producers and consumers.
- **Queues and Exchanges:** Queues are storage entities within the broker where messages are temporarily stored. Exchanges define the rules for routing messages from producers to queues based on criteria like message content or routing keys.
- **Addresses and Links:** Addresses identify message destinations within the messaging infrastructure. Links are communication channels between a sender (producer) and a receiver (consumer) associated with a specific address.
- **Sessions and Connections:** Sessions represent a logical channel for communication, allowing multiple streams of messages within a single connection. Connections manage the overall communication link between client applications and the message broker.
- **Security:** AMQP 1.0.0 supports various security mechanisms, including authentication and authorization, to ensure secure communication between clients and brokers.
- **Transport Agnostic:** The protocol is designed to be transport agnostic, meaning it can operate over different network transports such as TCP, TLS, or WebSockets, providing flexibility in deployment.
- **Flow Control:** AMQP 1.0.0 includes mechanisms for flow control, allowing consumers to indicate their ability to handle incoming messages at a given rate. This helps prevent overwhelming consumers with a large number of messages.
- **Error Handling:** The protocol specifies mechanisms for handling errors, including acknowledgment and rejection of messages, ensuring robustness and reliability in message delivery.
- **SASL Authentication:** Simple Authentication and Security Layer (SASL) is used for authenticating and securing connections between clients and brokers.

Overall, AMQP 1.0.0 provides a standardized and interoperable way for different software components and systems to communicate in a loosely coupled manner, making it suitable for various distributed and enterprise-level applications.

Components

TsgcWSPClient_AMQP1: it's the client component that implements **AMQP 1.0.0** protocol.

Most common uses

- **Connection**
  - Client AMQP1 Connect
  - Client AMQP1 Disconnect
  - Client AMQP1 Idle Timeout Connection
  - Client AMQP1 Connection State
  - Client AMQP1 Authentication
  - Client AMQP1 Azure Service Bus

- **Commands**
  - AMQP1 Sessions
  - AMQP1 Links
  - AMQP1 Sender Links
  - AMQP1 Receiver Links
  - AMQP1 Send Message
  - AMQP1 Read Message

TsgcWSPClient_AMQP1

The **TsgcWSClient_AMQP1** client implements the **AMQP 1.0.0** protocol following the OASIS specification. The client supports Plain TCP and WebSocket connections, TLS (secure) connections are supported too.

Configuration

The AMQP 1.0.0 client has the property AMQPOptions where you can configure the connection.

- **ChannelMax:** The channel-max value is the highest channel number that can be used on the connection. This
  value plus one is the maximum number of sessions that can be simultaneously active on the connection
- **ContainerId:** (optional) is the name of the source container, identifies uniquely the connection in the server.
- **CreditSize:** default size of the credit flow.
- **IdleTimeout:** The timeout is triggered by a local peer when no frames
  are received after a threshold value is exceeded. The idle timeout is measured in milliseconds, and starts from
  the time the last frame is received.
- **MaxFrameSize:** the max accepted frame size.
- **MaxLinksPerSession:** the max number of links per session.
- **WindowSize:** the default window size.

The AMQP Authentication must be configured in the Authentication property.

- **AuthType:** type of authentication
  - **amqp1authNone:** not configured.
  - **amqp1authSASLAnonymous:** anonymous authentication
  - **amqp1authSASLPlain:** user/password authentication. This type of authentication requires to fill the following properties:
    - Username
    - Password
  - **amqp1authSASLExternal:** external authentication

Connection

The connection starts with the client (usually a messaging application or service) initiating a TCP connection to the server (the message broker). The client connects to the server's port, typically 5672 for non-TLS connections and 5671 for TLS-secured connections. Once the TCP connection is established, the client and server negotiate the AMQP protocol version they will use. AMQP 1.0.0 supports various versions, and during negotiation, both parties agree on using version 1.0.0.

After protocol negotiation, the client may need to authenticate itself to the server, depending on the server's configuration. Authentication mechanisms can include SASL (Simple Authentication and Security Layer) mechanisms like PLAIN, EXTERNAL, or others supported by the server.

Example: connect to AMQP server listening on secure port 5671 and using SASL credentials

```
oAMQP = new TsgcWSPClient_AMQP1(this);
// Setting AMQP authentication options
oAMQP.AMQPOptions.Authentication.AuthType = amqp1authSASLPlain;
oAMQP.AMQPOptions.Authentication.Username = "sgc";
oAMQP.AMQPOptions.Authentication.Password = "sgc";
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient.Specifications.RFC6455 = false;
// Setting WebSocket client properties
oClient.Host = "www.esegece.com";
```

```
oClient.Port = 5671;
oClient.TLS = true;
// Assigning WebSocket client to AMQP client
oAMQP.Client = oClient;
// Activating WebSocket client
oClient.Active = true;
```

Sessions

Once authenticated, the client opens an AMQP session. A session is a logical context for communication between the client and server. Sessions are used to group related messaging operations together. Use the method **Create-Session** to create a new session, the method allows you to set the session name or leave empty and the component will assign automatically one.

If the session has been created successfully, the event **OnAMQPSessionOpen** will be fired with the details of the session.

```
oAMQP.OnAMQPSessionOpen += AMQP1AMQPSessionOpen;
oAMQP.CreateSession("MySession");
private void AMQP1AMQPSessionOpen(object sender, TsgcAMQP1Session aSession, TsgcAMQP1FrameBegin aBegin)
{
    Console.WriteLine("#session-open: " + aSession.Id);
}
```

Links

Within a session, the client creates links to communicate with specific entities like queues, topics, or other resources provided by the server. Links are bidirectional communication channels used for sending and receiving messages.

The component can work as a sender and receiver node. Allows to create any number of links for each session, up to the limit set in the **MaxLinksPerSession** property.

Sender Links

To create a new sender link, use the method **CreateSenderLink** and pass the name of the session and optionally the name of the sender link. If the link is created successfully, the event **OnAMQPLinkOpen** is raised.

```
oAMQP.OnAMQPLinkOpen += AMQP1AMQPLinkOpen;
oAMQP.CreateSenderLink("MySession", "MySenderLink");
private void AMQP1AMQPLinkOpen(object sender, TsgcAMQP1Session aSession, TsgcAMQP1Link aLink, TsgcAMQP1FrameAttac
{
    Console.WriteLine("#link-open: " + aLink.Name);
}
```

Receiver Links

To create a new receiver link, use the method **CreateReceiverLink** and pass the name of the session and optionally the name of the receiver link. If the link is created successfully, the event **OnAMQPLinkOpen** is raised.

```
oAMQP.AMQPLinkOpen += AMQP1AMQPLinkOpen;
oAMQP.OnCreateReceiverLink("MySession", "MySReceiverLink");
private void AMQP1AMQPLinkOpen(object sender, TsgcAMQP1Session aSession, TsgcAMQP1Link aLink, TsgcAMQP1FrameAttac
{
    Console.WriteLine("#link-open: " + aLink.Name);
}
```

Sending Messages

With the session established and links created, the client can start performing message operations such as sending messages to a destination. Use the method SendMessage to send a message using a sender link.

```
oAMQP.SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

Receiving Messages

By default, the Receiver Links are created in **Automatic mode**, which means that every time a new message arrives, it will be delivered to the client.

If the Receiver Links has been created in **manual mode**, use the Sync Method **GetMessage** to fetch and wait till a new message arrives.

In Automatic and Manual mode, every time a new message arrives, the event **OnAMQPMessage** is fired.

```
public void OnAMQPMessageEvent(object Sender, TsgcAMQP1Session aSession,
    TsgcAMQP1ReceiverLink aLink, TsgcAMQP1Message aMessage,
    ref TsgcAMQP1MessageDeliveryState DeliveryState)
{
    System.Windows.Forms.MessageBox.Show(aMessage.ApplicationData.AMQPValue.Value);
}
```

Connection | Client AMQP1 Connect

In order to connect to an AMQP Server, you must create first a TsgcWebSocketClient and a TsgcWSPClient_AMQP1. Then you must attach AMQP1 Component to WebSocket Client.

After a successful connection, the event OnAMQPConnect is fired.

Basic Usage

Connect to an AMQP 1.0.0 server without authentication. Define the AMQPOptions property values, virtual host and then set in the TsgcWebSocketClient the Host and Port of the server.
If you are using a TCP Plain connection, set the TsgcWebSocketClient property Specifications.RFC6455 to false.

```
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 = "amqp_host_address";
oClient.Port = 5672;
// Assigning WebSocket client to AMQP client
oAMQP.Client = oClient;
// Activating WebSocket client
oClient.Active = true;
```

Authentication

If the server requires authentication, use the properties AMQP:Authentication to set the values of the Username/ Password and set AuthType to the value "amqp1authSASLPlain".

```
oAMQP = new TsgcWSPClient_AMQP1(this);
// Setting AMQP authentication options
oAMQP.AMQPOptions.Authentication.AuthType = amqp1authSASLPlain;
oAMQP.AMQPOptions.Authentication.Username = "sgc";
oAMQP.AMQPOptions.Authentication.Password = "sgc";
// Creating WebSocket client
oClient = new TsgcWebSocketClient(this);
// Setting WebSocket specifications
oClient.Specifications.RFC6455 = false;
// Setting WebSocket client properties
oClient.Host = "www.esegece.com";
oClient.Port = 5671;
oClient.TLS = true;
// Assigning WebSocket client to AMQP client
oAMQP.Client = oClient;
// Activating WebSocket client
oClient.Active = true;
```

Connection | Client AMQP1 Disconnect

The client can disconnect a current active connection, using the following methods:

Sending a Close Reason

The AMQP client can inform the server that the connection will be closed and provide information about the reason why is closing the connection. Use the method Close to request a connection close to the server.

```
oAMQP.Close('invalid-frame', "The received frame has an invalid format.");
```

Await Close

By default, the **Close** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the Close method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void Close(string aCondition, string aDescription)
{
    TsgcAMQP1MethodOptions_Close oOptions = new TsgcAMQP1MethodOptions_Close();
    try
    {
        oOptions.ErrorCondition = aCondition;
        oOptions.ErrorDescription = aDescription;
        oOptions.Await = true;
        AMQP1.Close(oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
}
```

Closing Socket Connection

Just set the property Active of TsgcWebSocketClient to False. You can read more about closing connections.

Connection | Idle Timeout

Connections are subject to an **idle timeout** threshold. The timeout is triggered by the client when no frames are received from the server after a threshold value is exceeded. The idle timeout is measured in milliseconds, and starts from
the time the last frame is received. If the threshold is exceeded the component sends a Close Frame to the server. If the server does not respond after 10 seconds the client will close the TCP socket.

The Value of the Idle Timeout can be configured in the property:

AMQPOptions.IdleTimeout

The value set in this property will be sent to the server when opening the AMQP connection. If the value is greater than zero and less than half the MaxInt value, an internal timer will be enabled to check if the idle timeout has not been exceeded.

Example: set an IdleTimeout value of 60 seconds

AMQPOptions.IdleTimeout = 60000

Connection | Connection State

The AMQP 1.0.0 defines the following connection states:

- **amqp1csUnknown:** initial state.
- **amqp1csStart:** In this state a connection exists, but nothing has been sent or received. This is the state an implementation would be in immediately after performing a socket connect or socket accept.
- **amqp1csHeaderReceived:** In this state the connection header has been received from the peer but a connection header has not been sent.
- **amqp1csHeaderSent:** In this state the connection header has been sent to the peer but no connection header has been received.
- **amqp1csHeaderExchanged:** In this state the connection header has been sent to the peer and a connection header has been received from the peer.
- **amqp1csOpenPipe:** In this state both the connection header and the open frame have been sent but nothing has been received.
- **amqp1csOpenClosePipe:** In this state, the connection header, the open frame, any pipelined connection traffic, and the close frame have been sent but nothing has been received.
- **amqp1csOpenReceived:** In this state the connection headers have been exchanged. An open frame has been received from the peer but an open frame has not been sent.
- **amqp1csOpenSent:** In this state the connection headers have been exchanged. An open frame has been sent to the peer but no open frame has yet been received.
- **amqp1csClosePipe:** In this state the connection headers have been exchanged. An open frame, any pipelined connection traffic, and the close frame have been sent but no open frame has yet been received from the peer.
- **amqp1csOpened:** In this state the connection header and the open frame have been both sent and received.
- **amqp1csCloseReceived:** In this state a close frame has been received indicating that the peer has initiated an AMQP close. No further frames are expected to arrive on the connection; however, frames can still be sent. If desired, an implementation MAY do a TCP half-close at this point to shut down the read side of the connection.
- **amqp1csCloseSent:** In this state a close frame has been sent to the peer. It is illegal to write anything more onto the connection, however there could potentially still be incoming frames. If desired, an implementation MAY do a TCP half-close at this point to shutdown the write side of the connection.
- **amqp1csDiscarding:** The DISCARDING state is a variant of the CLOSE SENT state where the close is triggered by an error. In this case any incoming frames on the connection MUST be silently discarded until the peer's close frame is received.
- **amqp1csEnd:** In this state it is illegal for either endpoint to write anything more onto the connection. The connection can be safely closed and discarded.

The AMQP Client has the property **ConnectionState** where you can check in which connection state is the client component.

Connection | AMQP1 Authentication

The component has the following authentication methods:

- **amqp1authNone:** there is no authentication method to use when connecting to the server.
- **amqp1authSASLAnonymous:** connects as anonymous.
- **amqp1authSASLPlain:** the default, uses a user/password authentication.
- **amqp1authSASLExternal:** not currently supported.

SASL Authentication

The most common authentication is using **amqp1authSASLPlain** type. This authentication type, can be enabled in the AMQP1 component, accessing to the property AMQPOptions.Authentication.

- **AuthType:** select amqp1authSASLPlain
- **Username:** the user to use for SASL Authentication.
- **Password:** the secret value to use for SASL Authentication.

The result of the SASL Authentication can be obtained when the event **OnAMQPSASLAuthentication**.

```
public void OnAMQP1SASLAuthentication(object Sender, TsgcAMQP1SaslCode aCode,
        string aDescription, ref bool Handled)
{
  MessageBox.Show("#sasl-authentication: " + aDescription);
}
```

Connection | Azure MessageBus

AMQP (Advanced Message Queuing Protocol) is a robust messaging system designed to facilitate communication between diverse containers across various nodes. It standardizes both the protocol for transmitting messages and the structural framework of the messages themselves, ensuring consistent and reliable communication. To dive deeper into the fundamentals of AMQP, refer to our Getting Started with AMQP guide.

The AMQP component within the eSeGeCe library enables seamless integration with leading cloud messaging brokers, including Amazon MQ and Azure Service Bus. This guide focuses on using the AMQP component to connect with Azure Service Bus, demonstrating how to build a multi-tenant application capable of sending and receiving messages efficiently.

The component offers a comprehensive implementation with support for key features such as queues, topics, and subscriptions, making it an ideal choice for modern IoT and enterprise applications.

Azure Configuration

To begin, create a **Service Bus resource** within the Azure Portal. Once the resource is established, make sure to take note of the **resource's domain name**, as it will be essential for integration and configuration.

After the **namespace** has been successfully created, you can manage and monitor it directly from the **namespace overview** in the Azure Portal. This centralized interface provides access to key management tools and settings, enabling seamless administration of your Service Bus resource.

When using **SAS Authentication**, the **username is the SAS Policy name** and the **password is the primary or secondary key**.

```
// Create TCP client
var oClient = new TsgcWebSocketClient
{
    Specifications = { RFC6455 = false },
    Host = "esegece.servicebus.windows.net",
    Port = 5671,
    TLS = true
};
// Create AMQP1 protocol client
var oAMQP1 = new TsgcWSClient_AMQP1
{
    Specifications = { RFC6455 = false },
    AMQPOptions =
    {
        Authentication = {
        AuthType = AMQP1AuthType.SASLPlain,
        Username = "RootManageSharedAccessKey",
        Password = "BhJ78+w8kMXhS/eE/nBy0cRzodx9tipbi+ASbAXIaH8="
    }
},
Client = oClient;
// Connect to the server
oClient.Active = true;
```

Azure CBS Authentication

Azure Service Bus implements Claims-Based Security (CBS) over AMQP to authorize senders and receivers after the initial SASL handshake. The client opens a management link to the **$cbs** node and sends a **put-token** request containing either a Shared Access Signature (SAS) token or a JSON Web Token (JWT) issued by Microsoft Entra ID. Once the broker validates the token, the authorization is cached for its lifetime and the application can proceed to create sender and receiver links against queues, topics, or subscriptions.

The AMQP1 client automates this flow through two helper methods:

- **CreateAzureCbsSasToken** establishes a CBS sender/receiver link pair, generates a SAS token for the target entity, and publishes it to **$cbs**. Use it when authenticating with a shared access policy.

- **CreateAzureCbsJWT** follows the same CBS exchange but obtains an access token from Microsoft Entra ID (Azure AD) using the client-credentials grant before sending the JWT to **$cbs**.

Both methods require an active AMQP connection and accept the following parameters:

- **aName**: Identifier for the CBS link pair created internally.
- **aNameSpace** and **aEntityName**: The Service Bus namespace (without the *.servicebus.windows.net* suffix) and the queue, topic, or subscription path used to build the token audience.
- **aKeyName** / **aKeyValue**: Shared access policy name and key for SAS tokens. The component signs the token and sends it using the token type *servicebus.windows.net:sastoken*.
- **aTenant**, **aApplicationId**, **aSecret**: Microsoft Entra (Azure AD) directory ID, application (client) ID, and client secret used to request the JWT with the client credentials flow.
- **aListeningPort** (JWT): Local HTTP port for the OAuth 2.0 redirect (defaults to 8080 when not provided).
- **aExpiration** and **aTimeout**: Lifetime of the issued token (in seconds) and the maximum wait time (in milliseconds) for the CBS negotiation.
- **aRaiseIfError**: When set to *True*, the method raises an exception if token acquisition or the CBS response fails.

The following examples illustrate how to authenticate with CBS before sending messages.

```
// Create TCP client
var oClient = new TsgcWebSocketClient
{
    Specifications = { RFC6455 = false },
    Host = "esegece.servicebus.windows.net",
    Port = 5671,
    TLS = true
};
// Create AMQP1 protocol client
var oAMQP1 = new TsgcWSClient_AMQP1
{
    Specifications = { RFC6455 = false },
    AMQPOptions =
    {
        Authentication = { AuthType = AMQP1AuthType.SASLAnonymous }
    },
    Client = oClient
};
// Connect and publish SAS token through CBS
oClient.Active = true;
oAMQP1.CreateAzureCbsSasToken("cbs", "esegece", "queue1",
    "RootManageSharedAccessKey",
    "BhJ78+w8kMXhS/eE/nBy0cRzodx9tipbi+ASbAXIaH8=",
    3600, 10000, true);
```

The next example focuses solely on Microsoft Entra ID (Azure AD) authentication using JWTs. It shows how to request a token with the client credentials flow and publish it to **$cbs** before creating links to send or receive messages.

```
// Create TCP client
var oClient = new TsgcWebSocketClient
{
    Specifications = { RFC6455 = false },
    Host = "esegece.servicebus.windows.net",
    Port = 5671,
    TLS = true
};
// Create AMQP1 protocol client
var oAMQP1 = new TsgcWSClient_AMQP1
{
    Specifications = { RFC6455 = false },
    AMQPOptions =
    {
        Authentication = { AuthType = AMQP1AuthType.SASLAnonymous }
    },
    Client = oClient
};
// Connect and publish JWT through CBS
oClient.Active = true;
oAMQP1.CreateAzureCbsJWT("cbs", "esegece", "queue1",
    "00000000-0000-0000-0000-000000000000", // Tenant ID
    "11111111-1111-1111-1111-111111111111", // Application ID
```

```
    "client-secret", 8080, 3600, 10000, true);
```

Commands | AMQP1 Sessions

In the context of the AMQP (Advanced Message Queuing Protocol) 1.0.0 specification, a session represents a logical context for communication between a client and a message broker. Here's a breakdown of what an AMQP 1.0.0 session entails:

- **Logical Context:** A session establishes a logical context for messaging operations between an AMQP client (producer or consumer) and an AMQP broker. It provides a way to group related messaging operations together within a single connection.
- **Communication Channel:** Sessions serve as communication channels over which messages are sent and received. They encapsulate the exchange of messages, acknowledgments, and flow control mechanisms.
- **Transactional Boundaries:** Sessions define transactional boundaries for message operations. They enable the grouping of multiple message sends or receives into a single atomic unit, ensuring that either all operations within the session are processed successfully or none are processed at all.
- **Flow Control:** Sessions support flow control mechanisms to regulate the rate at which messages are exchanged between the client and the broker. Flow control helps prevent overwhelming the resources of either party, ensuring efficient and reliable message delivery.
- **Lifetime Management:** Sessions have a lifecycle that begins when they are created and ends when they are closed. Clients can establish multiple sessions within a single connection to parallelize message processing or isolate message streams.
- **Resource Allocation:** Sessions may be associated with specific resources such as queues, topics, or subscriptions within the broker. Messages sent or received within a session are bound to these resources, enabling targeted message routing and delivery.

In summary, an AMQP 1.0.0 session provides a logical context for message exchange between an AMQP client and broker, facilitating transactional integrity, flow control, and resource management within the messaging system. It defines the boundaries within which messaging operations are performed and helps ensure the efficient and reliable exchange of messages.

Open Session

The method **CreateSession** creates a new session with the given name (or if empty, it creates with a random name), if the session already exists an exception is raised. The client allows you to create multiple session using the same AMQP connection.

Once the session is successfully created, the event **OnAQMPSessionOpen** is fired.

```
oAMQP.OnAMQPSessionOpen += OnAMQPSessionOpenEvent;
oAMQP.CreateSession("MySession");
private void OnAMQP1AMQPSessionOpenEvent(object sender, TsgcAMQP1Session aSession, TsgcAMQP1FrameBegin aBegin)
{
    Console.WriteLine("#session-open: " + aSession.Id);
}
```

The **CreateSession** method returns the **TsgcAMQP1Session** class which contains the session information.

Await Open Session

By default, the **CreateSession** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CreateSession method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void OpenSession(string aSession)
{
    TsgcAMQP1MethodOptions_SessionOpen oOptions = new TsgcAMQP1MethodOptions_SessionOpen();
```

```
    try
    {
        oOptions.Await = true;
        AMQP1.CreateSession(aSession, oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
}
```

Close Session

To Close an existing session use the method **CloseSession** passing the name of the session to close.

Once the session is successfully closed, the event **OnAQMPSessionClose** is fired.

```
oAMQP.OnAMQPSessionClose += OnAMQPSessionCloseEvent;
oAMQP.CloseSession("MySession");
private void OnAMQPSessionCloseEvent(object Sender, TsgcAMQP1Session aSession, TsgcAMQP1FrameEnd aEnd)
{
    Console.WriteLine("#session-close: " + aSession.Id + " [" + aSession.Channel.ToString() + "] reason: " + aEnd
}
```

Await Close Session

By default, the **CloseSession** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CloseSession method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void CloseSession(string aSession)
{
    TsgcAMQP1MethodOptions_SessionClose oOptions = new TsgcAMQP1MethodOptions_SessionClose();
    try
    {
        oOptions.Await = true;
        AMQP1.CloseSession(aSession, oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
}
```

Commands | AMQP1 Links

In the AMQP (Advanced Message Queuing Protocol) 1.0.0 specification, a link represents a unidirectional communication channel between an AMQP client and a message broker. Let's delve deeper into what AMQP 1.0.0 links entail:

- **Communication Channel:** A link serves as a pathway through which messages flow between an AMQP sender and receiver. It allows for the transmission of messages in one direction, either from the sender to the receiver or vice versa.
- **Unidirectional Flow:** Each link is unidirectional, meaning that messages can only travel in one direction along the link. If bidirectional communication is needed, two links must be established—one for each direction.
- **Message Transfer:** Messages are transferred across links according to the AMQP protocol rules. These messages can include payloads, message properties, and additional metadata required for communication.
- **Resource Binding:** Links are associated with specific resources within the AMQP broker, such as queues, topics, or exchanges. Messages sent or received via a link are directed to or originate from these resources.
- **Flow Control:** Links support flow control mechanisms to regulate the rate at which messages are sent or received. Flow control ensures that neither the sender nor the receiver is overwhelmed by the volume of messages being exchanged.
- **Lifetime Management:** Links have a lifecycle that begins when they are established and ends when they are closed. They can be created dynamically as needed and closed when they are no longer required.
- **Addressing:** Links are identified by unique addresses that specify the source and target endpoints of the communication. These addresses allow clients and brokers to identify and establish connections to the appropriate endpoints.
- **Transactional Boundaries:** Links define transactional boundaries for message operations. They enable the grouping of multiple message sends or receives into a single atomic unit, ensuring consistency and reliability in message delivery.

In summary, AMQP 1.0.0 links provide a means for unidirectional communication between AMQP clients and brokers, facilitating the transfer of messages while supporting flow control, resource binding, addressing, and transactional integrity within the messaging system. They form the fundamental building blocks of message exchange in the AMQP protocol.

There are 2 types of Links:

- **Sender Links:** those links are used to send messages.
- **Receiver Links:** those links are used to receive messages.

Every time a new link is created or deletes, the following events are fired:

- **OnAMQPLinkOpen:** this event is triggered when a new link is created. Use the aLink.Mode property to check if the link is in receiver or sender mode.
- **OnAMQPLinkClose:** this event is triggered when a link is closed.

Commands | AMQP1 Sender Links

In the AMQP 1.0.0 protocol, a **Sender Link** is a **communication channel** established between an AMQP client and an AMQP server for the purpose of **sending messages**. It operates within the context of an AMQP session, which represents a logical channel for communication between the client and server.

Create Sender Link

To **Create a new Sender Link**, call the method **CreateSenderLink** which contains the following parameters:

- **Session:** the session name where the sender link will be attached.
- **Name:** (optional) the name of the sender link, if is not set, a random name will be assigned automatically.
- **Target:** (optional) you can specify the destination where messages should be received on the remote host by setting the "target" parameter. However, in certain scenarios, specifying the target may not be required. In such cases, providing an empty string will be sufficient.
- **SndSettleMode:** (mixed by default) AMQP offers the capability to discuss delivery assurances via the Message Settlement mechanism. Upon establishing a link, both the sender and the receiver discuss and agree upon a settlement mode (one for each role). Senders operate within one of these modes:
  - **amqp1ssmSettled:** The message is considered successfully delivered and acknowledged once it's sent.
  - **amqp1ssmUnsettled:** The message is not considered settled until it's explicitly accepted or rejected by the receiver. This allows for more control over message processing and handling.
  - **amqp1ssmMixed:** A combination of settled and unsettled modes can be used within a single AMQP session. Use the **MessageOptions** parameter of the **SendMessage** method to configure if the message is Settled or not.

When the Sender Link has been created successfully, the event **OnAMQPLinkOpen** will be fired.

```
oAMQP1.OnAMQPLinkOpen += AMQP1AMQPLinkOpen;
oAMQP1.CreateSenderLink("MySession", "MySenderLink");
private void AMQP1AMQPLinkOpen(object sender, TsgcAMQP1Session aSession, TsgcAMQP1Link aLink, TsgcAMQP1FrameAttac
{
    Console.WriteLine("#link-open: " + aLink.Name);
}
```

Await Create Sender Link

By default, the **CreateSenderLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CreateSenderLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void CreateSenderLink(string aSession, string aSender)
{
    TsgcAMQP1MethodOptions_CreateSenderLink oOptions = new TsgcAMQP1MethodOptions_CreateSenderLink();
    try
    {
        oOptions.Await = true;
        AMQP1.CreateSenderLink(aSession, aSender, "", oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
}
```

Sending Messages

To Send a new Message, call the method SendMessage which contains the following parameters:

- **Session:** name of the session.
- **Link:** name of the sender link.
- **Text:** the text of the string message.

```
oAMQP1.SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

Sending Messages Mixed Mode

When the Sender Link is created in Mixed mode (the default), when sending a message, the user can set if want the message is **settled** or **not**. Use the **MessageOptions** parameter to define if the message is settled or not.

```
TsgcAMQP1MessageOptions oMessageOptions = new TsgcAMQP1MessageOptions();
try
{
    oMessageOptions.Settled = true;
    oAMQP1.SendMessage("MySession", "MySenderLink", "MyMessage", "message-id", oMessageOptions);
}
finally
{
    oMessageOptions.Dispose();
}
```

Close Sender Link

To Close an existing Sender Link, call the method **CloseLink** which contains the following parameters:

- Session: name of the session that contains the link.
- Link: name of the sender link.
- Error: (optional) here you can set the reason why the link is closed.

When the Sender Link has been closed successfully, the event **OnAMQPLinkClose** will be fired.

```
oAMQP.CloseLink("MySession", "MySenderLink");
public void OnAMQPLinkCloseEvent(object sender, TsgcAMQP1Session aSession, TsgcAMQP1Link aLink, TsgcAMQP1FrameDet
{
    Console.WriteLine("#link-close: " + aLink.Name);
}
```

Await Close Sender Link

By default, the **CloseLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CloseLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void CloseSenderLink(string aSession, string aSenderLink)
{
    TsgcAMQP1MethodOptions_CloseLink oOptions = new TsgcAMQP1MethodOptions_CloseLink();
    try
    {
        oOptions.Await = true;
        AMQP1.CloseLink(aSession, aSenderLink, oOptions);
    }
    finally
    {
        oOptions.Dispose();
```

```
    }
}
```

Commands | AMQP1 Receiver Links

In the AMQP 1.0.0 protocol, a **Receiver Link** is a **communication channel** established between an AMQP client and an AMQP server for the purpose of **receiving messages**. It operates within the context of an AMQP session, which represents a logical channel for communication between the client and server.

Create Receiver Link

To **Create a new Receiver Link**, call the method **CreateReceiverLink** which contains the following parameters:

- **Session:** the session name where the sender link will be attached.
- **Name:** (optional) the name of the sender link, if is not set, a random name will be assigned automatically.
- **Source:** (optional) the source can be configured to indicate the location of the node on the remote host that is supposed to act as the sender. In some situations, specifying this address may not be required. In such cases, simply providing an empty string as the value for the parameters will be enough.
- **ReadMode:** (amqp1srmAuto by default) Receiver links can function in one of two modes for receiving messages:
  - **amqp1srmAuto:** Automatic Mode, in this mode the receiver actively works to ensure that messages are received promptly as soon as they become available. It automatically listens for and receives messages without any explicit instruction each time a new message arrives.
  - **amqp1srmManual:** Fetch-Based Mode, in this mode, the receiver will only retrieve or fetch a new message when it is specifically told to do so. Unlike the automatic mode, the receiver will not actively listen for new messages but will instead wait for manual instructions to fetch the next message.
- **RcvSettleMode:** (amqp1rsmFirst by default) Receiver Links operate within one of these modes:
  - **amqp1rsmFirst:** When messages arrive, they will be processed and confirmed right away. If the message hasn't already been confirmed by the time it was sent, the sender will be informed that the message has been received.
  - **amqp1rsmSecond:** Messages that arrive will only be confirmed after the sender has confirmed them first. Additionally, the sender will receive a notification when a message has been received, provided the message wasn't already confirmed when it was sent.

When the Receiver Link has been created successfully, the event **OnAMQPLinkOpen** will be fired.

```
oAMQP1.OnAMQPLinkOpen += AMQP1AMQPLinkOpen;
oAMQP1.CreateReceiverLink("MySession", "MyReceiverLink");
private void AMQP1AMQPLinkOpen(object sender, TsgcAMQP1Session aSession, TsgcAMQP1Link aLink, TsgcAMQP1FrameAttac
{
    Console.WriteLine("#link-open: " + aLink.Name);
}
```

Await Create Receiver Link

By default, the **CreateReceiverLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CreateReceiverLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void CreateReceiverLink(string aSession, string aReceiver)
{
    TsgcAMQP1MethodOptions_CreateReceiverLink oOptions = new TsgcAMQP1MethodOptions_CreateReceiverLink();
    try
    {
        oOptions.Await = true;
        AMQP1.CreateReceiverLink(aSession, aReceiver, "", oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
```

```
}
```

Sync Messages

When the Receiver Link works in Manual ReadMode, call the method **GetMessage** to get new messages. This method is synchronous, which means that waits till a timeout is exceeded (by default 10 seconds). When the method is called, the component increases the credit in one unit and waits till a new message arrives or the timeout has been exceeded. If no message arrives, the credit is set to zero again.

The method **GetMessage** has the following parameters:

- **Session:** name of the session that contains the link.
- **Link:** name of the receiver link.
- **Timeout:** (by default 1000 = 10 seconds) the max time the function will wait to get a new message.

Close Receiver Link

To Close an existing Receiver Link, call the method **CloseLink** which contains the following parameters:

- **Session:** name of the session that contains the link.
- **Link:** name of the receiver link.
- **Error:** (optional) here you can set the reason why the link is closed.

When the Receiver Link has been closed successfully, the event **OnAMQPLinkClose** will be fired.

```
oAMQP1.CloseLink("MySession", "MyReceiverLink");
public void OnAMQPLinkCloseEvent(object sender, TsgcAMQP1Session aSession, TsgcAMQP1Link aLink, TsgcAMQP1FrameDet
{
    Console.WriteLine("#link-close: " + aLink.Name);
}
```

Await Close Receiver Link

By default, the **CloseLink** method is **Asynchronous**, so after calling the method, the code continue. If you want to **wait till the CloseLink method is completed** and the confirmation sent by the server is received, set the property **Await** to True in the Options parameter.

```
public static void CloseReceiverLink(string aSession, string aReceiverLink)
{
    TsgcAMQP1MethodOptions_CloseLink oOptions = new TsgcAMQP1MethodOptions_CloseLink();
    try
    {
        oOptions.Await = true;
        AMQP1.CloseLink(aSession, aReceiverLink, oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
}
```

AMQP1 | Send Message

Read first AMQP1 Sender Links to know how to create a Sender Link.

Send Message

Use the method **SendMessage** passing the Session and SenderLink name to send a text message to the AMQP1 Server. The method has the following parameters:

- **Session:** name of the session.
- **Link:** name of the sender link.
- **Text:** text of the message.
- **MessageId:** (optional) the id of the message, it can be used when using unsettled mode, to know if the server has processed the message.
- **Options:** (optional) allows customizing some options when sending the message.
  - **Settled:** when using a sender link in mixed mode, when sending a message the Settled property can be customized.
  - **Await:** if the message is unsettled, and the value is true, the code will wait till the message is processed by the server or the timeout has exceeded.
  - **Timeout:** value in milliseconds if await is true (by default 10000).
  - **RaiseTimeoutException:** if the timeout is exceeded, an exception is raised (by default true).

```
oAMQP1.SendMessage("MySession", "MySenderLink", "My first AMQP Message");
```

Await Send Message

By default, the **SendMessage** method is asynchronous when sending a message unsettled, setting the property **Await** to true, the client will wait till receives a confirmation from the server that the message has been processed.

```
public static void SendMessageAwait(string aSession, string aSenderLink, string aText)
{
    TsgcAMQP1MethodOptions_SendMessageAck oOptions = new TsgcAMQP1MethodOptions_SendMessageAck();
    try
    {
        oOptions.Settled = false;
        oOptions.Await = true;
        AMQP1.SendMessage(aSession, aSenderLink, aText, "messsage-id", oOptions);
    }
    finally
    {
        oOptions.Dispose();
    }
}
```

Events

When sending a message, there are 2 Events that can be used to know when the message is sent and if the message has been processed by the server (when sending unsettled).

- **OnAMQPMessageSent:** this event is called after the message is sent to the server. When calling the method SendMessage, the message is stored in an internal queue and processed by a secondary thread, so after the message is sent, this event is called.

- **OnAMQPMessageSentAck:** this event is called, when the client receives a confirmation that the message has been processed by the AMQP1 Server.

```
public void OnAMQPMessageSentAck(object Sender, TsgcAMQP1Session aSession,
        TsgcAMQP1SenderLink aLink, string aMessageId, TsgcAMQP1FrameDeliveryStates aDeliveryState,
        TsgcAMQP1FrameDisposition aDisposition)
{
        string vMessageId = aMessageId;
        switch (aDeliveryState)
        {
                case TsgcAMQP1FrameDeliveryStates.amqp1fdtsAccepted:
                        MessageBox.Show("#msg-accepted: " + vMessageId);
                        break;
                case TsgcAMQP1FrameDeliveryStates.amqp1fdtsRejected:
                        MessageBox.Show("#msg-rejected: " + vMessageId + " " +
                                ((TsgcAMQP1FrameRejected)aDisposition.State).Error.Condition + " " +
                                ((TsgcAMQP1FrameRejected)aDisposition.State).Error.Description);
                        break;
                case TsgcAMQP1FrameDeliveryStates.amqp1fdtsReleased:
                        MessageBox.Show("#msg-released: " + vMessageId);
                        break;
                case TsgcAMQP1FrameDeliveryStates.amqp1fdtsModified:
                        MessageBox.Show("#msg-modified: " + vMessageId + " " +
                                ((TsgcAMQP1FrameModified)aDisposition.State).MessageAnnotations);
                        break;
                case TsgcAMQP1FrameDeliveryStates.amqp1fdtsReceived:
                        MessageBox.Show("#msg-received: " + vMessageId);
                        break;
        }
}
```

AMQP1 | Read Message

Every time a new message is received, the event **OnAMQPMessage** is fired.

The TsgcAMQP1Message instance contains the message received. You can access to the text message using the property **aMessage.ApplicationData.AMQPValue.Value**.

To specify the Delivery Outcome, use the **DeliveryState** parameter. By default, all the messages have the accepted state, but you can set one of the following:

- **amqp1mdtsAccepted:** The message has been processed successfully.
- **amqp1mdtsRejected:** The message failed to process successfully. Set the error using the property DeliveryState.Rejected.
- **amqp1mdtsReleased:** The message has not been and won't be processed.
- **amqp1mdtsModified:** Same as amqp1mdtsReleased, but you can add additional data using the property DeliveryState.Modified.

```
public void OnAMQPMessage(object Sender, TsgcAMQP1Session aSession,
        TsgcAMQP1ReceiverLink aLink, TsgcAMQP1Message aMessage,
        ref TsgcAMQP1MessageDeliveryState DeliveryState)
{
  if (aMessage.ApplicationData.AMQPValue.Value == "xxx")
  {
    DeliveryState.State = TsgcAMQP1MessageDeliveryStates.amqp1mdtsRejected;
    DeliveryState.Rejected.Error.Condition = "amqp-error-processing";
    DeliveryState.Rejected.Error.Description = "Value received was not the expected.";
  }
  else
  {
    DeliveryState.State = TsgcAMQP1MessageDeliveryStates.amqp1mdtsAccepted;
  }
}
```

Protocol STOMP

STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

Our STOMP client components support following STOMP versions: 1.0, 1.1 and 1.2.

Components

TsgcWSPClient_STOMP: generic STOMP Protocol client, allows you to connect to any STOMP Server.

TsgcWSPClient_STOMP_RabbitMQ: STOMP client for RabbitMQ Broker.

TsgcWSPClient_STOMP_ActiveMQ: STOMP client for ActiveMQ Broker.

TsgcWSPClient_STOMP

This is the Client Protocol STOMP Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

Send: The SEND frame sends a message to a destination in the messaging system.

Subscribe: The SUBSCRIBE frame is used to register to listen to a given destination.

UnSubscribe: The UNSUBSCRIBE frame is used to remove an existing subscription.

ACK: ACK is used to acknowledge the consumption of a message from a subscription.

NACK: NACK is the opposite of ACK. It is used to tell the server that the client did not consume the message.

BeginTransaction: is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be processed atomically based on the transaction.

CommitTransaction: is used to commit a transaction in progress.

AbortTransaction: is used to roll back a transaction in progress.

Disconnect: used to gracefully shut down the connection, where the client is assured that all previous frames have been received by the server.

Events

OnSTOMPConnected: this event is triggered after a new connection is established.

version : The version of the STOMP protocol the session will be using. See Protocol Negotiation for more details.
STOMP 1.2 servers MAY set the following headers:
heart-beat : The Heart-beating settings.
session : A session identifier that uniquely identifies the session.
server : A field that contains information about the STOMP server. The field MUST contain a server-name field and MAY be followed by optional comment fields delimited by a space character.

OnSTOMPMessage: this event is triggered when the client receives a message.

The MESSAGE frame MUST include a destination header indicating the destination the message was sent to. If the message has been sent using STOMP, this destination header SHOULD be identical to the one used in the corresponding SEND frame.
The MESSAGE frame MUST also contain a message-id header with a unique identifier for that message and a subscription header matching the identifier of the subscription that is receiving the message.
If the message is received from a subscription that requires explicit acknowledgment (either client or client-individual mode) then the MESSAGE frame MUST also contain an ack header with an arbitrary value. This header will be used to relate the message to a subsequent ACK or NACK frame.
MESSAGE frames SHOULD include a content-length header and a content-type header if a body is present.
MESSAGE frames will also include all user-defined headers that were present when the message was sent to the destination in addition to the server-specific headers that MAY get added to the frame.

OnSTOMPReceipt: this event is triggered once a server has successfully processed a client frame that requests a receipt.

A RECEIPT frame is an acknowledgment that the corresponding client frame has been processed by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been received by the server. However, these previous frames may not yet be fully processed. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

OnSTOMPError: this event is fired if something goes wrong.
The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).
If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.
ERROR frames SHOULD include a content-length header and a content-type header if a body is present.

OnSTOMPPing: this event is fired when a ping is sent or received
Handle here the pings received/sent between the client and server. The Parameter LastIncoming tells when the last ping was received and LastOutgoing when the last ping was sent.

Properties

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: Heart-beating can optionally be used to test the healthiness of the underlying TCP connection and to make sure that the remote end is alive and kicking. In order to enable heart-beating, each party has to declare what it can do and what it would like the other party to do. 0 means it cannot send/receive heart-beats, otherwise it is the desired number of milliseconds between heart-beats.

Options: The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.

Versions: Set which STOMP versions are supported.

ConnectHeaders: Allows sending custom headers when CONNECT method is sent.

TsgcWSPClient_STOMP_RabbitMQ

This is the Client Protocol STOMP Component for RabbitMQ Broker. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Destinations

The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific. The RabbitMQ STOMP adapter supports a number of different destination types:

- **Topic:** SEND and SUBSCRIBE to transient and durable topics.
- **Queue:** SEND and SUBSCRIBE to queues managed by the STOMP gateway.
- **QueueOutside:** SEND and SUBSCRIBE to queues created outside the STOMP gateway.
- **TemporaryQueue:** create temporary queues (in reply-to headers only).
- **Exchange:** SEND to arbitrary routing keys and SUBSCRIBE to arbitrary binding patterns.

Methods

Publish: The SEND frame sends a message to a destination in the messaging system.
PublishTopic
PublishQueue
PublishQueueOutside
PublishTemporaryQueue
PublishExchange

Subscribe: The SUBSCRIBE frame is used to register to listen to a given destination. Supports following subscriptions
SubscribeTopic
SubscribeQueue
SubscribeQueueOutside
SubscribeTemporaryQueue
SubscribeExchange

UnSubscribe: The UNSUBSCRIBE frame is used to remove an existing subscription. Supports following UnSubscriptions
UnSubscribeTopic
UnSubscribeQueue
UnSubscribeQueueOutside
UnSubscribeTemporaryQueue
UnSubscribeExchange

ACK: ACK is used to acknowledge the consumption of a message from a subscription.

NACK: NACK is the opposite of ACK. It is used to tell the server that the client did not consume the message.

BeginTransaction: is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be processed atomically based on the transaction.

CommitTransaction: is used to commit a transaction in progress.

AbortTransaction: is used to roll back a transaction in progress.

Disconnect: used to gracefully shut down the connection, where the client is assured that all previous frames have been received by the server.

Events

OnRabbitMQConnected: this event is triggered after a new connection is established.

version : The version of the STOMP protocol the session will be using. See Protocol Negotiation for more details.

STOMP 1.2 servers MAY set the following headers:

heart-beat : The Heart-beating settings.

session : A session identifier that uniquely identifies the session.

server : A field that contains information about the STOMP server. The field MUST contain a server-name field and MAY be followed by optional comment fields delimited by a space character.

OnRabbitMQMessage: this event is triggered when the client receives a message.

The MESSAGE frame MUST include a destination header indicating the destination the message was sent to. If the message has been sent using STOMP, this destination header SHOULD be identical to the one used in the corresponding SEND frame.

The MESSAGE frame MUST also contain a message-id header with a unique identifier for that message and a subscription header matching the identifier of the subscription that is receiving the message.

If the message is received from a subscription that requires explicit acknowledgment (either client or client-individual mode) then the MESSAGE frame MUST also contain an ack header with an arbitrary value. This header will be used to relate the message to a subsequent ACK or NACK frame.

MESSAGE frames SHOULD include a content-length header and a content-type header if a body is present.

MESSAGE frames will also include all user-defined headers that were present when the message was sent to the destination in addition to the server-specific headers that MAY get added to the frame.

OnRabbitMQReceipt: this event is triggered once a server has successfully processed a client frame that requests a receipt.

A RECEIPT frame is an acknowledgment that the corresponding client frame has been processed by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been received by the server. However, these previous frames may not yet be fully processed. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

OnRabbitMQError: this event is triggered if something goes wrong.

The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).

If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.

ERROR frames SHOULD include a content-length header and a content-type header if a body is present.

Properties

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: Heart-beating can optionally be used to test the healthiness of the underlying TCP connection and to make sure that the remote end is alive and kicking. In order to enable heart-beating, each party has to declare what it can do and what it would like the other party to do. 0 means it cannot send/receive heart-beats, otherwise it is the desired number of milliseconds between heart-beats.

Options: The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.

Versions: Set which STOMP versions are supported.

TsgcWSPClient_STOMP_ActiveMQ

This is the Client Protocol STOMP Component for ActiveMQ Broker. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Destinations

The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific. The Active STOMP adapter supports a number of different destination types:

- **Topic:** SEND and SUBSCRIBE to transient and durable topics.
- **Queue:** SEND and SUBSCRIBE to queues managed by the STOMP gateway.

Publish Options

Note that STOMP is designed to be as simple as possible - so any scripting language/platform can message any other with minimal effort. STOMP allows pluggable headers on each request such as sending & receiving messages. ActiveMQ has several extensions to the Stomp protocol, so that JMS semantics can be supported by Stomp clients. An OpenWire JMS producer can send messages to a Stomp consumer, and a Stomp producer can send messages to an OpenWire JMS consumer. And Stomp to Stomp configurations, can use the richer JMS message control.

STOMP supports the following standard JMS properties on SENT messages:

- **CorrelationId:** Good consumers will add this header to any responses they send.
- **Expires:** Expiration time of the message.
- **JMSXGroupID:** Specifies the Message Groups.
- **JMSXGroupSeq:** Optional header that specifies the sequence number in the Message Groups.
- **Persistent:** Whether or not the message is persistent.
- **Priority:** Priority on the message.
- **ReplyTo:** Destination you should send replies to.
- **MsgType:** Type of the message.

Methods

Publish: The SEND frame sends a message to a destination in the messaging system.
PublishTopic
PublishQueue

Subscribe: The SUBSCRIBE frame is used to register to listen to a given destination. Supports following subscriptions
SubscribeTopic
SubscribeQueue

UnSubscribe: The UNSUBSCRIBE frame is used to remove an existing subscription. Supports following UnSubscriptions
UnSubscribeTopic
UnSubscribeQueue

ACK: ACK is used to acknowledge the consumption of a message from a subscription.

NACK: NACK is the opposite of ACK. It is used to tell the server that the client did not consume the message.

BeginTransaction: is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be processed atomically based on the transaction.

CommitTransaction: is used to commit a transaction in progress.

AbortTransaction: is used to roll back a transaction in progress.

Disconnect: used to gracefully shut down the connection, where the client is assured that all previous frames have been received by the server.

Events

OnActiveMQConnected: this event is triggered after a new connection is established.

version : The version of the STOMP protocol the session will be using. See Protocol Negotiation for more details.

STOMP 1.2 servers MAY set the following headers:

heart-beat : The Heart-beating settings.

session : A session identifier that uniquely identifies the session.

server : A field that contains information about the STOMP server. The field MUST contain a server-name field and MAY be followed by optional comment fields delimited by a space character.

OnActiveMQMessage: this event is triggered when the client receives a message.

The MESSAGE frame MUST include a destination header indicating the destination the message was sent to. If the message has been sent using STOMP, this destination header SHOULD be identical to the one used in the corresponding SEND frame.

The MESSAGE frame MUST also contain a message-id header with a unique identifier for that message and a subscription header matching the identifier of the subscription that is receiving the message.

If the message is received from a subscription that requires explicit acknowledgment (either client or client-individual mode) then the MESSAGE frame MUST also contain an ack header with an arbitrary value. This header will be used to relate the message to a subsequent ACK or NACK frame.

MESSAGE frames SHOULD include a content-length header and a content-type header if a body is present.

MESSAGE frames will also include all user-defined headers that were present when the message was sent to the destination in addition to the server-specific headers that MAY get added to the frame.

OnActiveMQReceipt: this event is triggered once a server has successfully processed a client frame that requests a receipt.

A RECEIPT frame is an acknowledgment that the corresponding client frame has been processed by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been received by the server. However, these previous frames may not yet be fully processed. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

OnActiveMQError: this event is triggered if something goes wrong.

The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).

If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.

ERROR frames SHOULD include a content-length header and a content-type header if a body is present.

Properties

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: Heart-beating can optionally be used to test the healthiness of the underlying TCP connection and to make sure that the remote end is alive and kicking. In order to enable heart-beating, each party has to declare what it can do and what it would like the other party to do. 0 means it cannot send/receive heart-beats, otherwise it is the desired number of milliseconds between heart-beats.

Options: The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.

Versions: Set which STOMP versions are supported.

Protocol AppRTC

WebRTC (Web Real-Time Communication) is an API definition being drafted by the World Wide Web Consortium (W3C) to enable browser-to-browser applications for voice calling, video chat and P2P file sharing without plugins. The RTC in WebRTC stands for Real-Time Communications, a technology that enables audio/video streaming and data sharing between browser clients (peers). As a set of standards, WebRTC provides any browser with the ability to share application data and perform teleconferencing peer to peer, without the need to install plug-ins or third-party software.

WebRTC components are accessed with JavaScript APIs. Currently, in development are the Network Stream API, which represents an audio or video data stream, and the PeerConnection API, which allows two or more users to communicate browser-to-browser. Also under development is a DataChannel API that enables communication of other types of data for real-time gaming, text chat, file transfer, and so forth.

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

Components

TsgcWSPServer_AppRTC: Server Protocol AppRTC VCL Component.

TsgcWSPServer_AppRTC

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

Parameters

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

WebRTC Protocol requires STUN/TURN server, demos use public STUN/TURN servers for testing purposes. In order to put in a production system, a dedicated STUN/TURN server is required.
Registered users can download compiled binaries of **Coturn server for Windows**. Read more about COTURN STUN/TURN.

IceServers Configuration

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

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

Protocol WebRTC

WebRTC (Web Real-Time Communication) is an API definition being drafted by the World Wide Web Consortium (W3C) to enable browser-to-browser applications for voice calling, video chat and P2P file sharing without plugins. The RTC in WebRTC stands for Real-Time Communications, a technology that enables audio/video streaming and data sharing between browser clients (peers). As a set of standards, WebRTC provides any browser with the ability to share application data and perform teleconferencing peer to peer, without the need to install plug-ins or third-party software.

WebRTC components are accessed with JavaScript APIs. Currently, in development are the Network Stream API, which represents an audio or video data stream, and the PeerConnection API, which allows two or more users to communicate browser-to-browser. Also under development is a DataChannel API that enables communication of other types of data for real-time gaming, text chat, file transfer, and so forth.

Components

TsgcWSPServer_WebRTC: Server Protocol WebRTC VCL Component.

Parameters

- **IceServers:** here you can configure turn/stun servers for WebRTC connections. By default uses the following public STUN servers

```
{"iceServers": [{"url": "stun:stun.l.google.com:19302"}]}
```

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this url (you need to define your custom host and port)

```
http://host:port/webrtc.esegece.com.html
```

TsgcWSPServer_WebRTC

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

WebRTC Protocol requires STUN/TURN server, demos use public STUN/TURN servers for testing purposes. In order to put in a production system, a dedicated STUN/TURN server is required.

Registered users can download compiled binaries of **Coturn server for Windows**. Read more about COTURN STUN/TURN.

Properties

- **ICEServers:** define here the ICE Servers you want to use in the WebRTC sessions. Example:

{"iceServers": [{"url": "stun:stun.l.google.com:19302"}]}

- **CloseSessionOnHangup:** by default true, if enabled when a remote peer closes the connection, the other peer is disconnected too. If you want to maintain the other peer connection when the peer disconnects, set this property to false.

Protocol WebRTC Javascript

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/webrtc.esegece.com.js"></script>
```

Open Connection

When a WebSocket connection is opened, the browser requests access to the local camera and microphone, you need to allow access.

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  var socket = new sgcws_webrtc('ws://{%host%}:{%port%}');
</script>
```

Open WebRTC Channel

When a browser has access to local camera and microphone, 'sgcmediastart' event is fired and then you can attempt to connect to another client using webrtc_connect procedure

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  var socket = new sgcws_webrtc('ws://{%host%}:{%port%}');
  socket.on('sgcmediastart', function(event)
  {
    socket.webrtc_connect('custom channel');
  }
</script>
```

Close WebRTC channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  socket.webrtc_disconnect('custom channel');
</script>
```

Protocol WAMP

WAMP is an open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

Technically, WAMP is an officially registered WebSocket subprotocol (runs on top of WebSocket) that uses JSON as message serialization format.

What is RPC?

Remote Procedure Call (RPC) is a messaging pattern involving peers of two roles: client and server.
A server provides methods or procedures to call under well-known endpoints.
A client calls remote methods or procedures by providing the method or procedure endpoint and any arguments for the call.
The server will execute the method or procedure using the supplied arguments to the call and return the result of the call to the client.

What is PubSub?

Publish & Subscribe (PubSub) is a messaging pattern involving peers of three roles: publisher, subscriber and broker.
A publisher sends (publishes) an event by providing a topic (aka channel) as the abstract address, not a specific peer.
A subscriber receives events by first providing topics (aka channels) it is interested in. Subsequently, the subscriber will receive any events published to that topic.
The broker sits between publishers and subscribers and mediates messages publishes to subscribers. A broker will maintain lists of subscribers per topic so it can dispatch newly published events to the appropriate subscribers.
A broker may also dispatch events on its own, for example when the broker also acts as an RPC server and a method executed on the server should trigger a PubSub event.
In summary, PubSub decouples publishers and receivers via an intermediary, the broker.

Components

TsgcWSPServer_WAMP: Server Protocol WAMP VCL Component.

TsgcWSPClient_WAMP: Client Protocol WAMP VCL Component.

Javascript Component: Client Javascript Reference.

Most Common Uses

- **RPC**
  - Simple RPC
  - RPC Progress Results
- **PubSub**
  - Subscribers
  - Publishers

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/wamp.esegece.com.html
```

TsgcWSPServer_WAMP

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

Methods

CallResult: When the execution of the remote procedure finishes successfully, the server responds by sending a message with the result.

- **CallId:** this is the ID generated by client when request a call to a procedure
- **Result:** is the result, can be a number, a JSON object...

CallProgressResult: when an RPC has multiple results, this method is called when still there are more results to send. **Example:** if method has 20 results, from method 1 to 19, CallProgressResult must be called. And the final method, number 20, must be called with CallResult to finish method.

- **CallId:** this is the ID generated by client when request a call to a procedure
- **Result:** is the result, can be a number, a JSON object...

CallError: When the remote procedure call could not be executed, an error or exception occurred during the execution or the execution of the remote procedure finishes unsuccessfully for any other reason, the server responds by sending a message with error details.

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **ErrorURI:** identifies the error.
- **ErrorDesc:** error description.
- **ErrorDetails:** application error details, is optional.

Event: Subscribers receive PubSub events published by subscribers via the EVENT message.

- **TopicURI:** channel name where is subscribed.
- **Event:** message text.

Events

OnCall: event fired when the server receives RPC called by the client

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **ProcUri:** procedure identifier...
- **Arguments:** procedure params, can be an integer, a JSON object, a list...

OnBeforeCancelCall: event fired when the server receives a request to cancel a Call from client.

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **Cancel:** by default is True, which means that Call will be cancelled. If server doesn't want cancel this call, set this parameter to false.

OnPrefix: Procedures and Errors are identified using URIs or CURIEs, this event is triggered when a client sends a new prefix

- **Prefix:** compact URI expression.
- **URI:** full URI.

TsgcWSPClient_WAMP

This is the Client Protocol WAMP Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

Prefix: Procedures and Errors are identified using URIs or CURIEs, the client uses this method to send a new prefix.

- **aPrefix:** compact URI expression.
- **aURI:** full URI.

Subscribe: A client requests access to a valid topicURI (or CURIE from Prefix) to receive events published to the given topicURI. The request is asynchronous, the server will not return an acknowledgement of the subscription.

- **aTopicURI:** channel name.

UnSubscribe: Calling unsubscribe on a topicURI informs the server to stop delivering messages to the client previously subscribed to that topicURI.

- **aTopicURI:** channel name.

Call: sent by the client when requests a Remote Procedure Call (RPC)

- **aCallId:** this is the UUID generated by client
- **aProcURI:** procedure identifier.
- **aArguments:** procedure params, can be an integer, a JSON object, a list...

CancelCall: method called when the client wants to cancel an active Call.

- **aCallId:** this is the UUID generated by client

Publish: The client will send an event to all clients connected to the server who have subscribed to the topicURI.

- **TopicURI:** channel name.
- **Event:** message text.

Events

OnWelcome: is the first server-to-client message sent by a WAMP server

- **SessionId:** is a string that is randomly generated by the server and unique to the specific WAMP session. The sessionId can be used for at least two situations: 1) specifying lists of excluded or eligible clients when publishing event and 2) in the context of performing authentication or authorization.
- **ProtocolVersion:** is an integer that gives the WAMP protocol version the server speaks, currently it MUST be 1.
- **ServerIdent:** is a string the server may use to disclose its version, software, platform or identity.

OnCallError: event fired when the remote procedure call could not be executed, an error or exception occurred during the execution or the execution of the remote procedure finishes unsuccessfully for any other reason, the server responds by sending a message with error details

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **ErrorURI:** identifies the error.
- **ErrorDesc:** error description.

- **ErrorDetails:** application error details, is optional.

OnCallResult: event fired when the execution of the remote procedure finishes successfully, the server responds by sending a message with the result.

- **CallId:** this is the ID generated by client when request a call to a procedure
- **Result:** is the result, can be a number, a JSON object...

OnCallProgressResult: event fired when the execution of the remote procedure is in progress and there are still more pending results.

- **CallId:** this is the ID generated by client when request a call to a procedure
- **Result:** is the result, can be a number, a JSON object...

OnEvent: event fired when the client receives PubSub events published by subscribers via the EVENT message.

- **TopicURI:** channel name to which the client is subscribed.
- **Event:** message text.

Protocol WAMP Javascript

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/wamp.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
</script>
```

Send New Prefix

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.prefix('sgc', 'http://www.esegece.com');
</script>
```

Request RPC (Remote Procedure Call)

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.call('', 'sgc:CallTest', '20')
</script>
```

Subscribe to a TopicURI

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.subscribe('sgc:test')
</script>
```

UnSubscribe from a TopicURI

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.unsubscribe('sgc:test)
</script>
```

Publish message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.publish('sgc:channel', 'Test Message', [], []);
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Show Alert OnCallResult or OnCallError

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('wampcallresult', function(event)
  {
    alert('call result: ' + event.CallId + ' - ' + event.CallResult);
  }
  socket.on('wampcallprogressresult', function(event)
  {
    alert('call progress result: ' + event.CallId + ' - ' + event.CallResult);
  }
  socket.on('wampcallerror', function(event)
  {
    alert('call error: ' + event.CallId + ' - ' + event.ErrorURI + ' - ' + event.ErrorDesc +
          ' - ' + event.ErrorDetails);
  }
</script>
```

Show Alert OnEvent

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
```

```
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('wampevent', function(event)
  {
    alert('call result: ' + event.TopicURI + ' - ' + event.Event);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.state();
</script>
```

WAMP | Subscribers

A subscriber receives events by first providing topics (aka channels) it is interested in. Subsequently, the subscriber will receive any events published to that topic.
To receive events from a topic, the subscriber first has to subscribe to that topic.

WAMP Client

```
void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
  MessageBox.Show(Text);
}

oClient = new TsgcWebSocketClient();
oClient.Host = "127.0.0.1";
oClient.Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP.Client = oClient.
oClientWAMP.OnMessage = OnMessageEvent;
oClient.Active = true;

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

WAMP Server

```
void OnSubscriptionEvent(TsgcWSConnection Connection, string Subscription)
{
  MessageBox.Show("Subscribed: " + Subscription);
}

oServer = new TsgcWebSocketServer();
oServer.Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP.OnSubscription = OnSubscriptionEvent;
oServerWAMP.Server = oServer;
oServerWAMP.Active = true;
```

WAMP | Publishers

A publisher sends (publishes) an event by providing a topic (aka channel) as the abstract address, not a specific peer. Just call Publish method and pass as arguments the name of the topic and the message you want to send. This message will be delivered to all subscribers of this topic. As a note, there is no need to subscribe to a topic to publish messages on that topic.

There is no need to configure anything on server side, because messages are automatically broadcasted to clients when a publish message is received.

WAMP Client

```
oClient = new TsgcWebSocketClient();
oClient.Host = "127.0.0.1";
oClient.Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP.Client = oClient.
oClientWAMP.OnMessage = OnMessageEvent;
oClient.Active = true;

// Publish a message to all subscribers
oClient.Publish("myTopic", "Hello subscribers myTopic");
```

WAMP | Simple RPC

The most common use of the WAMP component is for a client to request a method from the server, and the server sends a response to the client. The client can send only the name of the method and/or can pass some parameters required by server to calculate the result. Server processes requests and if successful sends a response to client with the result. If there is any error, server sends an error response to client.
As you see, there is only One request and One response (successful or not).

Example: server has a method called **GetTime**, so every time a client requests this method, server returns server time.

WAMP Server

```
void OnServerCall(TsgcWSConnection Connection, string CallId, string ProcUri, string Arguments)
{
  if (ProcUri == "GetTime")
  {
    oServerWAMP.CallResult(CallId, DateTime.Now.ToString("yyyyMMdd HH:mm:ss"));
  }
  else
  {
    oServer.WAMP.CallError(CallId, "Unknown method");
  }
}
oServer = new TsgcWebSocketServer();
oServer.Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP.OnCall = OnServerCallEvent();
oServerWAMP.Server = oServer;
oServer.Active = true;
```

WAMP Client

```
void OnCallResultClient(TsgcWSConnection Connection, string CallId, string Result);
{
  MessageBox.Show(Result);
}
void OnCallErrorClient(TsgcWSConnection Connection, string Error)
{
  MessageBox.Show(Error);
}
oClient = new TsgcWebSocketClient();
oClient.Host = "127.0.0.1";
oClient.Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP.OnCallResult = OnCallResultClient;
oClientWAMP.OnCallError = OnCallErrorClient;
oClientWAMP.Client = oClient;
oClient.Active = true;
// After client has connected, request GetTime from server
oClientWAMP.Call("GetTime");
```

WAMP | RPC Progress Results

Sometimes, Remote Procedure Calls require more than one result to finish requests, by default WAMP 1.0 protocol doesn't allow Partial results in a call, this is a feature only for sgcWebSockets library.
The flow is very similar to a simple RPC, but here there are 1 or more partial results before CallResult is called to finish the process.
Basically, a client requests a procedure from the server, and the server can send a result or an error. If it sends a result, this can be the final result or it must send more results later. If it's final result, will call method **CallResult** and the process will be finished. If there are more results to send, will call method **CallProgressResult**.

Example: client requests server a method to receive every second the server time and stop after 20 messages.

WAMP Server

```
void OnServerCall(TsgcWSConnection Connection, string CallId, string ProcUri, string Arguments)
{
  if (ProcUri == "GetProgressiveTime")
  {
    int vNum = Int32.Parse(Arguments);
        for (int i = 1; i = vNum; i++)
        {
          if (i == 20)
          {
        oServerWAMP.CallResult(CallId, FormatDateTime("yyyymmdd hh:nn:ss", Now));
          }
          else
          {
                oServerWAMP.CallProgressiveResult(CallId, FormatDateTime("yyyymmdd hh:nn:ss", Now));
          }
        }
  }
  else
  {
    oServer.WAMP.CallError(CallId, "Unknown method");
  }
}


oServer = new TsgcWebSocketServer();
oServer.Port = 80;
oServerWAMP = new TsgcWSPServer_WAMP();
oServerWAMP.OnCall = OnServerCallEvent();
oServerWAMP.Server = oServer;
oServer.Active = true;
```

WAMP Client

```
void OnCallResultClient(TsgcWSConnection Connection, string CallId, string Result);
{
  MessageBox.Show(Result);
}


void OnCallProgressResultClient(TsgcWSConnection Connection, string CallId, string Result);
{
  MessageBox.Show(Result);
}


void OnCallErrorClient(TsgcWSConnection Connection, string Error)
{
  MessageBox.Show(Error);
}


oClient = new TsgcWebSocketClient();
oClient.Host = "127.0.0.1";
oClient.Port = 80;
oClientWAMP = new TsgcWSPClient_WAMP();
oClientWAMP.OnCallResult = OnCallResultClient;
oClientWAMP.OnCallProgressResult = OnCallProgressResultClient;
oClientWAMP.OnCallError = OnCallErrorClient;
```

```
oClientWAMP.Client = oClient;
oClient.Active = true;
// After client has connected, request GetTime from server
oClientWAMP.Call("GetProgressTime");
```

Protocol WAMP 2

WAMP provides Unified Application Routing in an open WebSocket protocol that works with different languages.

Using WAMP you can build distributed systems out of application components which are loosely coupled and communicate in (soft) real-time.

At its core, WAMP offers two communication patterns for application components to talk to each other:

- Publish & Subscribe (PubSub)
- Remote Procedure Calls (RPC)

WAMP is easy to use, simple to implement and based on modern Web standards: WebSocket, JSON and URIs.

Components

TsgcWSPClient_WAMP2: Client Protocol WAMP2 VCL Component.

TsgcWSPClient_WAMP2

This is the Client Protocol WAMP2 Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Session Methods

- **ABORT:** Both the Router and the Client may abort the opening of a WAMP session by sending an ABORT message.

  Reason MUST be an URI.
  Details MUST be a dictionary that allows you to provide additional, optional closing information (see below).

  No response to an ABORT message is expected.

- **GOODBYE:** A WAMP session starts its lifetime with the Router sending a WELCOME message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a GOODBYE message sent by one Peer and a GOODBYE message sent from the other Peer in response.

  Reason MUST be a URI.
  Details MUST be a dictionary that allows providing additional, optional closing information.

Publish/Subscribe Methods

- **PUBLISH:** When a Publisher requests to publish an event to some topic, it sends a PUBLISH message to a Broker:

  Request is a random, ephemeral ID chosen by the Publisher and used to correlate the Broker's response with the request.
  Options is a dictionary that allows you to provide additional publication request details in an extensible way. This is described further below.
  Topic is the topic published to.
  Arguments is a list of application-level event payload elements. The list may be of zero length.
  ArgumentsKw is an optional dictionary containing application-level event payload, provided as keyword arguments. The dictionary may be empty.

  If the Broker is able to fulfil and allowing the publication, the Broker will send the event to all current Subscribers of the topic of the published event.
  By default, publications are unacknowledged, and the Broker will not respond, whether the publication was successful indeed or not.

- **SUBSCRIBE:** A Subscriber communicates its interest in a topic to a Broker by sending a SUBSCRIBE message:

  Request MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
  Options MUST be a dictionary that allows providing additional subscription request details in an extensible way.
  Topic is the topic the Subscriber wants to subscribe to and MUST be a URI.

- **UNSUBSCRIBE:** When a Subscriber is no longer interested in receiving events for a subscription it sends an UNSUBSCRIBE message

  Request MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
  SUBSCRIBED.Subscription MUST be the ID for the subscription to unsubscribe from, originally handed out by the Broker to the Subscriber.

RPC Methods

- **CALL:** When a Caller wishes to call a remote procedure, it sends a CALL message to a Dealer:

  Request is a random, ephemeral ID chosen by the Caller and used to correlate the Dealer's response with the request.
  Options is a dictionary that allows you to provide additional call request details in an extensible way. This is described further below.
  Procedure is the URI of the procedure to be called.
  Arguments is a list of positional call arguments (each of arbitrary type). The list may be of zero length.
  ArgumentsKw is a dictionary of keyword call arguments (each of arbitrary type). The dictionary may be empty.

- **REGISTERCALL:** A Callee announces the availability of an endpoint implementing a procedure with a Dealer by sending a REGISTER message:

  Request is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
  Options is a dictionary that allows providing additional registration request details in an extensible way. This is described further below.
  Procedure is the procedure the Callee wants to register

- **UNREGISTERCALL:** When a Callee is no longer willing to provide an implementation of the registered procedure, it sends an UNREGISTER message to the Dealer:

  Request is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
  REGISTERED.Registration is the ID for the registration to revoke, originally handed out by the Dealer to the Callee.

- **INVOCATION:** If the Dealer is able to fulfil (mediate) the call and it allows the call, it sends a INVOCATION message to the respective Callee implementing the procedure:

  Request is a random, ephemeral ID chosen by the Dealer and used to correlate the Callee's response with the request.
  REGISTERED.Registration is the registration ID under which the procedure was registered at the Dealer.
  Details is a dictionary that allows you to provide additional invocation request details in an extensible way. This is described further below.
  CALL.Arguments is the original list of positional call arguments as provided by the Caller.
  CALL.ArgumentsKw is the original dictionary of keyword call arguments as provided by the Caller.

- **YIELD:** If the Callee is able to successfully process and finish the execution of the call, it answers by sending a YIELD message to the Dealer:

  INVOCATION.Request is the ID from the original invocation request.
  Options is a dictionary that allows providing additional options.
  Arguments is a list of positional result elements (each of arbitrary type). The list may be of zero length.
  ArgumentsKw is a dictionary of keyword result elements (each of arbitrary type). The dictionary may be empty.

Events

OnWAMPSession: After the underlying transport has been established, the opening of a WAMP session is initiated by the Client sending a `HELLO` message to the Router

- **Realm:** is a string identifying the realm this session should attach to
- **Details:** is a dictionary that allows you to provide additional opening information

OnWAMPWelcome: A Router completes the opening of a WAMP session by sending a WELCOME reply message to the Client.

- **Session:** MUST be a randomly generated ID specific to the WAMP session. This applies for the lifetime of the session.
- **Details:** is a dictionary that allows you to provide additional information regarding the open session.

OnWAMPChallenge: this event is raised when server requires client authenticate against server.

- **Authmethod:** this is the authentication method requested by server, example: ticket.
- **Details:** optional
- **Secret:** here client can set secret key which will be used to authenticate.

Example: Authentication using ticket method.

OnWAMPAbort: Both the Router and the Client may abort the opening of a WAMP session by sending an ABORT message.

- **Reason:** MUST be an URI.
- **Details:** MUST be a dictionary that allows providing additional, optional closing information.

OnWAMPGoodBye: A WAMP session starts its lifetime with the Router sending a `WELCOME` message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a GOODBYE message sent by one Peer and a `GOODBYE` message sent from the other Peer in response.

- **Reason:** MUST be an URI.
- **Details:** MUST be a dictionary that allows you to provide additional, optional closing information.

OnWAMPSubscribed: If the Broker is able to fulfill and allow the subscription, it answers by sending a SUBSCRIBED message to the Subscriber

- **SUBSCRIBE.Request:** MUST be the ID from the original request.
- **Subscription:** MUST be an ID chosen by the Broker for the subscription.

OnWAMPUnSubscribed: Upon successful unsubscription, the Broker sends an UNSUBSCRIBED message to the Subscriber

- **UNSUBSCRIBE.Request:** MUST be the ID from the original request.

OnWAMPPublished: If the Broker is able to fulfill and allowing the publication, and PUBLISH.Options.acknowledge == true, the Broker replies by sending a PUBLISHED message to the Publisher:

- **PUBLISH.Request:** is the ID from the original publication request.
- **Publication:** is a ID chosen by the Broker for the publication.

OnWAMPEvent: When a publication is successful and a Broker dispatches the event, it determines a list of receivers for the event based on Subscribers for the topic published to and, possibly, other information in the event. Note that the Publisher of an event will never receive the published event even if the Publisher is also a Subscriber of the topic published to. The Advanced Profile provides options for more detailed control over publication. When a Subscriber is deemed to be a receiver, the Broker sends the Subscriber an EVENT message.

- **SUBSCRIBED.Subscription:** is the ID for the subscription under which the Subscriber receives the event - the ID for the subscription originally handed out by the Broker to the Subscribe*.
- **PUBLISHED.Publication:** is the ID of the publication of the published event.
- **DETAILS:** is a dictionary that allows the Broker to provide additional event details in an extensible way.
- **PUBLISH.Arguments:** is the application-level event payload that was provided with the original publication request.
- **PUBLISH.ArgumentKw:** is the application-level event payload that was provided with the original publication request.

OnWAMPError: When the request fails, the Broker sends an ERROR

- **METHOD:** is the ID of the Method.
- **REQUEST.ID:** is the ID of the Request.
- **DETAILS:** is a dictionary that allows the Broker to provide additional event details in an extensible way.
- **ERROR:** describes the message error.
- **PUBLISH.Arguments:** is the application-level event payload that was provided with the original publication request.
- **PUBLISH.ArgumentKw:** is the application-level event payload that was provided with the original publication request.

OnWAMPResult: The Dealer will then send a RESULT message to the original Caller:

- **CALL.Request:** is the ID from the original call request.
- **DETAILS:** is a dictionary of additional details.
- **YIELD.Arguments:** is the original list of positional result elements as returned by the Callee.
- **YIELD.ArgumentsKw:** is the original dictionary of keyword result elements as returned by the Callee.

OnWAMPRegistered: If the Dealer is able to fulfill and allowing the registration, it answers by sending a REGISTERED message to the Callee:

- **REGISTER.Request:** is the ID from the original request.
- **Registration:** is an ID chosen by the Dealer for the registration.

OnWAMPUnRegistered: When a Callee is no longer willing to provide an implementation of the registered procedure, it sends an UNREGISTER message to the Dealer:

- **Request:** is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
- **REGISTERED.Registration:** is the ID for the registration to revoke, originally handed out by the Dealer to the Callee.

Protocol Default

This is default sub-protocol implemented using "JSONRPC 2.0" messages, every time you send a message using this protocol, a JSON object is created with the following properties:

jsonrpc: A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".

method: A String containing the name of the method to be invoked. Method names that begin with the word rpc followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else.

params: A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id: An identifier established by the Client that MUST contain a String, Number, or NULL value if included. If it is not included it is assumed to be a notification. The value SHOULD normally not be Null [1] and Numbers SHOULD NOT contain fractional parts [2]

JSON object example:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

Features

- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications. Supports Wildcard characters, so you can subscribe to a hierarchy of channels. Example: if you want to subscribe to all channels which start with 'news', then call Subscribe('news*').
- A messaging transport that is **agnostic** to the content of the payload
- **Acknowledgment** of messages sent.
- Supports **transactional messages** through server local transactions. When the client commits the transaction, the server processes all messages queued. If the client rolls back the transaction, then all messages are deleted.
- Implements **QoS** (Quality of Service) for message delivery.

Components

TsgcWSPClient_sgc: Server Protocol Default VCL Component.

TsgcWSPClient_sgc: Client Protocol Default VCL Component.

Javascript Component: Client Javascript Reference.

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/esegece.com.html
```

TsgcWSPServer_sgc

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

Methods

Subscribe / UnSubscribe: subscribe/unsubscribe to a channel. Supports wildcard characters, so you can subscribe to a hierarchy of channels. Example: if you want to subscribe to all channels which start with 'news', then call Subscribe('news*').

Publish: sends a message to all subscribed clients. Supports wildcard characters, so you can publish to a hierarchy of channels. Example: if you want to send a message to all subscribers to channels which start with 'news', then call Publish('news*').

RPCResult: if a call RPC from the client is successful, the server will respond with this method.

RPCError: if an RPC call from the client has an error, the server will respond with this method.

Broadcast: sends a message to all connected clients, if you need to broadcast a message to selected channels, use Channel argument.

WriteData: sends a message to single or multiple selected clients.

Properties

RPCAuthentication: if enabled, every time a client requests an RPC, method name needs to be authenticated against a username and password.

Methods: is a list of allowed methods. Every time a client sends an RPC first it will search if this method is defined on this list, if it's not in this list, OnRPCAuthentication event will be fired.

Subscriptions: returns a list of active subscriptions.

UseMatchesMasks: if enabled, subscriptions and publish methods accept wildcards, question marks... check MatchesMask Delphi function to see all supported masks.

Events

OnRPCAuthentication: if RPC Authentication is enabled, this event is triggered to define if a client can call this method or not.

OnRPC: fired when the server receives an RPC from a client.

OnNotification: fired every time the server receives a Notification from a client.

OnBeforeSubscription: fired every time before a client subscribes to a custom channel. Allows denying a subscription.

OnSubscription: fired every time a client subscribes to a custom channel.

OnUnSubscription: fired every time a client unsubscribes from a custom channel.

OnRawMessage: this event is triggered before a message is processed by the component.

TsgcWSPClient_sgc

This is the Client Protocol Default Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

Publish: sends a message to all subscribed clients.

RPC: Remote Procedure Call, the client requests a method and the response will be handled in OnRPCResult or OnRPCError events.

Notify: the client sends a notification to a server, this notification doesn't need a response.

Broadcast: sends a message to all connected clients, if you need to broadcast a message to selected channels, use Channel argument.

WriteData: sends a message to a server. If you need to send a message to a custom TsgcWSProtocol_Server_sgc, use "Guid" Argument. If you need to send a message to a single channel, use "Channel" Argument.

Subscribe: subscribe client to a custom channel. If the client is subscribed, OnSubscription event will be fired.

Unsubscribe: unsubscribe client from a custom channel. If the client is unsubscribed, OnUnsubscription event will be fired.

UnsubscribeAll: unsubscribe the client from all subscribed channels. If the client is unsubscribed, OnUnsubscription event will be fired for every channel.

GetSession: requests to server session id, data session is received OnSession Event.

StartTransaction: begins a new transaction.

Commit: server processes all messages queued in a transaction.

RollBack: server deletes all messages queued in a transaction.

Events

OnEvent: this event is fired every time a client receives a message from a custom channel.

OnRPCResult: this event is fired when the client receives a successful response from the server after an RPC is sent.

OnRPCError: this event is fired when the client receives an error response from the server after an RPC is sent.

OnAcknowledgment: this event is triggered when the client receives an acknowledgment from the server that message has been received.

OnRawMessage: this event is fired before a message is processed by the component.

OnSession: this event is fired after a successful connection or after a GetSession request.

Properties

Queue: disabled by default, if True all text/binary messages are not processed and queued until queue is disabled.

QoS: Three "Quality of Service" provided:

Level 0: "At most once", the message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the server either once or not at all.

Level 1: "At least once", the receipt of a message by the server is acknowledged by an ACKNOWLEDGMENT message. If there is an identified failure of either the communications link or the sending device, or the acknowledgement message is not received after a specified period of time, the sender resends the message. The message arrives at the server at least once. A message with QoS level 1 has an ID param in the message.

Level 2: "Exactly once", where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied. If there is an identified failure of either the communications link or the sending device, or the acknowledgement message is not received after a specified period of time, the sender resends the message.

Subscriptions: returns a list of active subscriptions.

TsgcIWWSPClient_sgc

This is the Intraweb Client Protocol Default Component. You need to drop this component in the form and select a TsgcIWWebSocketClient Component using Client Property.

Methods

WriteData: sends a message to a server. If you need to send a message to a custom TsgcWSProtocol_Server_sgc, use "Guid" Argument. If you need to send a message to a single channel, use "Channel" Argument.

Subscribe: subscribe client to a custom channel. If the client is subscribed, OnSubscription event will be fired.

Unsubscribe: unsubscribe client to a custom channel. If client is unsubscribed, OnUnsubscription event will be fired.

Protocol Default Javascript

Default Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and esegece.com.js** files.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Publish Message to test channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'test');
</script>
```

Show Alert with Event Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcevent', function(event)
  {
    alert('channel:' + event.channel + '. message: ' + event.message);
  }
</script>
```

Call RPC

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  var params = {param:10};
  socket.rpc(GUID(), 'test', JSON.stringify(params));
</script>
```

Handle RPC Response

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcrpcresult', function(event)
  {
    alert('result:' + event.result);
  }
  socket.on('sgcrpcerror', function(event)
  {
    alert('error:' + event.code + ' ' + event.message);
  }
</script>
```

Call Notify

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  var params = {param:10};
  socket.notify('test', JSON.stringify(params));
</script>
```

Send Messages in a Transaction

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.starttransaction('sgc:test');
  socket.publish('Message1', 'sgc:test');
  socket.publish('Message2', 'sgc:test');
```

```
  socket.publish('Message3', 'sgc:test');
  socket.commit('sgc:test');
</script>
```

Show Alert OnSubscribe or OnUnSubscribe to a channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcsubscribe', function(event)
  {
    alert('subscribed: ' + event.channel);
  }
  socket.on('sgcunsubscribe', function(event)
  {
    alert('unsubscribed: ' + event.channel);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Get Session

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcsession', function(event)
  {
    alert(event.guid);
  };
  socket.getsession();
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.state();
</script>
```

Set QoS

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.qoslevel1();
  socket.publish('message', 'channel');
</script>
```

Set Queue Level

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.queuelevel2();
  socket.publish('message1', 'channel1');
  socket.publish('message2', 'channel1');
</script>
```

Protocol Files

This protocol allows sending files using binary WebSocket transport. It can handle big files with a low memory usage.

Features

- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications.
- **Acknowledgment** of messages sent.
- Implements **QoS** (Quality of Service) for file delivery.
- Optionally can request **Authorization** for files received.
- **Low memory** usage.

Components

TsgcWSPServer_Files: Server Protocol Files VCL Component.

TsgcWSPClient_Files: Client Protocol Files VCL Component.

Classes

TsgcWSMessageFile: the object which encapsulates file packet information.

Most common uses

- **Send Files**
  - How Send Files To Server
  - How Send Files To Clients
- **Big Files**
  - How Send Big Files

TsgcWSPServer_Files

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

Methods

SendFile: sends a file to a client, you can set the following parameters
- **aSize:** size of every packet in bytes.
- **aData:** user custom data, here you can write any text you think is useful for client.
- **aChannel:** if you only want to send data to all clients subscribed to this channel.
- **aQoS:** type of quality of service.
- **aFileId:** if empty, will be set automatically.

BroadcastFile: sends a file to all connected clients. You can set several parameters:
- **aSize:** size of every packet in bytes.
- **aData:** user custom data, here you can write any text you think is useful for client.
- **aChannel:** if you only want to send data to all clients subscribed to this channel.
- **aExclude:** connection guids separated by a comma, which you don't want to send this file.
- **aInclude:** connection guids separated by a comma, which you want to send this file.
- **aQoS:** type of quality of service.
- **aFileId:** if empty, will be set automatically.

Properties

Files: files properties.

BufferSize: default size of every packet sent, in bytes.

SaveDirectory: the directory where all files will be stored.

QoS: quality of service

Interval: interval to check if a qosLevel2 message has been sent.

Level: level of quality of service.

qosLevel0: the message is sent.

qosLevel1: the message is sent and you get an acknowledgment if the message has been processed.

qosLevel2: the message is sent, you get an acknowledgment if the message has been processed and packets are requested by the receiver.

Timeout: maximum wait time.

ClearReceivedStreamsOnDisconnect: if disabled, when the client reconnects, it tries to resume file download for qosLevel2. Enabled by default.

ClearSentStreamsOnDisconnect: if disabled, when the client reconnects, it tries to resume file upload for qosLevel2. Enabled by default.

Events

OnFileBeforeSent: fired before a file is sent. You can use this event to check file data before it is sent.

OnFileReceived: fired when a file is successfully received.

OnFileReceivedAuthorization: fired to check if a file can be received.

OnFileReceivedError: fired when an error occurs receiving a file.

OnFileReceivedFragment: fired when a fragment file is received. Useful to show progress.

OnFileSent: fired when a file is successfully sent.

OnFileSentAcknowledgment: fired when a fragment is sent and the receiver has processed.

OnFileSentError: fired when an error occurs sending a file.

OnFileSentFragment: fired when a fragment file is sent. Useful to show progress.

TsgcWSPClient_Files

This is the Client Files Protocol Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

SendFile: sends a file to the server, you can set the following parameters
aSize: size of every packet in bytes.
aData: user custom data, here you can write any text you think is useful for the server.
aQoS: type of quality of service.
aFileId: if empty, will be set automatically.

Properties

Files: files properties

BufferSize: default size of every packet sent, in bytes.

SaveDirectory: the directory where all files will be stored.

QoS: quality of service

Interval: interval to check if a qosLevel2 message has been sent.

Level: level of quality of service.

qosLevel0: the message is sent.

qosLevel1: the message is sent and you get an acknowledgment if the message has been processed.

qosLevel2: the message is sent, you get an acknowledgment if the message has been processed and packets are requested by the receiver.

Timeout: maximum wait time.

ClearReceivedStreamsOnDisconnect: if disabled, when the client reconnects, it tries to resume file download for qosLevel2. Enabled by default.

ClearSentStreamsOnDisconnect: if disabled, when the client reconnects, it tries to resume file upload for qosLevel2. Enabled by default.

Events

OnFileBeforeSent: fired before a file is sent. You can use this event to check file data before it is sent.

OnFileReceived: fired when a file is successfully received.

OnFileReceivedAuthorization: fired to check if a file can be received.

OnFileReceivedError: fired when an error occurs receiving a file.

OnFileReceivedFragment: fired when a fragment file is received. Useful to show progress.

OnFileSent: fired when a file is successfully sent.

OnFileSentAcknowledgment: fired when a fragment is sent and the receiver has processed.

OnFileSentError: fired when an error occurs sending a file.

OnFileSentFragment: fired when a fragment file is sent. Useful to show progress.

TsgcWSMessageFile

This object is passed as a parameter every time a file protocol event is raised.

Properties

- BufferSize: default size of the packet.
- Channel: if specified, this file will only be sent to clients subscribed to specific channel.
- Method: internal method.
- FileId: identifier of a file; it is unique for all files received/sent.
- Data: user custom data. Here the user can set whatever text.
- FileName: name of the file.
- FilePosition: file position in bytes.
- FileSize: Total file size in bytes.
- Id: identifier of a packet; it is unique for every packet.
- QoS: quality of service of the message.
- Streaming: for internal use.
- Text: for internal use.

Protocol Files | How Send Files To Server

To send a file to the server, just call the method **SendFile** of Files Protocol and pass the full **FileName** as argument.
The file received by the server will be saved by default in the same directory where the server executable is located or in the Path set in the Files.SaveDirectory property.

```
// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files.Server = oServer;
oServer.Host = "127.0.0.1";
oServer.Port = 8080;


// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient.URL = "ws://127.0.0.1:8080";


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


// ... Start Server
oServer.Active = true;


// ... Connect client and Send File
if oClient.Connect() then
  oClient_Files.SendFile("c:\Documents\yourfile.txt");
```

Protocol Files | How Send Files To Clients

To send a file to a client, just call the method **SendFile** of Files Protocol and pass the **Guid** of the Connection and the full **FileName** as argument. The Guid of the client connection can be captured OnConnect event of Server Protocol Files.
The file received by the client will be saved by default in the same directory where the client executable is located or in the Path set in the Files.SaveDirectory property.

```
// ... capture the guid of the client connection to send later the file
void OnConnectEvent(TsgcWSConnection *Connection)
{
  FGuid = Connection.Guid;
}


// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files.Server = oServer;
oServer_Files.OnConnect += OnConnectEvent;
oServer.Host = "127.0.0.1";
oServer.Port = 8080;


// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient.URL = "ws://127.0.0.1:8080";


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


// ... Start Server
oServer.Active = true;
oClient.Connect();


// ... Send File to the client connected
oServer_Files.SendFile(FGuid, "c:\Documents\yourfile.txt");
```

Protocol Files | How Send Big Files

When you want to send big files to Server or Client, for example a File of some Gigabytes, you can experience some memory problems trying to load the full file. The Protocol Files allows you to send the files in smaller packets that when received by other peer are reassembled in a single file. Just use the **Size** parameter of **SendFile** method to set the Size in Bytes of every single packet.

```
// ... Create Server
TsgcWebSocketServer oServer = new TsgcWebSocketServer();
TsgcWSPServer_Files oServer_Files = new TsgcWSPServer_Files();
oServer_Files.Server = oServer;
oServer.Host = "127.0.0.1";
oServer.Port = 8080;


// ... Create Client
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient.URL = "ws://127.0.0.1:8080";


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


// ... Start Server
oServer.Active = true;


// ... Connect client and Send File in packets of 100000 bytes
if oClient.Connect() then
  oClient_Files.SendFile("c:\Documents\yourfile.txt", 100000, TwsQoS.qosLevel0, "");
```

Protocol Presence

Presence protocol allows you to know who is subscribed to a channel, this makes it easier to create chat applications and know who is online, example: game users, chat rooms, users viewing the same document...

Features

- By default user is **identified by a name**, but this can be **customized** passing more data: email, company, twitter...
- Events to **Authorize** if a **Channel** can be created, if a **member** is allowed...
- Every time a new **member joins** a channel, all members are **notified**.
- **Publish messages** to all channel subscribers.
- **Low memory** usage.

Components

TsgcWSPServer_Presence: Server Protocol Presence VCL Component.

TsgcWSPClient_Presence: Client Protocol Presence VCL Component.

Classes

TsgcWSPresenceMessage: the object which encapsulates presence packet information.

TsgcWSPServer_Presence

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

Methods

All methods are handled internally by the server in response to client requests.

Properties

You must link this component to a **Server** or to a **Broker** if you are using more than one protocol.

EncodeBase64: disabled by default. When enabled, string values are encoded in base64 to avoid problems with JSON encoding.

Acknowledgment: if enabled, every time a server sends a message to client assign an ID to this message and queues in a list. When the client receives a message, if detect it has an ID, it sends an Acknowledgment to the server, which means the client has processed message and server can delete from the queue.

- **Interval:** interval in seconds where server checks if there are messages not processed by client.
- **Timeout:** maximum wait time before the server sends the message again.

Methods

- **Broadcast:** use the Broadcast method to send a message to all connected clients using this protocol or to clients subscribed to a specific channel.

Events

There are several events to handle actions like: a new member request to join a channel, a new channel is created by a member, a member unsubscribes from a channel...

New Member

```
// When a new client connects to a server, first sends member data to the server to request a new member.
// Following events can be called:

// OnBeforeNewMember:
// Server receives a request from the client to join and the server accepts or not this member.
// Use Accept parameter to allow or not this member. By default all members are accepted.

void OnBeforeNewMember(TsgcWSConnection aConnection, TsgcWSPresenceMember aMember, bool &Accept)
{
  if (aMember.Name == "Spam")
  {
    Accept = false;
  }
}

// OnNewMember:
// After a new member is accepted, then this event is called and means this member has join member list.
// You can use aMember. Data property to store objects in memory like database access, session objects...

void OnNewMember(TsgcWSConnection aConnection, TsgcWSPresenceMember aMember)
{

}
```

Subscriptions

```
// When a client has joined as a member, can subscribe to new channels, if a channel not exists,
// the following events can be called:

// OnBeforeNewChannel:
// Server receives a subscription request from the client to join this channel but the channel doesn't exist,
// the server can accept or not to create this channel. Use Accept parameter to allow or not this channel.
// By default, all channels are accepted.

void OnBeforeNewChannelBeforeNewChannel(TsgcWSConnection Connection, TsgcWSPresenceChannel aChannel,
  TsgcWSPresenceMember aMember, bool &Accept)
{
  if (aChannel.Name == "Spam")
  {
    Accept = false;
  }
}

// OnNewChannel: After a new channel is accepted, then this event is called and means a new channel has been crea
// Channel properties can be customized in this event.

void OnNewChannel(TsgcWSConnection Connection, ref TsgcWSPresenceChannel aChannel)
{

}

// If the channel already exists or has been created, the server can accept or no new subscriptions.

// OnBeforeNewChannelMembers:
// Server receives a subscription request from a client to join this channel, the server can accept
// or not a member join. Use Accept parameter to allow or not this member. By default, all members are accepted.

void OnBeforeNewChannelMember(TsgcWSConnection Connection, TsgcWSPresenceChannel aChannel,
  TsgcWSPresenceMember aMember, bool &Accept)
{
  if (aMember.Name == "John")
  {
    Accept = true;
  }
  else if (aMember.Name == "Spam")
  {
    Accept = false;
  }
}

// OnNewChannelMember:
// After a new member is accepted, then this event is called and means a new member has joined the channel.
// All subscribers to this channel, will be notified about new members.

void OnNewChannelMember(TsgcWSConnection Connection, TsgcWSPresenceChannel aChannel,
  TsgcWSPresenceMember aMember)
{

}
```

UnSubscriptions

```
// Every time a member unjoin a channel or disconnects, the server is notified by following events:

// OnRemoveChannelMember:
// Server receives a subscription request from a client to join this channel but the channel doesn't exist,
// the server can accept or not to create this channel. Use Accept parameter to allow or not this channel.
// By default all channels are accepted.

void OnRemoveChannelMember(TsgcWSConnection Connection, TsgcWSPresenceChannel aChannel,
  TsgcWSPresenceMember aMember)
{
  Log("Member: " + aMember.Name + " unjoin channel: " + aChannel.Name);
}

// When a member disconnects, automatically server is notified:

// OnRemoveMember: when the client disconnects from protocol, this event is called and server is notified of
// which never has disconnected.

void OnRemoveMember(TsgcWSConnection Connection, TsgcWSPresenceMember aMember)
{
  Log("Member: " + aMember.Name);
}
Errors
// Every time there is an error, these events are called, example: server has denied a member
// to subscribe to a channel, a member try to subscribe to an already subscribed channel...

//OnErrorMemberChannel: this event is called every time there is an error trying to create a new channel,
// join a new member, subscribe to a channel...

void OnErrorMemberChannel(TsgcWSConnection Connection, TsgcWSPresenceError aError,
  TsgcWSPresenceChannel aChannel, TsgcWSPresenceMember aMember)
{
```

```
  Log("#Error: " + aError.Text);
}

// When a member disconnects, automatically server is notified:

// OnErrorPublishMsg: when a client publish a message and this is denied by the server, this event is raised.

void OnErrorPublishMsg(TsgcWSConnection Connection, TsgcWSPresenceError aError, TsgcWSPresenceMsg aMsg,
  TsgcWSPresenceChannel aChannel, TsgcWSPresenceMember aMember)
{
  Log("#Error: " + aError.Text);
}
```

TsgcWSPresenceMessage

This object encapsulates all internal messages exchanged by the server and client presence protocol.

TsgcWSPresenceMember

ID: internal identifier
Name: member name, provided by the client.
Info: member additional info, provided by the client.
Data: TObject which can be used for server purposes.

TsgcWSPresenceMemberList

Member[i]: member of a list by index
Count: number of members of the list

TsgcWSPresenceChannel

Name: channel name, provided by the client.

TsgcWSPresenceMsg

Text: text message, provided by the client when call Publish method

TsgcWSPresenceError

Code: integer value identifying the error
Text: error description.

TsgcWSPClient_Presence

This is the Client Presence Protocol Component. You need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Properties

EncodeBase64: disabled by default. When enabled, string values are encoded in base64 to avoid problems with JSON encoding.

Presence: member data

- **Name:** member name.
- **Info:** any additional info related to member (example: email, twitter, company...)

Acknowledgment: if enabled, every time a client sends a message to server assign an ID to this message and queues in a list. When the server receives the message, if detect it has an ID, it sends an Acknowledgment to the client, which means the server has processed message and the client can delete from the queue.

- **Interval:** interval in seconds where the client checks if there are messages not processed by server.
- **Timeout:** maximum wait time before the client sends the message again.

Methods

There are several methods to subscribe to a channel, get a list of members...

Connect

When a client connects, the first event called is OnSession, the server sends a session ID to the client, which identifies this client in the server connection list. After OnSession event is called, automatically client sends a request to the server to join as a member, if successful, the OnNewMember event is raised

```
void OnNewMember(TsgcWSConnection aConnection, TsgcWSPresenceMember aMember)
{
  Log("Connected: " + aMember.Name);
}
```

Subscriptions

When a client wants to subscribe to a channel, use the method "Subscribe" and pass the channel name as argument

```
Client.Subscribe("MyChannel");
```

If the client is successfully subscribed, the **OnNewChannelMember** event is called. All members of this channel will be notified using the same event.

```
void OnNewChannelMember(TsgcWSConnection Connection, TsgcWSPresenceChannel aChannel,
  TsgcWSPresenceMember aMember)
```

```
{
  Log("Subscribed: " + aChannel.Name);
}
```

if the server denies the access to a member, the **OnErrorMemberChannel** event is raised.

Unsubscriptions

When a client unsubscribes from a channel, use the method "Unsubscribe" and pass channel name as argument

```
Client.Unsubscribe("MyChannel");
```

If a client is successfully unsubscribed, the **OnRemoveChannelMember** event is called. All of the members of this channel will be notified using the same event.

```
void OnRemoveChannelMember(TsgcWSConnection Connection, TsgcWSPresenceChannel aChannel,
  TsgcWSPresenceMember aMember)
{
  Log("Unsubscribed: " + aChannel.Name);
}
```

If a client can't unsubscribe from a channel, example: because is not subscribed, the **OnErrorMemberChannel** event is raised.

```
void OnErrorMemberChannel(TsgcWSConnection Connection, TsgcWSPresenceError aError,
  TsgcWSPresenceChannel aChannel, TsgcWSPresenceMember aMember)
{
  Log("Error: " + aError.Text);
}
```

When a client disconnects from the server, the event **OnRemoveEvent** is called.

```
void OnRemoveMember(TsgcWSConnection aConnection, TsgcWSPresenceMember aMember)
{
  Log("#RemoveMember: " + aMember.Name);
}
```

Publish

When a client wants to send a message to all members or all subscribers of a channel, use the **Publish** method

```
Client.Publish("Hello All Members");

Client.Publish("Hello All Members of this channel", "MyChannel");
```

If a message is successfully published, the **OnPublishMsg** event is called. All members of this channel will be notified using the same event.

```
void OnPublishMsg(TsgcWSConnection Connection, TsgcWSPresenceMsg aMsg,
  TsgcWSPresenceChannel aChannel, TsgcWSPresenceMember aMember)
{
  Log("#PublishMsg: " + aMsg.Text + " " + aMember.Name);
}
```

if a message can't be published, the **OnErrorPublishMsg** event is raised.

```
void OnErrorPublishMsg(TsgcWSConnection Connection, TsgcWSPresenceError aError,
  TsgcWSPresenceMsg aMsg, TsgcWSPresenceChannel aChannel, TsgcWSPresenceMember aMember)
{
  Log("#Error: " + aError.Text);
}
```

GetMembers

A client can request from the server a list of all members or all members subscribed to a channel. Use the **GetMembers** method

```
Client.GetMembers();

Client.GetMembers("MyChannel");
```

If a message is successfully processed by the server, the **OnGetMembers** event is called

```
void OnGetMembers(TsgcWSConnection Connection, TsgcWSPresenceMemberList aMembers,
  TsgcWSPresenceChannel aChannel)
{
  for (int i = 0; i < aMembers.Count; i++)
  {
    Log("#GetMembers: " + aMembers.Member[i].ID + " " + aMembers.Member[i].Name);
  }
}
```

If there is an error because the member is not allowed or is not subscribed to channel, the **OnErrorMemberChannel** event is raised

```
void OnErrorMemberChannel(TsgcWSConnection Connection, TsgcWSPresenceError aError,
  TsgcWSPresenceChannel aChannel, TsgcWSPresenceMember aMember)
{
  Log("Error: " + aError.Text);
}
```

Invite

A client can invite another member to subscribe to a channel.

```
Client.Invite("MyChannel", "E54541D0F0E5R40F1E00FEEA");
```

When the other member receives the invitation, the **OnChannelInvitation** event is called and member can Accept or not the invitation.

```
void OnChannelInvitation(TsgcWSConnection Connection, TsgcWSPresenceMember aMember,
  TsgcWSPresenceChannel aChannel, ref bool Accept, ref int ErrorCode, ref string ErrorText)
{
  if (aChannel.Name == "MyChannel")
  {
    Accept = true;
  }
  else
  {
    Accept = false;
  }
}
```

The member who sends the invitation, can know if the invitation has been accepted or not using the **OnChannelInvitationResponse** event.

```
void PresenceClientChannelInvitationResponse(TsgcWSConnection Connection, TsgcWSPresenceMember aMember,
  TsgcWSPresenceChannel aChannel, bool Accept, TsgcWSPresenceError aError)
{
  if (Accept)
    DoLog("#invitation_accepted: [To] " + aMember.Name + " [Channel] " + aChannel.Name);
  else
    DoLog("#invitation_cancelled: [To] " + aMember.Name + " [Channel] " + aChannel.Name + " [Error] " + aError.Te
}
```

Protocol Presence Javascript

Presence Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and presence.esegece.com.js** files.

Here you can find available methods, you must replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/presence.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
</script>
```

New Member after connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcsession', function(event)
  {
    socket.newmember(event.id, 'John', 'Additional Info');
  });
</script>
```

Subscribe to Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.subscribe('Topic 1');
</script>
```

Unsubscribe from Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.unsubscribe('Topic 1');
</script>
```

Publish Message to Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'Topic 1');
</script>
```

Receive Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcpublishmsg', function(event)
  {
    console.log('#publishmsg: ' + event.message.text);
  });
</script>
```

Get All Members Connected

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcgetmembers', function(event)
  {
    for (var i  in event.members) {
     console.log(event.members[i].id + ' ' + event.members[i].name);
    }
  });
  socket.getmembers();
</script>
```

Show Alert when Members subscribe/unsubscribe

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcnewmember', function(event)
  {
    alert('#new member: ' + event.member.name);
  }
  );
  socket.on('sgcremovemember', function(event)
  {
    alert('#removed member: ' + event.member.name);
  }
  );
  socket.on('sgcnewchannelmember', function(event)
  {
    alert('#new member: ' + event.member.name + ' in channel: ' + event.channel.name);
  }
  );
  socket.on('sgcremovechannelmember', function(event)
  {
    alert('#remove member: ' + event.member.name + ' from channel: ' + event.channel.name);
  }
  );
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('sgcerrormemberchannel', function(event)
  {
    alert('#error member channel: ' + event.error.text);
  }
  );
  socket.on('sgcerrorpublishmsg', function(event)
  {
    alert('#error publish: ' + event.error.text);
  }
  );
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Protocol E2EE

End-to-End Encryption (E2EE) means that messages are encrypted on the sender device and can be decrypted only on recipient devices. The server routes packets but cannot read plaintext content.

This topic explains the technical flow for:

- **Direct messages (1:1)** between two peers.
- **Group messages** using a sender-key model with membership-based key rotation.

Cryptographic building blocks

- **Identity / key agreement:** Elliptic Curve Diffie-Hellman (ECDH) P-256.
- **Symmetric encryption:** AES-256-GCM.
- **Key derivation:** HKDF-SHA-256 to derive encryption keys from shared secrets.
- **Message authentication:** AEAD authentication tag (provided by AES-GCM).
- **Replay protection:** per-message nonce/IV plus sender sequence counters.

Direct messages (1:1) technical flow

1. **Public key discovery**
   Each peer publishes a public identity key. The server may distribute public keys, but private keys never leave the client device.
2. **Shared secret establishment**
   The sender and recipient perform ECDH (private key + peer public key) and obtain the same shared secret.
3. **Session key derivation**
   HKDF-SHA-256 derives one or more symmetric keys (encryption key, optional header key) from the ECDH output.
4. **Message encryption**
   The plaintext is encrypted with AES-256-GCM using a unique nonce/IV. Output includes ciphertext + authentication tag.
5. **Transport**
   The server forwards encrypted payloads and metadata (for example: sender id, key id, counter, timestamp) without plaintext access.
6. **Recipient decryption**
   The recipient derives the same session key, verifies the authentication tag, and decrypts. Any tampering causes authentication failure.

Group messages technical flow (Sender Keys)

For groups, encrypting each message separately for every member is expensive. A common optimization is a **sender key** design:

- Each sender maintains a **Sender Key State** per group.
- The state contains a symmetric **chain key** and message counter.
- For every outgoing group message, the sender derives a one-time message key from the chain key and advances the chain (hash ratchet).
- The message key encrypts the payload with AES-256-GCM.

How a sender key is distributed

1. When a sender joins a group, it creates a fresh sender key state.
2. The sender key state is shared to each current member over existing 1:1 encrypted sessions (pairwise E2EE).
3. After distribution, normal group payloads use the sender-key fast path (single encryption per message).

Membership changes and sender key rotation

To preserve forward and backward secrecy, group sender keys must rotate on membership events:

- **User joins group:** rotate sender keys so the new member cannot decrypt older history unless explicitly shared.
- **User leaves or is removed:** rotate sender keys immediately so the removed member cannot decrypt new traffic.

- **Periodic rotation:** optional time-based or message-count rotation reduces impact of key compromise.

Typical rotation sequence:

1. Create a new sender key state (new key id, new chain key).
2. Distribute it only to currently authorized members through pairwise E2EE channels.
3. Start encrypting new group messages with the new key id.
4. Accept old key id only during a short transition window, then retire it.

Security properties

- **Confidentiality:** only clients with valid keys can decrypt.
- **Integrity and authenticity:** AEAD verification detects tampering and forged ciphertexts.
- **Server blindness:** relay servers do not hold plaintext keys for message content.
- **Post-membership protection:** rotated keys block former members from future group messages.

Components

TsgcWSPServer_E2EE: Server Protocol E2EE component. It forwards encrypted messages between clients without knowing message contents.

TsgcWSPClient_E2EE: Client Protocol E2EE component. It manages key exchange, encryption and decryption on peer devices.

TsgcWSPServer_E2EE

The E2EE server protocol component routes encrypted direct and group messages between clients while keeping payloads opaque. Drop it on a form, assign a TsgcWebSocketHTTPServer (or other server component) to the **Server** property, and configure the E2EE options. The implementation lives in the unit **sgcWebSocket_Protocol_E2EE_Server**.

Configuration

To use the component:

- Set **Server** to a configured **TsgcWebSocketHTTPServer** instance.
- Configure acknowledgment handling in **E2EE_Options.Ack**.
- Choose how new public keys are handled with **E2EE_Options.PublicKeys.ReceiveNewPublicKey**.
- Set **Server.Active := True** to begin listening.

Properties

Server: references the WebSocket server that hosts the E2EE protocol.

E2EE_Options: options that control server-side E2EE behavior.

- **Ack.RcvDirectMessage:** when enabled, the server sends acknowledgments for direct messages.
- **Ack.RcvGroupMessage:** when enabled, the server sends acknowledgments for group messages.
- **PublicKeys.ReceiveNewPublicKey:** determines what the server does when it receives a new public key (for example, save and broadcast).

How Group Messages Work

The server manages group state and routes encrypted traffic, but never decrypts message content.

1. A client creates a group (**CreateGroup** request).
2. Clients join or leave the group (**JoinGroup** / **LeaveGroup** requests).
3. A client sends an encrypted group payload (**SendGroupMessage** request).
4. The server relays payloads to active group members and emits optional acknowledgments according to **E2EE_Options.Ack**.
5. If requested, a group is removed (**DeleteGroup** request) and clients receive corresponding notifications.

Methods

The server protocol is event-driven. Client operations such as group create/delete/join/leave and group message send are received through the protocol channel and processed automatically by the component.

Events

OnConnect / OnDisconnect: fired when client WebSocket connections are opened or closed.

OnMessage / OnRawMessage: triggered when raw WebSocket data arrives.

OnError / OnException: fired for protocol or runtime errors.

OnE2EEMessageIn: fired when the server receives an E2EE packet from a client (direct/group/membership operations).

OnE2EEMessageOut: fired when the server relays an E2EE packet to destination clients.

Example (Demo Flow)

The following example matches the demo **demos\02.WebSocket_Protocols\12.E2EE**, enabling acknowledgments and automatic public key handling while tracing incoming/outgoing E2EE packets.

```
private void FormCreate(object sender, EventArgs e)
{
    e2eeServer.E2EE_Options.Ack.RcvDirectMessage = true;
    e2eeServer.E2EE_Options.Ack.RcvGroupMessage = true;
    e2eeServer.E2EE_Options.PublicKeys.ReceiveNewPublicKey =
        TE2EEServerNewPubKey.e2eeServerNewPubKeySaveAndBroadcast;
}
private void e2eeServerE2EEMessageIn(object sender, string text)
{
    memoLog.AppendText("<-- " + text + Environment.NewLine);
}
private void e2eeServerE2EEMessageOut(object sender, string text)
{
    memoLog.AppendText("--> " + text + Environment.NewLine);
}
```

TsgcWSPClient_E2EE

The E2EE client protocol component adds end-to-end encryption over a WebSocket connection, including encrypted direct messages and encrypted group messages. Drop the component on a form, assign a TsgcWebSocketClient to the **Client** property, and configure the E2EE options before connecting. The implementation lives in the unit **sgcWebSocket_Protocol_E2EE_Client**.

Configuration

To use the component:

- Set **Client** to a configured **TsgcWebSocketClient** instance.
- Assign a unique user identifier using **E2EE_Options.UserId**.
- Enable message acknowledgments with **E2EE_Options.Ack.RcvDirectMessage** and **E2EE_Options.Ack.RcvGroupMessage** when you want delivery state callbacks.
- Set **Client.Active := True** to open the WebSocket connection.

Properties

Client: references the **TsgcWebSocketClient** that hosts the E2EE protocol.

E2EE_Options: options that define client-side E2EE behavior.

- **UserId:** unique identifier for the local user. This value is used by the server to route direct and group messages.
- **Ack.RcvDirectMessage:** when enabled, the client emits acknowledgments for received direct messages.
- **Ack.RcvGroupMessage:** when enabled, the client emits acknowledgments for received group messages.

How Group Messages Work

Group messaging keeps the same E2EE model as direct messaging: content is encrypted on the sender and decrypted on recipients. The server only coordinates members and relays encrypted payloads.

1. Create a group with **CreateGroup** (or use an existing group).
2. Join the group with **JoinGroup** to receive membership and key context. If you've created the group, you're already in.
3. Send encrypted data to the group with **SendGroupMessage**.
4. Handle membership updates using the group events (join/leave/member join/member leave).
5. Optionally delete the group with **DeleteGroup**.

Methods

SendDirectMessage(ToUserId, Text): sends an encrypted direct message to a remote user.

```
e2ee.SendDirectMessage("USER42", "Hello from E2EE");
```

CreateGroup(Group): creates a new encrypted group.

```
e2ee.CreateGroup("DEV_TEAM");
```

JoinGroup(Group): joins an existing encrypted group.

```
e2ee.JoinGroup("DEV_TEAM");
```

LeaveGroup(Group): leaves a joined group.

```
e2ee.LeaveGroup("DEV_TEAM");
```

DeleteGroup(Group): deletes a group.

```
e2ee.DeleteGroup("DEV_TEAM");
```

SendGroupMessage(Group, Text): sends an encrypted text message to all online members in a group.

```
e2ee.SendGroupMessage("DEV_TEAM", "Build finished");
```

Events

OnConnect / OnDisconnect: fired when the underlying WebSocket connection changes state.

OnError / OnException / OnE2EEError: fired for transport, runtime, or E2EE protocol errors.

OnE2EEMessageText / OnE2EEMessageBinary: fired when a decrypted direct message is received.

OnE2EEGroupMessageText: fired when a decrypted group text message is received.

OnE2EEMessageAck: fired when the server or peer acknowledges a direct/group message.

OnE2EEUserCreated / OnE2EEUserDeleted: fired when users are registered or removed from the E2EE user list.

OnE2EEGroupCreated / OnE2EEGroupDeleted: fired when groups are created or deleted.

OnE2EEGroupJoin / OnE2EEGroupLeave: fired when the local user joins or leaves a group. **OnE2EEGroupJoin** includes the current member list.

OnE2EEGroupMemberJoin / OnE2EEGroupMemberLeave: fired when other users join or leave a group you belong to.

Example (Demo Flow)

The following example follows the demo **demos\02.WebSocket_Protocols\12.E2EE**: configure a user id, create/join a group, and send direct/group messages.

```
private void FormCreate(object sender, EventArgs e)
{
    e2ee.E2EE_Options.UserId = "CLIENT01";
    e2ee.E2EE_Options.Ack.RcvDirectMessage = true;
    e2ee.E2EE_Options.Ack.RcvGroupMessage = true;
}
private void btnCreateJoinClick(object sender, EventArgs e)
{
    e2ee.CreateGroup("TEAM01");
    e2ee.JoinGroup("TEAM01");
}
private void btnSendClick(object sender, EventArgs e)
{
    e2ee.SendDirectMessage("CLIENT02", "Hello direct");
```

```
    e2ee.SendGroupMessage("TEAM01", "Hello group");
}
```

WebSocket APIs

There are several implementations based on WebSockets: finance, message publishing, queues... sgcWebSockets implements the most important APIs based on WebSocket protocol. In order to use an API, just attach API component to the client and all messages will be handled by API component (only one API component can be attached to a client).

Client APIs

API	Description
Binance	is an international multi-language cryptocurrency exchange.
Binance Futures	allows you to connect to Binance Futures WebSocket / REST Market Streams.
Coinbase	Coinbase is a US-based crypto exchange. Trade Bitcoin (BTC), Ethereum (ETH), a Support for WebSocket API and REST API.
SignalR	is a library for ASP.NET developers that makes developing real-time web functionali
SignalRCore	ASP.NET Core SignalR is an open-source library that simplifies adding real-time we
SocketIO	is a JavaScript library for real-time web applications. It enables real-time, bi-directio clients and servers.
Kraken	is a US-based cryptocurrency exchange.
Kraken Futures	allows you to connect to Kraken Futures WebSocket / REST Market data.
Pusher	Pusher is an easy and reliable platform with flexible pub/sub messaging, live user li
FXCM	also known as Forex Capital Markets, is a retail broker for trading on the foreign exc
Bitfinex	Bitfinex is one of the world's largest and most advanced cryptocurrency trading plat Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Classic...
Bitstamp	Bitstamp is one of the world's longest standing crypto exchange, supporting the blo
Huobi	is an international multi-language cryptocurrency exchange.
Cex	is a cryptocurrency exchange and former Bitcoin cloud mining provider.
Cex Plus	CEX.IO Exchange Plus is the ultimate crypto trading platform that features deep liq
Bitmex	is a cryptocurrency exchange and derivative trading platform.
3Commas	It's a crypto trading bot.
Kucoin	is a cryptocurrency exchange that allows you to buy, sell, and store cryptocurrencie DOGE
Kucoin Futures	allows you to connect to Kucoin Futures Servers (WebSocket and REST)
OKX	formerly known as OKEx, is one of the largest crypto spot and derivatives trading ex
XTB	FX and CFD trading, providing access to over +2000 financial markets.
Discord	is one of the most popular communication tools for online gaming and streaming.
Bybit	cryptocurrency exchange and trading platform
OpenAI API	The OpenAI Realtime API enables low-latency, multimodal interactions including sp periences and real-time transcription.
MEXC	Centralized cryptocurrency exchange and trading platform, this component impleme
MEXC Futures	Centralized cryptocurrency exchange and trading platform, this component impleme
Bitget	Cryptocurrency exchange and trading platform supporting Spot and Futures market
Gate.io	Cryptocurrency exchange supporting Spot and Futures trading with WebSocket and
Deribit	Cryptocurrency derivatives exchange offering futures and options trading on Bitcoin
Crypto.com	Cryptocurrency exchange supporting Market and User channels with WebSocket ar
HTX	International cryptocurrency exchange (formerly Huobi) with REST API for market d

WebSocket APIs can be registered at **runtime**, just call Method **RegisterAPI** and pass API component as a parameter.

Other Client APIs

API	Description
Telegram	is a cloud-based instant messaging and voice over IP service. Users can send messa stickers, audio and files of any type.
Whatsapp	is an internationally available American freeware, cross-platform centralized instant m
RCON	is a TCP/IP-based communication protocol which allows console commands to be iss
CryptoHopper	It's a crypto trading bot and portfolio manager.
CryptoRobotics	It's a crypto trading robot.

Server APIs

API	Description
RTCMultiConnection	RTCMultiConnection is a WebRTC JavaScript library for peer-to-peer applic conferencing, file sharing, media streaming etc.)
WebPush	The Web Push protocol allows web applications to send notifications to use open or active.
WebAuthn	FIDO2/WebAuthn server API for passwordless authentication using biometr

API Binance

Binance

Binance is an international multi-language cryptocurrency exchange. It offers some APIs to access Binance data. The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **UserData stream:** subscribed clients get account details. Requires an API key to authenticate and uses WebSocket as protocol.
3. **REST API:** Requires an API Key and Secret to authenticate and uses HTTPs as protocol.
    1. **Market Data**
    2. **Account and Trading Data**
    3. **Wallet**
4. **Futures:** WebSocket Futures Market Data Streams are supported through the Binance Futures Client API.

The client supports **Binance**.us too, the following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **UserData stream:** subscribed clients get account details. Requires an API key to authenticate and uses WebSocket as protocol.
3. **REST API:** clients can request to server market and account data. Requires an API Key and Secret to authenticate and uses HTTPs as protocol.

Properties

Binance API has 2 types of methods: public and private. Public methods can be accessed without authentication, for example: get ticker prices. Some are private and related to user data; those methods require the use of Binance API keys.

- **ApiKey:** you can request a new api key in your binance account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **TestNet:** if enabled it will connect to Binance Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.
- **UserStream:** if enabled the client will receive notifications on Account, Orders or Balance Updates (by default true).
- **BinanceUS:** if enabled, will connect to Binance.us Servers (instead of Binance.com servers which is the default).
- **ListenKeyOnDisconnect:** this property specifies what to do when the client disconnect from Binance servers with an Active ListenKey.
    - **blkodDeleteListenKey:** Delete the Active ListenKey doing an HTTP Request to Binance Servers (this is the default).
    - **blkodClearListenKey:** Doesn't delete the ListenKey from Binance Servers and just clears the value of the field.
    - **blkodDoNothing:** does nothing, so the next time that connects to Binance will try to use the same ListenKey.
- **UseCombinedStreams:** if enabled, will combine streams as follows: {"stream":"<streamName>","data":<rawPayload>} (by default disabled)

Most common uses

- **WebSockets API**
  - **How to Connect to WebSocket API**
  - **How to Subscribe to a WebSocket Channel**
- **REST API**
  - **How to Get Market Data**
  - **How to Use Private REST API**
  - **How to Trade Spot**
  - **Private Requests Time**
  - **Withdraw**

WebSocket Stream API

Base endpoint is wss://stream.binance.com:9443, client can subscribe / unsubscribe from events after a successful connection.
The following Subscription / Unsubscription methods are supported.

Method	Parameters	Description
AggregateTrades	Symbol	push trade information that is aggregated for a single taker order
Trades	Symbol	push raw trade information; each trade has a unique buyer and seller
KLine	Symbol, Interval	push updates to the current klines/candlestick every second, minute, hour...
MiniTicker	Symbol	24hr rolling window mini-ticker statistics. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs.
AllMiniTickers		24hr rolling window mini-ticker statistics for all symbols that changed in an array. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs. Note that only tickers that have changed will be present in the array.
Ticker	Symbol	24hr rolling window ticker statistics for a single symbol. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs.
AllMarketTickers		24hr rolling window ticker statistics for all symbols that changed in an array. These are NOT the statistics of the UTC day, but a 24hr rolling window for the previous 24hrs. Note that only tickers that have changed will be present in the array.
BookTicker	Symbol	Pushes any update to the best bid or ask's price or quantity in real-time for a specified symbol.
AllBookTickers		Pushes any update to the best bid or ask's price or quantity in real-time for all symbols.
PartialBookDepth	Symbol, Depth	Top <levels> bids and asks, pushed every second. Valid <levels> are 5, 10, or 20.
DiffDepth	Symbol	Order book price and quantity depth updates used to locally manage an order book.

After a successful subscription / unsubscription, client receives a message about it, where id is the result of Subscribed / Unsubscribed method.

```
{
  "result": null,
  "id": 1
}
```

User Data Stream API

Requires a valid ApiKey obtained from your binance account, and ApiKey must be set in Binance.ApiKey property of component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
Account Update	Account state is updated with the outboundAccountInfo event.
Balance Update	Balance Update occurs during the following: • Deposits or withdrawals from the account • Transfer of funds between accounts (e.g. Spot to Margin)
Order Update	Orders are updated with the executionReport event.

REST API

The base endpoint is: https://api.binance.com. All endpoints return either a JSON object or array. Data is returned in ascending order. Oldest first, newest last.

Access to the REST API Options, using the property REST_API.BinanceOptions.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
Ping		Test connectivity to the Rest API.
GetServerTime		Test connectivity to the Rest API and get the current server time.
GetExchangeInformation		Current exchange trading rules and symbol information

Market Data EndPoints

Method	Parameters	Description
GetOrderBook	Symbol	Get Order Book.
GetTrades	Symbol	Get recent trades
GetHistoricalTrades	Symbol	Get older trades.
GetAggregateTrades	Symbol	Get compressed, aggregate trades. Trades that fill at the time, from the same order, with the same price will have the quantity aggregated.
GetKLines	Symbol, Interval	Kline/candlestick bars for a symbol. Klines are uniquely identified by their open time.

GetAver-agePrice	Symbol	Current average price for a symbol.
Get24hrTicker	Symbol	24 hour rolling window price change statistics. Careful when accessing this with no symbol.
GetPriceTicker	Symbol	Latest price for a symbol.
GetPriceTickers	Symbols	Latest price for an array of symbols. Example: ["BTCUSDT","BNBUSDT"]
GetBookTicker	Symbol	Best price/qty on the order book for a symbol or symbols.
GetUIKLines	Symbol, Interval	Kline/candlestick bars for a symbol. The response is similar to GetKLines, optimized for presentation of candlestick charts.
GetRollingWin-dowTicker	Symbol, Symbols, WindowSize	Rolling window price change statistics. Note: WindowSize default is 1d if not specified.
GetTrading-DayTicker	Symbol, Symbols, Type	Price change statistics for a trading day.

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

Account Data EndPoints

Method	Parameters	Description
NewOrder	Symbol, Side, Type	Send in a new order.
PlaceMarke-tOrder	Side, Symbol, Quantity	Places a New Market Order
PlaceMar-ketQuoteOrder	Side, Symbol, QuoteOrderQty	Places a New Market Quote Order
PlaceLimitOrder	Side, Symbol, Quantity, Limit-Price	Places a New Limit Order
PlaceStopOrder	Side, Symbol, Quantity, Stop-Price, LimitPrice	Places a New Stop Order
PlaceStop-TrailingOrder	Side, Symbol, Quantity, Trail-ingDelta, LimitPrice	Places a New Stop Trailing Order
PlaceTakeProfi-tOrder	Side, Symbol, Quantity, Stop-Price, LimitPrice	Places a New Take Profit Order
PlaceTakeProf-itTrailingOrder	Side, Symbol, Quantity, Trail-ingDelta, LimitPrice	Places a New Take ProfitTrailing Order
PlaceLimit-MakerOrder	Side, Symbol, Quantity	Places a New Limit Market Order
TestNewOrder	Symbol, Side, Type	Test new order creation and signature/recvWindow long. Creates and validates a new order but does not send it into the matching engine.
QueryOrder	Symbol	Check an order's status.
CancelOrder	Symbol	Cancel an active order. Cancel an active order. Either OrderId or OrigClientOrderId must be sent.
CancelAl-lOpenOrders	Symbol (optional)	
GetOpenOrders		Get all open orders on a symbol. Careful when accessing this with no symbol.
GetAllOrders	Symbol	Get all account orders; active, canceled, or filled.
NewOCO	Symbol, Side, Quantity, Price, StopPrice	Send in a new OCO
CancelOCO	Symbol	Cancel an entire Order List
QueryOCO	Symbol	Retrieves a specific OCO based on provided optional parameters
GetAllOCO		Retrieves all OCO based on provided optional parameters

GetOpenOCO		Get All Open OCO.
GetAccountInformation		Get current account information.
GetAccountTradeList	Symbol	Get trades for a specific account and symbol.
CancelReplaceOrder	Symbol, Side, Type, CancelReplaceMode	Cancels an existing order and places a new order on the same symbol.
NewOrderListOCO	Symbol, Side, Quantity, AboveType, BelowType	Place a new OCO order list.
NewOrderListOTO	Symbol, WorkingType, WorkingSide, WorkingQuantity, WorkingPrice, PendingType, PendingSide, PendingQuantity	Place a new OTO (One-Triggers-the-Other) order list.
NewOrderListOTOCO	Symbol, WorkingType, WorkingSide, WorkingQuantity, WorkingPrice, PendingSide, PendingAboveType, PendingBelowType, PendingQuantity	Place a new OTOCO (One-Triggers-a-One-Cancels-the-Other) order list.
NewSOROrder	Symbol, Side, Type, Quantity	Places an order using Smart Order Routing (SOR).
TestSOROrder	Symbol, Side, Type, Quantity	Test new order using Smart Order Routing (SOR). Creates and validates a new order but does not send it into the matching engine.
GetOrderRateLimitUsage		Displays the user's current order count usage for all intervals.
GetPreventedMatches	Symbol	Displays the list of orders that were expired because of STP (Self Trade Prevention).
GetAllocations	Symbol	Retrieves allocations resulting from SOR order placement.
GetAccountCommission	Symbol	Get current account commission rates.

Convert EndPoints

Method	Parameters	Description
GetAllConvertPairs	FromAsset, ToAsset	Query for all convertible token pairs and the tokens' respective upper/lower limits
GetConvertAssetInfo		Query for supported asset's precision information
SendConvertQuoteRequest	FromAsset, ToAsset	Request a quote for the requested token pairs
AcceptConvertQuote	QuoteId	Accept the offered quote by quote ID.
GetConvertOrderStatus	OrderId or QuoteId	Query order status by order ID.
PlaceConvertLimitOrder	BaseAsset, QuoteAsset, Side, LimitPrice	Enable users to place a limit order. baseAsset or quoteAsset can be determined via exchangeInfo endpoint. Limit price is defined from baseAsset to quoteAsset. Either baseAmount or quoteAmount is used.
CancelConvertLimitOrder	OrderId	Enable users to cancel a limit order
GetConvertLimitOpenOrders		Enable users to query for all existing limit orders
GetConvertTradeHistory	StartTime, EndTime	The max interval between startTime and endTime is 30 days.

Wallet EndPoints

(*wallet endpoints only work with production server, not demo)

Method	Description
GetWalletSystemStatus	Fetch system status.
GetWalletAllCoinsInformation	Get information of coins (available for deposit and withdraw) for user.
GetWalletDailyAccountSnapshot	Type: "SPOT", "MARGIN", "FUTURES" • The query time period must be less then 30 days • Support query within the last one month only • If startTimeand endTime not sent, return records of the last 7 days by default
SetWalletDisableFastWithdrawSwitch	This request will disable fastwithdraw switch under your account. You need to enable "trade" option for the api key which requests this endpoint.
SetWalletEnableFastWithdrawSwitch	This request will enable fastwithdraw switch under your account. You need to enable "trade" option for the api key which requests this endpoint. When Fast Withdraw Switch is on, transferring funds to a Binance account will be done instantly. There is no on-chain transaction, no transaction ID and no withdrawal fee.
WalletWithdraw	Submit a withdraw request.
GetWalletDepositHistory	Fetch deposit history.
GetWalletWithdrawHistory	Fetch Withdraw history.
GetWalletDepositAddress	Fetch deposit address with network.
GetWalletAccountStatus	Fetch account status detail.
GetWalletAccountAPITradingStatus	Fetch account api trading status detail.
GetWalletDustLog	Only return last 100 records Only return records after 2020/12/01
GetWalletAssetsConvertedBNB	
WalletDustTransfer	Convert dust assets to BNB. You need to openEnable Spot & Margin Trading permission for the API Key which requests this endpoint.
GetWalletAssetDividendRecord	Query asset dividend record.
GetWalletAssetDetail	Fetch details of assets supported on Binance.
GetWalletTradeFee	Fetch trade fee
WalletUserUniversalTransfer	You need to enable Permits Universal Transfer option for the API Key which requests this endpoint.MAIN_UMFUTURE Spot account transfer to USDⓈ-M Futures account ENUM of Type: • MAIN_CMFUTURE Spot account transfer to COIN-M Futures account • MAIN_MARGIN Spot account transfer to Margin（cross）account • UMFUTURE_MAIN USDⓈ-M Futures account transfer to Spot account • UMFUTURE_MARGIN USDⓈ-M Futures account transfer to Margin（cross）-account • CMFUTURE_MAIN COIN-M Futures account transfer to Spot account • CMFUTURE_MARGIN COIN-M Futures account transfer to Margin(cross) account • MARGIN_MAIN Margin（cross）account transfer to Spot account • MARGIN_UMFUTURE Margin（cross）account transfer to USDⓈ-M Futures • MARGIN_CMFUTURE Margin（cross）account transfer to COIN-M Futures • ISOLATEDMARGIN_MARGIN Isolated margin account transfer to Margin(cross) account • MARGIN_ISOLATEDMARGIN Margin(cross) account transfer to Isolated margin account • ISOLATEDMARGIN_ISOLATEDMARGIN Isolated margin account transfer to Isolated margin account • MAIN_FUNDING Spot account transfer to Funding account • FUNDING_MAIN Funding account transfer to Spot account • FUNDING_UMFUTURE Funding account transfer to UMFUTURE account • UMFUTURE_FUNDING UMFUTURE account transfer to Funding account

	• MARGIN_FUNDING MARGIN account transfer to Funding account • FUNDING_MARGIN Funding account transfer to Margin account • FUNDING_CMFUTURE Funding account transfer to CMFUTURE account • CMFUTURE_FUNDING CMFUTURE account transfer to Funding account
GetWallet-QueryUserUniver-salTransferHistory	• fromSymbol must be sent when type are ISOLATEDMARGIN_MARGIN and ISOLATEDMARGIN_ISOLATEDMARGIN • toSymbol must be sent when type are MARGIN_ISOLATEDMARGIN and ISOLATEDMARGIN_ISOLATEDMARGIN • Support query within the last 6 months only • If startTimeand endTime not sent, return records of the last 7 days by default
GetWalletFundingWallet	Currently supports querying the following business assets：Binance Pay, Binance Card, Binance Gift Card, Stock Token
GetWalletUserAsset	Get user assets, just for positive data.
GetWalletApiKeyPermis-sion	

Events

Binance Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Binance server.
OnDisconnect
After a disconnection from Binance server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Binance API Component, called **OnBinanceHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket User Stream).

(*) Due to changes in Binance Servers, Indy versions before Rad Studio 10.1, won't be able to connect to Test Servers. This issue doesn't affect to Enterprise Edition or if the Indy version has been upgraded to latest.

Binance | Connect WebSocket API

In order to connect to Binance WebSocket API, just create a new Binance API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
oBinance.Client = oClient;
oClient.Active = true;
```

Binance | Subscribe WebSocket Channel

Binance offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how to subscribe to a Ticker:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
oBinance.Client = oClient;
oBinance.SubscribeTicker("bnbbtc");

void OnMessage(TsgcWSConnection Connection, const string aText)
{
// here you will receive the ticker updates
}
```

Binance | Get Market Data

Binance offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get a snapshot of the ticker BNBBTC, make the following call:

```
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
MessageBox.Show(oBinance.REST_API.GetPriceTicker("BNBBTC"));
```

Binance | Private REST API

The Binance REST API offers public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Binance account and create a new API, you will get the following values:

- ApiKey
- ApiSecret

These fields must be configured in the Binance property of the Binance API client component.
Once configured, you can start to do private requests to the Binance Pro REST API

*Private Requests, require that your local machine has the local time synchronized, if not, the requests will be rejected by Binance server. Check the following article about this, Binance Private Requests Time.

```
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
oBinance.Binance.ApiKey = "<your api key>";
oBinance.Binance.ApiSecret = "<your api secret>";
MessageBox.Show(oBinance.REST_API.GetAccountInformation());
```

Binance | Trade Spot

Binance allows you to trade spot using its REST API.

Configuration

First you must create an **API Key** in your binance account and add privileges to trading with Spot.
Once this is done, you can start spot trading.
First, **set your ApiKey and your ApiSecret** in the Binance Client Component, this will be used to sign the requests sent to Binance server.

Place an Order

To place a new order, just call the method **REST_API.NewOrder** of the Binance Client Component.

Depending on the type of order (market, limit...) the API requires more or fewer fields.

Mandatory Fields

- **Symbol:** the product id symbol, example: BNBBTC
- **Side:** BUY or SELL
- **type:** the order type
  - LIMIT
  - MARKET
  - STOP_LOSS
  - STOP_LOSS_LIMIT
  - TAKE_PROFIT
  - TAKE_PROFIT_LIMIT
  - LIMIT_MAKER

Additional Mandatory Fields based on Type

- **LIMIT:** timeInForce, quantity, price
- **MARKET:** quantity or quoteOrderQty
- **STOP_LOSS / TAKE_PROFIT:** quantity, stopPrice
- **STOP_LOSS_LIMIT / TAKE_PROFIT_LIMIT:** timeInForce, quantity, price, stopPrice
- **LIMIT_MAKER:** quantity, price

When you send an order, there are 2 possibilities:

1. **Successful:** the function NewOrder returns the message sent by binance server.
2. **Error:** the exception is returned in the event OnBinanceHTTPException.

Place Market Order 1 BNBBTC

```
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
oBinance.Binance.ApiKey = "<api key>";
oBinance.Binance.ApiSecret = "<api secret>";
MessageBox.Show(oBinance.REST_API.NewOrder("BNBBTC", "BUY", "MARKET", "", 1));
```

Place Limit Order 1 BNBBTC at 0.009260

```
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
oBinance.Binance.ApiKey = "<api key>";
```

```
oBinance.Binance.ApiSecret = "<api secret>";
MessageBox.Show(oBinance.REST_API.NewOrder("BNBBTC", "BUY", "LIMIT", "GTC", 1, 0, 0.009260));
```

-

Binance | Private Requests Time

When you do a private request to Binance, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 5 seconds with Binance servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

The logic is as follows:

```
if (timestamp < (serverTime + 1000) && (serverTime - timestamp) <= recvWindow) {
  // process request
} else {
  // reject request
}
```

It is recommended to use a small recvWindow of 5000 or less! The max cannot go beyond 60000 milliseconds.

You can check the Binance server time, calling method **GetServerTime**, which will return the time of the Binance server

The **RecvWindow** defaults to **5000**, this value can be increased using the property **REST_API.BinanceOptions.RecvWindow**.

Binance | Withdraw

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

- Coin
- Address
- Amount

```
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
oBinance.Binance.ApiKey = "<your api key>";
oBinance.Binance.ApiSecret = "<your api secret>";
MessageBox.Show(oBinance.REST_API.WalletWithdraw("BTC", "7213fea8e94b4a5593d507237e5a555b", 0.25));
```

API Binance Futures

Binance

Binance is an international multi-language cryptocurrency exchange. It offers some APIs to access Binance data. This component allows you to get Binance Futures WebSocket Market Streams.

https://binance-docs.github.io/apidocs/futures/en
https://binance-docs.github.io/apidocs/delivery/en

Futures Contracts

Binance API has 2 types of methods: public and private. Public methods can be accessed without authentication, example: get ticker prices. Some are private and related to user data; those methods require the use of Binance API keys.

- **ApiKey:** you can request a new api key in your binance account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **TestNet:** if enabled it will connect to Binance Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.
- **UserStream:** if enabled the client will receive notifications on Account, Orders or Balance Updates (by default true).
- **ListenKeyOnDisconnect:** this property specifies what to do when the client disconnect from Binance servers with an Active ListenKey.
    - **blkodDeleteListenKey:** Delete the Active ListenKey doing an HTTP Request to Binance Servers (this is the default).
    - **blkodClearListenKey:** Doesn't delete the ListenKey from Binance Servers and just clear the value of the field.
    - **blkodDoNothing:** does nothing, so the next time that connects to Binance will try to use the same ListenKey.
- **UseCombinedStreams:** if enabled, will combine streams as follows: {"stream":"<streamName>","data":<rawPayload>} (by default disabled)

Client can connect to USDT or COIN Binance Futures, set which contract you want to trade using **FuturesContracts** property:

- **bfcUSDT:** connects to USD-M Futures API.
- **bfcCOIN:** connects to COIN-M Futures API.

Client can connect to Production or Demo Binance accounts. If **TestNet** property is enabled, it will connect to Demo account, otherwise will connect to production Binance Servers.

WebSocket Stream API

Client can subscribe / unsubscribe from events after a successful connection.
The following Subscription / Unsubscription methods are supported.

Method	Parameters	Description
AggregateTrades	Symbol	The Aggregate Trade Streams push trade information that is aggregated for a single taker order every 100 milliseconds.

MarkPrice	Symbol, UpdateSpeed	Mark price and funding rate for a single symbol pushed every 3 seconds or every second.
AllMarkPrice	UpdateSpeed	Mark price and funding rate for all symbols pushed every 3 seconds or every second.
KLine	Symbol, Interval	The Kline/Candlestick Stream push updates to the current klines/candlestick every 250 milliseconds (if existing).
MiniTicker	Symbol	24hr rolling window mini-ticker statistics for a single symbol. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before.
AllMiniTicker		24hr rolling window mini-ticker statistics for all symbols. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before. Note that only tickers that have changed will be present in the array.
Ticker	Symbol	24hr rolling window ticker statistics for a single symbol. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before.
AllMarketTickers		24hr rolling window ticker statistics for all symbols. These are NOT the statistics of the UTC day, but a 24hr rolling window from requestTime to 24hrs before. Note that only tickers that have changed will be present in the array.
BookTicker	Symbol	Pushes any update to the best bid or ask's price or quantity in real-time for a specified symbol.
AllBookTickers		Pushes any update to the best bid or ask's price or quantity in real-time for all symbols.
LiquidationOrders	Symbol	The Liquidation Order Streams push force liquidation order information for specific symbol
AllLiquidationOrders		The All Liquidation Order Streams push force liquidation order information for all symbols in the market.
PartialBookDepth	Symbol, Depth	Top bids and asks, Valid are 5, 10, or 20.
DiffDepth	Symbol	Bids and asks, pushed every 250 milliseconds, 500 milliseconds, 100 milliseconds or in real time(if existing)

After a successful subcription / unsubscription, client receives a message about it, where id is the result of Subscribed / Unsubscribed method.

```
{
  "result": null,
  "id": 1
}
```

User Data Stream API

Requires a valid ApiKey obtained from your binance account, and ApiKey must be set in Binance.ApiKey property of component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
Margin Call	When the user's position risk ratio is too high, this stream will be pushed. This message is only used as risk guidance information and is not recommended for investment strategies. In the case of a highly volatile market, there may be the possibility that the user's position has been liquidated at the same time when this stream is pushed out.
Balance and Position Update	Balance Update occurs during the following: • When balance or position get updated, this event will be pushed. • When "FUNDING FEE" changes to the user's balance. •
Order Update	When new order created, order status changed will push such event.

REST API

All endpoints return either a JSON object or array. Data is returned in ascending order. Oldest first, newest last.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
Ping		Test connectivity to the Rest API.
GetServerTime		Test connectivity to the Rest API and get the current server time.
GetExchangeInformation		Current exchange trading rules and symbol information

Market Data EndPoints

Method	Parameters	Description
GetOrderBook	Symbol	Get Order Book.
GetTrades	Symbol	Get recent trades
GetHistoricalTrades	Symbol	Get older trades.
GetAggregateTrades	Symbol	Get compressed, aggregate trades. Trades that fill at the time, from the same order, with the same price will have the quantity aggregated.
GetKLines	Symbol, Interval	Kline/candlestick bars for a symbol. Klines are uniquely identified by their open time.
Get24hrTicker	Symbol	24 hour rolling window price change statistics. Careful when accessing this with no symbol.
GetPriceTicker	Symbol	Latest price for a symbol or symbols.
GetBookTicker	Symbol	Best price/qty on the order book for a symbol or symbols.
GetMarkPrice	Symbol	Mark Price and Funding Rate
GetFundingRateHistory	Symbol	
GetOpenInterest	Symbol	Get present open interest of a specific symbol.
GetOpenInterestStatistics	Symbol, Period	
GetTopTraderAccountRatio	Symbol, Period	
GetTopTraderPositionRatio	Symbol, Period	
GetGlobalAccountRatio	Symbol, Period	
GetTakerVolume	Symbol, Period	
GetContinuousKLines	Pair, ContractType, Interval	Kline/candlestick bars for a specific contract type.
GetIndexPriceKLines	Pair, Interval	Kline/candlestick bars for the index price of a pair.

GetMarkPriceK-Lines	Symbol, Interval	Kline/candlestick bars for the mark price of a symbol.
GetPremiumIn-dexKLines	Symbol, Interval	Premium index kline bars of a symbol.
GetFundingInfo		Get funding rate info for all symbols.
GetPriceTick-erV2	Symbol	Latest price for a symbol or symbols (V2).
GetIndexInfo	Symbol	Get index info.
GetAssetIndex	Symbol	Get asset index for multi-assets mode.
GetConstituents	Symbol	Get index constituents.
GetDeliv-eryPrice	Pair	Get delivery price.
GetBasis	Pair, ContractType, Period	Get basis data.

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

Account and Trades EndPoints

Method	Parameters	Description
ChangePosi-tionMode	DualPosition	Change user's position mode (Hedge Mode or One-way Mode) on EVERY symbol
GetCurrentPosi-tionMode		Get user's position mode (Hedge Mode or One-way Mode) on EVERY symbol
NewOrder	Symbol, Side, PositionSide, Type	Send in a new order.
PlaceMarke-tOrder	Side, Symbol, Quantity	
PlaceLimitOrder	Side, Symbol, Quantity, Limit-Price	
PlaceStopOrder	Side, Symbol, Quantity, Stop-Price, LimitPrice	
PlaceTrail-ingStopOrder	Side, Symbol, Quantity, aActi-vationPrice, aCallbackRate	
QueryOrder	Symbol	Check an order's status.
CancelOrder	Symbol	Cancel an active order. Either OrderId or OrigClientOrderId must be sent.
CancelAl-lOpenOrders	Symbol	
AutoCancelAl-lOpenOrders	Symbol, CountDownTimer	Cancel all open orders of the specified symbol at the end of the specified countdown.
QueryCurren-tOpenOrder	Symbol	
GetOpenOrders	Symbol	Get all open orders on a symbol. Careful when accessing this with no symbol.
GetAllOrders	Symbol	Get all account orders; active, canceled, or filled.
GetAccountBal-ance		
GetAccountIn-formation		Get current account information.
ChangeInitial-Leverage	Symbol, Leverage	Change user's initial leverage of specific symbol market.
ChangeMargin-Type	Symbol, MarginType	
ModifyIsolated-PositionMargin	Symbol, Amount, Type	

GetPositionMarginChangeHistory	Symbol	
GetPositionInformation	Symbol	
GetAccountTradeList	Symbol	
GetIncomeHistory	Symbol	
GetNotionalLeverageBracket	Symbol	
TestNewOrder	Symbol, Side, PositionSide, Type	Test new order creation and signature/recvWindow long. Creates and validates a new order but does not send it into the matching engine.
ModifyOrder	Symbol	Modify an existing order.
NewBatchOrders	BatchOrders	Place multiple orders.
ModifyBatchOrders	BatchOrders	Modify multiple orders.
CancelBatchOrders	Symbol	Cancel multiple orders.
GetOrderAmendment	Symbol	Get order modification history.
CountdownCancelAll	Symbol, CountdownTime	Cancel all open orders of the specified symbol at the end of the specified countdown.
GetForceOrders	Symbol	Get user's force liquidation orders.
GetADLQuantile	Symbol	Get ADL quantile estimation for positions.
GetAccountBalanceV3		Get futures account balance (V3).
GetAccountInformationV3		Get current account information (V3).
GetPositionInformationV3	Symbol	Get current position information (V3).
GetCommissionRate	Symbol	Get user commission rate.
GetAccountConfig		Get current account configuration.
GetSymbolConfig	Symbol	Get symbol configuration.
GetOrderRateLimit		Get user's order rate limit.
GetApiTradingStatus	Symbol	Get API trading quantitative rules indicators.
ChangeMultiAssetsMode	MultiAssetsMargin	Change user's multi-assets mode. Multi-Assets Mode: true; Single-Asset Mode: false.
GetMultiAssetsMode		Get user's current multi-assets mode.
SetFeeBurn	FeeBurn	Change user's BNB fee burn status.
GetFeeBurn		Get user's BNB fee burn status.
CreateListenKey		Start a new user data stream. The stream will close after 60 minutes unless a keepalive is sent.
KeepAliveListenKey		Keepalive a user data stream to prevent a timeout.
CloseListenKey		Close a user data stream.

Events

Binance Futures Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Binance server.
OnDisconnect
After a disconnection from Binance server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Binance API Component, called **OnBinanceHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket User Stream).

(*) Due to changes in Binance Servers, Indy versions before Rad Studio 10.1, won't be able to connect to Test Servers. This issue doesn't affect to Enterprise Edition or if the Indy version has been upgraded to the latest.

API Binance Futures | Trade

Binance allows you to trade futures using its REST API.

Configuration

First you must create an **API Key** in your binance account and add privileges to trading with Futures.
Once this is done, you can start to trading with futures.
First you must select if you want to trade with **USDT** or **COIN** futures, there is a property called FuturesContracts where you can set which future contract you want to trade
Then, **set your ApiKey and your ApiSecret** in the Binance Futures Client Component, this will be used to sign the requests sent to Binance server.

Place an Order

To place a new order, just call to method **REST_API.NewOrder** of Binance Futures Client Component.

Depending on the type of the order (market, limit...) the API requires more or less fields.

Mandatory Fields

- **Symbol:** the product id symbol, example: BTCUSD_210326
- **Side:** BUY or SELL
- **type:** the order type
  - LIMIT
  - MARKET
  - STOP
  - TAKE_PROFIT
  - STOP_MARKET
  - TAKE_PROFIT_MARKET
  - TRAILING_STOP_MARKET

Additional Mandatory Fields based on Type

- **LIMIT:** timeInForce, quantity, price
- **MARKET:** quantity
- **STOP/TAKE_PROFIT:** quantity, price, stopPrice
- **STOP_MARKET/TAKE_PROFIT_MARKET:** stopPrice
- **TRAILING_STOP_MARKET:** callbackRate

When you send an order, there are 2 possibilities:

1. **Successful:** the function NewOrder returns the message sent by binance server.
2. **Error:** the exception is returned in the event OnBinanceHTTPException.

API SocketIO

SocketIO

Socket.IO is a JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical API. Like Node.js, it is event-driven.

Messages Types

0: open (Sent from the server when a new transport is opened (recheck))

1: close (Request the close of this transport but does not shut down the connection itself.)

2: ping (Sent by the client. The server should answer with a pong packet containing the same data)
example
client sends: 2probe
server sends: 3probe

3: pong (Sent by the server to respond to ping packets.)

4: string message (actual message, client and server should call their callbacks with the data.)
example:
42/chat,[“join”,”{room:1}"]
4 is the message packet type in the engine.io protocol
2 is the EVENT type in the socket.io protocol
/chat is the data which is processed by socket.io
socket.io will fire the “join” event
will pass "room: 1" data. It is possible to omit namespace only when it is /.

5: upgrade (Before engine.io switches a transport, it tests, if server and client can communicate over this transport. If this test succeeds, the client sends an upgrade packets which requests the server to flush its cache on the old transport and switch to the new transport.)

6: noop (A noop packet. Used primarily to force a poll cycle when an incoming WebSocket connection is received.)

Properties

API: specifies SocketIO version:

ioAPI0: supports socket.io 0.* servers (selected by default)

ioAPI1: supports socket.io 1.* servers

ioAPI2: supports socket.io 2.* servers

ioAPI3: supports socket.io 3.* servers

ioAPI4: supports socket.io 4.* servers

Base64: if enabled, binary messages are received as base64.

HandShakeCustomURL: allows customizing the URL to get socket.io session.

HandShakeTimestamp: only enable if you want to send timestamp as a parameter when a new session is requested (enable this property if you try to access a gevent-socketio python server).

HandShakeAuthToken: if the server requires a token for authentication, set here the authentication token.

Namespace: allows setting a namespace when connects to the server.

Polling: disabling this property, client will connect directly to server using websocket as transport.

Parameters: allows you to set connection parameters.

EncodeParameters: if enabled, parameters are encoded.

Methods

Use the WriteData method to send messages to the socket.io server (following the Message Types section).
1. Call the method "add user" with one parameter using John as the user name.

```
WriteData("42[\"add user\", \"John\"]");
```

Events

OnHTTPRequest

Before a new websocket connection is established, the socket.io server requires the client to open a new HTTP connection to get a new session id. In some cases, the socket.io server requires authentication using HTTP headers. You can use this event to add custom HTTP headers, like Basic authorization or Bearer token authentication.

OnAfterConnect

This event is called after the socket.io connection is successful and the client can send messages to the server. Here you can subscribe to namespaces, for example.

API Coinbase

Coinbase

APIs supported

- WebSockets API: connect to a public websocket server and provides real-time market data updates.
- REST API: The REST API has endpoints for account and order management as well as public market data.

Most common uses

- **WebSockets API**
  - **How to Connect to WebSocket API**
  - **How to Subscribe to a WebSocket Channel**
- **REST API**
  - **How to Get Market Data**
  - **How to Use Private REST API**
  - **How to Place Orders**
  - **How to Use SandBox Account**
  - **Private Requests Time**

WebSockets API

The WebSocket feed is publicly available and provides real-time market data updates for orders and trades. Two endpoints are supported in production:

- **Market Data** is the feed that provides updates for both orders and trades. Most channels are now available without authentication.
- **User Order Data** provides updates for the orders of the user.

You can subscribe to the following channels:

Method	Arguments	Description
SubscribeHeart-Beat		Real-time server pings to keep all connections open
SubscribeStatus	**aProductId:** id of the product	Sends all products and currencies on a preset interval
SubscribeCandles	**aProductId:** id of the product	Real-time updates on product candles
SubscribeTicker	**aProductId:** id of the product	Real-time price updates every time a match happens
SubscribeTicker-Batch	**aProductId:** id of the product	Real-time price updates every 5000 milli-seconds
SubscribeLevel2	**aProductId:** id of the product	All updates and easiest way to keep order book snapshot
SubscribeMarket-Trades	**aProductId:** id of the product	Real-time updates every time a market trade happens
SubscribeUser	**aProductId:** id of the product	Only sends messages that include the authenticated user
SubscribeFutures-BalanceSummary		Real-time updates every time a user's futures balance changes

The User and FuturesBalanceSummary channels require authentication, so first request your API keys in your Coinbase account and then set the values in the property Coinbase of the component:

- ApiKey
- ApiSecret

Authentication will result in a couple of benefits:

1. Messages where you're one of the parties are expanded and have more useful fields
2. You will receive private messages, such as lifecycle information about stop orders you placed

REST API

Private Endpoints

Private endpoints are available for order management, and account management.
Before being able to sign any requests, you must create an API key via the Coinbase Pro website. The API key will be scoped to a specific profile. Upon creating a key you will have 3 pieces of information which you must remember:

- Key
- Secret
- Passphrase

The Key and Secret will be randomly generated and provided by Coinbase Pro; the Passphrase will be provided by you to further secure your API access. Coinbase Pro stores the salted hash of your passphrase for verification, but cannot recover the passphrase if you forget it.

You can restrict the functionality of API keys. Before creating the key, you must choose what permissions you would like the key to have. The permissions are:

- View - Allows a key read permissions. This includes all GET endpoints.
- Transfer - Allows a key to transfer currency on behalf of an account, including deposits and withdraws. Enable with caution - API key transfers WILL BYPASS two-factor authentication.
- Trade - Allows a key to enter orders, as well as retrieve trade data. This includes POST /orders and several GET endpoints.

Accounts

Method	Arguments	Description
ListAccounts		Get a list of trading accounts from the profile of the API key.
GetAccount	**aAccountId:** id of the account	Information for a single account. Use this endpoint when you know the account_id. API key must belong to the same profile as the account.

Orders

Method	Arguments	Description
PlaceNewOrder	**aOrder:** class that contains all possible fields of an order	Places a new order. Use only if you need to access to advanced order options.
PlaceMarketOrder	**aSide:** buy or sell **aProductId:** id of the product **aQuoteSize:** The amount of the second Asset in the Trading Pair. **aBaseSize:** The amount of the first Asset in the Trading Pair	Places a new Market order.

	aClient_oid: Order ID selected by you to identify your order	
PlaceLimitOrder	**aSide:** buy or sell **aProductId:** id of the product **aQuoteSize:** The amount of the second Asset in the Trading Pair. **aBaseSize:** The amount of the first Asset in the Trading Pair **aLimitPrice:** price limit **Client_oid:** Order ID selected by you to identify your order	Places a new Limit order.
PlaceStopOrder	**aSide:** buy or sell **ProductId:** id of the product **aBaseSize:** The amount of the first Asset in the Trading Pair **StopPrice:** price of the stop **aLimitPrice:** price limit **aStopDirection:** loss or entry **Client_oid:** Order ID selected by you to identify your order	Places a new Stop Order
CancelOrder	**aOrderId:** id of the order	Cancel a previously placed order. Order must belong to the profile that the API key belongs to.
EditOrder	**aOrderId:** id of the order **aPrice:** price **aSize:** Amount	Edit an order with a specified new size, or new price
EditOrderPreview	**aOrderId:** id of the order **aPrice:** price **aSize:** Amount	Preview an edit order request with a specified new size, or new price.
ListOrders		Get a list of orders filtered by optional query parameters (product_id, order_status, etc).
GetOrder	**aOrderId:** id of the order	Get a single order by order ID.
PreviewOrder		Preview an order.
ClosePosition	**aOrderId:** id of the order **aProductId:** id of the product **aSize:** amount	Places an order to close any open positions for a specified product_id.

Market Data

Method	Arguments	Description
GetPublicProducts		Get a list of the available currency pairs for trading.
GetPublicProduct	**aProductId:** id of the product	Get information on a single product by product ID.
GetPublicProductBook	**aProductId:** id of the product	Get a list of bids/asks for a single product. The amount of detail shown can be customized with the limit parameter.
GetPublicProductCandles	**aProductId:** id of the product **aStart:** start of the time interval **aEnd:** end of the time interval **aGranularity:** The timeframe each candle represents.	Get rates for a single product by product ID, grouped in buckets.
GetTrades	**aProductId:** id of the product	Get snapshot information by product ID about the last trades (ticks) and best bid/ask.
GetTime		Get the current time from the Coinbase Advanced API.

Fills

Method	Arguments	Description
GetFillsByOrderId		Get a list of fills filtered by order id
GetFillsByProductId		Get a list of fills filtered by product id
GetFillsByTradeId		Get a list of fills filtered by trade id

Convert

Method	Arguments	Description
CreateConvertQuote		Create a convert quote between currencies.
CommitConvertTrade		Commit a convert trade.
GetConvertTrade		Get convert trade details.

Fees

Method	Arguments	Description
GetTransactionSummary		Get transaction fee summary.

Products (Authenticated)

Method	Arguments	Description
ListProducts		List available products.
GetProduct	**aProductId:** id of the product	Get a specific product.
GetProductBook	**aProductId:** id of the product	Get product order book.
GetProductCandles	**aProductId:** id of the product	Get product OHLCV candles.
GetMarketTrades	**aProductId:** id of the product	Get recent market trades.
GetBestBidAsk		Get best bid/ask prices.

Portfolios

Method	Arguments	Description
ListPortfolios		List all portfolios.
CreatePortfolio		Create a new portfolio.
DeletePortfolio		Delete a portfolio.
GetPortfolioBreakdown		Get portfolio breakdown details.
MovePortfolioFunds		Move funds between portfolios.

Perpetuals

Method	Arguments	Description
GetPerpetualsPortfolioSummary		Get perpetuals portfolio summary.
ListPerpetualsPositions		List perpetuals positions.
GetPerpetualsPosition		Get a specific perpetuals position.

Coinbase | Connect WebSocket API

In order to connect to Coinbase WebSocket API, just create a new Coinbase API client and attach to TsgcWebSocketClient. See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase.Client = oClient;
oClient.Active = true;
```

Coinbase | Subscribe WebSocket Channel

Coinbase offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how subscribe to a Ticker:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase.Client = oClient;
oCoinbase.SubscribeTicker("ETH-USD");

void OnCoinbaseMessage(TObject Sender, string aType, string aRawMessage)
{
// here you will receive the ticker updates
}
```

Coinbase Pro | Get Market Data

Coinbase offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get an snapshot of the book BTC-USD, do the following call

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
MessageBox.Show(oCoinbase.REST_API.GetPublicProductBook("BTC-USD"));
```

Coinbase Pro | Private REST API

The Coinbase REST API offers public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Coinbase account and create a new API, you will get the following values:

- ApiKey
- ApiSecret

These fields must be configured in the Coinbase property of the Coinbase API client component.
Once configured, you can start to do private requests to the Coinbase REST API

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase.Coinbase.ApiKey = "<your api key>";
oCoinbase.Coinbase.ApiSecret = "<your api secret>";
MessageBox.Show(oCoinbase.REST_API.ListAccounts);
```

Coinbase Pro | Private Requests Time

When you do a private request to Coinbase, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 30 seconds with Coinbase servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

You can check the Coinbase Pro server time, calling method **GetTime**, which will return the time of the Coinbase Pro server

Coinbase Pro | Place Orders

In order to place new orders in Coinbase, you first need your API keys to access your private data. Check the following article: How to Use Private REST API.

Once you have configured your API keys, you can start to place orders

Market Order

Place a new Market Order, buy 0.002 contracts of BTC-USD

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase.Coinbase.ApiKey = "your api key";
oCoinbase.Coinbase.ApiSecret = "your api secret";
oCoinbase.Coinbase.ApiPassphrase = "your passphrase";
MessageBox.Show(oCoinbase.REST_API.PlaceMarketOrder(coisBuy, "BTC-USD", 0.002

, 0

));
```

Limit Order

Place a new Limit Order, buy 0.002 contracts of BTC-USD at price limit of 10000

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase.Coinbase.ApiKey = "your api key";
oCoinbase.Coinbase.ApiSecret = "your api secret";
oCoinbase.Coinbase.ApiPassphrase = "your passphrase";
MessageBox.Show(oCoinbase.REST_API.PlaceLimitOrder(coisBuy, "BTC-USD", 0.002,

0,

10000));
```

Coinbase Pro SandBox Account

Coinbase allows you to use a SandBox account where you can trade without real funds. This account requires creating API keys different from the production account.
To use the SandBox account, just set **Coinbase.SandBox** property to **true**, before doing any request to the REST API.

```
TsgcWSAPI_Coinbase oCoinbase = new TsgcWSAPI_Coinbase();
oCoinbase.Coinbase.ApiKey = "your api key";
oCoinbase.Coinbase.ApiSecret = "your api secret";
oCoinbase.Coinbase.SandBox = true;
MessageBox.Show(oCoinbase.REST_API.ListAccounts);
```

API SignalRCore

SignalRCore

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

- Apps that require high-frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
- Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
- Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
- Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

SignalRCore sgcWebSockets component uses WebSocket as transport to connect to a SignalRCore server, if this transport is not supported, an error will be raised.

Hubs

SignalRCore uses hubs to communicate between clients and servers. SignalRCore provides 2 hub protocols: text protocol based on JSON and binary protocol based on MessagePack. The sgcWebSockets component only implements JSON text protocol to communicate with SignalRCore servers.
To configure which Hub client will use, just set in **SignalRCore/Hub** property the name of the Hub before the client connects to the server.

Connection

When a client opens a new connection to the server, sends a request message which contains format protocol and version. sgcWebSockets always sends format protocol as JSON. The server will reply with an error if the protocol is not supported by the server, this error can be handled using **OnSignalRCoreError** event, and if the connection is successful, **OnSignalRCoreConnect** event will be called.

When a client connects to a SignalRCore server, it can send a ConnectionId which identifies client between sessions, so if you get a disconnection client can reconnect to server passing same prior connection id. In order to get a new connection id, just connect normally to the server and you can know ConnectionId using **OnBeforeConnectEvent**. If you want to reconnect to the server and pass a prior connection id, use **ReConnect** method and pass **ConnectionId** as a parameter.

SignalRCore Protocol

The SignalR Protocol is a protocol for two-way RPC over any Message-based transport. Either party in the connection may invoke procedures on the other party, and procedures can return zero or more results or an error. Example: the client can request a method from the server and the server can request a method from the client. The following messages are exchanged between server and clients:

- **HandshakeRequest:** the client sends to the server to agree on the message format.
- **HandshakeResponse:** server replies to the client an acknowledgement of the previous HandshakeRequest message. Contains an error if the handshake failed.
- **Close:** called by client or server when a connection is closed. Contains an error if the connection was closed because of an error.
- **Invocation:** client or server sends a message to another peer to invoke a method with arguments or not.

- **StreamInvocation:** client or server sends a message to another peer to invoke a streaming method with arguments or not. The Response will be split into different items.
- **StreamItem:** is a response from a previous StreamInvocation.
- **Completion:** means a previous invocation or StreamInvocation has been completed. Can contain a result if the process has been successful or an error if there is some error.
- **CancelInvocation:** cancel a previous StreamInvocation request.
- **Ping:** is a message to check if the connection is still alive.

SignalRCore Encoding

SignalRCore allows you to use the following encodings:

- **JSON**: currently the only supported encoding.
- **MessagePack**

Currently, only JSON is supported although MessagePack can be used encoding the messages sent using an external messagepack library. See the section MessagePack below for more information.

The configuration of the Encoding Protocol is defined in the property SignalRCore.Protocol. By default the value is **srcpJSON**.

Authorization

Authentication can be enabled to associate a user with each connection and filter which users can access resources. Authentication is implemented using Bearer Tokens: the client provides an access token and the server validates this token and uses it to identify the user.
In standard Web APIs, bearer tokens are sent in an HTTP Header, but when using websockets, the token is transmitted as a query string parameter.
The following methods are supported:

srcaRequestToken

If Authentication is enabled, the flow is:

1. First tries to get a valid token from server. Opens an HTTP connection against Authentication.RequestToken.URL and do a POST using User and Password data.
2. If previous is successful, a token is returned. If not, an error is returned.
3. If token is returned, then opens a new HTTP connection to negotiate a new connection. Here, token is passed as an HTTP Header.
4. If previous is successful, opens a websocket connection and pass token as query string parameter.

- **Authentication.Enabled:** if active, authorization will be used before a websocket connection is established.
- **Authentication.Username:** the username provided to server to authenticate.
- **Authentication.Password:** the secret word provided to server to authenticate.
- **Authentication.RequestToken.PostFieldUsername:** name of field to transmit username (depends on configuration, check http javascript page to see which name is used).
- **Authentication.RequestToken.PostFieldPassword:** name of field to transmit password (depends on configuration, check http javascript page to see which name is used).
- **Authentication.RequestToken.URL:** url where token is requested.
- **Authentication.RequestToken.QueryFieldToken:** name of query string parameter using in websocket connection.

srcaSetToken

Here, you pass token directly to SignalRCore server (because token has been obtained from another server).

- **Authentication.Enabled:** if active, authorization will be used before a websocket connection is established.
- **Authentication.SetToken.Token:** token value obtained.

The Access token can be sent as a query parameter (this is the option by default) or sent as an HTTP Header as a Bearer Token. Use the property Authentication.TokenParam to configure this behaviour.

- **srctQuery:** the access_token is passed in the query url of the websocket connection.
- **srctHeader:** the access_token is passed as an http header as a Bearer Token.

srcaBasic

This option uses **Basic Authentication**, this authentication method requires configuring the SignalRCore component and the TsgcWebSocketClient.

Example: if the server requires basic authentication and the username is "user" and the password is "secret", configure the components as shown below.

```
// websocket client
TsgcWebSocketClient WSClient = new TsgcWebSocketClient();
WSClient.Authentication.Enabled = true;
WSClient.Authentication.Basic.Enabled = true;
WSClient.Authentication.URL.Enabled = false;
WSClient.Authentication.Session.Enabled = false;
WSClient.Authentication.Token.Enabled = false;
WSClient.Authentication.User = "user";
WSClient.Authentication.Password = "secret";
// signalrcore
TsgcWSAPI_SignalRCore Signal = new TsgcWSAPI_SignalRCore();
Signal.SignalRCore.Authentication.Enabled = true;
Signal.SignalRCore.Authentication.Authentication = srcaBasic;
Signal.SignalRCore.Authentication.Username = "user";
Signal.SignalRCore.Authentication.Password = "secret";
Signal.Client = WSClient;
```

Communication between Client and Server

There are three kinds of interactions between server and clients:

Invocations

The Caller sends a message to the Callee and expects a message indicating that the invocation has been completed and optionally a result of the invocation

Example: client invokes SendMessage method and passes as parameters user name and text message. Sends an Invocation Id to
get a result message from the server.

```
SignalRCore.Invoke("SendMessage", new object[] {"John", "Hello All."}, "id-000001");

void OnSignalRCoreCompletion(object Sender, TSignalRCore_Completion Completion)
{
  if (Completion.Error != "")
  {
    MessageBox.Show("Something goes wrong.");
  }
  else
  {
    MessageBox.Show("Invocation Successful!");
  }
}
```

Non-Blocking Invocations

The Caller sends a message to the Callee and does not expect any further messages for this invocation. Invocations can be sent without an Invocation ID value. This indicates that the invocation is "non-blocking".

Example: client invokes SendMessage method and passes as parameters user name and text message. The client doesn't expect any response from the server about the result of the invocation.

```
SignalRCore.Invoke("SendMessage", new object[] {"John", "Hello All."});
```

Streaming Invocations

The Caller sends a message to the Callee and expects one or more results returned by the Callee followed by a message indicating the end of invocation.

Example: client invokes Counter method and requests 10 numbers with an interval of 500 milliseconds.

```
SignalRCore.InvokeStream("Counter", new object[] {10, 500}, "id-000002");

void OnSignalRCoreStreamItem(object Sender, TSignalRCore_StreamItem StreamItem, ref bool Cancel)
{
  DoLog("#stream item: " + StreamItem.Item);
}

void OnSignalRCoreCompletion(object Sender, TSignalRCore_Completion Completion)
{
  if (Completion.Error != "")
  {
    MessageBox.Show("Something goes wrong.");
  }
  else
  {
    MessageBox.Show("Invocation Successful!");
  }
}
```

Invocations

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

```
void Invoke(string aTarget, object[] aArguments, string aInvocationId = "");
void InvokeStream(string aTarget, object[] aArguments, string aInvocationId);
```

Allocate a unique Invocation ID value (arbitrary string, chosen by the Caller) to represent the invocation. Call Invoke or InvokeStream method containing the Target being invoked, Arguments and InvocationId (if you don't send InvocationId, you won't get completion result).

If the Invocation is marked as non-blocking (see "Non-Blocking Invocations" below), stop here and immediately yield back to the application. Handle StreamItem or Completion message with a matching Invocation ID.

```
SignalRCore.InvokeStream("Counter", new object[] {10, 500}, "id-000002");

void OnSignalRCoreStreamItem(object Sender, TSignalRCore_StreamItem StreamItem, ref bool Cancel)
{
  if (StreamItem.InvocationId == "id-000002")
  {
    DoLog("#stream item: " + StreamItem.Item);
  }
}

void OnSignalRCoreCompletion(object Sender, TSignalRCore_Completion Completion)
{
  if (Completion.InvocationId == "id-000002")
  {
    if (Completion.Error != "")
    {
      MessageBox.Show("Something goes wrong.");
    }
    else
    {
      MessageBox.Show("Invocation Successful!");
    }
  }
}
```

You can call a single invocation and wait for completion.

```
bool InvokeAndWait(string aTarget, object[] aArguments, string aInvocationId, out TSignalRCore_Completion Complet
bool InvokeStreamAndWait(string aTarget, object[] aArguments, string aInvocationId, out TSignalRCore_Completion C
```

Allocate a unique Invocation ID value (arbitrary string, chosen by the Caller) to represent the invocation. Call InvokeAndWait or InvokeStreamAndWait method containing the Target being invoked, Arguments and InvocationId. The program will wait till completion event is called or Time out has been exceeded.

```
TSignalRCore_Completion oCompletion;
if (SignalRCore.InvokeStreamAndWait("Counter", new object[] {10, 500}, "id-000002", out oCompletion))
{
  DoLog("#invoke stream ok: " + oCompletion.Result);
}
else
{
  DoLog("#invocke stream error: " + oCompletion.Error);
}

void OnSignalRCoreStreamItem(object Sender, TSignalRCore_StreamItem StreamItem, ref bool Cancel)
{
  if (StreamItem.InvocationId == "id-000002")
  {
    DoLog("#stream item: " + StreamItem.Item);
  }
}
```

Cancel Invocation

If the client wants to stop receiving StreamItem messages before the Server sends a Completion message, the client can send a CancelInvocation message with the same InvocationId used for the StreamInvocation message that started the stream.

```
void OnSignalRCoreStreamItem(object Sender, TSignalRCore_StreamItem StreamItem, ref bool Cancel)
{
  if (StreamItem.InvocationId == "id-000002")
  {
    Cancel = true;
  }
}
```

Client Results

An Invocation is only considered completed when the Completion message is received. If the client receives an Invocation from the server, OnSignalRCoreInvocation event will be called.

```
void OnSignalRCoreInvocation(object Sender, TSignalRCore_Invocation Invocation)
{
  if (Invocation.Target == "SendMessage")
  {
    // ... your code here ...
  }
}

// Once invocation is completed, call Completion method to inform server invocation is finished.
// If result is successful, then call CompletionResult method:
SignalRCore.CompletionResult("id-000002", "ok");

// If not, then call CompletionError method:
SignalRCore.CompletionError("id-000002", "Error processing invocation.");
```

Close Connection

Sent by the client when a connection is closed. Contains an error reason if the connection was closed because of an error.

```
SignalRCore.Close("Unexpected message");

// If the server close connection by any reason, OnSignalRCoreClose event will be called.
void OnSignalRCoreClose(object Sender, TSignalRCore_Close Close)
{
  DoLog("#closed: " + Close.Error);
}
```

Ping

The SignalR Hub protocol supports "Keep Alive" messages used to ensure that the underlying transport connection remains active. These messages help ensure:

Proxies don't close the underlying connection during idle times (when few messages are being sent). If the underlying connection is dropped without being terminated gracefully, the application is informed as quickly as possible.

Keep alive behaviour is achieved calling Ping method or enabling HeartBeat on WebSocket client. If the server sends a ping to the client, the client will send automatically a response and OnSignalRCoreKeepAlive event will be called.

```
void OnSignalRCoreKeepAlive(object Sender)
{
  DoLog("#keepalive");
}
```

MessagePack

In the MsgPack Encoding of the SignalR Protocol, each Message is represented as a single MsgPack array containing items that correspond to properties of the given hub protocol message. The array items may be primitive values, arrays (e.g. method arguments) or objects (e.g. argument value). The first item in the array is the message type.

Refer to the MessagePack documentation to see how encode the messages sent.

Every time a new message is received, this is dispatched in the event OnSignalRCoreMessagePack event. The message can be accessed reading the Data Stream parameter. The parameter JSON by default is empty, if you convert the MessagePack message to JSON, the component will process the JSON message as if the encoding was using JSON (so the events OnSignalRCoreCompletion, OnSignalRCoreInvocation... will be dispatched).

API SignalR

SignalR

SignalR component uses WebSocket as transport to connect to a SignalR server, if this transport is not supported, an error will be raised.
SignalR client component has a property called SignalR where you can set following data:

- **Hubs:** contains a list of hubs the client is subscribing to.
- **ProtocolVersion:** the version of the protocol used by the client, supports protocol versions from 1.2 to 1.5
- **UserAgent:** user agent used to connect to SignalR server.

The client supports sending Text or Binary data.

Hubs Messages

Hubs API makes it possible to invoke server methods from the client and client methods from the server. The protocol used for persistent connection is not rich enough to allow expressing RPC (remote procedure call) semantics. It does not mean however that the protocol used for hub connections is completely different from the protocol used for persistent connections. Rather, the protocol used for hub connections is mostly an extension of the protocol for persistent connections.

When a client invokes a server method it no longer sends a free-flow string as it was for persistent connections. Instead, it sends a JSON string containing all necessary information needed to invoke the method. Here is a sample message a client would send to invoke a server method:

```
WriteData("{\"H\":\"chathub\",\"M\":\"Send\",\"A\":[\"CSharp Client\",\"Test message\"],\"I\":0}");
```

The payload has the following properties:
I – invocation identifier – allows you to match up responses with requests
H – the name of the hub
M – the name of the method
A – arguments (an array, can be empty if the method does not have any parameters)

If the string argument has **double quotes** replace " by \"

Example: if the argument is {"test":1}, send the argument as {\"test\":1}

```
WriteData('{"H":"chathub","M":"Send","A":["{\"test\":1}"],"I":0}');
```

Authorization

Authentication can be enabled to associate a user with each connection and filter which users can access resources. Authentication is implemented using Bearer Tokens: the client provides an access token and the server validates this token and uses it to identify the user.

Currently only Bearer Tokens are supported:

Here, you pass token directly to Signal server (because token has been obtained from another server).

- **Authentication.Enabled:** if active, authorization will be used before a websocket connection is established.

- **Authentication.Authentication:** 2 types of authentication are supported: bearer token or cookies. Both require an external way to get the required values.
  - **BearerToken:** token value obtained.
  - **Cookie:** set the value of the cookie required.

```
TsgcWSAPI_Signal oSignalR = new TsgcWSAPI_Signal();
oSignalR.SignalR.Enabled = true;
oSignalR.SignalR.Authentication = srcBearerToken;
oSignalR.SignalR.BearerToken.Token = "token here";
```

The component has the following events:

OnSignalRConnect

This event is raised when the client connects successfully to the server.

OnSignalRDisconnect

This event is raised when the client is disconnected from the server.

OnSignalRError

This event is called when there is an error in WebSocket connection.

OnSignalRMessage

The protocol used for persistent connection is quite simple. Messages sent to the server are just raw strings. There is no specific format they have to be in. Messages sent to the client are more structured. The properties you can find in the message are as follows:

C – message id, present for all non-KeepAlive messages
M – an array containing actual data.

```
{"C":"d-9B7A6976-B,2|C,2","M":["Welcome!"]}
```

OnSignalRBinary

This event is called when binary data is received from the server.

OnSignalRResult

When a server method is invoked, the server returns a confirmation that the invocation has completed by sending the invocation id to the client and – if the method returned a value – the return value, or – if invoking the method failed – the error.
Here are sample results of a server method call:

```
{"I":"0"}
```

A server void method whose invocation identifier was "0" completed successfully.

```
{"I":"0", "R":42}
```

A server method returning a number whose invocation identifier was "0" completed successfully and returned the value 42.

```
{"I":"0", "E":"Error occurred"}
```

OnSignalRKeepAlive

This event is raised when a KeepAlive message is received from the server.

API Kraken

Kraken

Overview

WebSockets API offers real-time market data updates. WebSockets is a bidirectional protocol offering fastest real-time data, helping you build real-time applications. The public message types presented below do not require authentication. Private-data messages can be subscribed on a separate authenticated endpoint.

Kraken offers a REST API too with Public market data and Private user data (which requires an authentication).

Configuration

Private API requires creating an API key from your Kraken account.
Kraken allows Test environment on WebSocket protocol, enable Beta property from Kraken Property to use this beta feature.

APIs supported

- WebSockets Public API: connects to a public WebSocket server.
- WebSockets Private API: connects to a private WebSocket server and requires an API Key and API Secret to Authenticate against server.
- REST Public API: connects to a public REST server.
- REST Private API: connects to a public REST server and requires an API Key and API Secret to Authenticate against server.

Kraken Examples

How to Connect to Public WebSocket Server

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kraken oKraken = new TsgcWSAPI_Kraken();
oKraken.Client = oClient;
oClient.Active = true;
```

How to Connect to Private WebSocket Server

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kraken oKraken = new TsgcWSAPI_Kraken();
oKraken.Kraken.ApiKey = "your api key";
oKraken.Kraken.ApiSecret = "your api secret";
oKraken.Client = oClient;
oClient.Active = true;
```

How to Get Ticker from REST API

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kraken oKraken = new TsgcWSAPI_Kraken();
```

```
oKraken.Client = oClient;
MessageBox(oKraken.GetTicker("XBTUSD"));
```

REST API Methods

Public Endpoints

Method	Arguments	Description
GetSystemStatus		Get current system status.

Private Endpoints

Method	Arguments	Description
GetExtendedBalance		Get extended balance information.
AmendOrder		Amend an existing order.
CancelAllOrders		Cancel all open orders.
CancelAllOrdersAfter		Dead man's switch - cancel all orders after timeout.
EditOrder		Edit an existing order.
AddOrderBatch		Batch add multiple orders.
CancelOrderBatch		Batch cancel multiple orders.
GetWithdrawalMethods		Get available withdrawal methods.
GetWithdrawalAddresses		Get withdrawal addresses.

How to Get Account Balance from REST API

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kraken oKraken = new TsgcWSAPI_Kraken();
oKraken.Kraken.ApiKey = "your api key";
oKraken.Kraken.ApiSecret = "your api secret";
oKraken.Client = oClient;
MessageBox.Show(oKraken.GetAccountBalance());
```

API Kraken | WebSockets Public API

Connection

URL: wss://ws.kraken.com (v1) or wss://ws.kraken.com/v2 (v2, recommended)
The component now supports WebSocket API v2 via the **Version** property (default: 2). Set `Kraken.Version := 1` to use the legacy v1 endpoint.

Once the socket is open you can subscribe to a public channel by sending a subscribe request message.

General Considerations

- All messages sent and received via WebSockets are encoded in JSON format
- All floating point fields (including timestamps) are quoted to preserve precision.
- Format of each tradeable pair is A/B, where A and B are ISO 4217-A3 for standardized assets and popular unique symbol if not standardized.
- Timestamps should not be considered unique and not be considered as aliases for transaction ids. Also, the granularity of timestamps is not representative of transaction rates.

Supported Pairs

ADA/CAD, ADA/ETH, ADA/EUR, ADA/USD, ADA/XBT, ATOM/CAD, ATOM/ETH, ATOM/EUR, ATOM/USD, ATOM/XBT, BCH/EUR, BCH/USD, BCH/XBT, DASH/EUR, DASH/USD, DASH/XBT, EOS/ETH, EOS/EUR, EOS/USD, EOS/XBT, GNO/ETH, GNO/EUR, GNO/USD, GNO/XBT, QTUM/CAD, QTUM/ETH, QTUM/EUR, QTUM/USD, QTUM/XBT, USDT/USD, ETC/ETH, ETC/XBT, ETC/EUR, ETC/USD, ETH/XBT, ETH/CAD, ETH/EUR, ETH/GBP, ETH/JPY, ETH/USD, LTC/XBT, LTC/EUR, LTC/USD, MLN/ETH, MLN/XBT, REP/ETH, REP/XBT, REP/EUR, REP/USD, STR/EUR, STR/USD, XBT/CAD, XBT/EUR, XBT/GBP, XBT/JPY, XBT/USD, BTC/CAD, BTC/EUR, BTC/GBP, BTC/JPY, BTC/USD, XDG/XBT, XLM/XBT, DOGE/XBT, STR/XBT, XLM/EUR, XLM/USD, XMR/XBT, XMR/EUR, XMR/USD, XRP/XBT, XRP/CAD, XRP/EUR, XRP/JPY, XRP/USD, ZEC/XBT, ZEC/EUR, ZEC/JPY, ZEC/USD, XTZ/CAD, XTZ/ETH, XTZ/EUR, XTZ/USD, XTZ/XBT

Methods

Ping

Client can ping server to determine whether connection is alive, server responds with pong.
This is an application level ping as opposed to default ping in WebSockets standard which is server initiated

Ticker

Ticker information includes best ask and best bid prices, 24hr volume, last trade price, volume weighted average price, etc for a given currency pair. A ticker message is published every time a trade or a group of trade happens.
Subscribe to a ticker calling SubscribeTicker method:

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

UnSubscribe calling UnSubscribeTicker method:

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

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

Ticker updates will be notified in OnKrakenData event.


```
[
  0,
  {
    "a": [
      "5525.40000",
      1,
      "1.000"
    ],
    "b": [
      "5525.10000",
      1,
      "1.000"
    ],
    "c": [
      "5525.10000",
      "0.00398963"
    ],
    "v": [
      "2634.11501494",
      "3591.17907851"
    ],
    "p": [
      "5631.44067",
      "5653.78939"
    ],
    "t": [
      11493,
      16267
    ],
    "l": [
      "5505.00000",
      "5505.00000"
    ],
    "h": [
      "5783.00000",
      "5783.00000"
    ],
    "o": [
      "5760.70000",
      "5763.40000"
    ]
  },
  "ticker",
  "XBT/USD"
]
```


OHLC

When subscribed for OHLC, a snapshot of the last valid candle (irrespective of the endtime) will be sent, followed by updates to the running candle. For example, if a subscription is made to 1 min candle and there have been no trades for 5 mins, a snapshot of the last 1 min candle from 5 mins ago will be published. The endtime can be used to determine that it is an old candle.
Subscribe to a OHLC calling SubscribeOHLC method, you must pass pair and interval.

If subscription is successful, OnKrakenSubscribed event will be called:

UnSubscribe calling UnSubscribeOHLC method:

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

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

OHLC updates will be notified in OnKrakenData event.

```
[
  42,
  [
    "1542057314.748456",
    "1542057360.435743",
    "3586.70000",
    "3586.70000",
    "3586.60000",
    "3586.60000",
    "3586.68894",
    "0.03373000",
    2
  ],
  "ohlc-5",
  "XBT/USD"
]
```

Trade

Trade feed for a currency pair.
Subscribe to Trade feed calling SubscribeTrade method.

If subscription is successful, OnKrakenSubscribed event will be called:

UnSubscribe calling UnSubscribeTrade method:

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

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

Trade updates will be notified in OnKrakenData event.

```
[
  0,
  [
    [
      "5541.20000",
      "0.15850568",
      "1534614057.321597",
      "s",
      "l",
      ""
    ],
    [
      "6060.00000",
      "0.02455000",
      "1534614057.324998",
      "b",
      "l",
      ""
    ]
  ],
  "trade",
  "XBT/USD"
]
```

Book

Order book levels. On subscription, a snapshot will be published at the specified depth, following the snapshot, level updates will be published.
Subscribe to a Book calling SubscribeBook method, you must pass pair and depth.

If subscription is successful, OnKrakenSubscribed event will be called:

UnSubscribe calling UnSubscribeBook method:

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

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

Book updates will be notified in OnKrakenData event.

```
[
  0,
  {
    "as": [
      [
          "5541.30000",
          "2.50700000",
          "1534614248.123678"
      ],
      [
          "5541.80000",
          "0.33000000",
          "1534614098.345543"
      ],
      [
          "5542.70000",
          "0.64700000",
          "1534614244.654432"
      ]
    ],
    "bs": [
      [
          "5541.20000",
          "1.52900000",
          "1534614248.765567"
      ],
      [
          "5539.90000",
          "0.30000000",
          "1534614241.769870"
      ],
      [
          "5539.50000",
          "5.00000000",
          "1534613831.243486"
      ]
    ]
  },
  "book-100",
  "XBT/USD"
]
```

Spread

Spread feed to show best bid and ask price for subscribed asset pair. Bid volume and ask volume is part of the message too.
Subscribe to Spread feed calling SubscribeSpread method.

If subscription is successful, OnKrakenSubscribed event will be called:

UnSubscribe calling UnSubscribeSpread method:

If unsubscription is successful, OnKrakenUnSubscribed event will be called:

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

Spread updates will be notified in OnKrakenData event.

```
[
  0,
  [
    "5698.40000",
    "5700.00000",
    "1542057299.545897",
    "1.01234567",
    "0.98765432"
  ],
  "spread",
  "XBT/USD"
]
```

Other Methods

You can subscribe / unsubscribe to all channels with one method:

OHLC interval value is 1 if all channels subscribed.

Events

OnConnect: when websocket client is connected to server.
OnKrakenConnect: called after successful websocket connection and when server sends system status.
OnKrakenSystemStatus: called when system status changes.
OnKrakenSubscribed: called after a successful subscription to a channel.
OnKrakenUnSubscribed: called after a successful unsubscription from a channel.
OnKrakenSubscriptionError: called if there is an error trying to subscribe / unsubscribe.
OnKrakenData: called every time a channel subscription has an update.

API Kraken | WebSockets Private API

Connection

URL: wss://ws-auth.kraken.com (v1) or wss://ws-auth.kraken.com/v2 (v2, recommended)
The component now supports WebSocket API v2 via the **Version** property (default: 2).

Once the socket is open you can subscribe to private-data channels by sending an authenticated subscribe request message.

Authentication

The API client must request an authentication "token" via the following REST API endpoint "GetWebSocketsToken" to connect to WebSockets Private endpoints. The token should be used within 15 minutes of creation. The token does not expire once a connection to a WebSockets API private message (openOrders or ownTrades) is maintained.
In order to get a Websockets Token, an API Key and API Secret must be set in Kraken Options Component, the api key provided by Kraken in your account

```
Kraken.ApiKey = "api key";
Kraken.ApiSecret = "api secret";
```

Methods

OwnTrades

Get a list of own trades, on first subscription, you get a list of latest 50 trades

```
SubscribeOwnTrades();
```

Later, you can unsubscribe from OwnTrades, calling UnSubscribeOwnTrades method

```
UnSubscribeOwnTrades();
```

Response example from server

```
[
  [
    {
      "TDLH43-DVQXD-2KHVYY": {
        "cost": "1000000.00000",
        "fee": "600.00000",
        "margin": "0.00000",
        "ordertxid": "TDLH43-DVQXD-2KHVYY",
        "ordertype": "limit",
        "pair": "XBT/EUR",
        "postxid": "OGTT3Y-C6I3P-XRI6HX",
        "price": "100000.00000",
        "time": "1560520332.914664",
        "type": "buy",
        "vol": "1000000000.00000000"
      }
    }
  ],
  "ownTrades"
]
```

Open Orders

Feed to show all the open orders belonging to the user authenticated API key. Initial snapshot will provide list of all open orders and then any updates to the open orders list will be sent. For status change updates, such as 'closed', the fields orderid and status will be present in the payload

```
SubscribeOpenOrders();
```

Later, you can unsubscribe from OpenOrders, calling UnSubscribeOpenOrders method

```
UnSubscribeOpenOrders();
```

Response example from server


```
[
  [
    {
      "OGTT3Y-C6I3P-XRI6HX": {
        "cost": "0.00000",
        "descr": {
          "close": "",
          "leverage": "0:1",
          "order": "sell 0.00001000 XBT/EUR @ limit 9.00000 with 0:1 leverage",
          "ordertype": "limit",
          "pair": "XBT/EUR",
          "price": "9.00000",
          "price2": "0.00000",
          "type": "sell"
        },
        "expiretm": "0.000000",
        "fee": "0.00000",
        "limitprice": "9.00000",
        "misc": "",
        "oflags": "fcib",
        "opentm": "0.000000",
        "price": "9.00000",
        "refid": "OKIVMP-5GVZN-Z2D2UA",
        "starttm": "0.000000",
        "status": "open",
        "stopprice": "0.000000",
        "userref": 0,
        "vol": "0.00001000",
        "vol_exec": "0.00000000"
      }
    }
  ],
  "openOrders"
]
```


Add Order

Send a new Order to Kraken

```
TsgcWSKrakenOrder oKrakenOrder = new TsgcWSKrakenOrder();
oKrakenOrder.Pair = "XBT/USD";
oKrakenOrder._Type = kosBuy;
oKrakenOrder.OrderType = kotMarket;
oKrakenOrder.Volume = 1;
AddOrder(oKrakenOrder);
```

List of Order parameters

```
pair = asset pair
type = type of order (buy/sell)
ordertype = order type:
    market
    limit (price = limit price)
    stop-loss (price = stop loss price)
    take-profit (price = take profit price)
    stop-loss-profit (price = stop loss price, price2 = take profit price)
    stop-loss-profit-limit (price = stop loss price, price2 = take profit price)
    stop-loss-limit (price = stop loss trigger price, price2 = triggered limit price)
```

```
    take-profit-limit (price = take profit trigger price, price2 = triggered limit price)
    trailing-stop (price = trailing stop offset)
    trailing-stop-limit (price = trailing stop offset, price2 = triggered limit offset)
    stop-loss-and-limit (price = stop loss price, price2 = limit price)
    settle-position
price = price (optional.  dependent upon ordertype)
price2 = secondary price (optional.  dependent upon ordertype)
volume = order volume in lots
leverage = amount of leverage desired (optional.  default = none)
oflags = comma delimited list of order flags (optional):
    viqc = volume in quote currency (not available for leveraged orders)
    fcib = prefer fee in base currency
    fciq = prefer fee in quote currency
    nompp = no market price protection
    post = post only order (available when ordertype = limit)
starttm = scheduled start time (optional):
    0 = now (default)
    +<n> = schedule start time <n> seconds from now
    <n> = unix timestamp of start time
expiretm = expiration time (optional):
    0 = no expiration (default)
    +<n> = expire <n> seconds from now
    <n> = unix timestamp of expiration time
userref = user reference id.  32-bit signed number.  (optional)
validate = validate inputs only.  do not submit order (optional)
optional closing order to add to system when order gets filled:
    close[ordertype] = order type
    close[price] = price
    close[price2] = secondary price
```

Response example from server

```
{
  "descr": "buy 0.01770000 XBTUSD @ limit 4000",
  "event": "addOrderStatus",
  "status": "ok",
  "txid": "ONPNXH-KMKMU-F4MR5V"
}
```

Cancel Order

Cancel order

```
CancelOrder("Order Id");
```

Response example from server

```
{
  "event": "cancelOrderStatus",
  "status": "ok"
}
```

API Kraken | REST Public API

Connection

URL: https://api.kraken.com

Kraken Public API doesn't require any authentication.

Configuration

The only configuration is whether to enable a log for REST HTTP requests. Enable HTTPLogOptions if you want to save in a text file log all HTTP Requests/Responses

Events

OnKrakenHTTPException: this event is called if there is any exception doing an HTTP Request from REST Api.

Methods

GetServerTime

This method is to aid in approximating the skew time between the server and client. Returns Time in Unix format.

```
{"error":[],"result":{"unixtime":1586705546,"rfc1123":"Sun, 12 Apr 20 15:32:26 +0000"}}
```

GetAssets

Returns information about Assets

```
{"error":[],"result":{"ADA":{"aclass":"currency","altname":"ADA","decimals":8,"display_decimals":6}}}}
```

GetAssetPairs

Returns information about a pair of assets

```
Kraken.REST_API.GetAssetPairs("XBTUSD");
```

GetTicker

Returns ticker information

```
Kraken.REST_API.GetTicker("XBTUSD");
```

GetOHLC

Returns Open-High-Low-Close data.

```
Kraken.REST_API.GetOHLC("XBTUSD");
```

GetOrderBook

Returns Array pair name and market depth.

```
Kraken.REST_API.GetOrderBook("XBTUSD");
```

GetTrades

Returns recent trade data of a pair.

```
Kraken.REST_API.GetTrades("XBTUSD");
```

GetSpread

Returns recent spread data of a pair.

```
Kraken.REST_API.GetSpread("XBTUSD");
```

API Kraken | REST Private API

Connection

URL: https://api.kraken.com

Authentication

REST Private API requires an API Key and API Secret, these values are provided by Kraken in your account.

```
Kraken.ApiKey = "api key";
Kraken.ApiSecret = "api secret";
```

Methods

GetAccountBalance

Returns your account balance.

```
Kraken.REST_API.GetAccountBalance();
```

GetTradeBalance

Returns information about your trades.

```
Kraken.REST_API.GetTradeBalance();
```

GetOpenOrders

Returns a list of open orders.

```
Kraken.REST_API.GetOpenOrders();
```

GetClosedOrders

Returns a list of closed orders.

```
Kraken.REST_API.GetClosedOrders();
```

QueryOrders

Query information about an order.

```
Kraken.REST_API.QueryOrders("1234");
```

GetTradesHistory

Returns an array of trade info.

```
Kraken.REST_API.GetTradesHistory();
```

QueryTrades

Query information about a trade.

```
Kraken.REST_API.QueryTrades("1234");
```

GetOpenPositions

Returns position info.

```
Kraken.REST_API.GetOpenPositions("1234");
```

GetLedgers

Returns associative array of ledgers info.

```
Kraken.REST_API.GetLedgers();
```

QueryLedgers

Returns associative array of ledgers info.

```
Kraken.REST_API.QueryLedgers("1234");
```

GetTradeVolume

Returns trade volume info.

```
Kraken.REST_API.GetTradeVolume();
```

AddExport

Adds a new report export.

```
Kraken.REST_API.AddExport("Report All Trades");
```

ExportStatus

Get Status of reports

```
Kraken.REST_API.ExportStatus();
```

RetrieveExport

Get Report by report id.

```
Kraken.REST_API.RetrieveExport("GOCO");
```

RemoveExport

Remove Report by report id.

```
Kraken.REST_API.RemoveExport("GOCO");
```

Add Order

Adds a new order

```
pair = asset pair
type = type of order (buy/sell)
ordertype = order type:
    market
    limit (price = limit price)
    stop-loss (price = stop loss price)
    take-profit (price = take profit price)
    stop-loss-profit (price = stop loss price, price2 = take profit price)
    stop-loss-profit-limit (price = stop loss price, price2 = take profit price)
    stop-loss-limit (price = stop loss trigger price, price2 = triggered limit price)
    take-profit-limit (price = take profit trigger price, price2 = triggered limit price)
    trailing-stop (price = trailing stop offset)
    trailing-stop-limit (price = trailing stop offset, price2 = triggered limit offset)
    stop-loss-and-limit (price = stop loss price, price2 = limit price)
    settle-position
price = price (optional.  dependent upon ordertype)
price2 = secondary price (optional.  dependent upon ordertype)
volume = order volume in lots
leverage = amount of leverage desired (optional.  default = none)
oflags = comma delimited list of order flags (optional):
    viqc = volume in quote currency (not available for leveraged orders)
    fcib = prefer fee in base currency
    fciq = prefer fee in quote currency
    nompp = no market price protection
    post = post only order (available when ordertype = limit)
starttm = scheduled start time (optional):
    0 = now (default)
    +n = schedule start time n seconds from now
    n = unix timestamp of start time
expiretm = expiration time (optional):
    0 = no expiration (default)
    +n = expire n seconds from now
    n = unix timestamp of expiration time
userref = user reference id.  32-bit signed number.  (optional)
validate = validate inputs only.  do not submit order (optional)
optional closing order to add to system when order gets filled:
    close[ordertype] = order type
    close[price] = price
    close[price2] = secondary price
```

```
TsgcHTTPKrakenOrder oKrakenOrder = new TsgcHTTPKrakenOrder();
oKrakenOrder.Pair = "XBT/USD";
oKrakenOrder._Type = koshBuy;
oKrakenOrder.OrderType = kothMarket;
oKrakenOrder.Volume = 1;
Kraken.REST_API.AddOrder(oKrakenOrder);
```

CancelOrder

Cancels an open order by id

```
Kraken.REST_API.CancelOrder("1234");
```

API Kraken Futures

Kraken Futures

Overview

The **REST API** allows you to securely access the methods of your Kraken Futures account. Examples of REST API Methods:

- request current or historical price information
- check your account balance and PnL
- your margin parameters and estimated liquidation thresholds
- place or cancel orders (individually or in batch)
- see your open orders
- open positions or trade history
- request a digital asset withdrawal

These methods are called "endpoints" and are explained in REST API section.

The **Websocket API** allows you to securely establish a communication channel to the Kraken Futures platform to receive information in real time. This allows listening to updates instead of continuously sending requests. These channels are called subscriptions.

Some of the endpoints allow performing sensitive tasks, such as initiating a digital asset withdrawal. To access these endpoints securely, the API uses encryption techniques developed by the National Security Agency.

Configuration

In order to use the API, you need to generate a pair of unique **API keys** (if you want access to private APIs):

1. Sign in to your **Kraken Futures account**.
2. Click on your name on the upper-right corner.
3. Select "Settings" from the drop-down menu.
4. Select the "Create Key" tab in the API panel.
5. Press the "Create Key" button.
6. View your Public and Private keys and record them somewhere safe.

Copy the Public and Private Keys to the **KrakenOptions** property of the component.

KrakenOptions.ApiKey
KrakenOptions.ApiSecret

APIs supported

- WebSockets Public API: connects to a public WebSocket server.
- WebSockets Private API: connects to a private WebSocket server and requires an API Key and API Secret to Authenticate against server.
- REST Public API: connects to a public REST server.
- REST Private API: connects to a public REST server and requires an API Key and API Secret to Authenticate against server.

REST API Methods

Private Endpoints

Method	Arguments	Description
BatchOrder		Submit batch orders (place/cancel/edit).
GetOrderStatus		Get status of specific orders.
GetPNLCurrencyPreferences		Get PNL currency preferences.
SetPNLCurrencyPreference		Set PNL currency preference for a symbol.
GetLeverageSettings		Get leverage preferences.
SetLeverageSettings		Set leverage for a symbol.

API Kraken Futures | WebSockets Public API

Connection

URL: wss://futures.kraken.com/ws/v1

Once the socket is open you can subscribe to a public channel by sending a subscribe request message.

Methods

Ticker

This endpoint returns current market data for all currently listed Futures contracts and indices. Authentication is not required.
Subscribe to a ticker calling **SubscribeTicker** method:

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

UnSubscribe calling **UnSubscribeTicker** method:

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

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

Ticker updates will be notified in **OnKrakenData** event.


```
{    "result":"success",

 "tickers":[

{

"tag": "perpetual",

"pair": "XBT:USD",

"symbol": "pi_xbtusd",

"markPrice": 9520.2,

"bid": 9520,

"bidSize": 30950,

"ask": 9520.5,

"askSize": 3779,
```

"vol24h": 68238712,

"openInterest": 29308193,

"open24h": 10137,

"last": 9521,

"lastTime": "2020-06-03T08:14:26.624Z",

"lastSize": 1,

"suspended": false,

"fundingRate": 4.943012455e-9,

"fundingRatePrediction": 4.414499215e-9

}

{

"tag": "quarter",

"pair": "XBT:USD",

"symbol": "fi_xbtusd_200925",

"markPrice": 9659.8,

"bid": 9659.5,

"bidSize": 6480,

"ask": 9660,

"askSize": 17100,

"vol24h": 4562580,

"openInterest": 3573325,

"open24h": 10370.5,

"last": 9660,

"lastTime": "2020-06-03T08:10:37.800Z",

"lastSize": 5000,

"suspended": false

},

{

"symbol": "in_xbtusd",

"last": 9519,

"lastTime": "2020-06-03T08:14:49.000Z"

}

],

"serverTime": "2020-06-03T08:14:49.865Z"

}

Trade

The trade feed returns information about executed trades
Subscribe to Trade feed calling **SubscribeTrade** method.

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

UnSubscribe calling **UnSubscribeTrade** method:

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

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

Trade updates will be notified in **OnKrakenData** event.


```
{  "feed": "trade_snapshot",

 "product_id": "PI_XBTUSD",

 "trades": [

{

  "feed": "trade",

  "product_id": "PI_XBTUSD",

  "uid": "caa9c653-420b-4c24-a9f1-462a054d86f1",

  "side": "sell",

  "type": "fill",

  "seq": 655508,

  "time": 1612269657781,

  "qty": 440,

  "price": 34893

},

{

  "feed": "trade",
```

"product_id": "PI_XBTUSD",

"uid": "45ee9737-1877-4682-bc68-e4ef818ef88a",

"side": "sell",

"type": "fill",

"seq": 655507,

"time": 1612269656839,

"qty": 9643,

"price": 34891

}

]

}

Book

This feed returns information about the order book.
Subscribe to a Book calling SubscribeBook method, you must pass the Symbol.

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

UnSubscribe calling **UnSubscribeBook** method:

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

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

Book updates will be notified in **OnKrakenData** event.

{

"feed": "book_snapshot",

"product_id": "PI_XBTUSD",

"timestamp": 1612269825817,

"seq": 326072249,

"tickSize": null,

"bids": [{

"price": 34892.5,

"qty": 6385

},

{

"price": 34892,

"qty": 10924

}

],

"asks": [{

"price": 34911.5,

"qty": 20598

},

{

"price": 34912,

"qty": 2300

}

]

}

Ticker Lite

The ticker lite feed returns ticker information about listed products.
Subscribe to Spread feed calling **SubscribeTickerLite** method.

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

UnSubscribe calling **UnSubscribeTickerLite** method:

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

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

Spread updates will be notified in **OnKrakenData** event.

{ "feed": "ticker_lite",

"product_id": "PI_XBTUSD",

"bid": 34932,

"ask": 34949.5,

"change": 3.3705205220015966,

"premium": 0.1,

"volume": 264126741,

"tag": "perpetual",

"pair": "XBT:USD",

"dtm": 0,

"maturityTime": 0

}

{

"feed":"ticker_lite",

"product_id":"FI_ETHUSD_210625",

"bid":1753.45,

"ask":1760.35,

"change":13.448175559936647,

"premium":9.1,

"volume":6899673.0,

"tag":"semiannual",

"pair":"ETH:USD",

"dtm":141,

"maturityTime":1624633200000

}

HeartBeat

The heartbeat feed publishes a heartbeat message at timed intervals.

Events

OnConnect: when websocket client is connected to server.
OnKrakenFuturesConnect: called after successful websocket connection and when server sends system status.
OnKrakenFuturesSubscribed: called after a successful subscription to a channel.
OnKrakenFuturesUnSubscribed: called after a successful unsubscription from a channel.
OnKrakenFuturesError: called if there is any error while subscribing/unsubscribing.
OnKrakenData: called every time a channel subscription has an update.

API Kraken Futures | WebSockets Private API

Connection

URL: wss://futures.kraken.com/ws/v1

Authentication

The subscribe and unsubscribe requests to WebSocket private feeds require a signed challenge message with the user api_secret.
The challenge is obtained as is shown in Section WebSocket API Public (using the api_key).
Authenticated requests must include both the original challenge message (original_challenge) and the signed (signed_challenge) in JSON format.
In order to get a Websockets Challenge, an API Key and API Secret must be set in Kraken Options Component, the api key provided by Kraken in your account

```
Kraken.ApiKey = "api key";
Kraken.ApiSecret = "api secret";
```

Methods

Open Orders Verbose

This subscription feed publishes information about user open orders. This feed adds extra information about all the post-only orders that failed to cross the book.

```
SubscribeOpenOrdersVerbose();
```

Later, you can unsubscribe from OpenOrdersVerbose, calling **UnSubscribeOpenOrdersVerbose** method

```
UnSubscribeOpenOrdersVerbose();
```

Response example from server

```
{        'feed':'open_orders_verbose_snapshot',    'account':'0f9c23b8-63e2-40e4-9592-6d5aa57c12
        {           'instrument':'PI_XBTUSD',         'time':1567428848005,         'last_updat
```

Open Positions

This subscription feed publishes the open positions of the user account.

```
SubscribeOpenPositions();
```

Later, you can unsubscribe from OpenPositions, calling **UnSubscribeOpenPositions** method

```
UnSubscribeOpenPositions();
```

Response example from server

```
{
    "feed": "open_positions",
    "account": "DemoUser",
    "positions": [{
        "instrument": "fi_xbtusd_180316",
        "balance": 2000.0,
        "entry_price": 11675.86541981,
        "mark_price": 11090.0,
        "index_price": 12290.550000000001,
        "pnl": -0.00905299
    }]
}
```

Account Log

This subscription feed publishes account information.

```
SubscribeAccountLog();
```

Later, you can unsubscribe from AccountLog, calling **UnSubscribeAccountLog** method

```
UnSubscribeAccountLog();
```

Response example from server

```
{
    'feed': 'account_log_snapshot',
    'logs': [{
                'id': 1690,
                'date': '2019-07-11T08:00:00.000Z',
                'asset': 'bch',
                'info': 'funding
                rate change ','
                booking_uid ':'
                86 fdc252 - 1 b6e - 40 ec - ac1d - c7bd46ddeebf ','
                margin_account ':'
                f - bch: usd ','
                old_balance ':0.01215667051,'
                new_balance ':0.01215736653,'
                old_average_entry_price ':0.0,'
                new_average_entry_price ':0.0,'
                trade_price ':0.0,'
                mark_price ':0.0,'
                realized_pnl ':0.0,'
                fee ':0.0,'
                execution ':'
                ','
                collateral ':'
                bch ','
                funding_rate ':-8.7002552653e-08,'
                realized_funding ':6.9602e-07}]
}
```

Fills

This subscription feed publishes fills information.

```
SubscribeFills();
```

Later, you can unsubscribe from Fills, calling **UnSubscribeFills** method

```
UnSubscribeFills();
```

Response example from server

{

"feed":"fills_snapshot",

"account":"DemoUser",

"fills":[

{

"instrument":"FI_XBTUSD_200925",

"time":1600256910739,

"price":10937.5,

"seq":36,

"buy":true,

"qty":5000.0,

"order_id":"9e30258b-5a98-4002-968a-5b0e149bcfbf",

"fill_id":"cad76f07-814e-4dc6-8478-7867407b6bff",

"fill_type":"maker",

"fee_paid":-0.00009142857,

"fee_currency":"BTC"

}]

}

Open Orders

This subscription feed publishes information about user open orders.

```
SubscribeOpenOrders();
```

Later, you can unsubscribe from OpenOrders, calling **UnSubscribeOpenOrders** method

```
UnSubscribeOpenOrders();
```

Response example from server

{

"feed": "open_orders_snapshot",

"account": "e258dba9-4dd4-4da5-bfef-75beb91c098e",

"orders": [

{

"instrument": "PI_XBTUSD",

"time": 1612275024153,

"last_update_time": 1612275024153,

"qty": 1000,

"filled": 0,

"limit_price": 34900,

"stop_price": 13789,

"type": "stop",

"order_id": "723ba95f-13b7-418b-8fcf-ab7ba6620555",

"direction": 1,

"reduce_only": false,

"triggerSignal": "last"

}

]

}

Account Balance And Margins

This subscription feed returns balance and margin information for the client's account.

```
SubscribeAccountBalanceAndMargins();
```

Later, you can unsubscribe from AccounBalance, calling **UnSubscribeAccountBalanceAndMargins** method

```
UnSubscribeAccountBalanceAndMargins();
```

Response example from server

{

"feed": "account_balances_and_margins",

"account": "DemoUser",

"margin_accounts": [

{

"name": "xbt",
"balance": 0,
"pnl": 0,
"funding": 0,
"pv": 0,
"am": 0,
"im": 0,
"mm": 0
},
{
"name": "f-xbt:usd",
"balance": 9.99730211055,
"pnl": -0.00006034858674327812,
"funding": 0,
"pv": 9.997241761963258,
"am": 9.99666885201038,
"im": 0.0005729099528781564,
"mm": 0.0002864549764390782
},
],
"seq": 14
}

Notifications

This subscription feed publishes notifications to the client.

```
SubscribeNotifications();
```

Later, you can unsubscribe from Notifications, calling **UnSubscribeNotifications** method

```
UnSubscribeNotifications();
```

Response example from server

{
"feed":"notifications_auth",

```
  "notifications":[
    {
      "id":5,
      "type":"market",
      "priority":"low",
      "note":"A note describing the notification.",
      "effective_time":1520288300000
    },
    ...
  ]
}
```

API Kraken Futures | REST Public API

Connection

URL: https://futures.kraken.com/derivatives/api/v3

Kraken Futures Public API doesn't require any authentication.

Configuration

The only configuration is whether to enable a log for REST HTTP requests. Enable HTTPLogOptions if you want to save in a text file log all HTTP Requests/Responses

Events

OnKrakenHTTPException: this event is called if there is any exception doing an HTTP Request from REST Api.

Methods

GetFeeSchedules

This endpoint lists all fee schedules. Authentication is not required.

```
KrakenFutures.REST_API.GetFeeSchedules();
```

Order Book

This endpoint returns the entire non-cumulative order book of currently listed Futures contracts.

```
KrakenFutures.REST_API.GetFeeSchedules("PI_XBTUSD");
```

Tickers

This endpoint returns current market data for all currently listed Futures contracts and indices.

```
KrakenFutures.REST_API.GetTickers();
```

Instruments

This endpoint returns specifications for all currently listed Futures contracts and indices.

```
KrakenFutures.REST_API.GetInstruments();
```

History

This endpoint returns the last 100 trades from the specified lastTime value - if no value specified will return the last 100 trades. is endpoint only returns trade history for a maximum of 7 days from the time it is called or since last .trading engine release (whichever is sooner).

```
KrakenFutures.REST_API.GetHistory("PI_XBTUSD");
```

API Kraken Futures | REST Private API

Connection

URL: https://futures.kraken.com/derivatives/api/v3

Authentication

REST Private API requires an API Key and API Secret, these values are provided by Kraken in your account.

```
Kraken.ApiKey = "api key";
Kraken.ApiSecret = "api secret";
```

Methods

EditOrderByOrderId

This endpoint allows editing an existing order for a currently listed Futures contract.

aOrderId: ID of the order you wish to edit
aSize: The size associated with the order
aLimitPrice: The limit price associated with the order.
aStopPrice: The stop price associated with a stop order. Required if old Order Type is Stop.

```
KrakenFutures.REST_API.EditOrderByOrderId("Order_Id", 2, 1000);
```

EditOrderByCliOrderId

This endpoint allows editing an existing order for a currently listed Futures contract.

aCliOrderId: The order identity that is specified from the user. It must be globally unique.
aSize: The size associated with the order
aLimitPrice: The limit price associated with the order.
aStopPrice: The stop price associated with a stop order. Required if Order Type is Stop.

```
KrakenFutures.REST_API.EditOrderByCliOrderId("Cli_Order_Id", 2, 1000);
```

SendMarketOrder

This endpoint allows you to send a Market Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.

```
KrakenFutures.REST_API.SendMarketOrder(kosfBuy, "PI_XBTUSD", 1);
```

SendLimitOrder

This endpoint allows you to send a Limit Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.
aLimitPrice: The limit price associated with the order.

```
KrakenFutures.REST_API.SendLimitOrder(kosfBuy, "PI_XBTUSD", 1, 1000);
```

SendStopOrder

This endpoint allows you to send a Stop Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.
aStopPrice: The stop price associated with a stop order.
aLimitPrice: The limit price associated with the order.

```
KrakenFutures.REST_API.SendStopOrder(kosfBuy, "PI_XBTUSD", 1, 1000, 900);
```

SendTakeProfitOrder

This endpoint allows you to send a Take Profit Order.

aSide: The direction of the order: buy or sell.
aSymbol: The symbol of the futures
aSize: The size associated with the order.
aStopPrice: The stop price associated with a stop order.
aLimitPrice: The limit price associated with the order.

```
KrakenFutures.REST_API.SendTakeProfitOrder(kosfBuy, "PI_XBTUSD", 1, 1000, 900);
```

SendOrder

This endpoint allows sending a limit, stop, take profit or immediate-or-cancel order for a currently listed Futures contract.

OrderType: select one of the following kotfLMT, kotfPOST, kotfMKT, kotfSTP, kotfTAKE_PROFIT, kotfIOC
Symbol: The symbol of the futures
Side: The direction of the order (buy or sell).
Size: The size associated with the order.
StopPrice: The stop price associated with a stop order.
LimitPrice: The limit price associated with the order.
TriggerSignal: If placing a Stop or TakeProfit order, the signal used for trigger, select one of the following kots-Mark, kotsIndex, kotsLast
CliOrderId: The order identity that is specified from the user. It must be globally unique.
ReduceOnly: Set as true if you wish the order to only reduce an existing position. Any order which increases an existing position will be rejected. Default false.

```
TsgcHTTPKrakenFuturesOrder oOrder = new TsgcHTTPKrakenFuturesOrder();
oOrder.Side = TsgcHTTPKrakenFuturesOrderSide.kosfBuy;
oOrder.Symbol = "PI_XBTUSD";
oOrder.OrderType = TsgcHTTPKrakenFuturesOrderType.kotfMKT;
oOrder.Size = 1;
KrakenFutures.REST_API.SendOrder(oOrder);
```

CancelOrderByOrderId

This endpoint allows cancelling an open order for a Futures contract.

aOrderId: ID of the order you wish to edit

```
KrakenFutures.REST_API.CancelOrderByOrderId("Order_Id");
```

CancelOrderByCliOrderId

This endpoint allows cancelling an open order for a Futures contract.

aCliOrderId: The order identity that is specified from the user. It must be globally unique.

```
KrakenFutures.REST_API.CancelOrderByCliOrderId("Cli_Order_Id");
```

GetFills

This endpoint returns information on filled orders for all futures contracts.

aLastFillDate: If not provided, returns the last 100 fills in any futures contract. If provided, returns the 100 entries before lastFillTime.

```
KrakenFutures.REST_API.GetFills("2020-07-22T13:45:00.000Z");
```

Transfer

This endpoint allows you to transfer funds between two margin accounts with the same collateral currency, or between a margin account and your cash account.

aFromAccount: The name of the cash or margin account to move funds from.
aToAccount: The name of the cash or margin account to move funds to.
aUnit: The unit to transfer.
aAmount: The amount to transfer.

```
KrakenFutures.REST_API.Transfer("FI_XBTUSD", "cash", "xbt", 1.5);
```

GetOpenPositions

This endpoint returns the size and average entry price of all open positions in Futures contracts. This includes Futures contracts that have matured but have not yet been settled.

```
KrakenFutures.REST_API.GetOpenPositions();
```

GetNotifications

This endpoint provides the platform's notifications.

```
KrakenFutures.REST_API.GetNotifications();
```

GetAccounts

This endpoint returns key information relating to all your Kraken Futures accounts which may either be cash accounts or margin accounts. This includes digital asset balances, instrument balances, margin requirements, margin trigger estimates and auxiliary information such as available funds, PnL of open positions and portfolio value.

```
KrakenFutures.REST_API.GetAccounts();
```

CancelAllOrders

This endpoint allows cancelling an open order for a Futures contract.

Symbol: A futures product to cancel all open orders (optional)

```
KrakenFutures.REST_API.CancelAllOrders();
```

CancelAllOrdersAfter

This endpoint provides a Dead Man's Switch mechanism to protect the client from network malfunctions. The client can send a request with a timeout in seconds which will trigger a countdown timer that will cancel all client orders when timeout expires.

aTimeout: The timeout specified in seconds.

```
KrakenFutures.REST_API.CancelAllOrdersAfter(60);
```

GetOpenOrders

This endpoint returns information on all open orders for all Futures contracts.

```
KrakenFutures.REST_API.OpenOrders();
```

GetHistoricalOrders

This endpoint returns historical orders made on an account.

aSince: The DateTime Since
aBefore: The DateTime Before
aSort: "asc" for ascending sort "desc" for descending
aContinuationToken: Continuation token provided from a prior response which can be used in call to return the next set of available results

```
KrakenFutures.REST_API.GetHistoricalOrders(1604937694000, 1604937700000);
```

GetHistoricalTriggers

This endpoint returns allows historical triggers made on an account.

aSince: The DateTime Since
aBefore: The DateTime Before
aSort: "asc" for ascending sort "desc" for descending
aContinuationToken: Continuation token provided from a prior response which can be used in call to return the next set of available results

```
KrakenFutures.REST_API.GetHistoricalTriggers(1604937694000, 1604937700000);
```

GetHistoricalExecutions

This endpoint returns allows historical executions made on an account.

aSince: The DateTime Since
aBefore: The DateTime Before
aSort: "asc" for ascending sort "desc" for descending
aContinuationToken: Continuation token provided from a prior response which can be used in call to return the next set of available results

```
KrakenFutures.REST_API.GetHistoricalExecutions(1604937694000, 1604937700000);
```

WithdrawalToSpotWallet

This endpoint allows submitting a request to withdraw digital assets from a Kraken Futures wallet to your Kraken Spot wallet.

aCurrency: The digital asset that shall be withdrawn, e.g. xbt or xrp.
aAmount: The amount of currency that shall be withdrawn.

```
KrakenFutures.REST_API.WithdrawalToSpotWallet("xbt", 1000);
```

GetFeeScheduleVolumes

This endpoint returns your 30-day USD volume.

```
KrakenFutures.REST_API.GetFeeScheduleVolumes();
```

GetAccountLogCSV

This endpoint allows clients to download a csv file of their account logs.

```
KrakenFutures.REST_API.GetAccountLogCSV();
```

API Pusher

Pusher

Pusher is an easy and reliable platform with nice features based on WebSocket protocol: flexible pub/sub messaging, live user lists (presence), authentication...
Pusher WebSocket API is 7.
Data is sent bi-directionally over a WebSocket as text data containing UTF8 encoded JSON (Binary WebSocket frames are not supported).
You can call **Ping** method to test connection to the server. Essentially any messages received from the other party are considered to mean that the connection is alive. In the absence of any messages, either party may check that the other side is responding by sending a ping message, to which the other party should respond with a pong.
Before you connect, you must complete the following fields:

Important

Pusher requires that websocket client connects to a URL using previous fields (key, cluster...), these fields are used to build the url and this is done when you assign the client in pusher component. So, to be sure that URL is built correctly, set the client after you have filled the pusher configuration fields. Find below pseudo-code:

```
// configure pusher fields
pusher.cluster = ...
pusher.key = ...
// set client
pusher.client = websocket client
// start connection
websocket client.Active = true;
```

After a successful connection, **OnPusherConnect** event is raised and you get following fields:

- Socket ID: A unique identifier for the connected client.
- Timeout: The number of seconds of server inactivity after which the client should initiate a ping message (this is handled automatically by component).

In case of error, **OnPusherError** will be raised, and information about error provided. An error may be sent from Pusher in response to invalid authentication, an invalid command, etc.

4000-4099

Indicates an error resulting in the connection being closed by Pusher, and that attempting to reconnect using the same parameters will not succeed.

4000: Application only accepts SSL connections, reconnect using wss://
4001: Application does not exist
4003: Application disabled
4004: Application is over connection quota
4005: Path not found
4006: Invalid version string format
4007: Unsupported protocol version
4008: No protocol version supplied

4100-4199

Indicates an error resulting in the connection being closed by Pusher, and that the client may reconnect after 1s or more.

4100: Over capacity

4200-4299

Indicates an error resulting in the connection being closed by Pusher, and that the client may reconnect immediately.

4200: Generic reconnect immediately
4201: Pong reply not received: ping was sent to the client, but no reply was received - see ping and pong messages
4202: Closed after inactivity: The client has been inactive for a long time (currently 24 hours) and client does not support ping. Please upgrade to a newer WebSocket draft or implement version 5 or above of this protocol.

4300-4399

Any other type of error.

4301: Client event rejected due to rate limit

Channels

Channels are a fundamental concept in Pusher. Each application has a number of channels, and each client can choose which channels it subscribes to.
Channels provide:

- A way of filtering data. For example, in a chat application, there may be a channel for people who want to discuss 'dogs'
- A way of controlling access to different streams of information. For example, a project management application would want to authorise people to get updates about 'projectX'

It's strongly recommended that channels are used to filter your data and that it is not achieved using events. This is because all events published to a channel are sent to all subscribers, regardless of their event binding.
Channels don't need to be explicitly created and are instantiated on client demand. This means that creating a channel is easy. Just tell a client to subscribe to it.
The following types of channels are supported:

- **Public channels** can be subscribed to by anyone who knows their name
- **Private channels** introduce a mechanism which lets your server control access to the data you are broadcasting
- **Presence channels** are an extension of private channels. They let you register user information on subscription, and let other members of the channel know who's online
- **Cache channels** remember the last triggered event and send it as the first event to new subscribers (public, private and presence variants)
- **Private-Encrypted channels** provide end-to-end encryption using NaCl secretbox, ensuring that even Pusher cannot read the message data

Public Channels

Public channels should be used for publicly accessible data as they do not require any form of authorisation in order to be subscribed to.
You can subscribe and unsubscribe from channels at any time. There's no need to wait for the Pusher to finish connecting first.
Example: subscribe to channel "my-channel".
If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.
When Publish method is called and the channel is Public, the component instead of using the WebSocket protocol, uses the HTTP protocol and calls the method TriggerEvent (publish is not allowed using websocket protocol).

Private Channels

Requires Indy 10.5.7 or later
Private channels should be used when access to the channel needs to be restricted in some way. In order for a user to subscribe to a private channel permission must be authorised.
Example: subscribe to channel "my-private-channel".
If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.

Presence Channels

Requires Indy 10.5.7 or later
Presence channels build on the security of Private channels and expose the additional feature of an awareness of who is subscribed to that channel. This makes it extremely easy to build chat room and "who's online" type functionality to your application. Think chat rooms, collaborators on a document, people viewing the same web page, competitors in a game, that kind of thing.
Presence channels are subscribed to from the client API in the same way as private channels but the channel name must be prefixed with presence-. As with private channels an HTTP Request is made to a configurable authentication URL to determine if the current user has permissions to access the channel.
Information on users subscribing to, and unsubscribing from a channel can then be accessed by binding to events on the presence channel and the current state of users subscribed to the channel is available via the channel.members property.
Example: subscribe to channel "my-presence-channel".

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.

Cache Channels

A cache channel remembers the last triggered event, and sends this as the first event to new subscribers.
When an event is triggered on a cache channel, Pusher Channels caches this event, and when a client subscribes to a cache channel, if a cached value exists, this is sent to the client as the first event on that channel. This behavior helps developers to provide the initial state without adding additional logic to fetch it from elsewhere.
The following Cache Channels are supported:

- Public Cache Channel
- Private Cache Channel
- Presence Cache Channel

Example: subscribe to public cache channel "my-cache-channel".

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.
All messages from the subscribed channel will be received **OnPusherEvent** event.
If there is no cached event when subscribing to a cache channel, the **OnPusherCacheMiss** event will be raised, providing the channel name. This allows your application to handle the case where no cached data is available.

Private-Encrypted Channels

Private-Encrypted channels provide end-to-end encryption for messages. Like private channels, they require authentication, but additionally all data payloads are encrypted using NaCl secretbox so that only authorized subscribers can read the content. Even Pusher itself cannot decrypt the messages.
To use private-encrypted channels, you must provide a **SharedSecret** during authentication. The shared secret is used for encrypting and decrypting message data.
Example: subscribe to a private-encrypted channel "my-encrypted-channel".

A private-encrypted-cache variant is also available, combining encryption with cache channel behavior:

When using the **OnPusherAuthentication** event with private-encrypted channels, you can set the **SharedSecret** property on the response object to provide the encryption key:

Presence Events

Presence channels provide additional events that notify your application when users join or leave a channel, and allow you to track subscription counts.

OnPusherMemberAdded

Raised when a new member subscribes to a presence channel. Provides the channel name, user ID, and user info of the member that joined.

OnPusherMemberRemoved

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

OnPusherSubscriptionCount

Raised when the subscription count changes on a channel. Provides the channel name and the current number of subscribers. This event must be enabled on your Pusher dashboard.

OnPusherCacheMiss

Raised when subscribing to a cache channel that has no cached event. Provides the channel name. This allows your application to handle the case when no cached data is available, for example by fetching the data from another source.

Publish Messages

Not only you can receive messages from subscribed channels, but you can also send messages to other subscribed users.
Call method **Publish** to send a message to all subscribed users of channel.
Example: send an event to all subscribed users of "my-channel"

Publish no more than 10 messages per second per client (connection). Any events triggered above this rate limit will be rejected by Pusher API. This is not a system issue, it is a client issue. 100 clients in a channel sending messages at this rate would each also have to be processing 1,000 messages per second! Whilst some modern browsers might be able to handle this it's most probably not a good idea.

REST API

The API is hosted at http://api-CLUSTER.pusher.com , where CLUSTER is replaced with your own apps cluster (for instance, eu).
HTTP status codes are used to indicate the success or otherwise of requests. The following status are common:
200 Successful request. Body will contain a JSON hash of response data
400 Error: details in response body
401 Authentication error: response body will contain an explanation
403 Forbidden: app disabled or over message quota
The following REST API functions have been implemented.

Function	Description
TriggerEvent	Triggers a new event on the specified channel. Supports optional SocketId (to exclude a client) and Info parameters.
Trigger-BatchEvents	Triggers multiple events in a single HTTP request. Accepts a JSON array of event objects.
GetChannels	Provides a list of all active channels. Supports optional FilterByPrefix and Info parameters.
GetChannel	Provides information about a specific channel. Supports an optional Info parameter.
GetUsers	Provides a list of all users connected to a channel.
TerminateUser-Connections	Terminates all connections for a given user by their user ID.

TriggerEvent

Triggers an event on one or more channels. Requires the event name, channel name, and data payload.

Parameter	Description
aEventName	The name of the event to trigger.
aChannel	The channel name to trigger the event on.
aData	The event data (JSON string).
aSocketId (optional)	A socket ID to exclude from receiving the event. Useful to prevent the sender from receiving its own message.
aInfo (optional)	A comma-separated list of attributes to include in the response (e.g. "subscription_count").

TriggerBatchEvents

Triggers multiple events in a single API call, which is more efficient than making separate requests for each event. The batch parameter must be a JSON string containing an array of event objects, where each object has "channel", "name", and "data" fields.

GetChannels

Returns a list of active channels. Supports optional parameters to filter the results and request additional information.

Parameter	Description
aFilterByPrefix (optional)	Filter channels by a name prefix (e.g. "presence-" to list only presence channels).
aInfo (optional)	A comma-separated list of attributes to include in the response (e.g. "user_count").

GetChannel

Returns information about a specific channel.

Parameter	Description
aChannel	The channel name to get information about.
aInfo (optional)	A comma-separated list of attributes to include (e.g. "user_count,subscription_count").

GetUsers

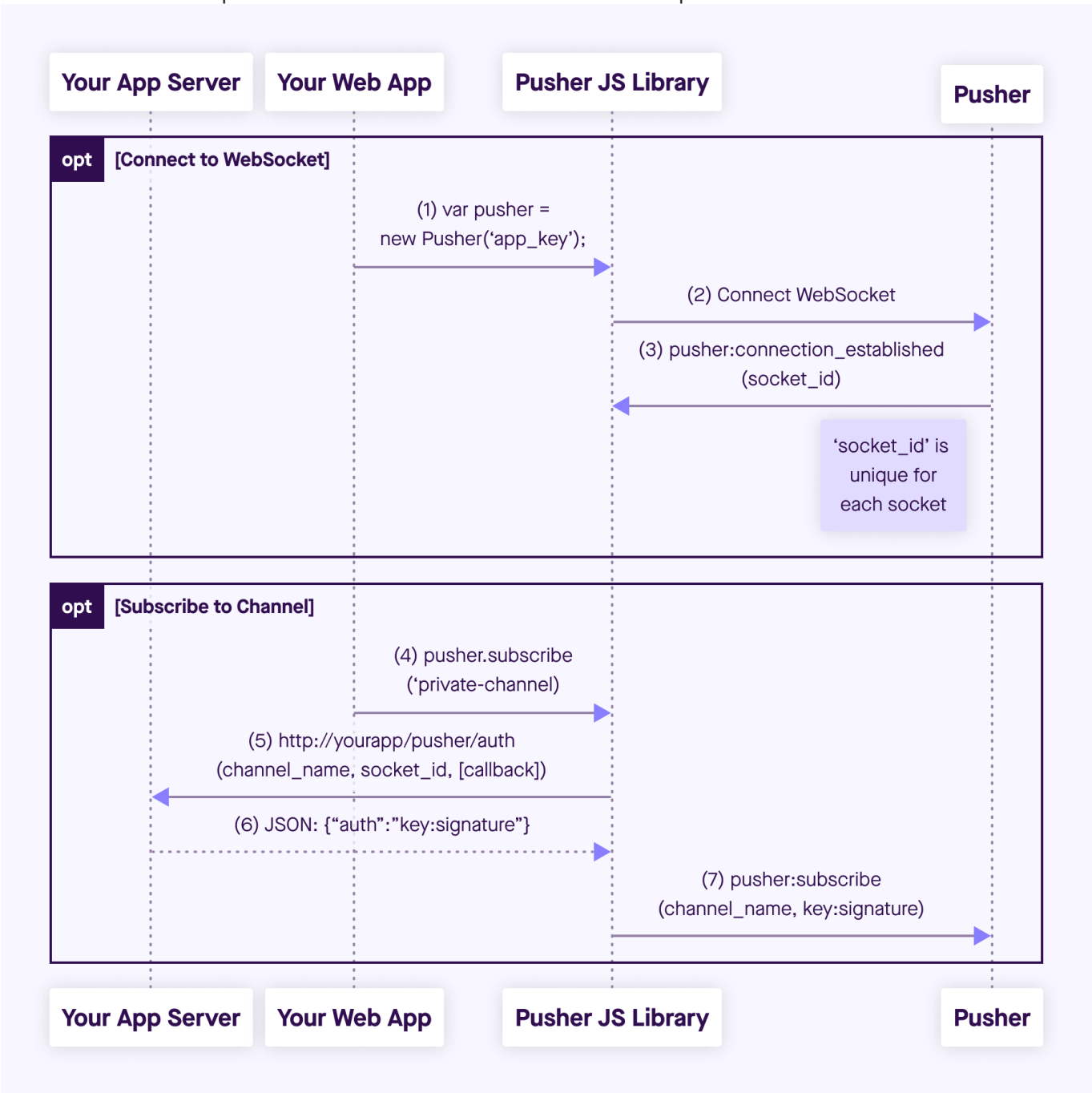
Returns a list of users connected to a presence channel. The channel name must include the full prefix (e.g. "presence-my-channel").

TerminateUserConnections

Terminates all connections established by a given user. This can be used to force a specific user to disconnect from all channels. The user ID must match the "user_id" used when the user subscribed to a presence channel.

Custom Authentication

Pusher only allows subscribing to private or presence channels, if the connection provides an authentication token, this allows you to restrict the access.
You can build your own Authentication flow, using **OnPusherAuthentication** event, this event is called before the subscription message is signed with the secret key provided by Pusher. This event has 2 parameters a request authentication with fields like SocketId, channel name... which can be used by your own authentication server to authenticate or not the request. Find below a screenshot which shows the pusher authentication flow



When a client connects to the pusher server, it sends the Key provided by pusher and the server returns an identification id (socket_id).
When a client subscribes to a private (or presence) channel, the sgcWebSockets client uses the Secret Key provided by pusher to create a signature which is included in the subscription message. Using the OnPusherAutentication event, you can capture the fields required to sign the message, implement your own authentication methods and if

successful, return the signature and this signature will be included in the subscription message and sent to the server.
Example:

The format of the signature is:
Private channels: key:HMAC256(SocketID, ChannelName)
Presence channels: key: HMAC256(SocketID, ChannelName, Data)
The **TsgcWSPusherResponseAuthentication** object provides the following properties:

Property	Description
Secret	The Pusher secret key used to compute the HMAC signature. Pre-filled with Pusher.Secret if configured.
Signature	The computed authentication signature. If left empty, the component will calculate it automatically using the Secret.
SharedSecret	The shared secret key for private-encrypted channels. Required when subscribing to pscPrivateEncryptedChannel or pscPrivateEncryptedCacheChannel. Used for end-to-end encryption of message data.

API Bitmex

Bitmex

Bitmex is a cryptocurrency exchange and derivative trading platform.

The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **REST API:** clients can request to server market and account data. Requires an API Key and Secret to authenticate and uses HTTPs as protocol.

Properties

Bitmex API has 2 types of methods: public and private. Public methods can be accessed without authentication, for example: get ticker prices. Some are private and related to user data; those methods require the use of Bitmex API keys.

- **ApiKey:** you can request a new api key in your Bitmex account, just copy the value to this property.
- **ApiSecret:** it's the secret of the API, keep safe.
- **TestNet:** if enabled it will connect to Bitmex Demo Account (by default false).
- **HTTPLogOptions:** stores in a text file a log of HTTP requests
  - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
  - **FileName:** full path of filename where logs will be stored

Most common uses

- **WebSockets API**
  - **How to Connect to WebSocket API**
  - **How to Subscribe to a WebSocket Channel**
- **REST API**
  - **How to Place a Bitmex Order**

WebSocket API

Subscribe / Unsubscribe

BitMEX allows subscribing to real-time data. This access is not rate-limited once connected and is the best way to get the most up-to-date data to your programs. In some topics, you can pass a Symbol to filter events by symbol, example: trades, quotes...

The following subscription topics are available without authentication:

- **btmAnnouncement**: Site Announcements
- **btmChat**: Trollbox chat
- **btmConnected**: Statistics of connected users/bots
- **btmFunding**: Updates of swap funding rates. Sent every funding interval (usually 8hrs)
- **btmInstrument**: Instrument updates including turnover and bid/ask
- **btmInsurance**: Daily Insurance Fund updates
- **btmLiquidation**: Liquidation orders as they're entered into the book
- **btmOrderBookL2_25**: Top 25 levels of level 2 order book
- **btmOrderBookL2**: Full level 2 order book
- **btmOrderBook10**: Top 10 levels using traditional full book push

- **btmPublicNotifications**: System-wide notifications (used for short-lived messages)
- **btmQuote**: Top level of the book
- **btmQuoteBin1m**: 1-minute quote bins
- **btmQuoteBin5m**: 5-minute quote bins
- **btmQuoteBin1h**: 1-hour quote bins
- **btmQuoteBin1d**: 1-day quote bins
- **btmSettlement**: Settlements
- **btmTrade**: Live trades
- **btmTradeBin1m**: 1-minute trade bins
- **btmTradeBin5m**: 5-minute trade bins
- **btmTradeBin1h**:1-hour trade bins
- **btmTradeBin1d**: 1-day trade bins

The following subjects require authentication:

- **btmAffiliate**: Affiliate status, such as total referred users & payout %
- **btmExecution**: Individual executions; can be multiple per order
- **btmOrder**: Live updates on your orders
- **btmMargin**: Updates on your current account balance and margin requirements
- **btmPosition**: Updates on your positions
- **btmPrivateNotifications**: Individual notifications - currently not used
- **btmTransact**: Deposit/Withdrawal updates
- **btmWallet**: Bitcoin address balance data, including total deposits & withdrawals

Example of messages received:

```
{
      "table":"orderBookL2_25",
      "keys":["symbol","id","side"],
      "types":{"id":"long","price":"float","side":"symbol","size":"long","symbol":"symbol"}
      "foreignKeys":{"side":"side","symbol":"instrument"},
      "attributes":{"id":"sorted","symbol":"grouped"},
      "action":"partial",
      "data":[
        {"symbol":"XBTUSD","id":17999992000,"side":"Sell","size":100,"price":80},
        {"symbol":"XBTUSD","id":17999993000,"side":"Sell","size":20,"price":70},
        {"symbol":"XBTUSD","id":17999994000,"side":"Sell","size":10,"price":60},
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy","size":10,"price":50},
        {"symbol":"XBTUSD","id":17999996000,"side":"Buy","size":20,"price":40},
        {"symbol":"XBTUSD","id":17999997000,"side":"Buy","size":100,"price":30}
      ]
}

{
      "table":"orderBookL2_25",
      "action":"update",
      "data":[
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy","size":5}
      ]
    }
{
      "table":"orderBookL2_25",
      "action":"delete",
      "data":[
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy"}
      ]
}
{
      "table":"orderBookL2_25",
      "action":"insert",
      "data":[
        {"symbol":"XBTUSD","id":17999995500,"side":"Buy","size":10,"price":45},
      ]
}
```

Authentication

If you wish to subscribe to user-locked streams, you must authenticate first. Note that invalid authentication will close the connection.

BitMEX API usage requires an API Key.

Permanent API Keys can be locked to IP address ranges and revoked at will without compromising your main credentials. They also do not require renewal.

To use API Key auth, you must generate an API Key in your account.

Call method **Authenticate** before subscribing to any Authenticated Topic.

CancelAllAfter (Dead Man's Switch)

The **CancelAllAfter** method implements the Dead Man's Switch feature. When called with a timeout value (in milliseconds), it instructs the server to cancel all open orders if no subsequent CancelAllAfter call is received within the timeout period. This is useful to ensure orders are canceled in case of network disconnection.

REST API

Method	Description
GetExecutions	This returns all raw transactions, which includes order opening and cancelation, and order status changes.
GetExecutionsTradeHistory	This returns more focused Transactions.
GetInstruments	This returns all instruments and indices, including those that have settled or are unlisted. Use this endpoint if you want to query for individual instruments or use a complex filter.
GetOrders	To get open orders only
PlaceOrder	Place a raw order using TsgcHTTPBitmexOrder object.
PlaceMarketOrder	Place a new MARKET order.
PlaceLimitOrder	Place a new LIMIT order.
PlaceStopOrder	Place a new STOP order.
PlaceStopLimitOrder	Place a new STOPLIMIT order.
AmendOrder	Modify an existing order.
CancelOrder	Cancels an active Order.
CancelAllOrders	Cancel All Active Orders.
CancelAllOrdersAfter	Cancel All Orders after some time.
ClosePosition	Close an open position.
GetOrderBook	Get Current OrderBook in vertical format
GetPosition	Get your positions.
SetPositionIsolate	Enable isolated margin or cross-margin per position.
SetPositionLeverage	Choose leverage per position.
SetPositionRiskLimit	Update your risk limit.
SetPositionTransferMargin	Transfer equity in or out of a position.
GetQuotes	Get Quotes
GetTrades	Get Trades
GetFunding	Get funding data.
GetInsurance	Get insurance fund data.
GetTradeBucketed	Get bucketed trade data (OHLCV) with configurable bin sizes.
GetQuoteBucketed	Get bucketed quote data with configurable bin sizes.
GetSettlement	Get settlement data.
GetLiquidation	Get liquidation orders.
GetInstrumentIndices	Get instrument indices.
GetInstrumentCompositeIndex	Get composite index data for instruments.
GetStats	Get exchange-wide statistics.
GetStatsHistory	Get historical exchange statistics.

GetStatsHistoryUSD	Get historical USD exchange statistics.
GetUserMargin	Get your account margin data.
GetUserWallet	Get your wallet information.
GetUserWalletHistory	Get your wallet transaction history.
GetUserWalletSummary	Get a summary of your wallet.

Bitmex | Connect WebSocket API

In order to connect to Bitmex WebSocket API, just create a new Bitmex API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bitmex oBitmex = new TsgcWSAPI_Bitmex();
oBitmex.Client = oClient;
oClient.Active = true;
```

Bitmex | Subscribe WebSocket Channel

Bitmex offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how subscribe to a Trade Channel:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bitmex oBitmex = new TsgcWSAPI_Bitmex();
oBitmex.Client = oClient;
oBitmex.Subscribe(btmTrade, "xbtusd");
void OnBitmexMessage(Sender: TObject; const aTopic: TwsBitmexTopics; const aMessage: string)
{
// here you will receive the tradeupdates
}
```

Bitmex | How to Place Orders

The Bitmex REST API offer public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Bitmex account and create a new API, you will get the following values:

- ApiKey
- ApiSecret

These fields must be configured in the Bitmex property of the Bitmex API client component. Once configured, you can start to do private requests to the Bitmex REST API.

Order Types

All orders require a symbol. All other fields are optional except when otherwise specified.
These are the valid ordTypes:

- **Limit:** The default order type. Specify an orderQty and price.
- **Market:** A traditional Market order. A Market order will execute until filled or your bankruptcy price is reached, at which point it will cancel.
- **Stop:** A Stop Market order. Specify an orderQty and stopPx. When the stopPx is reached, the order will be entered into the book.
    - On sell orders, the order will trigger if the triggering price is lower than the stopPx. On buys, higher.
    - Note: Stop orders do not consume margin until triggered. Be sure that the required margin is available in your account so that it may trigger fully.
    - Close Stops don't require an orderQty. See Execution Instructions below.
- **StopLimit:** Like a Stop Market, but enters a Limit order instead of a Market order. Specify an orderQty, stopPx, and price.
- **MarketIfTouched:** Similar to a Stop, but triggers are done in the opposite direction. Useful for Take Profit orders.
- **LimitIfTouched:** As above; use for Take Profit Limit orders.
- **Pegged:** Pegged orders allow users to submit a limit price relative to the current market price. Specify a pegPriceType, and pegOffsetValue.
    - Pegged orders must have an execInst of Fixed. This means the limit price is set at the time the order is accepted and does not change as the reference price changes.
    - PrimaryPeg: Price is set relative to near touch price.
    - MarketPeg: Price is set relative to far touch price.
    - A pegPriceType submitted with no ordType is treated as a Pegged order.

Execution Instructions

The following execInsts are supported. If using multiple, separate with a comma (e.g. LastPrice,Close).

- **ParticipateDoNotInitiate:** Also known as a Post-Only order. If this order would have executed on placement, it will cancel instead. This is intended to protect you from the far touch moving towards you while the order is in transit. It is not intended for speculating on the far touch moving away after submission - we consider such behaviour abusive and monitor for it.
- **MarkPrice, LastPrice, IndexPrice:** Used by stop and if-touched orders to determine the triggering price. Use only one. By default, MarkPrice is used. Also used for Pegged orders to define the value of LastPeg.
- **ReduceOnly:** A ReduceOnly order can only reduce your position, not increase it. If you have a ReduceOnly limit order that rests in the order book while the position is reduced by other orders, then its order quantity will be amended down or canceled. If there are multiple ReduceOnly orders the least aggressive will be amended first.
- **Close:** Close implies ReduceOnly. A Close order will cancel other active limit orders with the same side and symbol if the open quantity exceeds the current position. This is useful for stops: by canceling these orders, a Close Stop is ensured to have the margin required to execute, and can only execute up to the full size of your position. If orderQty is not specified, a Close order has an orderQty equal to your current position's size.

    - Note that a Close order without an orderQty requires a side, so that BitMEX knows if it should trigger above or below the stopPx.
- **LastWithinMark:** Used by stop orders with LastPrice to allow stop triggers only when:
    - For Sell Stop Market / Stop Limit Order
        - Last Price <= Stop Price
        - Last Price >= Mark Price × (1 - 5%)
    - For Buy Stop Market / Stop Limit Order:
        - Last Price >= Stop Price
        - Last Price <= Mark Price × (1 + 5%)
- **Fixed:** Pegged orders must have an execInst of Fixed. This means the limit price is set at the time the order is accepted and does not change as the reference price changes.

Pegged Orders

Pegged orders allow users to submit a limit price relative to the current market price. The limit price is set once when the order is submitted and does not change with the reference price. This order type is not intended for speculating on the far touch moving away after submission - we consider such behaviour abusive and monitor for it.

Pegged orders have an ordType of Pegged, and an execInst of Fixed.

A pegPriceType and pegOffsetValue must also be submitted:

- PrimaryPeg - price is set relative to the near touch price
- MarketPeg - price is set relative to the far touch price

Trailing Stop Pegged Orders

Use pegPriceType of TrailingStopPeg to create Trailing Stops.

The price is set at submission and updates once per second if the underlying price (last/mark/index) has moved by more than 0.1%. stopPx then moves as the market moves away from the peg, and freezes as the market moves toward it.

Use pegOffsetValue to set the stopPx of your order. The peg is set to the triggering price specified in the execInst (default MarkPrice). Use a negative offset for stop-sell and buy-if-touched orders.

Requires ordType: Stop, StopLimit, MarketIfTouched, LimitIfTouched.

Trailing Stops

You may use pegPriceType of 'TrailingStopPeg' to create Trailing Stops. The pegged stopPx will move as the market moves away from the peg, and freeze as the market moves toward it.

To use, combine with pegOffsetValue to set the stopPx of your order. The peg is set to the triggering price specified in the execInst (default 'MarkPrice'). Use a negative offset for stop-sell and buy-if-touched orders.

Requires ordType: 'Stop', 'StopLimit', 'MarketIfTouched', 'LimitIfTouched'.

Tracking Your Orders

If you want to keep track of order IDs yourself, set a unique clOrdID per order. This clOrdID will come back as a property on the order and any related executions (including on the WebSocket), and can be used to get or cancel the order. Max length is 36 characters.

Examples:

```
// buy market order
BITMEX.REST_API.PlaceMarketOrder(bmosBuy, "XBTUSD", 100);
// sell limit order at 45000
BITMEX.REST_API.PlaceLimitOrder(bmosSell, "XBTUSD", 100, 45000.00);
// stop order at 48000
BITMEX.REST_API.PlaceStopOrder(bmosSell, "XBTUSD", 100, 48000.00);
```

API Bitfinex

Bitfinex

Bitfinex is one of the world's largest and most advanced cryptocurrency trading platform. Users can exchange Bitcoin, Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Classic...

Bitfinex WebSocket API version is 2.0

Each message sent and received via the Bitfinex's WebSocket channel is encoded in JSON format

A symbol can be a trading pair or a margin currency:

- Trading pairs symbols are formed prepending a "t" before the pair (i.e tBTCUSD, tETHUSD).
- Margin currencies symbols are formed prepending an "f" before the currency (i.e fUSD, fBTC, ...)

After a successful connection, **OnBitfinexConnect** event is raised and you get Bitfinex API Version number as a parameter.

You can call **Ping** method to test connection to the server.

If the server sends any information, this can be handled using **OnBitfinexInfoMessage** event, where a Code and a Message are parameters with information about the message sent by the server. Example codes:

20051 : Stop/Restart WebSocket Server (please reconnect)
20060 : Entering in Maintenance mode. Please pause any activity and resume after receiving the info message 20061 (it should take 120 seconds at most).
20061 : Maintenance ended. You can resume normal activity. It is advised to unsubscribe/subscribe again all channels.

In case of error, **OnBitfinexError** will be raised, and information about error provided. Example error codes:

10000 : Unknown event
10001 : Unknown pair

In order to change the configuration, call **Configuration** method and pass as a parameter one of the following flags:

CS_DEC_S = 8; // Enable all decimal as strings.
CS_TIME_S = 32; // Enable all times as date strings.
CS_SEQ_ALL = 65536; // Enable sequencing BETA FEATURE
CHECKSUM = 131072; // Enable checksum for every book iteration. Checks the top 25 entries for each side of the book. The checksum is a signed int.

Subscribe Public Channels

There are channels which are public and there is no need to authenticate against the server. All messages are raised **OnBitfinexUpdate** event.

SubscribeTicker

The ticker is a high level overview of the state of the market. It shows you the current best bid and ask, as well as the last trade price. It also includes information such as daily volume and how much the price has moved over the last day.

```
// Trading pairs
[
  CHANNEL_ID,
  [
    BID,
    BID_SIZE,
    ASK,
    ASK_SIZE,
    DAILY_CHANGE,
    DAILY_CHANGE_PERC,
    LAST_PRICE,
    VOLUME,
    HIGH,
    LOW
  ]
]
// Funding pairs
[
  CHANNEL_ID,
  [
    FRR,
    BID,
    BID_PERIOD,
    BID_SIZE,
    ASK,
    ASK_PERIOD,
    ASK_SIZE,
    DAILY_CHANGE,
    DAILY_CHANGE_PERC,
    LAST_PRICE,
    VOLUME,
    HIGH,
    LOW
  ]
]
```

SubscribeTrades

This channel sends a trade message whenever a trade occurs at Bitfinex. It includes all the pertinent details of the trade, such as price, size and time.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      ID,
      MTS,
      AMOUNT,
      PRICE
    ],
    ...
  ]
]
// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      ID,
      MTS,
      AMOUNT,
      RATE,
      PERIOD
    ],
    ...
  ]
]
```

SubscribeOrderBook

The Order Books channel allows you to keep track of the state of the Bitfinex order book. It is provided on a price aggregated basis, with customizable precision. After receiving the response, you will receive a snapshot of the book, followed by updates upon any changes to the book.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      PRICE,
      COUNT,
      AMOUNT
    ],
    ...
  ]
]

// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      RATE,
      PERIOD,
      COUNT,
      AMOUNT
    ],
    ...
  ]
]
```

SubscribeRawOrderBook

These are the most granular books.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      ORDER_ID,
      PRICE,
      AMOUNT
    ],
    ...
  ]
]

// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      OFFER_ID,
      PERIOD,
      RATE,
      AMOUNT
    ],
    ...
  ]
]
```

SubscribeCandles

Provides a way to access charting candle info. Time Frames:

1m: one minute
5m : five minutes
15m : 15 minutes
30m : 30 minutes
1h : one hour
3h : 3 hours
6h : 6 hours
12h : 12 hours
1D : one day
7D : one week
14D : two weeks
1M : one month

```
[
  CHANNEL_ID,
  [
    [
      MTS,
      OPEN,
      CLOSE,
      HIGH,
      LOW,
      VOLUME
    ],
    ...
  ]
]
```

Subscribe Authenticated Channels

This channel allows you to keep up to date with the status of your account. You can receive updates on your positions, your balances, your orders and your trades.

Use **Authenticate** method in order to Authenticate against the server and set required parameters.

Once authenticated, you will receive updates of: Orders, positions, trades, funding offers, funding credits, funding loans, wallets, balance info, margin info, funding info, funding trades...

You can request **UnAuthenticate** method if you want to log off from the server.

API Kucoin

Kucoin

Kucoin is an international multi-language cryptocurrency exchange. It offers some APIs to access Kucoin data. The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **REST API:** clients can request to server market and account data. Requires an API Key, Secret and Passphrase to authenticate and uses HTTPs as protocol.

Properties

Kucoin API has 2 types of methods: public and private. Public methods can be accessed without authentication, example: get ticker prices. Private methods related to user data require the use of Kucoin API keys.

- **ApiKey:** you can request a new api key in your kucoin account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **Passphrase:** string required to connect to Kucoin Servers.
- **Sandbox:** if enabled it will connect to Kucoin Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.

Most common uses

- **WebSockets API**
    - **How to Connect to WebSocket API**
    - **How to Subscribe to a WebSocket Channel**
- **REST API**
    - **How to Get Market Data**
    - **How to Use Private REST API**
    - **How to Trade Spot**
    - **Private Requests Time**

WebSocket Feed

To subscribe channel messages from a certain server, the client side should send subscription message to the server.
If the subscription succeeds, the system will send ack messages to you, when the response is set as true.

```
{
   "id":"1545910660739",
   "type":"ack"
}
```

While there are topic messages generated, the system will send the corresponding messages to the client side.

The following Subscription / Unsubscription methods are supported.

Public Channels

Method	Parameters	Description
SubscribeSymbolTicker	Symbol	Subscribe to this topic to get the push of BBO changes. If there is no change within one second, it will not be pushed. It will be pushed per 100ms with the newest BBO. If there was no change compared with last data, it will not be pushed.
SubscribeAllSymbolsTicker		Subscribe to this topic to get the push of all market symbols BBO change.
SubscribeSymbolSnapshot	Symbol	Subscribe to get snapshot data for a single symbol. The snapshot data is pushed at 2 seconds intervals.
SubscribeMarketSnapshot	Market	Subscribe this topic to get the snapshot data for the entire market. The snapshot data is pushed at 2 seconds intervals.
SubscribeLevel2MarketData	Symbol	Subscribe to this topic to get Level2 order book data. When the websocket subscription is successful, the system would send the increment change data pushed by the websocket to you.
SubscribeLevel2_5BestAskBid	Symbol	The system will return the 5 best ask/bid orders data, which is the snapshot data of every 100 milliseconds (in other words, the 5 best ask/bid orders data returned every 100 milliseconds in real-time).
SubscribeLevel2_50BestAskBid	Symbol	The system will return the 50 best ask/bid orders data, which is the snapshot data of every 100 milliseconds (in other words, the 50 best ask/bid orders data returned every 100 milliseconds in real-time).
SubscribeKlines	Symbol	Subscribe to this topic to get K-Line data.
SubscribeMatchExecutionData	Symbol	Subscribe to this topic to get the matching event data flow of Level 3. For each order traded, the system would send you the match messages in the following format.
SubscribeIndexPrice	Symbol	Subscribe to this topic to get the index price for the margin trading.
SubscribeMarkPrice	Symbol	Subscribe to this topic to get the mark price for margin trading.
SubscribeOrderBookChanged	Symbol	Subscribe to this topic to get the order book changes on margin trade.
SubscribeLevel1	Symbol	Subscribe to Level 1 best bid/ask data for a symbol.

If ACK parameter is sent to true, after a successful subscription / unsubscription, client receives a message about it.

Private Channels

Requires a valid ApiKey obtained from your Kucoin account. The ApiKey, ApiSecret and Passphrase must be set in the Kucoin property of the client API component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
SubscribeTradeOrders	This topic will push all change events of your orders.
SubscribeAccountBalance	You will receive this message when an account balance changes. The message contains the details of the change.
SubscribePositionStatus	The system will push the change event when the position status changes.
SubscribeMarginTradeOrders	The system will push this message to the lenders when the order enters the order book.
SubscribeStopOrder	When a stop order is received by the system, you will receive a message with "open" type. It means that this order entered the system and waited to be triggered.
SubscribeTradeOrdersV2	Subscribe to trade orders V2 channel for enhanced order update notifications.

SubscribeCrossMargin-Position	Subscribe to cross margin position updates. The system will push the change event when the cross margin position changes.
SubscribeIsolatedMargin-Position	Subscribe to isolated margin position updates. The system will push the change event when the isolated margin position changes.

REST API

All endpoints return either a JSON object or array.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
GetServiceSta-tus		Test connectivity to the Rest API and get the Service Status
GetServerTime		Test connectivity to the Rest API and get the current server time.

Market Data EndPoints

Method	Parameters	Description
GetSymbolList	Market	Request via this endpoint to get a list of available currency pairs for trading. If you want to get the market information of the trad-ing symbol
GetTicker	Symbol	Request via this endpoint to get Level 1 Market Data. The re-turned value includes the best bid price and size, the best ask price and size as well as the last traded price and the last trad-ed size.
GetAllTickers		Request market tickers for all the trading pairs in the market (in-cluding 24h volume).
Get24hrStats	Symbol	Request via this endpoint to get the statistics of the specified ticker in the last 24 hours.
GetMarketList		Request via this endpoint to get the transaction currency for the entire trading market.
GetPartOrder-Book20	Symbol	Request via this endpoint to get a list of open orders for a sym-bol. Level-2 order book includes all bids and asks (aggregated by price), this level returns only one size for each active price (as if there was only a single order for that price). The system will return you 20 pieces of data (ask and bid data) on the order book.
GetPartOrder-Book100	Symbol	Request via this endpoint to get a list of open orders for a sym-bol. Level-2 order book includes all bids and asks (aggregated by price), this level returns only one size for each active price (as if there was only a single order for that price). The system will return you 100 pieces of data (ask and bid data) on the or-der book.
GetFullOrder-Book	Symbol	Request via this endpoint to get the order book of the specified symbol. Level 2 order book includes all bids and asks (aggre-gated by price). This level returns only one aggregated size for each price (as if there was only one single order for that price). This API will return data with full depth.

GetKLines	Symbol	Request via this endpoint to get the kline of the specified symbol. Data are returned in grouped buckets based on requested type.
GetCurrencies		Request via this endpoint to get the currency list.
GetCurrencyDetail	Currency	Request via this endpoint to get the currency details of a specified currency
GetFiatPrice		Request via this endpoint to get the currency details of a specified currency
GetPartOrderBook1	Symbol	Request via this endpoint to get the Level 1 best bid/ask for a symbol.

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

User EndPoints

Method	Parameters	Description
GetAllSubAccounts		You can get the user info of all sub-users via this interface.
GetListAccounts		Get a list of accounts.
GetAccount	AccountId	Information for a single account. Use this endpoint when you know the accountId.
GetAccountBalanceSubAccount	SubUserId	This endpoint returns the account info of a sub-user specified by the subUserId.
InnerTransfer		This API endpoint can be used to transfer funds between accounts internally. Users can transfer funds between their main account, trading account, cross margin account, and isolated margin account free of charge. Transfer of funds from the main account, cross margin account, and trading account to the futures account is supported, but transfer of funds from futures accounts to other accounts is not supported.
GetDepositAddresses	Currency	Get deposit addresses for a currency.
CreateDepositAddress	Currency	Create a new deposit address for a currency.
GetDepositList		Get deposit history.
GetAccountLedgers		Get account ledger entries.
GetTradeFees	Symbols	Get trade fees for the specified symbols.

Withdraw EndPoints

Method	Parameters	Description
GetWithdrawalsList		Get a list of the Withdrawals.
GetHistoricalWithdrawalsList		List of KuCoin V1 historical withdrawals.
GetWithdrawalsQuotas	Currency	Get Withdrawals Quotas
ApplyWithdraw	Currency, Address, Amount	Create a Withdraw

CancelWithdraw	WithdrawalId	Only withdrawals requests of PROCESSING status could be canceled.

Trade Endpoints

Method	Parameters	Description
PlaceOrder		You can place two types of orders: limit and market. Orders can only be placed if your account has sufficient funds. Once an order is placed, your account funds will be put on hold for the duration of the order. How much and which funds are put on hold depends on the order type and parameters specified
PlaceMarketOrder		Places a Market Order.
PlaceLimitOrder		Places a Limit Order.
PlaceMarginOrder		Places a Margin Order.
CancelOrder		Cancels an Order by Order Id.
CancelOrderByClientOid		Cancels an Order by Client Order Id.
CancelAllOrders		Cancel all open orders.
ListOrders		Request via this endpoint to get your current order list. Items are paginated and sorted to show the latest first
GetRecentOrders		Request via this endpoint to get 1000 orders in the last 24 hours.
GetOrder		Request via this endpoint to get a single order info by order ID.
GetOrderByClientOid		Request via this endpoint to get a single order info by Client order ID.
ListFills		Request via this endpoint to get the recent fills.
GetRecentFills		Request via this endpoint to get a list of 1000 fills in the last 24 hours.
PlaceStopOrder		Places a Stop Order.
PlaceStopMarketOrder		Places a Stop Market Order.
PlaceStopLimitOrder		Places a Stop Limit Order.
CancelStopOrder		Cancels a Open Stop Order by Order Id
CancelStopOrderByClientOid		Cancels a Open Stop Order by Client Order Id
CancelAllStopOrders		Cancel All Stop Orders
GetStopOrder		Request via this interface to get a stop order information via the order ID.
GetStopOrderByClientOid		Request via this interface to get a stop order information via the Client order ID.
ListStopOrders		Request via this endpoint to get your current untriggered stop order list. Items are paginated and sorted to show the latest first.
PlaceHFOrder		Place a high-frequency order.
CancelHFOrder		Cancel a high-frequency order by order ID.
CancelHFOrderByClientOid		Cancel a high-frequency order by client order ID.
CancelAllHFOrders		Cancel all high-frequency orders.
GetHFActiveOrders		Get active high-frequency orders.

GetHFDone-Orders		Get completed high-frequency orders.
GetHFOrder		Get a specific high-frequency order by order ID.

Events

Kucoin Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Kucoin server.
OnDisconnect
After a disconnection from Kucoin server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Kucoin API Component, called **OnKucoinHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket Feeds).

Kucoin | Connect WebSocket API

In order to connect to Kucoin WebSocket API, just create a new Kucoin API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
oKucoin.Client = oClient;
oClient.Active = true;
```

Kucoin | Subscribe WebSocket Channel

Kucoin offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how to subscribe to a Ticker:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
oKucoin.Client = oClient;
oKucoin.SubscribeSymbolTicker("BTC-USDT");
void OnMessage(TsgcWSConnection Connection, const string aText)
{
// here you will receive the ticker updates
}
```

Kucoin | Get Market Data

Kucoin offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get the snapshot of the ticker BTC-USDT, do the following call

```
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
MessageBox.Show(oKucoin.REST_API.GetTicker("

BTC-USDT

"));
```

Kucoin | Private REST API

The Kucoin REST API offer public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Kucoin account and create a new API, you will get the following values:

- ApiKey
- ApiSecret
- Passphrase

These fields must be configured in the Kucoin property of the Kucoin API client component.
Once configured, you can start to do private requests to the Kucoin Pro REST API

*Private Requests, require that your local machine has the local time synchronized, if not, the requests will be rejected by Kucoin server. Check the following article about this, Kucoin Private Requests Time.

```
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
oKucoin.Kucoin.ApiKey = "<your api key>";
oKucoin.Kucoin.ApiSecret = "<your api secret>";
oKucoin.Kucoin.Passphrase = "<your passphrase>";
MessageBox.Show(oKucoin.REST_API.GetListAccounts());
```

Kucoin | Trade Spot

Kucoin allows you to trade spot using its REST API.

Configuration

First you must create an **API Key** in your Kucoin account and add privileges to trading with Spot.
Once this is done, you can start spot trading.
First, **set your ApiKey, ApiSecret and Passphrase** in the Kucoin Client Component, this will be used to sign the requests sent to Kucoin server.

Place an Order

To place a new order, just call to method **REST_API.PlaceOrder** of Kucoin Client Component.

Depending on the type of the order (market, limit...) the API requires more or less fields.

Parameters

Param	type	Description
clientOid	String	Unique order id created by users to identify their orders, e.g. UUID.
side	String	**buy** or **sell**
symbol	String	a valid trading symbol code. e.g. ETH-BTC
type	String	*[Optional]* **limit** or **market** (default is **limit**)
remark	String	*[Optional]* remark for the order, length cannot exceed 100 utf8 characters
stp	String	*[Optional]* self trade prevention , **CN**, **CO**, **CB** or **DC**
trade-Type	String	*[Optional]* The type of trading : **TRADE**（Spot Trade）, **MARGIN_TRADE** (Margin Trade). Default is **TRADE**. Note: To improve the system performance and to accelerate order placing and processing, KuCoin has added a new interface for order placing of margin. For traders still using the current interface, please move to the new one as soon as possible. The current one will no longer accept margin orders by May 1st, 2021 (UTC). At the time, KuCoin will notify users via the announcement, please pay attention to it.

LIMIT ORDER PARAMETERS

Param	type	Description
price	String	price per base currency
size	String	amount of base currency to buy or sell
timeInForce	String	*[Optional]* **GTC**, **GTT**, **IOC**, or **FOK** (default is **GTC**), read Time In Force.
cancelAfter	long	*[Optional]* cancel after **n** seconds, requires **timeInForce** to be **GTT**
postOnly	boolean	*[Optional]* Post only flag, invalid when **timeInForce** is **IOC** or **FOK**
hidden	boolean	*[Optional]* Order will not be displayed in the order book
iceberg	boolean	*[Optional]* Only a portion of the order is displayed in the order book
visibleSize	String	*[Optional]* The maximum visible size of an iceberg order

MARKET ORDER PARAMETERS

Param	type	Description
size	String	*[Optional]* Desired amount in base currency
funds	String	*[Optional]* The desired amount of quote currency to use

When you send an order, there are 2 possibilities:

1. **Successful:** the function PlaceOrder returns the message sent by Kucoin server.
2. **Error:** the exception is returned in the event OnKucoinHTTPException.

Place Market Order 1 BTC-USDT

```
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
oKucoin.Kucoin.ApiKey = "<api key>";
oKucoin.Kucoin.ApiSecret = "<api secret>";
oKucoin.Kucoin.Passphrase = "<passphrase>";
MessageBox.Show(oKucoin.REST_API.

PlaceMarketOrder

(

kosBuy, "BTC-USDT", 1

));
```

Place Limit Order 1 BTC-USDT at 40000

```
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
oKucoin.Kucoin.ApiKey = "<api key>";
oKucoin.Kucoin.ApiSecret = "<api secret>";
oKucoin.Kucoin.Passphrase = "<passphrase>";
MessageBox.Show(oKucoin.REST_API.

PlaceLimitOrder

(

kosBuy, "BTC-USDT", 1, 40000

));
```

Kucoin | Private Requests Time

When you do a private request to Kucoin, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 5 seconds with Kucoin servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

You can check the Kucoin server time, calling method **GetServerTime**, which will return the time of the Kucoin server

API Kucoin Futures

Kucoin Futures

Kucoin is an international multi-language cryptocurrency exchange. It offers some APIs to access Kucoin data. The following APIs are supported:

1. **WebSocket streams:** allows you to subscribe to some methods and get data in real-time. Events are pushed to clients by server to subscribers. Uses WebSocket as protocol.
2. **REST API:** clients can request to server market and account data. Requires an API Key, Secret and Passphrase to authenticate and uses HTTPs as protocol.

Properties

Kucoin API has 2 types of methods: public and private. Public methods can be accessed without authentication, example: get ticker prices. Private and related to user data methods require the use of Kucoin API keys.

- **ApiKey:** you can request a new api key in your kucoin account, just copy the value to this property.
- **ApiSecret:** API secret is only required for REST_API, websocket api only requires ApiKey for some methods.
- **Passphrase:** string required to connect to Kucoin Servers.
- **Sandbox:** if enabled it will connect to Kucoin Demo Account (by default false).
    - **HTTPLogOptions:** stores in a text file a log of HTTP requests
        - **Enabled:** if enabled, will store all HTTP requests of WebSocket API.
        - **FileName:** full path of filename where logs will be stored
        - **REST:** stores in a text file a log of REST API requests
            - **Enabled:** if enabled, will store all HTTP Requests of REST API.
            - **FileName:** full path of filename where logs will be stored.

Most common uses

- **WebSockets API**
    - **How to Connect to WebSocket API**
    - **How to Subscribe to a WebSocket Channel**
- **REST API**
    - **How to Get Market Data**
    - **How to Use Private REST API**
    - **How to Trade Futures**
    - **Private Requests Time**

WebSocket Feed

To subscribe channel messages from a certain server, the client side should send subscription message to the server.
If the subscription succeeds, the system will send ack messages to you, when the response is set as true.

```
{
   "id":"1545910660739",
   "type":"ack"
}
```

While there are topic messages generated, the system will send the corresponding messages to the client side.

The following Subscription / Unsubscription methods are supported.

Public Channels

Method	Parameters	Description
SubscribeSymbolTickerV2	Symbol	Subscribe this topic to get the realtime push of BBO changes. After subscription, when there are changes in the order book, the system will push the real-time ticker symbol information to you. It is recommended to use the new topic for timely information.
SubscribeSymbolTicker	Symbol	Subscribe this topic to get the realtime push of BBO changes. The ticker channel provides real-time price updates whenever a match happens. If multiple orders are matched at the same time, only the last matching event will be pushed.
SubscribeLevel2MarketData	Symbol	Subscribe this topic to get Level 2 order book data.
SubscribeExecutionData	Symbol	For each order executed, the system will send you the match messages in the format as following.
SubscribeLevel2_5BestAskBid	Symbol	Returned for every 100 milliseconds at most.
SubscribeLevel2_50BestAskBid	Symbol	Returned for every 100 milliseconds at most.
SubscribeContractMarketData	Symbol	Subscribe this topic to get the market data of the contract.
SubscribeSystemAnnouncements	Symbol	Subscribe this topic to get the system announcements.
SubscribeTransactionStatistics	Symbol	The transaction statistics will be pushed to users every 5 seconds.
SubscribeKlines	Symbol	Subscribe to contract klines (candlestick) data.
SubscribeFundingFeeSettlement	Symbol	Subscribe to funding fee settlement notifications.

If ACK parameter is sent to true, after a successful subscription / unsubscription, client receives a message about it.

Private Channels

Requires a valid ApiKey obtained from your Kucoin account. The ApiKey, ApiSecret and Passphrase must be set in the Kucoin property of the client API component.

The following data is pushed to client every time there is a change. There is no need to subscribe to any method, this is done automatically if you set a valid ApiKey.

Method	Description
SubscribeTradeOrders	This topic will push all change events of your orders.
SubscribeAccountBalance	You will receive this message when an account balance changes. The message contains the details of the change.
SubscribePositionChange	The system will push the change event when the position status changes.
SubscribeStopOrder	When a stop order is received by the system, you will receive a message with "open" type. It means that this order entered the system and waited to be triggered.
SubscribeMarginMode	Subscribe to margin mode changes. The system will push the change event when the margin mode is updated.
SubscribeCrossMarginLeverage	Subscribe to cross margin leverage changes. The system will push the change event when the cross margin leverage is updated.

REST API

All endpoints return either a JSON object or array.

Public API EndPoints

These endpoints can be accessed without any authorization.

General EndPoints

Method	Parameters	Description
GetServiceStatus		Test connectivity to the Rest API and get the Service Status
GetServerTime		Test connectivity to the Rest API and get the current server time.

Market Data EndPoints

Method	Parameters	Description
GetOpenContractList		Submit request to get the info of all open contracts.
GetOrderInfoContract		Submit request to get info of the specified contract.
GetTicker	Symbol	The real-time ticker includes the last traded price, the last traded size, transaction ID, the side of liquidity taker, the best bid price and size, the best ask price and size as well as the transaction time of the orders. These messages can also be obtained through Websocket. The Sequence Number is used to judge whether the messages pushed by Websocket is continuous.
GetPartOrderBook20	Symbol	Get a snapshot of aggregated open orders for a symbol.
GetPartOrderBook100	Symbol	Get a snapshot of aggregated open orders for a symbol.
GetFullOrderBook	Symbol	Get a snapshot of aggregated open orders for a symbol.
GetLevel2PullingMessages	Symbol	If the messages pushed by Websocket is not continuous, you can submit the following request and re-pull the data to ensure that the sequence is not missing. In the request, the start parameter is the sequence number of your last received message plus 1, and the end parameter is the sequence number of your current received message minus 1. After re-pulling the messages and applying them to your local exchange order book, you can continue to update the order book via Websocket incremental feed. If the difference between the end and start parameter is more than 500, please stop using this request and we suggest you to rebuild the Level 2 orderbook.
GetTradeHistory	Symbol	List the last 100 trades for a symbol.
GetInterestRateList	Symbol	Check interest rate list.
GetIndexList	Symbol	Check index list
GetCurrentMarkPrice	Symbol	Check the current mark price.
GetPremiumIndex	Symbol	Submit request to get premium index.
GetCurrentFundingRate	Symbol	Submit request to check the current mark price.
GetKLine	Symbol	Get K Line Data of Contract

Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

User EndPoints

Method	Parameters	Description
GetAccoun-tOverview		Get Account Overview
GetTransaction-History		If there are open positions, the status of the first page returned will be Pending, indicating the realised profit and loss in the current 8-hour settlement period. Please specify the minimum offset number of the current page into the offset field to turn the page.

Trade Endpoints

Method	Parameters	Description
PlaceOrder		You can place two types of orders: limit and market. Orders can only be placed if your account has sufficient funds. Once an order is placed, your funds will be put on hold for the duration of the order. The amount of funds on hold depends on the order type and parameters specified.
PlaceMarketOrder		Places a Market Order.
PlaceLimitOrder		Places a Limit Order.
CancelOrder		Cancels an Order by Order Id.
LimitOrderMassCancellation		Cancel all open orders (excluding stop orders). The response is a list of orderIDs of the canceled orders.
StopOrderMassCancellation		Cancel all untriggered stop orders. The response is a list of orderIDs of the canceled stop orders. To cancel triggered stop orders, please use 'Limit Order Mass Cancelation'.
GetOrderList		List your current orders.
GetUntriggeredStopOrderList		Get the un-triggered stop orders list.
GetListOrdersCompleted24hr		Get a list of recent 1000 orders in the last 24 hours. If you need to get your recent traded order history with low latency, you may query this endpoint.
GetOrder		Get a single order by order id (including a stop order).
GetOrderByClientOid		Get a single order by client order id (including a stop order).
GetFills		Get a list of recent fills.
GetRecentFills		Get a list of recent 1000 fills in the last 24 hours. If you need to get your recent traded order history with low latency, you may query this endpoint.
ActiveOrderValueCalculation		You can query this endpoint to get the total number and value of all your active orders.
GetPositionDetails		Get the position details of a specified position.
GetPositionList		Get the position details of a specified position.
AutoDepositMargin		Enable/Disable of Auto-Deposit Margin
AddMarginManually		Add Margin Manually
ObtainFuturesRiskLimitLevel		This interface can be used to obtain information about risk limit level of a specific contract
AdjustRiskLimitLevel		This interface is for the adjustment of the risk limit level. To adjust the level will cancel the open order, the response can only indicate whether the submit of the adjustment request is successful or not.
GetFundingHistory		Submit request to get the funding history.

GetMaxOpenSize		Get maximum open position size for a contract.
SwitchMarginMode		Switch between cross margin and isolated margin modes.
GetMarginMode		Get the current margin mode for a contract.

Events

Kucoin Messages are received in TsgcWebSocketClient component, you can use the following events:

OnConnect
After a successful connection to Kucoin server.
OnDisconnect
After a disconnection from Kucoin server
OnMessage
Messages sent by server to client are handled in this event.
OnError
If there is any error in protocol, this event will be called.
OnException
If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Kucoin API Component, called **OnKucoinHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket Feeds).

Kucoin | Futures Connect WebSocket API

In order to connect to Kucoin WebSocket API, just create a new Kucoin API client and attach to TsgcWebSocket-Client.
See below an example:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();

TsgcWSAPI_Kucoin_Futures

oKucoin = new

TsgcWSAPI_Kucoin_Futures

();
oKucoin.Client = oClient;
oClient.Active = true;
```

Kucoin | Futures Subscribe WebSocket Channel

Kucoin offers a variety of channels where you can subscribe to get real-time updates of market data, orders... Find below a sample of how to subscribe to a Ticker:

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Kucoin oKucoin = new TsgcWSAPI_Kucoin();
oKucoin.Client = oClient;
oKucoin.SubscribeSymbolTickerV2("XBTUSDM");
void OnMessage(TsgcWSConnection Connection, const string aText)
{
// here you will receive the ticker updates
}
```

Kucoin | Futures Get Market Data

Kucoin offers public Market Data through REST Endpoints, when you call one of these endpoints, you will get a snapshot of the market data requested.
The Market Data Endpoints don't require authentication, so are freely available to all users.

Example: to get the snapshot of the ticker BTC-USDT, do the following call

```
TsgcWSAPI_Kucoin_Futures
oKucoin = new
TsgcWSAPI_Kucoin_Futures
();
MessageBox.Show(oKucoin.REST_API.GetTicker("
XBTUSDM
"));
```

Kucoin | Futures Private REST API

The Kucoin REST API offer public and private endpoints. The Private endpoints require that messages are signed to increase the security of transactions.
First you must login to your Kucoin account and create a new API, you will get the following values:

- ApiKey
- ApiSecret
- Passphrase

These fields must be configured in the Kucoin property of the Kucoin API client component.
Once configured, you can start to do private requests to the Kucoin Pro REST API

*Private Requests, require that your local machine has the local time synchronized, if not, the requests will be rejected by Kucoin server. Check the following article about this, Kucoin Private Requests Time.

```
TsgcWSAPI_Kucoin_Futures

oKucoin = new

TsgcWSAPI_Kucoin_Futures

();
oKucoin.Kucoin.ApiKey = "<your api key>";
oKucoin.Kucoin.ApiSecret = "<your api secret>";
oKucoin.Kucoin.Passphrase = "<your passphrase>";
MessageBox.Show(oKucoin.REST_API.GetAccountOverview());
```

Kucoin | Futures Trade

Kucoin allows you to trade Futures using its REST API.

Configuration

First you must create an **API Key** in your Kucoin account and add privileges to trading with Futures.
Once this is done, you can start futures trading.
First, **set your ApiKey, ApiSecret and Passphrase** in the Kucoin Client Component, this will be used to sign the requests sent to Kucoin server.

Place an Order

To place a new order, just call to method **REST_API.PlaceOrder** of Kucoin Client Component.

Depending on the type of the order (market, limit...) the API requires more or less fields.

Parameters

Param	type	Description
clientOid	String	Unique order id created by users to identify their orders, e.g. UUID, Only allows numbers, characters, underline(_), and separator(-)
side	String	**buy** or **sell**
symbol	String	a valid contract code. e.g. XBTUSDM
type	String	*[optional]* Either **limit** or **market**
leverage	String	Leverage of the order
remark	String	*[optional]* remark for the order, length cannot exceed 100 utf8 characters
stop	String	*[optional]* Either **down** or **up**. Requires **stopPrice** and **stopPriceType** to be defined
stopPrice-Type	String	*[optional]* Either **TP**, **IP** or **MP**, Need to be defined if **stop** is specified.
stopPrice	String	*[optional]* Need to be defined if **stop** is specified.
reduceOnly	boolean	*[optional]* A mark to reduce the position size only. Set to false by default. Need to set the position size when reduceOnly is true.
closeOrder	boolean	*[optional]* A mark to close the position. Set to false by default. It will close all the positions when closeOrder is true.
forceHold	boolean	*[optional]* A mark to forcibly hold the funds for an order, even though it's an order to reduce the position size. This helps the order stay on the order book and not get canceled when the position size changes. Set to false by default.

LIMIT ORDER PARAMETERS

Param	type	Description
price	String	Limit price

Param	type	Description
size	Integer	Order size. Must be a positive number
timeIn-Force	String	*[optional]* **GTC**, **IOC**(default is **GTC**), read Time In Force
postOnly	boolean	*[optional]* Post only flag, invalid when **timeInForce** is **IOC**. When postOnly chose, not allowed choose hidden or iceberg.
hidden	boolean	*[optional]* Orders not displaying in order book. When hidden chose, not allowed choose postOnly.
iceberg	boolean	*[optional]* Only visible portion of the order is displayed in the order book. When iceberg chose, not allowed choose postOnly.
visibleSize	Integer	*[optional]* The maximum visible size of an iceberg order

MARKET ORDER PARAMETERS

Param	type	Description
size	Integer	*[optional]* amount of contract to buy or sell

When you send an order, there are 2 possibilities:

1. **Successful:** the function PlaceOrder returns the message sent by Kucoin server.
2. **Error:** the exception is returned in the event OnKucoinHTTPException.

Place Market Order 1 XBTUSDM

```
TsgcWSAPI_Kucoin_Futures

oKucoin = new

TsgcWSAPI_Kucoin_Futures

();
oKucoin.Kucoin.ApiKey = "<api key>";
oKucoin.Kucoin.ApiSecret = "<api secret>";
oKucoin.Kucoin.Passphrase = "<passphrase>";
MessageBox.Show(oKucoin.REST_API.

PlaceMarketOrder

(

kosBuy, "XBTUSDM", 1

));
```

Place Limit Order 1 XBTUSDM at 40000

```
TsgcWSAPI_Kucoin_Futures

oKucoin = new

TsgcWSAPI_Kucoin_Futures

();
oKucoin.Kucoin.ApiKey = "<api key>";
oKucoin.Kucoin.ApiSecret = "<api secret>";
oKucoin.Kucoin.Passphrase = "<passphrase>";
MessageBox.Show(oKucoin.REST_API.
```

```
PlaceLimitOrder
(
kosBuy, "XBTUSDM", 1, 40000
));
```

Kucoin | Futures Private Requests Time

When you do a private request to Kucoin, the message is signed to increase the security of requests. The message takes the local time and sends inside the signed message, if the local time has a difference greater than 5 seconds with Kucoin servers, the request will be rejected. So, it's important to verify that your local time is synchronized, you can do this using the synchronization time method for your OS.

You can check the Kucoin server time, calling method **GetServerTime**, which will return the time of the Kucoin server

API 3Commas

3Commas

APIs supported

- WebSockets API: connect to a public websocket server and provides real-time market data updates.
- REST API: The REST API has endpoints for account and order management as well as public market data.

WebSockets API

The websocket feed provides real-time market data updates for Trades and Deals

You can subscribe to the following **Public channels**:

Method	Argu-ments	Description
Sub-scribeSmart-Trades		
Sub-scribeDeals		

These channels require **Authentication** against 3Commas servers. So first request your API keys in your 3Com-mas Account and then set the values in the property ThreeComas of the component:

- ApiKey
- ApiSecret

If the subscription is successful, the event **OnThreeCommasConfirmSubscription** will be called. If not, the event **OnThreeCommasRejectSubscription** is called, you can get the reason of the rejection using the **aRawMessage** parameter.

REST API

Test Connectivity

Method	Arguments	Description
GetPing		
Get-Server-Time		Returns the server time

Account

Method	Arguments	Description
GetAccounts		User connected ex-changes list
GetMarketList		Supported Market List

GetMarket-Pairs	**aMarketCode:** code of the market	All market pairs
GetCurren-cyRatesWith-LeverageData	**aMarketCode:** code of the market **aPair:** pair name	Currency rates and lim-its with leverage data
GetCurren-cyRates	**aMarketCode:** code of the market **aPair:** pair name	Currency rates and lim-its
GetBalances	**aAccountId:** id of the account	Load balances for specified exchange
GetAccount-TableData	**aAccountId:** id of the account	Information about all user balances on speci-fied exchange
GetAc-countLeverage	**aAccountId:** id of the account **aPair:** pair name	Information about ac-count leverage
GetAccountIn-fo	**aAccountId:** id of the account	Single Account Info

Smart Trades

Method	Arguments	Description
GetSmart-TradeHistory		Get the Trade History
PlaceMarke-tOrder	**aAccountId:** id of the account **aOrderSide:** buy or sell **aPair:** pair name **aQuantity:** amount	Places a Market Order
PlaceLimi-tOrder	**aAccountId:** id of the account **aOrderSide:** buy or sell **aPair:** pair name **aQuantity:** amount **aPrice:** limit price	Places a Limit Order
GetSmart-Trade	**aId:** id of the trade	Get a Smart Trade by the Id of the Trade
CancelS-martTrade	**aId:** id of the trade	Cancel a Smart Trade by the Id of the Trade
CloseByMar-ketSmart-Trade	**aId:** id of the trade	
EditSmart-Trade	**aId:** id of the trade	Edit an existing Smart Trade
ForceStartS-martTrade	**aId:** id of the trade	Force start a Smart Trade
AddFundsS-martTrade	**aId:** id of the trade	Add funds to a Smart Trade
GetSmart-TradeTrades	**aId:** id of the trade	Get trades of a Smart Trade

DCA Bot

Method	Arguments	Description
CreateDCABot		Create a new DCA Bot
GetDCABot	**aId:** id of the bot	Get a DCA Bot by Id
GetDCABots		Get all DCA Bots
EnableDCABot	**aId:** id of the bot	Enable a DCA Bot
DisableDCABot	**aId:** id of the bot	Disable a DCA Bot
DeleteDCABot	**aId:** id of the bot	Delete a DCA Bot
CancelDCABot	**aId:** id of the bot	Cancel a DCA Bot
GetDCABotStats		Get DCA Bot statistics
GetAvailableStrategyList		Get available strategy list
GetBlacklistPairs		Get blacklist pairs
AddBlacklistPairs		Add blacklist pairs

Deals

Method	Arguments	Description
GetDeals		Get all deals
GetDeal	**aId:** id of the deal	Get a deal by Id
UpdateDeal	**aId:** id of the deal	Update a deal
CancelDeal	**aId:** id of the deal	Cancel a deal
CloseAtMarketDeal	**aId:** id of the deal	Close a deal at market price

Grid Bot

Method	Arguments	Description
CreateGridBot		Create a new Grid Bot
GetGridBot	**aId:** id of the bot	Get a Grid Bot by Id
GetGridBots		Get all Grid Bots
EnableGridBot	**aId:** id of the bot	Enable a Grid Bot
DisableGridBot	**aId:** id of the bot	Disable a Grid Bot

Delete-GridBot	**aId:** id of the bot	Delete a Grid Bot

Events

OnConnect

When a new WebSocket connection is open

OnDisconnect

When a WebSocket connection is closed

OnThreeCommasConnect

When the client receives a Welcome message from 3Commas server, means the connection is ready.

OnThreeCommasConfirmSubscription

Confirms a previous subscription sent by the client.

OnThreeCommasRejectSubscription

There is an error trying to subscribe to a 3Commas channel

OnThreeCommasMessage

Here the client receives the data sent by server related to the channels subscribed

OnThreeCommasPing

Ping sent by server to the client.

OnThreeCommasHTTPException

If there is any error while calling HTTP REST methods, this event will be called.

API OKX

OKX

APIs supported

- WebSockets API: connect to a websocket server and provides real-time market data updates, account changes and place trading orders.

Properties

WebSocket channels are divided into two categories: public and private channels.

- **Public channels:** include tickers channel, K-Line channel, limit price channel, order book channel, and mark price channel, etc -- do not require log in.
- **Private channels:** including account channel, order channel, and position channel, etc -- require log in.

You can configure the following properties in the OKX property.

- **ApiKey:** you can request a new api key in your OKX account, just copy the value to this property.
- **ApiSecret:** it's the secret value of the api.
- **Passphrase:** it's the custom string defined when creating a new api key.
- **IsDemo:** if enabled, will connect to the OKX Demo account (disabled by default).
- **IsPrivate:** if enabled, you will be able to connect to private channels (disabled by default).
- **IsBusiness:** if enabled, will connect to the Business WebSocket endpoint (wss://ws.okx.com:8443/ws/v5/business). Use this for candle channels, algo order channels, and other advanced channels (disabled by default).

Connection

When the client successfully connects to OKX servers, the event **OnOKXConnect** is fired. If there is any error while trying to connect, the event **OnOKXError** will be fired with the error details.
After the event **OnOKXConnect** is fired, then you can start to **send** and **receive messages** from OKX servers.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX oOKX = new TsgcWSAPI_OKX();
oOKX.Client = oClient;
oOKX.OKX.ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oOKX.OKX.ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oOKX.OKX.Passphrase = "secret_passphrase";
oClient.Active = true;
void OnOKXConnect(TsgcWSAPI_OKX Sender, string aMessage, string aCode, string aRawMessage)
{
  DoLog("#OKX Connected");
}
void OnOKXError(TsgcWSAPI_OKX Sender, string aCode, string aMessage, string aRawMessage)
{
  DoLog("#error: " + aMessage);
}
```

Public Channels

The websocket feed provides real-time market data updates for orders and trades. The websocket feed has some public channels like ticker, trades...

You can subscribe to the following **Public channels**:

Method	Description
SubscribeInstruments	The full instrument list will be pushed for the first time after subscription. Subsequently, the instruments will be pushed if there is any change to the instrument's state (such as delivery of FUTURES, exercise of OPTION, listing of new contracts / trading pairs, trading suspension, etc.).
SubscribeTicker	Retrieve the last traded price, bid price, ask price and 24-hour trading volume of instruments. Data will be pushed every 100 ms.
SubscribeOpenInterest	Retrieve the open interest. Data will be pushed every 3 seconds.
SubscribeCandlestick	Retrieve the candlesticks data of an instrument. the push frequency is the fastest interval 500ms push the data.
SubscribeTrades	Retrieve the recent trades data. Data will be pushed whenever there is a trade.
SubscribeEstimated-Prices	Retrieve the estimated delivery/exercise price of FUTURES contracts and OPTION. Only the estimated delivery/exercise price will be pushed an hour before delivery/exercise, and will be pushed if there is any price change.
SubscribeMarkPrice	Retrieve the mark price. Data will be pushed every 200 ms when the mark price changes, and will be pushed every 10 seconds when the mark price does not change.
SubscribeMarkPrice-Candlestick	Retrieve the candlesticks data of the mark price. Data will be pushed every 500 ms.
SubscribePriceLimit	Retrieve the maximum buy price and minimum sell price of the instrument. Data will be pushed every 5 seconds when there are changes in limits, and will not be pushed when there is no changes on limit.
SubscribeOrderBook	Retrieve order book data. Use books for 400 depth levels, book5 for 5 depth levels, bbo-tbt tick-by-tick 1 depth level, books50-l2-tbt tick-by-tick 50 depth levels, and books-l2-tbt for tick-by-tick 400 depth levels. • **books:** 400 depth levels will be pushed in the initial full snapshot. Incremental data will be pushed every 100 ms when there is change in order book. • **books5:** 5 depth levels will be pushed every time. Data will be pushed every 100 ms when there is change in order book. • **bbo-tbt**: 1 depth level will be pushed every time. Data will be pushed every 10 ms when there is change in order book. • **books-l2-tbt:** 400 depth levels will be pushed in the initial full snapshot. Incremental data will be pushed every 10 ms when there is change in order book. • **books50-l2-tbt:** 50 depth levels will be pushed in the initial full snapshot. Incremental data will be pushed every 10 ms when there is change in order book. If asks or bids is an empty array, it means that there are changes in 400 depth, instead of 50 depth. If you maintain the order book data locally, please ignore empty asks and bids.
SubscribeOptionSummary	Retrieve detailed pricing information of all OPTION contracts. Data will be pushed at once.
SubscribeFundingRate	Retrieve funding rate. Data will be pushed in 30s to 90s.
SubscribeIndexCandlestick	Retrieve the candlesticks data of the index. Data will be pushed every 500 ms.
SubscribeIndexTicker	Retrieve index tickers data
SubscribeStatus	Get the status of system maintenance and push when the system maintenance status changes. First subscription: "Push the latest change data"; every time there is a state change, push the changed content
SubscribePublicStructureBlockTrades	Data will be pushed whenever there is a block trade.
SubscribeBlockTickers	Retrieve the latest block trading volume in the last 24 hours. The data will be pushed when triggered by transaction execution event. In addition, it will also be pushed in 5 minutes interval according to subscription granularity.
SubscribeAllTrades	Retrieve all trades data. Data will be pushed whenever there is a trade.
SubscribeLiquidationOrders	Retrieve liquidation orders. Data will be pushed when there is a liquidation order.
SubscribeADLWarning	Retrieve ADL warning data. Data will be pushed when the auto-deleveraging risk increases.
SubscribeEconomicCalendar	Retrieve the economic calendar events. Data will be pushed when there are updates to economic events.

SubscribePublicBlock-Trades	Retrieve public block trades. Data will be pushed whenever there is a block trade.
SubscribeOptionTrades	Retrieve option trades data. Data will be pushed whenever there is an option trade.
SubscribeCallAuction-Details	Retrieve call auction details. Data will be pushed when there are updates to call auction information.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX oOKX = new TsgcWSAPI_OKX();
oOKX.Client = oClient;
oOKX.OKX.ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oOKX.OKX.ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oOKX.OKX.Passphrase = "secret_passphrase";
oClient.Active = true;
void OnOKXConnect(TsgcWSAPI_OKX Sender, string aMessage, string aCode, string aRawMessage)
{
  oOKX.SubscribeInstruments(okxitFutures);
}
```

Private Channels

Including account channel, order channel, and position channel, etc -- require log in.

You can subscribe to the following **Private channels**:

Method	Description
SubscribeAccount	Retrieve account information. Data will be pushed when triggered by events such as placing order, canceling order, transaction execution, etc. It will also be pushed in regular interval according to subscription granularity.
SubscribePositions	Retrieve position information. Initial snapshot will be pushed according to subscription granularity. Data will be pushed when triggered by events such as placing/canceling order, and will also be pushed in regular interval according to subscription granularity.
SubscribeBalanceAnd-Position	Retrieve account balance and position information. Data will be pushed when triggered by events such as filled order, funding transfer.
SubscribeOrders	Retrieve order information. Data will not be pushed when first subscribed. Data will only be pushed when triggered by events such as placing/canceling order.
SubscribeOrdersAlgo	Retrieve algo orders (includes trigger order, oco order, conditional order). Data will not be pushed when first subscribed. Data will only be pushed when triggered by events such as placing/canceling order.
SubscribeAdvanceAlgo	Retrieve advance algo orders (including Iceberg order, TWAP order, Trailing order). Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribePositionRisk	This push channel is only used as a risk warning, and is not recommended as a risk judgment for strategic trading In the case that the market is not moving violently, there may be the possibility that the position has been liquidated at the same time that this message is pushed.
SubscribeAccount-Greeks	Retrieve account greeks information. Data will be pushed when triggered by events such as increase/decrease positions or cash balance in account, and will also be pushed in regular interval according to subscription granularity.
SubscribeRfqs	Retrieve the Rfqs.
SubscribeQuotes	Retrieve the Quotes.
SubscribePrivateStruc-tureBlockTrades	Retrieve Structure Block Trades.
SubscribeSpotGridAl-goOrders	Retrieve spot grid algo orders. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribeContactGri-dAlgoOrders	Retrieve contract grid algo orders. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.
SubscribeGridPosi-tions	Retrieve grid positions. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing/canceling order.

SubscribeGridSubOrders	Retrieve grid sub orders. Data will be pushed when first subscribed. Data will be pushed when triggered by events such as placing order.
SubscribeFills	Retrieve filled orders data. Data will be pushed when an order is filled.
SubscribeDepositInfo	Retrieve deposit information. Data will be pushed when there is a deposit status update.
SubscribeWithdrawalInfo	Retrieve withdrawal information. Data will be pushed when there is a withdrawal status update.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_OKX oOKX = new TsgcWSAPI_OKX();
oOKX.Client = oClient;
oOKX.OKX.ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oOKX.OKX.ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oOKX.OKX.Passphrase = "secret_passphrase";
oClient.Active = true;
void OnOKXConnect(TsgcWSAPI_OKX Sender, string aMessage, string aCode, string aRawMessage)
{
  oOKX.SubscribeOrders(okxitFutures, "BTC-USD", "BTC-USD-200329");
}
```

Trading

The WebSocket Trade requires **Authentication**.

You can place an order only if you have sufficient funds. Find below a table with the request parameters:

Parameter	Type	Required	Description
id	String	Yes	Unique identifier of the message Provided by client. It will be returned in response message for identifying the corresponding request. A combination of case-sensitive alphanumerics, all numbers, or all letters of up to 32 characters.
> instId	String	Yes	Instrument ID,e.g. `BTC-USD-190927-5000-C`
> tdMode	String	Yes	Trade mode Margin mode `isolated` `cross` Non-Margin mode `cash`
> ccy	String	No	Margin currency Only applicable to `cross` `MARGIN` orders in `Single-currency margin`.
> clOrdId	String	No	Client-supplied order ID A combination of case-sensitive alphanumerics, all numbers, or all letters of up to 32 characters.
> tag	String	No	Order tag A combination of case-sensitive alphanumerics, all numbers, or all letters of up to 16 characters.
> side	String	Yes	Order side, `buy` `sell`
> posSide	String	Optional	Position side The default is `net` in the `net` mode It is required in the `long/short` mode, and can only be `long` or `short`. Only applicable to `FUTURES/SWAP`.
> ordType	String	Yes	Order type `market`: market order `limit`: limit order `post_only`: Post-only order `fok`: Fill-or-kill order `ioc`: Immediate-or-cancel order `optimal_limit_ioc`:Market order with immediate-or-cancel order
> sz	String	Yes	Quantity to buy or sell.

Parameter	Type	Required	Description
> px	String	Optional	Price Only applicable to `limit` , `post_only` , `fok` , `ioc` order.
> reduceOnly	Boolean	No	Whether to reduce position only or not, `true` `false` , the default is `false` . Only applicable to `MARGIN` orders, and `FUTURES/SWAP` orders in `net` mode Only applicable to `Single-currency margin` and `Multi-currency margin`
> tgtCcy	String	No	Quantity type `base_ccy` : Base currency , `quote_ccy` : Quote currency Only applicable to `SPOT` traded with Market order Default is `quote_ccy` for buy, `base_ccy` for sell
> banAmend	Boolean	No	Whether to ban amending spot order or not, true or false, the default is false. It will fail to place orders if the balance is not enough when banAmend is true. Only applicable to SPOT market order

Place Order Example

You can place an order only if you have sufficient funds.

```
// Place Martket Order
TsgcWSAPI_OKX1.PlaceMarketOrder(okxosBuy, "ETH-BTC", 1);
// Place Limit Order
TsgcWSAPI_OKX1.PlaceLimitOrder(okxosBuy, "ETH-BTC", 1, 0.25);
```

Cancel Order Example

Cancel an incomplete order

```
TsgcWSAPI_OKX1.CancelOrder("ETH-BTC", "457589362405027840");
```

Amend Order

Amend an incomplete order.

```
TsgcWSAPI_OKX1.AmendOrder("ETH-BTC", "457589362405027840", "", 2);
```

Batch Trade Operations

The WebSocket Trade API also supports batch operations for placing, canceling, and amending multiple orders at once. These operations require **Authentication**.

Method	Description
BatchPlaceOrders	Place multiple orders in a single request. Maximum 20 orders can be placed at a time.
BatchCancelOrders	Cancel multiple orders in a single request. Maximum 20 orders can be canceled at a time.
BatchAmendOrders	Amend multiple incomplete orders in a single request. Maximum 20 orders can be amended at a time.
MassCancelOrders	Mass cancel all pending orders for a specific instrument type.

API XTB

XTB

APIs supported

- WebSockets API: connect to a websocket server and provides real-time market data updates, account changes and place trading orders.

Properties

The WebSocket protocol allows 2 types of requests: **Streaming commands** (receive live updates) and **Retrieve Trading Data** (send a request to server retrieving some information).

You can configure the following properties in the XTB property.

- **User:** the username that identifies the connection.
- **Password:** it's the secret value of the user.
- **Demo:** if enabled, will connect to the XTB Demo account (disabled by default).

Connection

When the client successfully connects to XTB servers, the event **OnXTBConnect** is fired. If there is any error while trying to connect, the event **OnXTBError** will be fired with the error details.
After the event **OnXTBConnect** is fired, then you can start to **send** and **receive messages** from XTB servers.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_XTB oXTB = new TsgcWSAPI_XTB();
oXTB.Client = oClient;
oXTB.XTB.User = "user_0001";
oXTB.XTB.Password = "secret_0001";
oClient.Active = true;
void OnXTBConnect(TsgcWSAPI_XTB Sender, string aStreamSessionId)
{
  DoLog("#XTB Connected");
}
void OnXTBError(TsgcWSAPI_XTB Sender, string aCode, string aDescription, string aRawMessage)
{
  DoLog("#error: " + aDescription);
}
```

Connection Commands

Method	Description
Login	In order to perform any action client application have to perform login process. No functionality is available before proper login process. The login method is called automatically after the client connects to the websocket server and the User/Password values are set.
Logout	

Streaming Commands

You can subscribe to the following channels:

Method	Description
SubscribeBalance	Allows to get actual account indicators values in real-time, as soon as they are available in the system.
SubscribeCandles	Subscribes for and unsubscribes from API chart candles. The interval of every candle is 1 minute. A new candle arrives every minute.
SubscribeKeepAlive	Subscribes for and unsubscribes from 'keep alive' messages. A new 'keep alive' message is sent by the API every 3 seconds.
SubscribeNews	Subscribes for and unsubscribes from news.
SubscribeProfits	Subscribes for and unsubscribes from profits.
SubscribeTickPrices	Establishes subscription for quotations and allows you to obtain the relevant information in real-time, as soon as it is available in the system. The getTickPrices command can be invoked many times for the same symbol, but only one subscription for a given symbol will be created. Please beware that when multiple records are available, the order in which they are received is not guaranteed.
SubscribeTrades	Establishes subscription for user trade status data and allows you to obtain the relevant information in real-time, as soon as it is available in the system. Please beware that when multiple records are available, the order in which they are received is not guaranteed.
SubscribeTradeStatus	Allows to get status for sent trade requests in real-time, as soon as it is available in the system. Please beware that when multiple records are available, the order in which they are received is not guaranteed
SubscribePing	Regularly calling this function is enough to refresh the internal state of all the components in the system. Streaming connection, when any command is not sent by client in the session, generates only one way network traffic. It is recommended that any application that does not execute other commands, should call this command at least once every 10 minutes.

Retrieving Trading Data

You can send the following Requests:

Method	Description
GetAllSymbols	Returns array of all symbols available for the user.
GetCalendar	Returns calendar with market events.
GetChartLastRequest	Please note that this function can be usually replaced by its streaming equivalent getCandles which is the preferred way of retrieving current candle data. Returns chart info, from start date to the current time. If the chosen period of CHART_LAST_INFO_RECORD is greater than 1 minute, the last candle returned by the API can change until the end of the period (the candle is being automatically updated every minute).
GetChartRangeRequest	Please note that this function can be usually replaced by its streaming equivalent getCandles which is the preferred way of retrieving current candle data. Returns chart info with data between given start and end dates.
GetCommissionDef	Returns calculation of commission and rate of exchange. The value is calculated as expected value, and therefore might not be perfectly accurate.
GetCurrentUserData	Returns information about account currency, and account leverage.

GetIbsHistory	Returns IBs data from the given time range.
GetMarginLevel	Please note that this function can be usually replaced by its streaming equivalent getBalance which is the preferred way of retrieving account indicators. Returns various account indicators
GetMarginTrade	Returns expected margin for given instrument and volume. The value is calculated as expected margin value, and therefore might not be perfectly accurate.
GetNews	Please note that this function can be usually replaced by its streaming equivalent getNews which is the preferred way of retrieving news data. Returns news from trading server which were sent within specified period of time.
GetProfitCalculation	Calculates estimated profit for given deal data Should be used for calculator-like apps only. Profit for opened transactions should be taken from server, due to higher precision of server calculation
GetServerTime	Returns current time on trading server
GetStepRules	Returns a list of step rules for DMAs
GetSymbol	Returns information about symbol available for the user
GetTickPrices	Please note that this function can be usually replaced by its streaming equivalent getTickPrices which is the preferred way of retrieving ticks data. Returns array of current quotations for given symbols, only quotations that changed from given timestamp are returned. New timestamp obtained from output will be used as an argument of the next call of this command.
GetTradeRecords	Returns array of trades listed in orders argument
GetTrades	Please note that this function can be usually replaced by its streaming equivalent getTrades which is the preferred way of retrieving trades data. Returns array of user's trades.
GetTradesHistory	Please note that this function can be usually replaced by its streaming equivalent getTrades which is the preferred way of retrieving trades data. Returns array of user's trades which were closed within specified period of time.
GetTradingHours	Returns quotes and trading times.
GetVersion	Returns the current API version.
Ping	Regularly calling this function is enough to refresh the internal state of all the components in the system. It is recommended that any application that does not execute other commands, should call this command at least once every 10 minutes. Please note that the streaming counterpart of this function is combination of ping and getKeepAlive
TradeTransaction	Starts trade transaction. tradeTransaction sends main transaction information to the server.
TradeTransactionStatus	Please note that this function can be usually replaced by its streaming equivalent getTradeStatus which is the preferred way of retrieving transaction status data. Returns current transaction status. At any time of transaction processing client might check the status of transaction on server side. In order to do that client must provide unique order taken from tradeTransaction invocation.

API Bybit

Bybit

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Currently, the API supported version is V5. The V5 API brings uniformity and efficiency to Bybit's product lines, unifying Spot, Derivatives, and Options in one set of specifications.

OpenAPI Version	Account Type	Linear			Inverse		Spot	Options
		USDT Perpetual	USDC Perpetual	USDC Futures	Perpetual	Futures		
V5	Unified trading account	✓	✓	✓	see note		✓	✓
	Classic account	✓			✓	✓	✓	
V3	Unified trading account	✓	✓					✓
	Classic account	✓			✓	✓	✓	

*Note: the Unified account supports inverse trading. However, the margin used is from the inverse derivatives wallet instead of the unified wallet.

Properties

You can configure the following properties in the Bybit property.

- **ApiKey:** you can request a new api key in your Bybit account, just copy the value to this property. If the APIKey is set, the client will connect to the websocket private server. If it's empty, will connect to the WebSocket public server.
- **ApiSecret:** it's the secret value of the api.
- **SignatureExpires:** number of seconds after the signature expires (by default 10 seconds).
- **TestNet:** if enabled, will connect to the Bybit TestNet Demo account (disabled by default).

Connection

When the client successfully connects to Bybit servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Bybit servers. If you are connecting to the private websocket channel, you must wait till **OnBybitAuthentication** event is fired and check if the success parameter is true, before subscribing to any channel.

The client supports several APIs, so use the property BybitClient to set which API you want to use:

- **bybSpot**
- **bybInverse**
- **bybLinear**
- **bybPerpetual**

Find below an example of connecting to WebSocket Spot Private API.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit oBybit = new TsgcWSAPI_Bybit();
oBybit.Client = oClient;
oBybit.Bybit.ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oBybit.Bybit.ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oBybit.BybitClient = bybSpot;
oClient.Active = true;
void OnBybitConnect(TsgcWSConnection Connection)
{
  DoLog("#Bybit Connected");
}
```

After a successful connection to the Spot WebSocket Server, you can start to subscribe to WebSocket channels, just access the **SPOT** property and then call any of the subscribe/unsubscribe methods available.

Events

The bybit client implements the following events to control the connection flow and get data sent from the Web-Socket server:

- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnBybitAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnBybitSubscribe:** when the client subscribes to a websocket channel.
- **OnBybitUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnBybitData:** when the client receives data from the server.
- **OnBybitError:** when there is any error during the bybit websocket connection.
- **OnBybitHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades. The websocket feed has some public channels like ticker, trades...

You can subscribe to the following **channels**:

Method	Public or Private	Description
SubscribeOrderBook	Public	Subscribe to the orderbook stream. Supports different depths.
SubscribeTrade	Public	Subscribe to the recent trades stream.
SubscribeTicker	Public	Subscribe to the ticker stream.
SubscribeKLine	Public	Subscribe to the klines stream.
SubscribeLiquidation	Public	Subscribe to the liquidation stream
SubscribeLT_KLine	Public	Subscribe to the leveraged token kline stream.
SubscribeLT_Ticker	Public	Subscribe to the leveraged token ticker stream.
SubscribeLT_Nav	Public	Subscribe to the leveraged token ticker stream.
SubscribePosition	Private	Subscribe to the leveraged token nav stream.
SubscribeExecution	Private	Subscribe
SubscribeOrder	Private	Subscribe
SubscribeWallet	Private	Subscribe

SubscribeGreek	Private	Subscribe
SubscribeDcp	Private	Subscribe
SubscribeInsurance	Public	Subscribe to the insurance fund stream.
SubscribeOrderPriceLimit	Public	Subscribe to the order price limit stream.
SubscribeADLAlert	Public	Subscribe to the auto-deleverage alert stream.
SubscribeFastExecution	Private	Subscribe to the fast execution stream.

Find below an example of subscribing to private websocket channels after a successful authentication.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit oBybit = new TsgcWSAPI_Bybit();
oBybit.Client = oClient;
oBybit.Bybit.ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oBybit.Bybit.ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oBybit.BybitClient = bybSpot;
oClient.Active = true;
void OnBybitAuthentication(TObject Sender, bool aSuccess, const string aError, const string aRawMessage)
{
  if (aSuccess == true)
  {
    oClient.SubscribeOrderBook("BTCUSDT");
    oClient.SubscribeTrade("BTCUSDT");
  }
}
```

REST API

The REST API has a list of Public and Private methods to request data from: markets, private account and wallet. Find below a list of available methods.

Method	Public / Private
GetServerTime	Public
GetKLine	Public
GetMarkPriceKLine	Public
GetIndexPriceKLine	Public
GetPremiumIndexPriceKLine	Public
GetInstrumentsInfo	Public
GetOrderBook	Public
GetTickers	Public
GetFundingRateHistory	Public
GetPublicRecentTradingHistory	Public
GetOpenInterest	Public
GetHistoricalVolatility	Public
GetInsurance	Public
GetRiskLimit	Public
GetDeliveryPrice	Public
GetLongShortRatio	Public
PlaceOrder	Private
PlaceMarketOrder	Private
PlaceLimitOrder	Private
AmendOrder	Private
CancelOrder	Private
GetOpenOrders	Private
CancelAllOrders	Private
GetOrderHistory	Private
GetPositionInfo	Private
SetLeverage	Private
SwitchCrossIsolatedMargin	Private
SetTPSLMode	Private
SwitchPositionMode	Private

SetRiskLimit	Private
SetTradingStop	Private
SetAutoAddMargin	Private
AddOrReduceMargin	Private
GetExecution	Private
GetClosedPNL	Private
ConfirmNewRiskLimit	Private
GetWalletBalance	Private
GetAccountInfo	Private
GetTransactionLog	Private
BatchPlaceOrder	Private
BatchAmendOrder	Private
BatchCancelOrder	Private
SetDCP	Private
GetFeeRate	Private
GetCollateralInfo	Private
SetMarginMode	Private
GetBorrowHistory	Private
GetCoinGreeks	Private
GetCoinInfo	Private
GetAllCoinsBalance	Private
CreateInternalTransfer	Private
GetInternalTransferList	Private
GetDepositRecords	Private
GetDepositAddress	Private
CreateWithdrawal	Private
CancelWithdrawal	Private
GetWithdrawalRecords	Private

Find below an example of getting the open orders.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_Bybit oBybit = new TsgcWSAPI_Bybit();
oBybit.Client = oClient;
oBybit.Bybit.ApiKey = "alsdjk23kandfnasbdfdkjhsdf";
oBybit.Bybit.ApiSecret = "aldskjfk3jkadknfajndsjfj23j";
oBybit.BybitClient = bybSpot;
oBybit.REST_API.GetAccountInfo();
```

API Cex

Cex

WebSocket API allows getting real-time notifications without sending extra requests, making it a faster way to obtain data from the exchange
Cex component has a property called Cex where you can fill API Keys provided by Cex to get access to your account data.

Message encoding

- All messages are encoded in JSON format.
- Prices are presented as strings to avoid rounding errors at JSON parsing on the client side.
- Compression of WebSocket frames is not supported by the server.
- Time is presented as integer UNIX timestamp in seconds.

Authentication

To get access to CEX.IO WebSocket data, you should be authorized.

- Log in to CEX.IO account.
- Go to https://cex.io/trade/profile#/api page.
- Select the type of required permissions.
- Click "Generate Key" button and save your secret key, as it will become inaccessible after activation.
- Activate your key.

Connectivity

- If a connected Socket is inactive for 15 seconds, CEX.IO server will send a PING message.
- Only server can be an Initiator of PING request.
- The server sends ping only to the authenticated user.
- The user has to respond with a PONG message. Otherwise, the WebSocket will be DISCONNECTED. This is handled automatically by the library.
- For the authenticated user, in case there is no notification or ping from the server within 15 seconds, it would be safer to send a request like 'ticker' or 'get-balance' and receive a response, in order to ensure connectivity and authentication.

Public Channels

These channels don't require authentication. Responses from the server are received through the OnCexMessage event.

- **SubscribeTickers:** Ticker feed with only price of transaction made on all pairs (deprecated)

```
{
    "e": "tick",
    "data": {
        "symbol1": "BTC",
        "symbol2": "USD",
        "price": "428.0123"
    }
}
```

- **SubscribeChart:** OHLCV chart feeds with Open, High, Low, Close, Volume numbers (deprecated)

```
{
    'e': 'ohlcv24',
    'pair': 'BTC:USD',
    'data': [
        '418.2936',
        '420.277',
        '412.09',
        '416.9778',
        '201451078368'
    ]
}
```

- **Subscribe Pair:** Market Depth feed (deprecated)

```
{
    'e': 'md_groupped',
    'data': {
        'pair': 'BTC:USD',
        'id': 11296131,
        'sell': {
            '427.5000': 1000000,
            '480.0000': 263544334,
            ...
        },
        'buy': {
            '385.0000': 3630000,
            '390.0000': 1452458642,
            ... 400+ pairs togather with 'sell' pairs
        }
    }
}
```

- **Subscribe Pair:** Order Book feed (deprecated)

```
{
    'e': 'md',
    'data': {
        'pair': 'BTC:USD',
        'buy_total': 63221099,
        'sell_total': 112430315118,
        'id': 11296131,
        'sell': [
            [426.45, 10000000],
            [426.5, 66088429300],
            [427, 1000000],
            ... 50 pairs overaall
        ],
        'buy': [
            [423.3, 4130702],
            [423.2701, 10641168],
            [423.2671, 1000000],
            ... 50 pairs overaall
        ]
    }
}
```

Private Channels

To access these channels, first call the Authenticate method. Responses from the server are received through the OnCexMessage event.

GetTicker

```
{
    "e": "ticker",
    "data": {
        "timestamp": "1471427037",
        "low": "290",
        "high": "290",
        "last": "290",
        "volume": "0.02062068",
        "volume30d": "14.38062068",
        "bid": 240,
        "ask": 290,
        "pair": [
            "BTC",
            "USD"
        ]
    },
    "oid": "1471427036908_1_ticker",
    "ok": "ok"
}
```


GetBalance


```
{
    "e": "get-balance",
    "data": {
           "balance": {
            'LTC': '10.00000000',
            'USD': '1024.00',
            'RUB': '35087.98',
            'EUR': '217.53',
            'GHS': '10.00000000',
            'BTC': '9.00000000'
        },
           "obalance": {
              'BTC': '0.12000000',
              'USD': "512.00",
           },
       },
       "time": 1435927928597
    "oid": "1435927928274_2_get-balance",
    "ok": "ok"
}
```


SubscribeOrderBook


```
{
"e": "order-book-subscibe",
"data": {
   "timestamp": 1435927929,
   "bids": [
      [
         241.947,
         155.91626
      ],
      [
         241,
         981.1255
      ],
   ],
   "asks": [
      [
         241.95,
         15.4613
      ],
      [
         241.99,
         17.3303
      ],
   ],
   "pair": "BTC:USD",
   "id": 67809
},
"oid": "1435927928274_5_order-book-subscribe",
"ok": "ok"
}
```

UnSubscribeOrderBook


```
{
"e": "order-book-unsubscribe",
"data": {
   "pair": "BTC:USD"
},
"oid": "1435927928274_4_order-book-unsubscribe",
"ok": "ok"
}
```


GetOpenOrders


```
{
"e": "open-orders",
"data": [
   {
      "id": "2477098",
      "time": "1435927928618",
      "type": "buy",
      "price": "241.9477",
      "amount": "0.02000000",
      "pending": "0.02000000"
   },
   {
      "id": "2477101",
      "time": "1435927928634",
      "type": "sell",
      "price": "241.9493",
      "amount": "0.02000000",
      "pending": "0.02000000"
   }
],
"oid": "1435927928274_9_open-orders",
"ok": "ok"
}
```


PlaceOrder


```
{
"e": "place-order",
"data": {
   "complete": false,
   "id": "2477098",
   "time": 1435927928618,
   "pending": "0.02000000",
   "amount": "0.02000000",
   "type": "buy",
   "price": "241.9477"
},
"oid": "1435927928274_7_place-order",
"ok": "ok"
}
```


CancelReplaceOrder


```
{
"e": "cancel-replace-order",
"data": {
   "complete": false,
   "id": "2689009",
   "time": 1443464955904,
   "pending": "0.04000000",
   "amount": "0.04000000",
```

```
   "type": "buy",
   "price": "243.25"
},
"oid": "1443464955209_16_cancel-replace-order",
"ok": "ok"
}
```


GetOrderRequest

In CEX.IO system, orders can be present in the trade engine or in an archive database. There can be time periods (~2 seconds or more), when the order is done/cancelled, but still not moved to the archive database. That means you cannot see it using calls: archived-orders/open-orders. This call allows getting order information in any case. Responses can have different format depending on orders location.


```
{
"e": "get-order",
"data": {
   "user": "XXX",
   "type": "buy",
   "symbol1": "BTC",
   "symbol2": "USD",
   "amount": "0.02000000",
   "remains": "0.02000000",
   "price": "50.75",
   "time": 1450214742160,
   "tradingFeeStrategy": "fixedFee",
   "tradingFeeBuy": "5",
   "tradingFeeSell": "5",
   "tradingFeeUserVolumeAmount": "nil",
   "a:USD:c": "1.08",
   "a:USD:s": "1.08",
   "a:USD:d": "0.00",
   "status": "a",
   "orderId": "5582060"
},
"oid": "1450214742135_10_get-order",
"ok": "ok"
}
```


CancelOrderRequest


```
{
"e": "cancel-order",
"data": {
   "order_id": "2477098"
   "time": 1443468122895
},
"oid": "1435927928274_12_cancel-order",
"ok": "ok"
}
```


GetArchivedOrders


```
{
"e": "archived-orders",
"data": [
   {
       "type": "buy",
       "symbol1": "BTC",
       "symbol2": "USD",
       "amount": 0,
       "amount2": 5000,
       "remains": 0,
       "time": "2015-04-17T10:46:27.971Z",
       "tradingFeeBuy": "2",
       "tradingFeeSell": "2",
       "ta:USD": "49.00",
       "fa:USD": "0.98",
       "orderId": "2340298",
       "status": "d",
```

```
        "a:BTC:cds": "0.18151851",
        "a:USD:cds": "50.00",
        "f:USD:cds": "0.98"
    },
    {
        "type": "buy",
        "symbol1": "BTC",
        "symbol2": "USD",
        "amount": 0,
        "amount2": 10000,
        "remains": 0,
        "time": "2015-04-08T15:46:04.651Z",
        "tradingFeeBuy": "2.99",
        "tradingFeeSell": "2.99",
        "ta:USD": "97.08",
        "fa:USD": "2.91",
        "orderId": "2265315",
        "status": "d",
        "a:BTC:cds": "0.39869578",
        "a:USD:cds": "100.00",
        "f:USD:cds": "2.91"
    }
],
"oid": "1435927928274    15_archived-orders",
"ok": "ok"
}
```


OpenPosition


```
{
    "e": "open-position",
    "oid": "1435927928274_7_open-position",
    "data": {
        'amount': '1',
        'symbol': 'BTC',
        "pair": [
            "BTC",
            "USD"
        ],
        'leverage': '2',
        'ptype': 'long',
        'anySlippage': 'true',
        'eoprice': '650.3232',
        'stopLossPrice': '600.3232'
    }
}
```


GetPosition


```
{
  "e": "get_position",
  "ok": "ok",
  "data": {
    "user": "ud100036721",
    "pair": "BTC:USD",
    "amount": "1.00000000",
    "symbol": "BTC",
    "msymbol": "USD",
    "omamount": "1528.77",
    "lsymbol": "USD",
    "lamount": "3057.53",
    "slamount": "3380.11",
    "leverage": "3",
    "stopLossPrice": "3380.1031",
    "dfl": "3380.10310000",
    "flPrice": "3057.53333333",
    "otime": 1513002370342,
    "psymbol": "BTC",
    "ptype": "long",
    "ofee": "10",
    "pfee": "10",
    "cfee": "10",
    "tfeeAmount": "152.88",
    "rinterval": "14400000",
    "okind": "Manual",
    "a:BTC:c": "1.00000000",
    "a:BTC:s": "1.00000000",
```

```
    "oorder": "89101551",
    "pamount": "1.00000000",
    "lremains": "3057.53",
    "slremains": "3380.11",
    "oprice": "4586.3000",
    "status": "a",
    "id": "125531",
    "a:USD:cds": "4739.18"
  }
}
```

GetOpenPositions

```
{
    'e': 'open_positions',
    "oid": "1435927928256_7_open-positions",
    'ok': 'ok',
    'data': [
        {
            'user': 'ud100036721',
            'id': '104102',
            'otime': 1475602208467,
            'symbol': 'BTC',
            'amount': '1.00000000',
            'leverage': '2',
            'ptype': 'long',
            'psymbol': 'BTC',
            'msymbol': 'USD',
            'lsymbol': 'USD',
            'pair': 'BTC:USD',
            'oprice': '607.5000'
            'stopLossPrice': '520.3232',
            'ofee': '1',
            'pfee': '3',
            'cfee': '4',
            'tfeeAmount': '3.04',
            'pamount': '1.00000000',
            'omamount': '303.75',
            'lamount': '303.75',
            'oorder': '34106774',
            'rinterval': '14400000',
            'dfl': '520.32320000',
            'slamount': '520.33',
            'slremains': '520.33',
            'lremains': '303.75',
            'flPrice': '303.75000000',
            'a:BTC:c': '1.00000000',
            'a:BTC:s': '1.00000000',
            'a:USD:cds': '610.54',
        },
        ...
    ]
}
```

ClosePosition

```
{
    'e': 'close_position',
    "oid": "1435927928364_7_close-position",
    'ok': 'ok',
    'data': {
        'id': 104034,
        'ctime': 1475484981063,
        'ptype': 'long',
        'msymbol': 'USD'
        'pair': {
            'symbol1': 'BTC',
            'symbol2': 'USD'
        }
        'price': '607.1700',
        'profit': '-12.48',
    }
}
```

API Cex Plus

Cex Plus

APIs supported

- WebSockets API: connect to a public websocket server and provides real-time market data updates.

WebSockets API

WebSocket is a TCP-based full-duplex communication protocol. Full-duplex means that both parties can send each other messages asynchronously using the same communication channel. This section describes which messages should Exchange Plus and Client send each other. All messages should be valid JSON objects.

WebSocket API is mostly used to obtain information or do actions which are not available or not easy to do using REST API. However, some requests or actions are possible to do in both REST API and WebSocket API. Exchange Plus sends messages to Client as a response to request previously sent by Client, or as a notification about some event (without prior Client's request).

Public API Calls

Public API rate limit is implied in order to protect the system from DDoS attacks and ensuring all Clients can have same level of stable access to Exchange Plus API endpoints. Public requests are limited by IP address from which public API requests are made. Request limits are determined from cost associated with each public API call. By default, each public request has a cost of 1 point, but for some specific requests this cost can be higher. See up-to-date request rate limit cost information in specification of each method.

Exchange Plus limits Public API calls to maximum of 100 points per minute, considering that each Public API call has its cost (see below). If request rate limit is reached then Exchange Plus replies with error, sends disconnected event to Client and closes WS connection afterwards. Exchange Plus will continue to serve Client starting from the next calendar minute. In the following example, request counter will be reset at 11:02:00.000.

Method	Description
GetTicker	This method is designed to obtain current information about Ticker, including data about current prices, 24h price & volume changes, last trade event etc. of certain assets.
GetOrderBook	This method allows Client to receive current order book snapshot for specific trading pair.
GetCandles	By using Candles method Client can receive historical OHLCV candles of different resolutions and data types. Client can indicate additional timeframe and limit filters to make response more precise to Client's requirements.
GetTradeHistory	This method allows Client to obtain historical data as to occurred trades upon requested trading pair. Client can supplement Trade History request with additional filter parameters, such as timeframe period, tradeIds range, side etc. to receive trades which match request parameters.
GetServerTime	This method is used to get the current time on Exchange Plus server. It can be useful for applications that have to be synchronized with the server's time.
GetPairsInfo	Pair Info method allows Client to receive the parameters for all supported trading pairs.
GetCurrenciesInfo	Currencies Info method allows Client to receive the parameters for all currencies configured in Exchange Plus as well as the deposit and withdrawal availability between Exchange Plus and CEX.IO Wallet.
GetProcessingInfo	This request allows Client to receive detailed information about available options to make deposits from external wallets and withdrawals to external wallets as to each supported cryptocurrency, including cryptocurrency name and available blockchains for deposit\withdrawals. Also, as to each supported blockchain there are indicated type of cryptocurrency on indicated blockchain, current deposit\withdrawal availability, minimum amounts for deposits\withdrawals, external withdrawal fees. Processing Information makes Client more flexible in choosing desired blockchain

	for receiving Deposit address and initiating external withdrawals via certain blockchain, so that Client uses more convenient way of transferring his crypto assets to or from CEX.IO Ecosystem.
SubscribeOrder-Book	Client by subscribing via WebSocket can subscribe to order book feed upon requested trading pair. In response to Order Book Subscribe request Client will receive current (initial) order book snapshot for requested pair with indicated seqId number. To track following updates to Order Book Client needs to subscribe via WebSocket to “order_book_increment“ messages, which would contain trading pair name, seqId number, Bids and Asks price levels deltas.
UnSubscribeOrder-Book	UnSubscribe from the order book channel.
SubscribeTrade	By using the Trade Subscribe method Client can subscribe via WebSocket to live feed of trade events which occur on requested trading pair. In response to Trade Subscribe request Client will receive a unique identifier of trade subscription which should further be used for unsubscription when trade subscription is not longer needed for Client. Client should subscribe via WebSocket to “tradeHistorySnapshot” and “tradeUpdate” messages to receive initial and periodical Trade History snapshots, and live trade events for requested trading pair.
UnSubscribeTrade	UnSubscribe from the trade channel.

Example: get the latest ticker of BTC-USD pair

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_CexPlus oCexPlus = new TsgcWSAPI_CexPlus();
oCexPlus.Client = oClient;
oCexPlus.OnCexPlusConnect += OnCexPlusConnectEvent;
oCexPlus.OnCexPlusMessage += OnCexPlusMessageEvent;
oClient.Active = true;

void OnCexPlusConnectEvent(object Sender)
{
  oCexPlus.GetTicker("BTC-USD");
}

void OnCexPlusMessageEvent(object Sender, string Event, string Msg)
{
  MessageBox.Show("Ticker data: " + Msg);
}
```

Private API Calls

Exchange Plus uses API keys to allow access to Private APIs.

Client can generate, configure and manage api keys, set permission levels, whitelisted IPs for API key etc. via Exchange Plus Web Terminal in the API Keys Management Profile section.

API Keys limit: By default Client can have up to 5 API Keys.

To restrict access to certain functionality while using of API Keys there should be defined specific set of permissions for each API Key. The defined set of permissions can be edited further if necessary.

The following permission levels are available for API Keys:

- **Read:** permission level for viewing of account related data, receiving reports, subscribing to market data etc.
- **Trade:** permission level, which allows placing and cancelling orders on behalf of account.
- **Funds Internal:** permission level, which allows transferring funds between accounts (between sub-accounts or main account and sub-accounts) of CEX.IO Exchange Plus Portfolio.
- **Funds Wallet:** permission level, which allows transferring funds from CEX.IO Exchange Plus Portfolio accounts (main account and sub-accounts) to CEX.IO Wallet and vice versa.

Method	Description

GetCurrentFee	This method indicates current fees at specific moment of time with consideration of Client' up-to-date 30d volume and day of week (fees can be different for e.g. on weekends).
GetFeeStrategy	Fee Strategy returns all fee options, which could be applied for Client, considering Client's trading volume, day of week, pairs, group of pairs etc. This method provides information about general fee strategy, which includes all possible trading fee values that can be applied for Client. To receive current trading fees, based on Client's current 30d trading volume, Client should use [Current Fee] method. To receive current 30d trading volume, Client should use [Volume] method.
GetVolume	This request allows Client to receive his trading volume for the last 30 days in USD equivalent.
CreateAccount	This request allows Client to create new sub-account. By default Client can have up to 5 sub-accounts, including main account.
GetAccountStatus	By using Account Status V3 method, Client can find out current balance and it's indicative equivalent in converted currency (by default "USD"), amounts locked in open (active) orders as to each sub-account and currency. If trading fee balance is available for Client, then response will also contain general trading fee balance data such as promo name, currency name, total balance and expiration date of this promo on Trading Fee Balance. It's Client's responsibility to track his sub-accounts available trading balance as current sub-account balance reduced by the balance amount locked in open (active) orders on sub-account.
GetOrders	This request allows Client to find out info about his orders.
NewOrder	Client can place new orders via WebSocket API by using Do My New Order Request. Along with a response to this request, Exchange Plus sends Account Event and Execution Report messages to Client if the request is successful. Response message indicates the last up-to-date status of order which is available in the system at the moment of sending the response. If the Client did not receive a Response message to Do My New Order Request - the Client can query current status of the order by using Get My Orders Request with clientOrderId parameter. When sending a request for new order, it is highly recommended to use clientOrderId parameter which corresponds to the specific new order request on the client's side. Exchange Plus avoids multiple placing the orders with the same clientOrderId. If more than one new orders with identical clientOrderId and other order parameters are identified - Exchange Plus places only the first order and returns the status of such order to the Client in response to the second and subsequent new order requests with the same parameters. If more than one new orders with identical clientOrderId but with different other order parameters are identified - Exchange Plus processes only the first order and rejects the second and subsequent new order requests with the same clientOrderID but with different other order parameters.
NewMarketOrder	Places a new market order.
NewLimitOrder	Places a new limit order.
CancelOrder	Client can cancel orders. Along with a response to this request, Exchange Plus sends Account Event and Execution Report messages to Client if this request is successful. Also, if request to cancel an order is declined, Exchange Plus sends Order Cancellation Rejection message.
CancelAllOrders	Client can cancel all open orders via WebSocket API. Along with a response to this request Exchange Plus will start cancellation process for all open orders and send corresponding Account Event and Execution Report messages to the Client.
GetTransactionHistory	This request allows Client to find out his financial transactions (deposits, withdrawals, internal transfers, commissions or trades).
GetFundingHistory	This request allows Client to find his deposit and withdrawal transactions.
InternalTransfer	Client can request to transfer money between his sub-accounts or between his main account and sub-account. Exchange Plus does not charge Client any commission for transferring funds between his accounts. Along with a response to this request, Exchange Plus sends Account Event messages to Client if this request is successful.
GetDepositAddress	This method can be used by Client for receiving a crypto address to deposit cryptocurrency. Deposit address can be generated for main and sub-accounts. The list of available blockchains for generating deposit address can be received by Client via Get Processing Info request.
FundsDepositFromWallet	Client can deposit funds from CEX.IO Wallet to Exchange Plus account. The system avoids processing of multiple deposit requests with the same clientTxId. If multiple deposit requests with identical clientTxId are received - the system processes only the first deposit request and rejects the second and subsequent deposit requests with the same clientTxId.
FundsWithdrawalToWallet	Client can withdraw funds from Exchange Plus account to CEX.IO Wallet. The system avoids multiple withdrawal requests with the same clientTxId. If multiple withdrawal requests with identical clientTxId are received - the system processes only the first withdrawal request and rejects the second and subsequent withdrawal requests with the same clientTxId.
GetWalletBalance	Retrieves the CEX.IO Wallet balance information.
SubscribeAccountEvents	Subscribes to real-time account event notifications (balance changes, order fills).

UnSubscribeAccountEvents	Unsubscribes from account event notifications.

Example: get the orders.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_CexPlus oCexPlus = new TsgcWSAPI_CexPlus();
oCexPlus.Client = oClient;
oCexPlus.CexPlus.ApiKey = "your-api-key";
oCexPlus.CexPlus.ApiSecret = "your-api-secret";
oCexPlus.OnCexPlusAuthenticated += OnCexPlusAuthenticatedEvent;
oCexPlus.OnCexPlusMessage += OnCexPlusMessageEvent;
oClient.Active = true;

void OnCexPlusAuthenticatedEvent(object Sender)
{
  oCexPlus.GetOrders();
}

void OnCexPlusMessageEvent(object Sender, string Event, string Msg)
{
  MessageBox.Show("Orders: " + Msg);
}
```

API Discord

Discord
Note: As of API v10, the Discord API domain has changed from `discordapp.com` to `discord.com`. The component has been updated to use API version 10 and the new domain.

Gateways are Discord's form of real-time communication over secure WebSockets. Clients will receive events and data over the gateway they are connected to and send data over the REST API.

Authorization

First you must generate a new Bot, and copy Bot Token which will be used to authenticate through API. Then set this token in API Component.

```
TsgcWSAPI_Discord1.DiscordOptions.BotOptions.Token = "...bot token here...";
```

Intents

Maintaining a stateful application can be difficult when it comes to the amount of data you're expected to process, especially at scale. Gateway Intents are a system to help you lower that computational burden.
When identifying to the gateway, you can specify an intents parameter which allows you to conditionally subscribe to pre-defined "intents", groups of events defined by Discord. If you do not specify a certain intent, you will not receive any of the gateway events that are batched into that group. The valid intents are (zero value means all events are received):

GUILDS (1 << 0) = Integer (1)
 - GUILD_CREATE
 - GUILD_DELETE
 - GUILD_ROLE_CREATE
 - GUILD_ROLE_UPDATE
 - GUILD_ROLE_DELETE
 - CHANNEL_CREATE
 - CHANNEL_UPDATE
 - CHANNEL_DELETE
 - CHANNEL_PINS_UPDATE
GUILD_MEMBERS (1 << 1) = Integer (2)
 - GUILD_MEMBER_ADD
 - GUILD_MEMBER_UPDATE
 - GUILD_MEMBER_REMOVE
GUILD_BANS (1 << 2) = Integer (4)
 - GUILD_BAN_ADD
 - GUILD_BAN_REMOVE
GUILD_EMOJIS (1 << 3) = Integer (8)
 - GUILD_EMOJIS_UPDATE
GUILD_INTEGRATIONS (1 << 4) = Integer (16)
 - GUILD_INTEGRATIONS_UPDATE
GUILD_WEBHOOKS (1 << 5) = Integer (32)
 - WEBHOOKS_UPDATE
GUILD_INVITES (1 << 6) = Integer (64)
 - INVITE_CREATE
 - INVITE_DELETE
GUILD_VOICE_STATES (1 << 7) = Integer (128)
 - VOICE_STATE_UPDATE
GUILD_PRESENCES (1 << 8) = Integer (256)
 - PRESENCE_UPDATE

GUILD_MESSAGES (1 << 9) = Integer (512)
 - MESSAGE_CREATE
 - MESSAGE_UPDATE
 - MESSAGE_DELETE
GUILD_MESSAGE_REACTIONS (1 << 10) = Integer (1024)
 - MESSAGE_REACTION_ADD
 - MESSAGE_REACTION_REMOVE
 - MESSAGE_REACTION_REMOVE_ALL
 - MESSAGE_REACTION_REMOVE_EMOJI
GUILD_MESSAGE_TYPING (1 << 11) = Integer (2048)
 - TYPING_START
DIRECT_MESSAGES (1 << 12) = Integer (4096)
 - CHANNEL_CREATE
 - MESSAGE_CREATE
 - MESSAGE_UPDATE
 - MESSAGE_DELETE
 - CHANNEL_PINS_UPDATE
DIRECT_MESSAGE_REACTIONS (1 << 13) = Integer (8192)
 - MESSAGE_REACTION_ADD
 - MESSAGE_REACTION_REMOVE
 - MESSAGE_REACTION_REMOVE_ALL
 - MESSAGE_REACTION_REMOVE_EMOJI
DIRECT_MESSAGE_TYPING (1 << 14) = Integer (16384)
 - TYPING_START

Heartbeat

Heartbeats are automatically handled by the component so you don't need to worry about them. When the client connects to the server, the server sends a HELLO response with a heartbeat interval, and the component reads the response and automatically adjusts the heartbeat to send a ping every x seconds. Sometimes the server can send a ping to the client; this is handled automatically by the client too.

Connection Ready

When the connection is ready, after a successful login and authorization by the server, **OnDiscordReady** event is raised and then you can start to receive updates from server.

Connection Resume

If the connection closes unexpectedly, when the client tries to reconnect, it calls **OnDiscordBeforeReconnect** event. The component automatically saves all data needed to make a successful resume, but parameters can be changed if needed. If you don't want to reconnect and start a new clean session, just set Reconnect to False.

If session is resumed, **OnDiscordResumed** event is fired. If it's a new session, **OnDiscordReady** will be fired.

Dispatch Events

Events are dispatched through **OnDiscordDispatch**, so here you can read events sent by the server to the client.

```
void OnDiscordDispatch(object Sender, string aEvent, string RawData)
{
  DoLog("#discord dispatch: " + aEvent + " " + RawData);
}
```

aEvent parameter contains the event name.
RawData contains full JSON message.

HTTP Requests

In order to request info about guilds, users, or update data... instead of using gateway websocket messages, Discord requires the use of HTTP requests, Find below all methods available to make an HTTP request:

function GET_Request(const aPath: String): string;
function POST_Request(const aPath, aMessage: String): string;
function PUT_Request(const aPath, aMessage: String): string;
function PATCH_Request(const aPath, aMessage: String): string;
function DELETE_Request(const aPath: String): string;

Example: get current user info

```
result = GET_Request("/users/@me");
```

sample response from server:

```
{
"id": "637423922035480852",
"username": "test",
"avatar": null,
"discriminator": "5125",
"bot": true,
"email": null,
"verified": true,
"locale": "en-US",
"mfa_enabled": false,
"flags": 0
}
```

API | OpenAI

The OpenAI Realtime API enables low-latency, multimodal interactions including speech-to-speech conversational experiences and real-time transcription.

The component **TsgcWSAPI_OpenAI** implements the RealTime OpenAI API.

Configuration

Use the **method** property to select **Conversation** or **Transcription**, currently only Transcription mode is supported.

The **InputAudio** property allows you to customize the following data:

- **Language:** example english (value = 'en').
- **Model:** which model will be used, example: whisper-1
- **Prompt:** optional prompt to provide instructions to the model.

OpenAI

The OpenAI API uses API keys for authentication. Visit your API Keys page to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code (browsers, apps). Production requests must be routed through your own backend server where your API key can be securely loaded from an environment variable or key management service.

This **API Key** must be configured in the **OpenAIOptions.ApiKey** property of the component. Optionally, for users who belong to multiple organizations, you can set your Organization in the property **OpenAIOptions.Organization** if your account belongs to an organization.

Once the API Key is configured, find below a list of available functions to interact with the OpenAI API.

Azure

The client supports Microsoft Azure OpenAI Services, so you can use your Azure account to interact with the Azure OpenAI API too. In order to configure the client to work with Azure, follow the next steps:

1. Configure the property **OpenAIOptions.Provider** = oapvAzure
2. Set the values of ResourceName and DeploymentId (these values can be located in your Azure Account)
    1. **OpenAIOptions.AzureOptions.ResourceName** = <your resource name>.
    2. **OpenAIOptions.AzureOptions.DeploymentId** = <your deployment id>.
3. Set the API Key of your Azure Account
    1. **OpenAIOptions.ApiKey** = <azure api key>.

Transcription Example

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

```
var wsClient = new TsgcWebSocketClient(null);
var audioRecorder = new TsgcAudioRecorderWave(null);
var openAI = new TsgcWSAPI_OpenAI(null);

openAI.Client = wsClient;
openAI.AudioRecorder = audioRecorder;
openAI.OpenAIOptions.APIKey = "your-api-key-here";
openAI.OpenAIOptions.Method = OpenAIMethod.Transcription;
openAI.OpenAIOptions.Provider = OpenAIProvider.OpenAI;
openAI.InputAudio.Language = "en";
openAI.InputAudio.Model = "whisper-1";
openAI.AudioTranscriptionCompleted += OnAudioTranscriptionCompleted;
```

```
private void OnAudioTranscriptionCompleted(object sender, TsgcWSOpenAIConversation_Item_Completed aItem)
{
    Console.WriteLine("#transcription_completed: " + aItem.Transcript);
}
```

API MEXC

MEXC is a global cryptocurrency exchange that exposes streaming market and account data over secure WebSocket connections.

The WebSocket Spot API follows the specifications published in the Market Streams and User Data Streams documentation. Market channels are delivered as Protocol Buffers frames while private streams use JSON. The **TsgcWSAPI_MEXC** component manages the connection lifecycle, authentication and listen key maintenance so that applications can focus on processing the pushed data.

In addition to streaming over WebSockets, the toolkit also exposes the complete Spot HTTP API. The **TsgcHTTP_API_MEXC_Spot** component (unit **sgcHTTP_API_MEXC**) wraps every official REST endpoint so applications can query market data or trade with signed requests.

Component configuration

The base WebSocket endpoint is **wss://wbs-api.mexc.com/ws**. When **MEXCUserDataStreams.UserStream** is enabled and a valid API key is provided, the component automatically requests a listen key and appends it to the connection URL.

Drop a **TsgcWebSocketClient** and a **TsgcWSAPI_MEXC** component on the form (or create them in code), assign the client instance to the API component and activate the client to start the session.

```
TsgcWebSocketClient wsClient = new TsgcWebSocketClient();
TsgcWSAPI_MEXC mexc = new TsgcWSAPI_MEXC();
mexc.Client = wsClient;
mexc.MEXCAPI.ApiKey = "YOUR_KEY";
mexc.MEXCAPI.ApiSecret = "YOUR_SECRET";
mexc.MEXCUserDataStreams.UserStream = true;
if (wsClient.Connect()) {
  mexc.SubscribeTrade("BTCUSDT");
}
```

Properties

- **Client:** reference to the **TsgcWebSocketClient** transport. The component enables the client heartbeat (15 seconds) to keep the session alive.
- **MEXCAPI.ApiKey / ApiSecret:** Spot API credentials. The key is required for private channels; the secret is needed to sign listen key requests.
- **MEXCUserDataStreams.UserStream:** when **True** (default) the component retrieves a listen key before connecting and refreshes it every 10 minutes. Disable it to work with public feeds only.
- **MEXCUserDataStreams.ListenKeyOnDisconnect:** action executed when the WebSocket disconnects while a listen key is active:
  - **blkmxodDeleteListenKey:** revoke the listen key via REST (default).
  - **blkmxodClearListenKey:** keep the listen key server-side and clear the cached value.
  - **blkmxodDoNothing:** reuse the same listen key on the next connection.
- **ListenKey:** read-only property exposing the active listen key returned by the REST API.
- **RawMessages:** if enabled, incoming text frames are not parsed and are forwarded as plain JSON through the inherited WebSocket events.

Events

- **OnConnect / OnDisconnect:** raised when the underlying WebSocket session is established or terminated.
- **OnMEXCMarketStream:** triggered for every Protobuf market frame. The handler receives a **TsgcMEXCSpotProtoMessage** whose **MessageType** property identifies the payload (Trade, KLine, DiffDepth,

LimitDepth, BookTicker, BookTickerBatch, MiniTicker or MiniTickers). Use the helper classes in **sgcWebSocket_API_MEXC_Proto** to inspect the decoded structure.
- **OnMEXCResponse:** receives subscription acknowledgements and ping/pong responses in JSON format (**Id**, **Code**, **Message**).
- **OnMEXCUserDataStream:** fired when private account messages arrive (orders, deals, balance updates). The channel name and raw JSON payload are supplied.
- **OnMEXCException:** captures REST or WebSocket exceptions (listen key errors, HTTP failures, parsing issues, etc.).

The following sample demonstrates how to decode a trade frame:

WebSocket Subscription methods

The component exposes helpers for each public Spot channel. Invoke the **Subscribe*** method to start streaming data and the corresponding **UnSubscribe*** to stop it.

Public channels

Method	Parameters	Description
SubscribeTrade	Symbol, Interval (ms)	Aggregated trade executions delivered as Protobuf **publicdeals**.
SubscribeKline	Symbol, Interval	Candlestick updates for configurable intervals (1 minute to 1 month).
SubscribeDiffDepth	Symbol, Interval (ms)	Order book incremental depth deltas for local book maintenance.
SubscribeBookDepth	Symbol, Levels	Periodic snapshots of the limit order book with the requested depth (default 5).
SubscribeBookTicker	Symbol, Interval (ms)	Best bid/ask updates for a single trading pair.
SubscribeBookTicker-Batch	UTC flag	Aggregated book ticker updates for multiple symbols.
SubscribeMiniTickers	UTC flag	Rolling 24h mini ticker statistics for all active instruments.
SubscribeMiniTicker	Symbol, UTC flag	Mini ticker for a single symbol.

Private channels

Private topics require a valid API key. The component subscribes using the active listen key and renews it periodically.

Method	Description
SubscribeAccountUpdate	Account balance and position changes.
SubscribeAccountDeals	Execution reports for filled orders.
SubscribeAccountOrders	Order life-cycle updates (new, cancelled, rejected, etc.).

Use the corresponding **UnSubscribe*** methods to terminate a stream. The component automatically sends **PING** commands and validates the returned **PONG** to monitor connectivity.

Public REST endpoints

Publicly available REST methods.

Method	Description
Ping	Connectivity check that calls **/api/v3/ping** and returns **True** when the exchange replies with an empty object.
GetServerTime	Retrieves the current exchange timestamp from **/api/v3/time**.
GetExchangeInformation	Returns trading rules and symbol metadata from **/api/v3/exchangeInfo**.
GetOrderBook	Downloads order book snapshots from **/api/v3/depth**; set the optional **limit** parameter to control the number of levels (default 100).
GetTrades	Retrieves recent public trades from **/api/v3/trades** with an optional **limit** (default 100).
GetHistoricalTrades	Provides historical trade data with optional **limit** and **fromId** filters.
GetAggregateTrades	Returns compressed/aggregated trades via **/api/v3/aggTrades** supporting optional **limit**, **fromId**, **startTime** and **endTime** filters.
GetKlines	Downloads candlesticks from **/api/v3/klines** supporting time range and limit filters.
GetAveragePrice	Returns the current weighted average price from **/api/v3/avgPrice**.
Get24hrTicker	Retrieves 24 hour ticker statistics for one or all symbols via **/api/v3/ticker/24hr**.
GetPriceTicker	Fetches the latest price using **/api/v3/ticker/price**; pass an empty symbol to retrieve prices for every trading pair.
GetBookTicker	Obtains best bid/ask quotes from **/api/v3/ticker/bookTicker** with support for single-symbol or full-exchange requests.

Private REST endpoints

Private endpoints require valid API keys and a signed query string. The component automatically appends the timestamp, recvWindow and HMAC signature.

Method	Description
GetAccountInformation	Returns balances and permissions from **/api/v3/account**.
GetOpenOrders	Lists current open orders (optionally filtered by symbol).
GetAllOrders	Retrieves historical orders with optional time and limit filters.
GetOrder	Queries the status of a specific order by supplying either **orderId** or **origClientOrderId**.
GetMyTrades	Lists private trade executions with optional **limit**, **fromId**, **startTime** and **endTime** filters.
NewOrder	Places a live order on **/api/v3/order** (market, limit, stop, etc.) with optional **timeInForce**, **quantity**, **quoteOrderQty**, **price**, **newClientOrderId**, **stopPrice**, **icebergQty** and extra parameters.
TestNewOrder	Sends a validation-only request to **/api/v3/order/test** accepting the same parameter set as **NewOrder**.
CancelOrder	Cancels a specific order on **/api/v3/order** using **orderId** or **origClientOrderId**.
CancelAllOrders	Bulk cancels all open orders for a symbol via **/api/v3/openOrders**.
GetSubAccounts	Lists managed sub-accounts (**/api/v3/sub-account/list**).
GetSubAccountAssets	Returns balances for a specific sub-account.
TransferSubAccount	Transfers assets between sub-accounts through **/api/v3/sub-account/transfer**; provide the amount and optionally the transfer **type**.
GetDepositAddress	Requests deposit addresses with an optional **network** filter.
GetDepositHistory	Fetches deposit records with optional **coin**, **status**, **startTime** and **endTime** filters.
GetWithdrawHistory	Returns withdrawal history filtered by **coin**, **status** and optional time range.
Withdraw	Submits a withdrawal request via **/api/v3/capital/withdraw/apply** including the mandatory **coin**, **address** and **amount** plus optional **network**, **addressTag**, **withdrawOrderId** and extra parameters.
BatchOrders	Submit batch orders for multiple symbols in a single request.
GetTradeFee	Get the trade fee rate for a specific symbol.

GetDepositHistory	Fetches deposit records with optional **coin**, **status**, **startTime** and **endTime** filters.
CancelWithdraw	Cancel a pending withdrawal request by withdrawal ID.
CreateInternalTransfer	Transfer assets between accounts internally.
GetTransferHistory	Get the history of internal transfers between accounts.

API MEXC Futures

MEXC perpetual and delivery contracts expose a dedicated streaming API documented in the official Futures WebSocket specification. The **TsgcWSAPI_MEXC_Futures** component encapsulates the connection, login handshake and topic management required to consume real-time derivatives data.

MEXC also maintains a REST interface for derivatives trading. The **TsgcHTTP_API_MEXC_Futures** component contained in **sgcHTTP_API_MEXC** complements the WebSocket feeds with HTTP helpers that map one-to-one to the official endpoints.

Component configuration

The Futures WebSocket endpoint is **wss://contract.mexc.com/edge**. Messages are encoded as JSON objects and the exchange requires periodic **ping**/**pong** frames which are handled automatically by the component.

To receive private notifications (order and account data) you must authenticate. Set **MEXCAPI.ApiKey** and **MEXCAPI.ApiSecret** before activating the WebSocket client. When credentials are present the component signs a login request (HMAC SHA256) as soon as the socket connects.

```
TsgcWebSocketClient wsClient = new TsgcWebSocketClient();
TsgcWSAPI_MEXC_Futures futures = new TsgcWSAPI_MEXC_Futures();
futures.Client = wsClient;
futures.MEXCAPI.ApiKey = "YOUR_KEY";
futures.MEXCAPI.ApiSecret = "YOUR_SECRET";
if (wsClient.Connect()) {
  futures.SubscribeDepth("BTC_USDT", true);
}
```

Properties

- **Client:** WebSocket transport used by the API component. Heartbeat support is enabled to keep the socket active.
- **MEXCAPI.ApiKey / ApiSecret:** required to sign the login request. Without credentials only public channels are accessible.
- **RawMessages:** bypasses the JSON parser when enabled so that the raw text frames are exposed through the base events.

Events

- **OnConnect / OnDisconnect:** WebSocket lifecycle notifications.
- **OnMEXCLogin:** raised after a successful authenticated login.
- **OnMEXCSubscribed / OnMEXCUnsubscribed:** confirmation of subscription changes. The event exposes the channel name returned by MEXC.
- **OnMEXCMessage:** delivers every market or account update together with the channel identifier.
- **OnMEXCError:** triggered when the server replies with an **rs.error** payload (invalid parameters, authentication failure, etc.).
- **OnMEXCException:** reports local exceptions raised while processing JSON data or performing network operations.

The following handler records subscription confirmations and prints incoming depth snapshots:

WebSocket Subscription methods

Each helper wraps a **sub**/**unsub** request as described by the official API. Use the matching **UnSubscribe*** method to cancel the stream.

Method	Parameters	Description
SubscribeDeal	Symbol	Trades executed on the contract (channel: **deal**).
SubscribeTickers	–	Global ticker statistics for all contracts.
SubscribeTicker	Symbol	Ticker summary for a single instrument.
SubscribeDepth	Symbol, Compress	Incremental order book updates
SubscribeDepthFull	Symbol, Level	Full depth snapshots with configurable number of levels (default 20).
SubscribeKline	Symbol, Interval	Candlestick data for supported timeframes (Min1, Min5, Min15, Min30, Min60, Hour4, Hour12, Day1, Week1, Month1).
SubscribeFundingRate	Symbol	Latest funding rate announcements.
SubscribeIndexPrice	Symbol	Underlying index price stream.
SubscribeFairPrice	Symbol	Mark (fair) price updates pushed by the exchange.

When the server acknowledges a request the **OnMEXCSubscribed** event fires with the channel name (for example **rs.sub.depth.BTC_USDT**). Errors are forwarded through **OnMEXCError** including the server message for troubleshooting.

Private channels

Private channels require a valid API key. The component authenticates automatically when credentials are provided.

Method	Description
SubscribePersonalOrder	Personal order updates.
SubscribePersonalOrderDeal	Personal order deal (fill) updates.
SubscribePersonalPosition	Personal position updates.
SubscribePersonalPlanOrder	Personal plan (trigger) order updates.
SubscribePersonalStopOrder	Personal stop order updates.
SubscribePersonalStop-PlanOrder	Personal stop plan order updates.
SubscribePersonalRiskLimit	Personal risk limit updates.
SubscribePersonalADLLevel	Personal ADL (auto-deleveraging) level updates.
SubscribePersonalAsset	Personal asset updates.

Public REST endpoints

Publicly available REST methods.

Method	Description
GetPing	Checks API reachability via **/api/v1/ping**.
GetServerTime	Returns the server timestamp from **/api/v1/time**.
GetContracts	Provides the list of available contracts (**/api/v1/contract/detail**).
GetDepth	Downloads order book depth from **/api/v1/contract/depth**; use the optional **limit** parameter (default 50) to select the number of levels.
GetDeals	Retrieves recent trades using **/api/v1/contract/deals** with an optional **limit** (default 100).

GetKlines	Returns candlestick data through **/api/v1/contract/kline** with optional **startTime**, **endTime** and **limit** filters (default 200).
GetIndexPrice	Fetches the underlying index price for the requested symbol (**/api/v1/contract/indexPrice**).
GetFairPrice	Returns the fair (mark) price for a contract from **/api/v1/contract/fairPrice**.
GetFundingRate	Reports the latest funding rate for a symbol using **/api/v1/contract/fundingRate**.
GetAllTickers	Get all futures contract tickers with real-time price summaries.
GetFundingRateHistory	Get historical funding rate records for a contract.
GetFairPriceKline	Get fair (mark) price kline/candlestick data for a contract.
GetIndexPriceKline	Get index price kline/candlestick data for a contract.

Private REST endpoints

Private derivatives endpoints require API credentials. The component signs each request with the timestamp and recvWindow automatically.

Method	Description
GetAccountAssets	Returns margin balances from **/api/v1/private/account/assets**.
GetPositionList	Lists current positions via **/api/v1/private/position/list**, optionally filtered by symbol.
SetPositionLeverage	Updates leverage for a symbol and, if supplied, the margin mode.
PlaceOrder	Places a futures order on **/api/v1/private/order** providing **symbol**, **side**, **positionSide**, **type** and **quantity** plus optional **price**, **clientOrderId** and extra parameters.
CancelOrder	Cancels a specific order using **/api/v1/private/order/cancel** by **orderId** or **clientOrderId**.
CancelAllOrders	Cancels all open orders for a symbol via **/api/v1/private/order/cancel-all**.
GetOpenOrders	Lists current open orders through **/api/v1/private/order/list/open** with an optional symbol filter.
GetOrderHistory	Retrieves order history (**/api/v1/private/order/list/history**) with optional symbol, time range and limit parameters.
GetFundingHistory	Returns past funding payments from **/api/v1/private/account/funding** with optional **startTime**, **endTime** and **limit** filters.
GetOpenPositions	Get all currently open positions.
ChangeMargin	Change the margin amount for a position.
GetPositionMode	Get the current position mode (one-way or hedge mode).
ChangePositionMode	Change the position mode between one-way and hedge mode.
PlaceBatchOrder	Place multiple futures orders in a single batch request.
GetOrderDetail	Get detailed information for a specific order.
GetOrderDealDetails	Get fill/deal details for a specific order.
PlaceTriggerOrder	Place a trigger (plan) order that executes when conditions are met.
CancelTriggerOrder	Cancel a specific trigger order.
CancelAllTriggerOrders	Cancel all open trigger orders.
GetTriggerOrders	Get the list of trigger (plan) orders.
GetStopOrders	Get the list of stop orders.
CancelStopOrder	Cancel a specific stop order.
CancelAllStopOrders	Cancel all open stop orders.

API Bitget

Bitget

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

Use the **BitgetChannel** property to select the channel type:
- **bgcSpot:** connect to the Spot WebSocket API.
- **bgcFutures:** connect to the Futures WebSocket API.

You can configure the following properties in the Bitget property.
- **ApiKey:** you can request a new api key in your Bitget account, just copy the value to this property. If the APIKey is set, the client will connect to the websocket private server. If it's empty, will connect to the WebSocket public server.
- **ApiSecret:** it's the secret value of the api.
- **Passphrase:** it's the custom string defined when creating a new api key.

Connection

When the client successfully connects to Bitget servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Bitget servers. If you are connecting to the private websocket channel, you must wait till **OnBitgetAuthentication** event is fired and check if the success parameter is true, before subscribing to any channel.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWS_API_Bitget oBitget = new TsgcWS_API_Bitget();
oBitget.Client = oClient;
oBitget.Bitget.ApiKey = "your_api_key";
oBitget.Bitget.ApiSecret = "your_api_secret";
oBitget.Bitget.Passphrase = "your_passphrase";
oBitget.BitgetChannel = bgcSpot;
oClient.Active = true;
void OnConnect(TsgcWSConnection Connection)
{
  DoLog("#Bitget Connected");
}
```

Events

The Bitget client implements the following events to control the connection flow and get data sent from the WebSocket server:
- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnBitgetAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnBitgetSubscribe:** when the client subscribes to a websocket channel.
- **OnBitgetUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnBitgetData:** when the client receives data from the server.
- **OnBitgetError:** when there is any error during the Bitget websocket connection.
- **OnBitgetHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **channels**:

Method	Public or Private	Description
SubscribeTicker	Public	Subscribe to the ticker stream for an instrument.
SubscribeTrade	Public	Subscribe to the recent trades stream.
SubscribeCandle	Public	Subscribe to the candlestick stream. Supports multiple intervals (e.g. candle1m, candle5m, candle1H).
SubscribeOrderBook	Public	Subscribe to the order book stream. Supports different depth levels.
SubscribeOrders	Private	Subscribe to order updates. Requires authentication.
SubscribePositions	Private	Subscribe to position updates. Requires authentication.
SubscribeAccount	Private	Subscribe to account balance updates. Requires authentication.

Find below an example of subscribing to private websocket channels after a successful authentication.

```
void OnBitgetAuthentication(TObject Sender, bool aSuccess, string aError, string aRawMessage)
{
  if (aSuccess == true)
  {
    oBitget.SubscribeOrders("BTCUSDT");
    oBitget.SubscribeAccount();
  }
}
```

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. Access the REST API through the **REST_API** property of the component.

Method	Public / Private	Description
GetServerTime	Public	Get the server time.
GetTickers	Public	Get ticker information for one or all symbols.
GetOrderBook	Public	Get the order book for a symbol.
GetCandles	Public	Get candlestick/kline data for a symbol.
GetRecentTrades	Public	Get the most recent trades for a symbol.
PlaceOrder	Private	Place a new order.
CancelOrder	Private	Cancel an existing order.
GetOpenOrders	Private	Get the list of open orders.
GetOrderDetail	Private	Get details of a specific order.
GetOrderHistory	Private	Get the order history.
GetAccountAssets	Private	Get account asset information.
GetAccountInfo	Private	Get account information.

Find below an example of using the REST API.

```
TsgcWS_API_Bitget oBitget = new TsgcWS_API_Bitget();
oBitget.Bitget.ApiKey = "your_api_key";
oBitget.Bitget.ApiSecret = "your_api_secret";
oBitget.Bitget.Passphrase = "your_passphrase";
// Get tickers
oBitget.REST_API.GetTickers("SPOT");
// Place an order
oBitget.REST_API.PlaceOrder("BTCUSDT", "buy", "market", "normal", "0.01");
```

API GateIO

Gate.io

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

Use the **GateIOChannel** property to select the channel type:
- **giocSpot:** connect to the Spot WebSocket API.
- **giocFutures:** connect to the Futures WebSocket API.

You can configure the following properties in the GateIO property.
- **ApiKey:** you can request a new api key in your Gate.io account, just copy the value to this property. If the APIKey is set, the client will authenticate against the private server. If it's empty, only public channels will be available.
- **ApiSecret:** it's the secret value of the api.

Connection

When the client successfully connects to Gate.io servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Gate.io servers. If you are connecting to the private websocket channel, you must wait till **OnGateIOAuthentication** event is fired and check if the success parameter is true, before subscribing to any private channel.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWS_API_GateIO oGateIO = new TsgcWS_API_GateIO();
oGateIO.Client = oClient;
oGateIO.GateIO.ApiKey = "your_api_key";
oGateIO.GateIO.ApiSecret = "your_api_secret";
oGateIO.GateIOChannel = giocSpot;
oClient.Active = true;
void OnConnect(TsgcWSConnection Connection)
{
  DoLog("#GateIO Connected");
}
```

Events

The GateIO client implements the following events to control the connection flow and get data sent from the WebSocket server:
- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnGateIOAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnGateIOSubscribe:** when the client subscribes to a websocket channel.
- **OnGateIOUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnGateIOData:** when the client receives data from the server.
- **OnGateIOError:** when there is any error during the GateIO websocket connection.
- **OnGateIOHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **channels**:

Method	Public or Private	Description
SubscribeTicker	Public	Subscribe to the ticker stream for a symbol.
SubscribeTrades	Public	Subscribe to the recent trades stream.
SubscribeCandlesticks	Public	Subscribe to the candlestick stream. Supports multiple intervals (e.g. 1m, 5m, 15m, 30m, 1h).
SubscribeOrderBook	Public	Subscribe to the order book stream. Supports configurable depth levels and update intervals.
SubscribeOrders	Private	Subscribe to order updates. Requires authentication.
SubscribeBalances	Private	Subscribe to balance updates. Requires authentication.

Find below an example of subscribing to public websocket channels.

```
void OnConnect(TsgcWSConnection Connection)
{
  oGateIO.SubscribeTicker("BTC_USDT");
  oGateIO.SubscribeOrderBook("BTC_USDT", "20", "100ms");
}
```

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. Access the REST API through the **REST_API** property of the component.

Method	Public / Private	Description
GetCurrencyPairs	Public	Get information about available currency pairs.
GetTickers	Public	Get ticker information for one or all currency pairs.
GetOrderBook	Public	Get the order book for a currency pair.
GetCandlesticks	Public	Get candlestick/kline data for a currency pair.
GetTrades	Public	Get the most recent trades for a currency pair.
PlaceOrder	Private	Place a new order. Supports limit and market orders.
CancelOrder	Private	Cancel an existing order.
GetOrder	Private	Get details of a specific order.
GetOpenOrders	Private	Get the list of open orders.
GetSpotAccounts	Private	Get spot account information.

Find below an example of using the REST API.

```
TsgcWS_API_GateIO oGateIO = new TsgcWS_API_GateIO();
oGateIO.GateIO.ApiKey = "your_api_key";
oGateIO.GateIO.ApiSecret = "your_api_secret";
// Get tickers
oGateIO.REST_API.GetTickers("BTC_USDT");
// Place an order
oGateIO.REST_API.PlaceOrder("BTC_USDT", "buy", "0.001", "50000", "limit");
```

API Deribit

Deribit
Deribit is a cryptocurrency derivatives exchange offering futures and options trading on Bitcoin and Ethereum.

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

You can configure the following properties in the Deribit property.

- **ApiKey:** you can request a new api key in your Deribit account, just copy the value to this property. If the APIKey is set, the client will authenticate against the private server. If it's empty, only public channels will be available.
- **ApiSecret:** it's the secret value of the api.
- **TestNet:** if enabled, will connect to the Deribit TestNet environment (disabled by default). Useful for testing without risking real funds.

Connection

When the client successfully connects to Deribit servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Deribit servers. If you are connecting to the private websocket channel, you must wait till **OnDeribitAuthentication** event is fired and check if the success parameter is true, before subscribing to any private channel.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWS_API_Deribit oDeribit = new TsgcWS_API_Deribit();
oDeribit.Client = oClient;
oDeribit.Deribit.ApiKey = "your_api_key";
oDeribit.Deribit.ApiSecret = "your_api_secret";
oDeribit.Deribit.TestNet = false;
oClient.Active = true;
void OnConnect(TsgcWSConnection Connection)
{
  DoLog("#Deribit Connected");
}
```

Events

The Deribit client implements the following events to control the connection flow and get data sent from the WebSocket server:

- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnDeribitAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnDeribitSubscribe:** when the client subscribes to a websocket channel.
- **OnDeribitUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnDeribitData:** when the client receives data from the server.
- **OnDeribitError:** when there is any error during the Deribit websocket connection.
- **OnDeribitHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **Public channels**:

Method	Description

SubscribeTicker	Subscribe to the ticker stream for an instrument. Provides real-time price and volume data.
SubscribeTrades	Subscribe to the recent trades stream for an instrument.
SubscribeOrderBook	Subscribe to the order book stream. Supports configurable grouping, depth levels and update intervals.
SubscribeCandle	Subscribe to the candlestick stream. Supports multiple resolutions (e.g. 1, 3, 5, 10, 15, 30, 60, 120, 180, 360, 720, 1D).
SubscribePerpetual	Subscribe to perpetual contract data for an instrument. Provides funding rate and other perpetual-specific information.
SubscribeBookChange	Subscribe to order book change notifications for an instrument.

You can subscribe to the following **Private channels** (require authentication):

Method	Description
SubscribeOrders	Subscribe to order updates for an instrument.
SubscribePositions	Subscribe to position updates. Supports filtering by currency and instrument kind.
SubscribeAccountChanges	Subscribe to account balance changes for a currency.
SubscribeUserTrades	Subscribe to user trade updates for an instrument.

Find below an example of subscribing to websocket channels.

```
void OnDeribitAuthentication(TObject Sender, bool aSuccess, string aError, string aRawMessage)
{
  if (aSuccess == true)
  {
    oDeribit.SubscribeOrders("BTC-PERPETUAL");
    oDeribit.SubscribePositions("BTC");
  }
}
```

REST API

The REST API provides Public and Private methods to request data from markets, trading, account, and wallet. Access the REST API through the **REST_API** property of the component.

Market Data (Public)

Method	Description
GetInstruments	Get available instruments for a currency.
GetTicker	Get ticker information for an instrument.
GetOrderBook	Get the order book for an instrument.
GetTrades	Get the most recent trades for an instrument.
GetCurrencies	Get the list of supported currencies.
GetIndexPrice	Get the current index price for an index name.
GetFundingRateHistory	Get the funding rate history for an instrument.
GetFundingRateValue	Get the current funding rate value for an instrument.
GetBookSummaryByCurrency	Get book summary for all instruments of a currency.
GetBookSummaryByInstrument	Get book summary for a specific instrument.

Trading (Private)

Method	Description
Buy	Place a buy order. Supports market and limit order types.
Sell	Place a sell order. Supports market and limit order types.
CancelOrder	Cancel an existing order by order ID.
CancelAllOrders	Cancel all open orders.
CancelAllByInstrument	Cancel all open orders for a specific instrument.
EditOrder	Edit an existing order (change amount or price).
GetOpenOrders	Get the list of open orders for a currency.
GetOrderState	Get the state of a specific order.

GetOrderHistory	Get the order history for a currency.

Account (Private)

Method	Description
GetPositions	Get positions for a currency.
GetAccountSummary	Get the account summary for a currency.
GetSubAccounts	Get the list of sub-accounts.
GetTransactionLog	Get the transaction log for a currency.

Wallet (Private)

Method	Description
GetDeposits	Get the deposit history for a currency.
GetWithdrawals	Get the withdrawal history for a currency.
GetTransfers	Get the transfer history for a currency.

Find below an example of using the REST API.

```
TsgcWS_API_Deribit oDeribit = new TsgcWS_API_Deribit();
oDeribit.Deribit.ApiKey = "your_api_key";
oDeribit.Deribit.ApiSecret = "your_api_secret";
// Get ticker
oDeribit.REST_API.GetTicker("BTC-PERPETUAL");
// Place a buy order
oDeribit.REST_API.Buy("BTC-PERPETUAL", 10, "market");
// Get account summary
oDeribit.REST_API.GetAccountSummary("BTC");
```

API Crypto.com

Crypto.com

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

Use the **Channel** property to select the channel type:
- **cccMarket:** connect to the Market WebSocket API for public market data.
- **cccUser:** connect to the User WebSocket API for private account data (requires authentication).

You can configure the following properties in the CryptoCom property.
- **ApiKey:** you can request a new api key in your Crypto.com account, just copy the value to this property. Required for private channels and authenticated REST API calls.
- **ApiSecret:** it's the secret value of the api.

Connection

When the client successfully connects to Crypto.com servers, the event **OnConnect** is fired. After the event **OnConnect** is fired, then you can start to **send** and **receive messages** to/from Crypto.com servers. If you are connecting to the User channel, you must wait till **OnCryptoComAuthentication** event is fired and check if the success parameter is true, before subscribing to any private channel.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWS_API_CryptoCom oCryptoCom = new TsgcWS_API_CryptoCom();
oCryptoCom.Client = oClient;
oCryptoCom.CryptoCom.ApiKey = "your_api_key";
oCryptoCom.CryptoCom.ApiSecret = "your_api_secret";
oCryptoCom.Channel = cccMarket;
oClient.Active = true;
void OnConnect(TsgcWSConnection Connection)
{
  DoLog("#CryptoCom Connected");
}
```

Events

The Crypto.com client implements the following events to control the connection flow and get data sent from the WebSocket server:
- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnCryptoComAuthentication:** fired when the client authenticates against the Private WebSocket Server.
- **OnCryptoComSubscribe:** when the client subscribes to a websocket channel.
- **OnCryptoComUnSubscribe:** when the client unsubscribes from a websocket channel.
- **OnCryptoComData:** when the client receives data from the server.
- **OnCryptoComError:** when there is any error during the Crypto.com websocket connection.
- **OnCryptoComHTTPException:** when there is any error during the REST request.

WebSocket API

The websocket feed provides real-time market data updates for orders and trades.
You can subscribe to the following **Market channels** (public):

Method	Description

SubscribeTicker	Subscribe to the ticker stream for an instrument. Provides real-time price and volume data.
SubscribeTrade	Subscribe to the recent trades stream for an instrument.
SubscribeCandlestick	Subscribe to the candlestick stream. Supports multiple intervals (e.g. 5m, 15m, 30m, 1h, 4h, 1D).
SubscribeBook	Subscribe to the order book stream. Supports configurable depth levels (e.g. 10, 50).

You can subscribe to the following **User channels** (private, require authentication):

Method	Description
SubscribeOrders	Subscribe to order updates. Optionally filter by instrument name.
SubscribeUserTrades	Subscribe to user trade updates. Optionally filter by instrument name.
SubscribeBalance	Subscribe to balance updates for your account.

Find below an example of subscribing to market channels.

```
void OnConnect(TsgcWSConnection Connection)
{
  oCryptoCom.SubscribeTicker("BTC_USDT");
  oCryptoCom.SubscribeBook("BTC_USDT", "10");
}
```

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. Access the REST API through the **REST_API** property of the component.

Method	Public / Private	Description
GetInstruments	Public	Get the list of available instruments.
GetTicker	Public	Get ticker information for one or all instruments.
GetOrderBook	Public	Get the order book for an instrument.
GetTrades	Public	Get the most recent trades for an instrument.
GetCandlestick	Public	Get candlestick/kline data for an instrument.
CreateOrder	Private	Create a new order. Supports limit and market orders.
CancelOrder	Private	Cancel an existing order.
CancelAllOrders	Private	Cancel all open orders for an instrument.
GetOpenOrders	Private	Get the list of open orders.
GetOrderDetail	Private	Get details of a specific order.
GetOrderHistory	Private	Get the order history.
GetAccountSummary	Private	Get account summary information.

Find below an example of using the REST API.

```
TsgcWS_API_CryptoCom oCryptoCom = new TsgcWS_API_CryptoCom();
oCryptoCom.CryptoCom.ApiKey = "your_api_key";
oCryptoCom.CryptoCom.ApiSecret = "your_api_secret";
// Get instruments
oCryptoCom.REST_API.GetInstruments();
// Create an order
oCryptoCom.REST_API.CreateOrder("BTC_USDT", "BUY", "MARKET", "", "0.001");
// Get account summary
oCryptoCom.REST_API.GetAccountSummary();
```

API HTX

HTX
HTX (formerly Huobi) is an international multi-language cryptocurrency exchange. The HTX API component provides an HTTP REST API to complement the existing Huobi WebSocket API.

APIs supported

- **WebSocket API:** connect to a websocket server and provides real-time market data updates, account changes and more. See the Huobi API documentation for WebSocket subscription methods.
- **REST API:** send HTTP requests to get market data, place orders, account data...

Properties

You can configure the following properties in the Huobi property (for WebSocket) or HTXOptions property (for REST API).

- **ApiKey:** you can request a new api key in your HTX account, just copy the value to this property. If the APIKey is set, the client will connect to the websocket private server. If it's empty, will connect to the WebSocket public server.
- **ApiSecret:** it's the secret value of the api.

Connection

The HTX component extends the Huobi WebSocket API client. When the client successfully connects to HTX servers, the event **OnConnect** is fired. If the ApiKey is not empty, the client will attempt to connect to the private websocket server, so only the private methods will be available. If the ApiKey is empty, the client will connect to the public websocket server and only the public methods will be available.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWS_API_Huobi oHTX = new TsgcWS_API_Huobi();
oHTX.Client = oClient;
oHTX.Huobi.ApiKey = "your_api_key";
oHTX.Huobi.ApiSecret = "your_api_secret";
oClient.Active = true;
void OnConnect(TsgcWSConnection Connection)
{
  DoLog("#HTX Connected");
}
```

Events

The HTX client implements the following events to control the connection flow and get data sent from the WebSocket server:

- **OnConnect:** fired when the websocket client connects to the WebSocket Server.
- **OnDisconnect:** fired when the websocket client disconnects from the WebSocket Server.
- **OnHuobiSubscribed:** event called after a successful subscription.
- **OnHuobiUnSubscribed:** event called after a successful unsubscription.
- **OnHuobiUpdate:** every time there is an update in data (kline, market depth...) this event is called.
- **OnHuobiError:** if there is an error in the HTX API, this event will provide information about the error.

WebSocket API

The WebSocket API provides real-time market data updates. See the Huobi API documentation for the full list of WebSocket subscription methods including public channels (SubscribeKLine, SubscribeMarketDepth, SubscribeTradeDetail, etc.) and private channels (SubscribeOrderUpdates, SubscribeTradeClearing, SubscribeAccountChange).

REST API

The REST API provides Public and Private methods to request data from markets and private accounts. The REST API is accessed through the **TsgcHTTP_API_HTX** component.

Market Data (Public)

Method	Description
GetServerTime	Get the server time.
GetSymbols	Get the list of available trading symbols.
GetCurrencys	Get the list of supported currencies.
GetMarketTickers	Get tickers for all trading symbols.
GetMarketDetail	Get the 24-hour market detail for a symbol.
GetMarketDepth	Get the order book depth for a symbol. Supports configurable depth type and depth levels.
GetMarketTrade	Get the most recent trade for a symbol.
GetMarketHistoryTrade	Get the most recent trades history for a symbol.
GetMarketHistoryKline	Get candlestick/kline data. Supports multiple periods (1min, 5min, 15min, 30min, 60min, 4hour, 1day, 1mon, 1week, 1year).

Trading (Private)

Method	Description
PlaceOrder	Place a new order. Supports buy-market, sell-market, buy-limit, sell-limit and other order types.
CancelOrder	Cancel an existing order by order ID.
GetOrder	Get details of a specific order.
GetOpenOrders	Get the list of open orders for an account.
GetOrderHistory	Get the order history for a symbol.

Account (Private)

Method	Description
GetAccounts	Get the list of accounts.
GetAccountBalance	Get the balance of a specific account.
GetAccountAssetValuation	Get the total asset valuation for an account type (e.g. spot).

Find below an example of using the REST API.

```
TsgcHTTP_API_HTX oHTX = new TsgcHTTP_API_HTX();
oHTX.HTXOptions.ApiKey = "your_api_key";
oHTX.HTXOptions.ApiSecret = "your_api_secret";
// Get market tickers
oHTX.GetMarketTickers();
// Get kline data
oHTX.GetMarketHistoryKline("btcusdt", "1min", 150);
// Place an order
oHTX.PlaceOrder("12345678", "btcusdt", "buy-limit", "0.001", "50000");
// Get account balance
oHTX.GetAccountBalance("12345678");
```

WhatsApp Cloud API

Whatsapp

Send and **receive messages** using a cloud-hosted version of the **WhatsApp Business Platform**. The **Cloud API** allows you to implement WhatsApp Business APIs without the cost of hosting of your own servers and also allows you to more easily scale your business messaging. The Cloud API supports up to 80 messages per second of combined sending and receiving (inclusive of text and media messages).

The WhatsApp Business API allows medium and large businesses to communicate with their customers at scale. Using the API, businesses can build systems that connect thousands of customers with agents or bots, enabling both programmatic and manual communication. Additionally, you can integrate the API with numerous backend systems, such as CRM and marketing platforms.

Features

Businesses will get all the new features faster on Cloud API. Right now, WhatsApp Business Cloud API comes with all the features that are available with WhatsApp Business API.

Useful features of WhatsApp Cloud API:

- **Integrate** WhatsApp messaging with tools like **CRM**, **analytics**, and **third-party** apps
- **Green Tick**, verified WhatsApp Business profile
- WhatsApp **Broadcast** & **Bulk Messaging**
- No app or interface, use via BSPs or CRM
- **WhatsApp Chatbot** & chat **automation** using third-party apps
- **Schedule** WhatsApp messages at a large scale
- **Interactive messaging** features include List messages, reply buttons, CTA messages

Most common uses

- **Configuration**
  - **WhatsApp Create App**
  - **WhatsApp Phone Number Id**
  - **WhatsApp Token**
  - **WhatsApp Webhook**
  - **WhatsApp Security**

- **Messages**
  - **WhatsApp Send Messages**
  - **WhatsApp Send Interactive Messages**
  - **WhatsApp Send Template Messages**
  - **WhatsApp Receive Messages and Status Notifications**
  - **WhatsApp Send Files**
  - **WhatsApp Download Media**

Get Started

To send and receive a first message using a test number, complete the following steps:

1. Set up Developer Assets and Platform Access

- Register as a Meta Developer
- Enable two-factor authentication for your account

- Create a Meta App: Go to **developers.facebook.com** > **My Apps** > **Create App**. Select the "Business" type and follow the prompts on your screen.

From the App Dashboard, click on the app you would like to connect to WhatsApp. Scroll down to find the "WhatsApp" product and click **Set up**.

Next, you will see the option to select an existing Business Manager (if you have one) or, if you would like, the onboarding process can create one automatically for you (you can customize your business later, if needed). Make a selection and click **Continue**.

When you click **Continue**, the onboarding process performs the following actions:

- Your App is associated with the Business Manager that you chose, or that was created automatically.
- A WhatsApp test phone number is added to your business. You can use this test phone number to explore the WhatsApp Business Platform without registering or migrating a real phone number. Test phone numbers can send unlimited messages to up to 5 recipients (which can be anywhere in the world).

2. Send a Test Message

Now, you can open your IDE and create a new project. Drop a TsgcWhatsapp_Client component and fill the following properties:

- **WhatsappOptions.PhoneNumberId:** is the ID of the Phone Number used to send messages.
- **WhatsappOptions.Token:** is the Temporary Access Token valid for 24 hours.

Once those 2 properties have been properly configured, call the method **SendTest** to send your **First message** to a phone number using the **Whatsapp Business Platform**.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendTest("34605889421");
```

3. Configure a Webhook

To get alerted when you receive a message or when a message's status has changed, you need to set up a Webhooks endpoint for your app. Setting up Webhooks doesn't affect the status of your phone number and does not interfere with you sending or receiving messages.

To get started, first you need to create the endpoint, so first configure the **ServerOptions** property of WhatsApp Client component and configure the following properties:

- **ServerOptions:** here you can configure the IP Address to bind, the Listening Port, if it's using SSL (the WebHook must run in a secure server, you can configure your server to use SSL or Proxy the WebHook requests to a none HTTPs server). The server is based on TsgcWebSocketHTTPServer.
  - **WebhookOptions:** this property allows you to set the Webhook properties that later will be configured in your developer facebook account.
    - **Endpoint:** it's the name of the endpoint, by default is /webhook. Example: if your server is listening on https://www.esegece.com, the endpoint will be "https://www.esegece.com/webhook"
    - **Token:** it's a security string that can contain any value defined by you. It's used to verify the Webhook registration is correct.

After configuring the server, you can use the method **StartServer** to start the server and accept the incoming requests.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.ServerOptions.WebhookOptions.PhoneNumberId = "/webhook";
oClient.ServerOptions.WebhookOptions.Token = "MySecretToken";
oClient.StartServer();
```

Once your endpoint is ready, go to your App Dashboard.

In your App Dashboard, find the WhatsApp product and click **Configuration**. Then, find the webhooks section and click **Configure a webhook**. After the click, a dialog appears on your screen and asks you for two items:

- Callback URL: This is the URL Meta will be sending the events to.
- Verify Token: This string is set up by you, when you create your webhook endpoint.

After adding the information, click **Verify and Save**.

Back in the App Dashboard, click **WhatsApp** > **Configuration** in the left-side panel. Under Webhooks, click **Manage**. A dialog box will open with all the objects you can get notified about. To receive messages from your users, click **Subscribe** for **messages**.

4. Receive a test message

Every time a new message is received, the client event **OnMessageReceived** will be called.

```
void OnMessageReceived(TsgcWhatsApp_Client Sender, TsgcWhatsApp_Receive_Message Message, ref bool MarkAsRead)
{
  DoLog("Received: " + Message.Text);
}
```

Now that your Webhook is set up, send a message to the test number you have used. You should immediately get a Webhooks notification with the content of your message!

The WhatsApp API does not allow sending free text messages to phones that have not contacted you before (within the latest 24 hours). The only way to send a text message to a phone that has never texted your developer account number is by sending a Template (previously approved by Meta). To override this limitation for testing free text messages, first send a WhatsApp message from the destination number to your developer account number, and then you will be able to send free text messages for 24 hours.

Events

OnBeforeSendMessage

This event is called before the message is sent to the WhatsApp servers. You can access the internal message through the RawMessage parameter.

OnBeforeSubscribe

This event is called before the server subscribes to a topic. Use the Accept parameter to allow or deny the subscription. By default, the server will subscribe to all events requested.

OnRawMessage

This event is called when the server receives a new message that has not yet been parsed, so you get access to the raw message.

OnMessageReceived

This event is called after the server receives and parses a new message. If you set the MarkAsRead parameter to True, the sender will receive a double check.

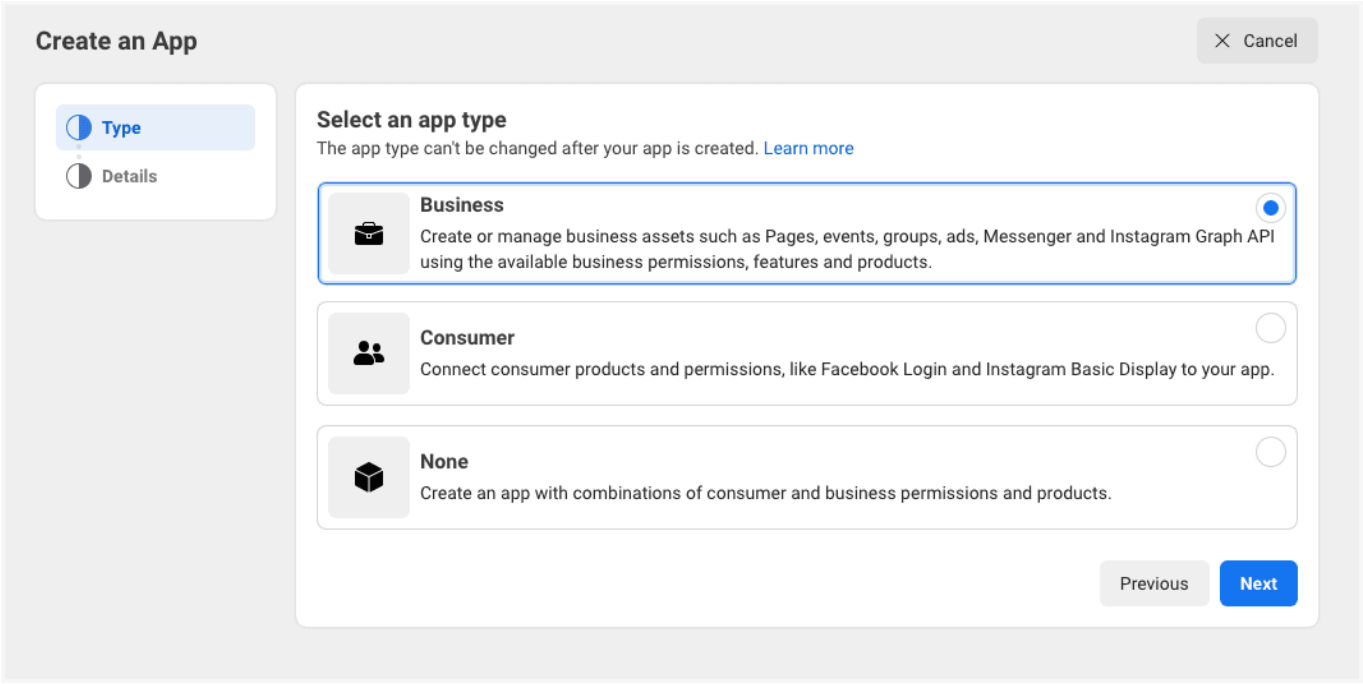
OnMessageSent

This event is called every time the server receives a new status message about the message previously sent. Read the Status property to know if the message has been sent, delivered or read.

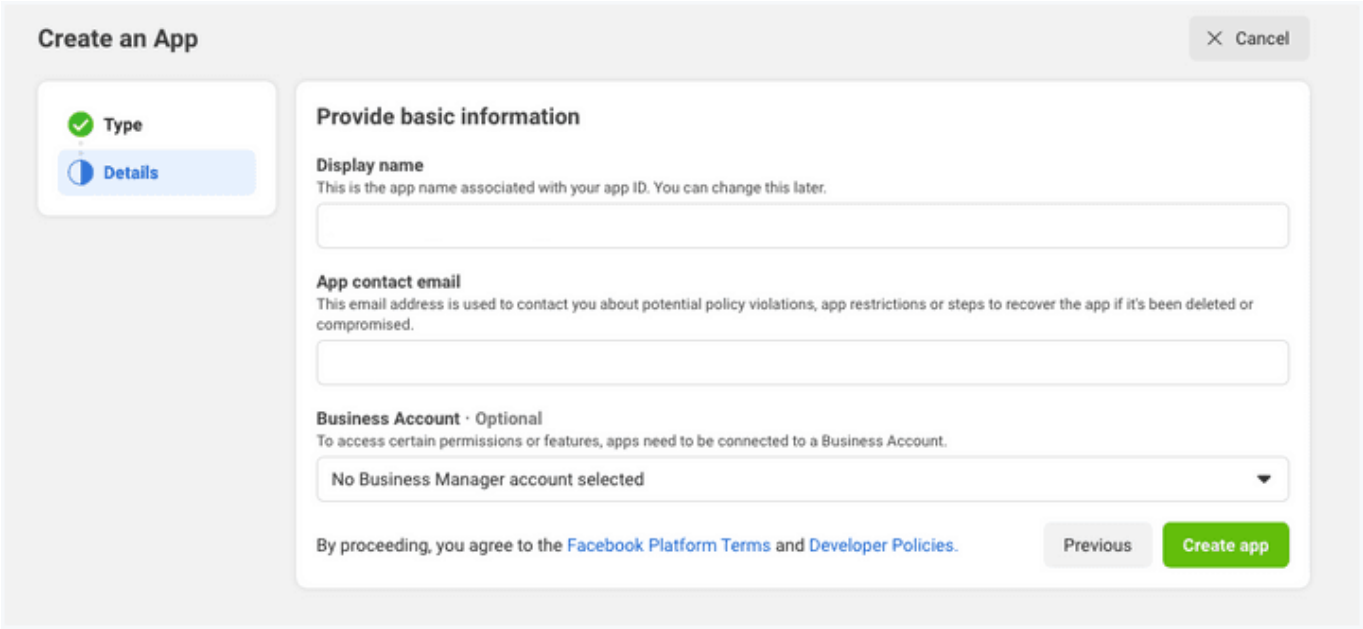
WhatsApp Create App

Go to developers.facebook.com and **Create App**.

Select **Business Type** as the app type and proceed.



Provide a name for your app (avoid using trademarked names such as “WhatsApp” or “Facebook”).



Once the app has been created, click the **WhatsApp button** on the next screen to add WhatsApp sending capabilities to your app.

On the next screen, you will be required to link your WhatsApp app to your Facebook business account. You will also have the option to create a new business account if you don’t have one yet.

WhatsApp Phone Number Id

When you register with WhatsApp Cloud API, Facebook provides a Test WhatsApp phone number that will be the default sending address of your Application. For recipients, you will have the option to add a maximum of 5 phone numbers during the development phase without having to make any payment.

Later you can register your own Phone Number to avoid the limitation of 5 phone numbers.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
```

WhatsApp Token

WhatsApp Cloud API requires a valid token to send any message using the Cloud API.

Facebook provides a Test WhatsApp phone number that allows you to send messages up to 5 phone numbers. You can override later this limitation registering your own phone number.

The WhatsApp API provides a **Temporary Access Token** that will be valid for 23 hours. This token must be configured in the TsgcWhatsApp_Client component to allow sending messages.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
```

If you need a long-valid token, you can create (or update) a system user and generate a new token with the **whatsapp_business_messaging** permission. This will allow you to send and receive WhatsApp messages without updating the token every 23 hours.

WhatsApp Webhook

Subscribe to Webhooks to get notifications about messages your business receives and customer profile updates.

Create Endpoint

Before you can start receiving notifications you will need to create an endpoint on your server to receive notifications.

Your endpoint must be able to process two types of HTTPS requests: Verification Requests and Event Notifications. Since both requests use HTTPs, your server must have a valid TLS or SSL certificate correctly configured and installed. Self-signed certificates are not supported.

When you configure the Webhook in the WhatsApp Settings, you must define the endpoint where is listening your server and a Token that can be any value, this token is used when registering the webhook endpoint and verify the subscriber is valid.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.ServerOptions.WebhookOptions.PhoneNumberId = "/webhook";
oClient.ServerOptions.WebhookOptions.Token = "MySecretToken";
oClient.StartServer();
```

Once the Webhook is configured, subscribe to **Messages** Webhook Fields to be notified every time a new message is received.

You can read more about configuring SSL Server.

WhatsApp Security

Every time a new message is received or there is a new status of a message, the server receives a notification in the endpoint configured in the Webhook. To be sure the request comes from WhatsApp Cloud API Servers, the request contains a header with a signature, you can configure the WhatsApp client to verify the signatures before process the message.

To do this, first you need to set the Application Secret in the property **ServerOptions.Application.Secret** and enable **VerifySignature** property.

Once configured, every time a new message is received, first the signature is verified, and if it's wrong, returns an error 500 and the message is not processed.

WhatsApp Send Messages

All API calls must be authenticated with an Access Token. Developers can authenticate their API calls with the access token generated in **App Dashboard** > **WhatsApp** > **Getting Started**

The API calls return the Message Id as a string.

Text Messages

Call the method **SendMessageText** and pass the following parameters:

- **aTo:** phone number
- **aText:** text of the message.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageText("34605889421", "Hello from sgcWebSockets!!!");
```

Image Messages

Call the method **SendMessageImage** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the image to send
- **aCaption:** title of the image (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageImage("34605889421", "Hello from sgcWebSockets!!!", "logo");
```

Document Messages

Call the method **SendMessageDocument** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the document to send
- **aCaption:** title of the document (optional).
- **aFileName:** name of the file (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageDocument("34605889421", "https://www.documents.com/file.txt", "Document", "file.txt");
```

Audio Messages

Call the method **SendMessageAudio** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the audio to send

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageAudio("34605889421", "https://www.audio.com/audio.mp3");
```

Video Messages

Call the method **SendMessageVideo** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the video to send

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageVideo("34605889421", "https://www.video.com/audio.mp4");
```

Sticker Messages

Call the method **SendMessageSticker** and pass the following parameters:

- **aTo:** phone number
- **aLink:** url where is the sticker to send

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageSticker("34605889421", "https://www.stickers.com/sticker");
```

Location Messages

Call the method **SendMessageLocation** and pass the following parameters:

- **aTo:** phone number
- **aLongitude:** Longitude of the location.
- **aLatitude:** Latitude of the location.
- **aName:** Name of the location.
- **aAddress:** Address of the location. Only displayed if aName is set.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageLocation("34605889421", "50.159305", "9.762686", "My Location", "My Address");
```

Contact Messages

Call the method **SendMessageContact** and pass the following parameters:

- **aTo:** phone number
- **aName:** Full name, as it normally appears (required).
- **aPhone:** the phone number (optional).
- **aEmail:** the email (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageLocation("34605889421", "John Smith", "15550386570", "john@mail.com");
```

WhatsApp Send Interactive Messages

Interactive messages give your users a simpler way to find and select what they want from your business on WhatsApp. During testing, chatbots using interactive messaging features achieved significantly higher response rates and conversions compared to those that are text-based.

The following messages are considered interactive:

- **List Messages**: Messages including a menu of up to 10 options. This type of message offers a simpler and more consistent way for users to make a selection when interacting with a business.
- **Reply Buttons**: Messages including up to 3 options —each option is a button. This type of message offers a quicker way for users to make a selection from a menu when interacting with a business. Reply buttons have the same user experience as interactive templates with buttons.

Interactive Message Specifications

- Interactive messages can be combined together in the same flow.
- Users cannot select more than one option at the same time from a list or button message, but they can go back and re-open a previous message.
- List or reply button messages cannot be used as notifications. Currently, they can only be sent within 24 hours of the last message sent by the user. If you try to send a message outside the 24-hour window, you get an error message.

When You Should Use It

List Messages are best for presenting several options, such as:

- A customer care or FAQ menu
- A take-out menu
- Selection of nearby stores or locations
- Available reservation times
- Choosing a recent order to repeat

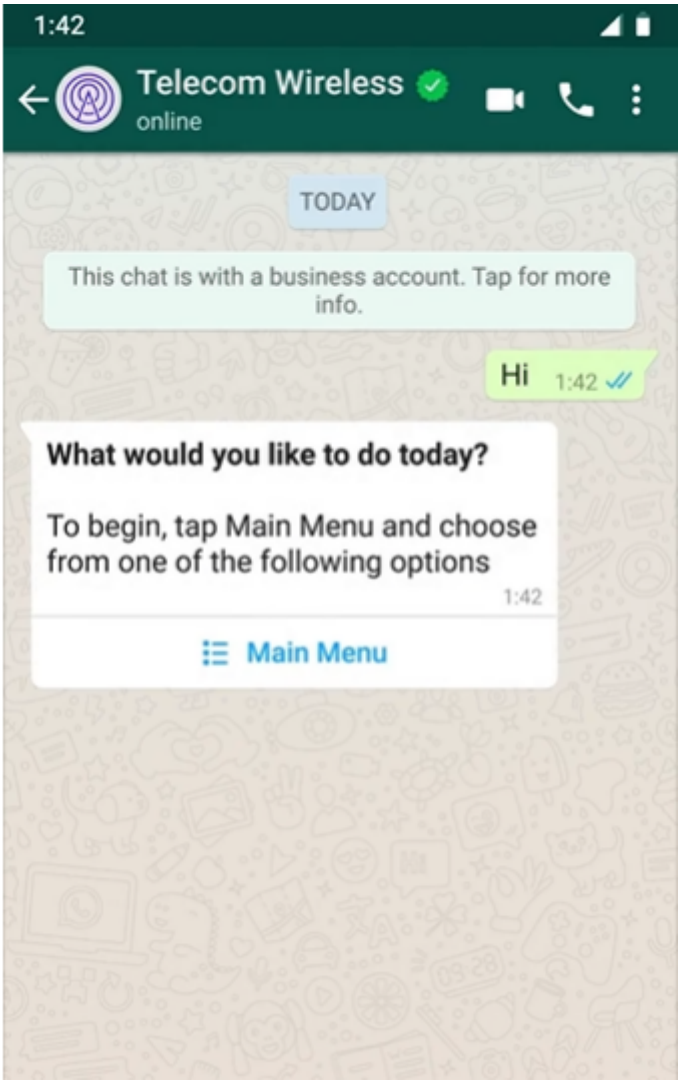
Reply Buttons are best for offering quick responses from a limited set of options, such as:

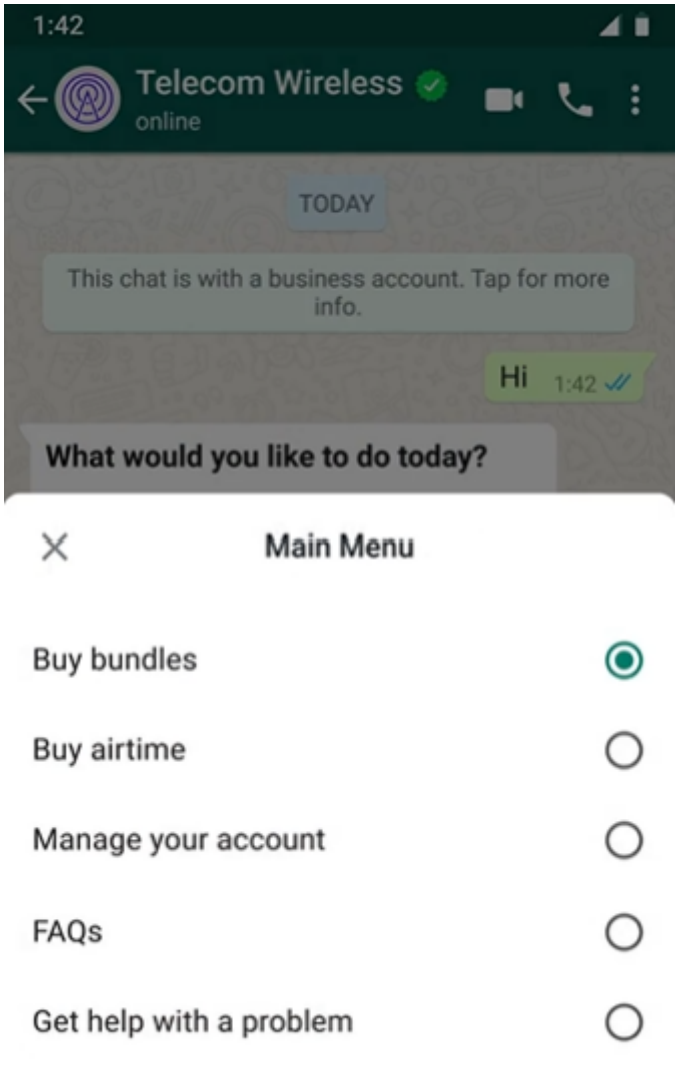
- Airtime recharge
- Changing personal details
- Reordering a previous order
- Requesting a return
- Adding optional extras to a food order
- Choosing a payment method

Reply buttons are particularly valuable for ‘personalized’ use cases where a generic response is not adequate.

Interactive List

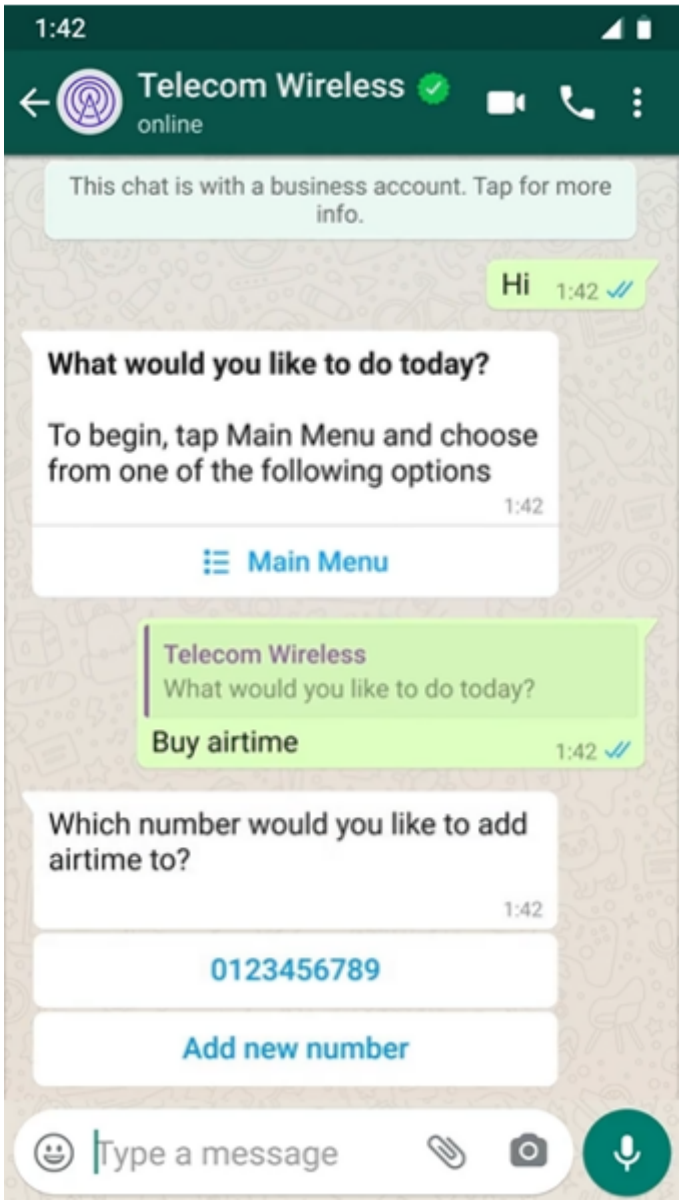
```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageInteractiveList("

34605889421

", "What Would you like to do today?", "To begin, Tap Main Menu and choose from of the following options", "", "M
```

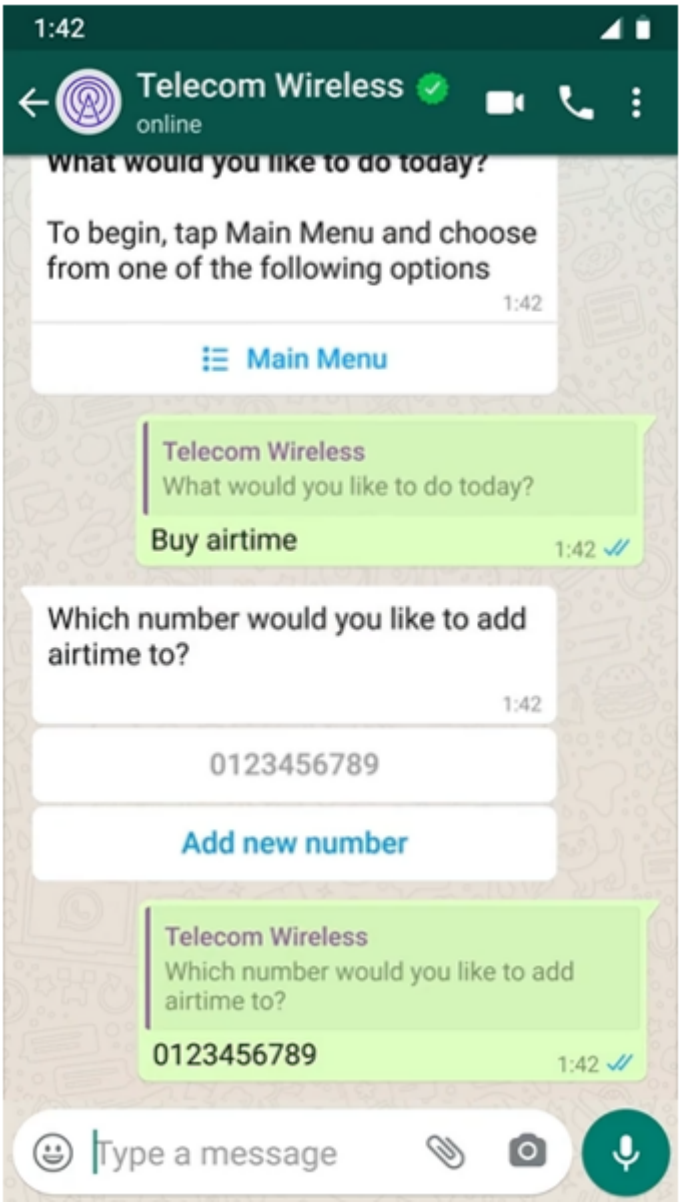




Reply Buttons

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageInteractiveButtons("34605889421", "Select an option", "Which number would you like to add airt
```


1:42
Telecom Wireless
online
This chat is with a business account. Tap for more info.
Hi 1:42
What would you like to do today?
To begin, tap Main Menu and choose from one of the following options
1:42
Main Menu
Telecom Wireless
What would you like to do today?
Buy airtime 1:42
Which number would you like to add airtime to?
1:42
0123456789
Add new number
Type a message


1:42
Telecom Wireless
online
What would you like to do today?
To begin, tap Main Menu and choose from one of the following options
1:42
Main Menu
Telecom Wireless
What would you like to do today?
Buy airtime
1:42
Which number would you like to add airtime to?
1:42
0123456789
Add new number
Telecom Wireless
Which number would you like to add airtime to?
0123456789
1:42
Type a message

WhatsApp Send Template Messages

Call the method **SendMessageTemplate** and pass the following parameters:

- **aTo:** phone number
- **aTemplate:** template identifier.
- **aLanguageCode:** template language.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageTemplate("34605889421", "hello_world", "en_US");
```

Template Message Parameters

Templates can include parameters, see below an example of default template with parameters

```
void SendSamplePurchaseFeedbackTemplate(string aName)
{
  TsgcWhatsApp_Send_Message_Template oTemplate = new TsgcWhatsApp_Send_Message_Template();
  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 oComponent2 = new TsgcWhatsApp_Send_Message_Template_Component();
  oComponent2._Type = wapctBody;
  oTemplate.Components.Add(oComponent2);
  TsgcWhatsApp_Send_Message_Template_Parameter oParameter2 = new TsgcWhatsApp_Send_Message_Template_Parameter();
  oParameter2.Text = aName;
  oParameter2._Type = wapptText;
  oComponent2.Parameters.Add(oParameter2);
  whatsapp.SendMessageTemplate("107809351952205", oTemplate);
}
```

Template Message Uploaded Image

Find below an example of a template where instead of using a link to an image, first uploads the image to the server and then sets the Id of the document.

```
void SendSamplePurchaseFeedbackTemplate(string aName)
{
  // ... first upload the file
  string vId = whatsapp.UploadMedia("c:\images\file.png", "image/png");

  // ... send message
  TsgcWhatsApp_Send_Message_Template oTemplate = new TsgcWhatsApp_Send_Message_Template();
  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 oComponent2 = new TsgcWhatsApp_Send_Message_Template_Component();
  oComponent2._Type = wapctBody;
  oTemplate.Components.Add(oComponent2);
  TsgcWhatsApp_Send_Message_Template_Parameter oParameter2 = new TsgcWhatsApp_Send_Message_Template_Parameter();
  oParameter2.Text = aName;
  oParameter2._Type = wapptText;
  oComponent2.Parameters.Add(oParameter2);
  whatsapp.SendMessageTemplate("107809351952205", oTemplate);
}
```

WhatsApp Receive Messages and Status Notifications

Subscribe to Webhooks to get notifications about messages your business receives and customer profile updates.

Whenever a trigger event occurs, the WhatsApp Business Platform sees the event and sends a notification to a Webhook URL you have previously specified. You can get two types of notifications:

- **Received messages**: This alert lets you know when you have received a message.
- **Message status and pricing notifications**: This alert lets you know when the status of a message has changed —for example, the message has been read or delivered.

Received Messages

Every time a new message is received the event **OnMessageReceived** is called, where you can access to the content of the Message and mark the message as read.

Find below an example, when a new text message is received, it's echoed to user who sent it.

```
void OnWhatsAppMessageReceived(TsgcWhatsApp_Client Sender, TsgcWhatsApp_Receive_Message Message, ref bool MarkAsR
{
  DoLog("Message Received: [" + Message.From + "] " + Message.Text);
  MarkAsRead = true;
}
```

Sent Messages

The WhatsApp Business Platform sends notifications to inform you of the status of the messages between you and users. When a message is sent successfully, you receive a notification when the message is sent, delivered, and read. The order of these notifications in your app may not reflect the actual timing of the message status. View the timestamp to determine the timing, if necessary.

- **sent:** The following notification is received when a business sends a message as part of a user-initiated conversation (if that conversation did not originate in a free entry point):
- **delivered:** The following notification is received when a business' message is delivered and that message is part of a user-initiated conversation (if that conversation did not originate in a free entry point):
- **read:** The following notification is received when the user reads the message.

Every time a new status is received, the event **OnMessageSent** is called.

```
void OnWhatsAppMessageSent(TsgcWhatsApp_Client Sender, TsgcWhatsApp_Receive_Message Message, TsgcWhatsAppSendMess
{
  string status = "unknown";
  if (Status == TsgcWhatsAppSendMessageStatusType.wapsmstRead)
  {
    status = "read";
  }
  else if (Status == TsgcWhatsAppSendMessageStatusType.wapsmstSent)
  {
    status = "sent";
  }
  else if (Status == TsgcWhatsAppSendMessageStatusType.wapsmstDelivered)
  {
```

```
    status = "delivered";
  }
  DoLog("Message Sent: " + Message.Id + " [" + status + "]");
}
```

WhatsApp Send Files

All API calls must be authenticated with an Access Token. Developers can authenticate their API calls with the access token generated in **App Dashboard** > **WhatsApp** > **Getting Started**

The API calls return the Message Id as a string.

When you send a File using the WhatsApp API, first the message is uploaded to WhatsApp servers and then a new message is sent with the object id returned after upload the file.

Image Messages

Call the method **SendMessageImage** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the image file to send.
- **aFileType:**
  - image/jpeg
  - image/png
- **aCaption:** title of the image (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendFileImage("34605889421", "c:\images\image.png", "image/png");
```

Document Messages

Call the method **SendMessageDocument** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the document file to send.
- **aFileType:**
  - text/plain
  - application/pdf
  - application/vnd.ms-powerpoint
  - application/msword
  - application/vnd.ms-excel
  - application/vnd.openxmlformats-officedocument.wordprocessingml.document
  - application/vnd.openxmlformats-officedocument.presentationml.presentation
  - application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
- **aCaption:** title of the document (optional).

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendFileDocument("34605889421", "c:\MyDocuments\invoice.pdf", "application/pdf");
```

Audio Messages

Call the method **SendMessageAudio** and pass the following parameters:

- **aTo:** phone number

- **aFileName:** full filename (with path) of the audio file to send.
- **aFileType:**
  - audio/aac
  - audio/mp4
  - audio/mpeg
  - audio/amr
  - audio/ogg

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendFileAudio("34605889421", "c:\Music\audio.mp3", "audio/mp4");
```

Video Messages

Call the method **SendMessageVideo** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the video file to send.
- **aFileType:**
  - video/mp4
  - video/3gp

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendFileVideo("34605889421", "c:\Videos\video.mp4", "video/mp4");
```

Sticker Messages

Call the method **SendMessageSticker** and pass the following parameters:

- **aTo:** phone number
- **aFileName:** full filename (with path) of the sticker file to send.
- **aFileType:**
  - image/webp

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendFileSticker("34605889421", "c:\Stickers\MySicker.webp", "image/webp");
```

WhatsApp Download Media

If you receive a message with a media file link, you can download the media file using the method **DownloadMedia**.

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAAO4OpgZAs98BAGj3nCFGr...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.DownloadMedia("38923878928822", "c:\whatsapp\media\image.png");
```

To delete a previously uploaded media file, just call **DeleteMedia** and pass the object id as argument.

API Telegram

Telegram

Telegram offers two kinds of APIs, one is **Bot API** which allows you to create programs that use Bots and HTTPs as protocol. **Telegram API and TDLib** allow you to build customized Telegram clients and is much more powerful than Bot API.

sgcWebSockets **supports TDLib through tdjson** library, which means that you can build your own telegram client. TDLib takes care of all network implementation details, encryption and local data storage. TDLib supports all Telegram features.

TDLib (Telegram Database Library) Advantages

- **Cross-platform:** can be used on Windows, Android, iOS, MacOS, Linux...
- **Easy to use:** uses json messages to communicate between application and telegram.
- **High-performance:** In the Telegram Bot API, each TDLib instance handles more than 24000 bots.
- **Consistent:** TDLib guarantees that all updates will be delivered in the right order.
- **Reliable:** TDLib remains stable on slow and unreliable internet connections.
- **Secure:** All local data is encrypted using a user-provided encryption key.
- **Fully Asynchronous:** Requests to TDLib don't block each other. Responses will be sent when they are available.

Configuration

Windows

TDLib requires other third-parties libraries: OpenSSL and ZLib. These libraries must be deployed with tdjson library.

* Windows versions require VCRuntime, which can be downloaded from Microsoft: https://www.microsoft.com/en-us/download/details.aspx?id=52685, If after installing, the problem persist, try to copy the following dll in the same folder where your application is: VCRUNTIME140.dll and VCRUNTIME140_1.dll.

Copy the following libraries to the same directory where your application is located:

Windows 32	Windows 64
tdjson.dll	tdjson.dll
libcrypto-1_1.dll	libcrypto-1_1-x64.dll
libssl-1_1.dll	libssl-1_1-x64.dll
zlib1.dll	zlib1.dll

Creating your Telegram Application

In order to obtain an API id and develop your own application using the Telegram API you need to do the following:

- Sign up for Telegram using any application.
- Log in to your Telegram core: https://my.telegram.org.
- Go to **API development tools** and fill out the form.
- You will get basic addresses as well as the **api_id** and **api_hash** parameters required for user authorization.
- For the moment each number can only have one **api_id** connected to it.

These values must be set in **Telegram.API** property of Telegram component. In order to authenticate, you can authenticate as an user or as a bot, there are 2 properties which you can set to login to Telegram:

- **PhoneNumber:** if you login as an user, you must set your **phone number** (with international code), example: +34699123456
- **BotToken:** if you login as a bot, set your token in this property (as provided by telegram).
- **DatabaseDirectory:** allows you to specify where is the tdlib database. Leave empty and will take the default configuration.

The following parameters can be configured:

- **ApplicationVersion:** application version, example: 1.0
- **DeviceModel:** device model, example: desktop
- **LanguageCode:** user language code, example: en.
- **SystemVersion:** verison of operating system, example: windows.

Optionally, you can configure the path where is located tdjson library using **SetTDJsonPath** method. Just pass the path before start a new telegram session.

Once you have configured Telegram Component, you can set Active property to true and program will attempts to connect to Telegram.

Sample Code

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram.Telegram.API.ApiHash = "your api hash";
oTelegram.Telegram.API.ApiId = "your api id";
oTelegram.PhoneNumber = "your phone number";
oTelegram.ApplicationVersion = "1.0";
oTelegram.DeviceModel = "Desktop";
oTelegram.LanguageCode = "en";
oTelegram.SystemVersion = "Windows";
oTelegram.Active = true;
```

Authorization

There are two events which can be called by library in order to get an Authentication Code (delivered in Telegram Application, not SMS) or to provide a password.

OnAuthenticationCode

This event is called when Telegram sends an Authorization Code to Telegram Application and user must copy this code and set in Code argument of this event.

```
void OnAuthenticationCode(TObject Sender, ref string Code)
{
  Code = "telegram code here";
}
```

OnAuthenticationPassword

This event is called when Telegram requires that user set a password.

Authorization Status

Once authorization has started, you can check the status of authorization **OnAuthorizationStatus** event, this event is called every time there is a change in status of authorization. Some values of Status are:

- authorizationStateWaitTdlibParameters
- authorizationStateWaitEncryptionKey

- authorizationStateWaitPhoneNumber
- authorizationStateWaitCode
- authorizationStateLoggingOut
- authorizationStateClosed
- authorizationStateReady

Connection Status

Once connection has started, you can check the status of connection **OnConnectionStatus** event, this event is called every time there is a change in status of connection. Some values of Status are:

- connectionStateConnecting
- connectionStateUpdating
- connectionStateReady

Main Methods

TsgcTDLib_Telegram API Component support several Telegram methods, find below the most used.

Method	Parameters	Description
SendTextMessage	**aChatId:** Id of Chat which message will be sent **aText:** Text of Message. **InlineKeyboard:** Optional Buttons (only bots).	Sends a Text Message to a Chat
SendRichTextMessage	**aChatId:** Id of Chat which message will be sent **aText:** Text of Message. **InlineKeyboard:** Optional Buttons (only bots).	Sends a Rich Text Message to a Chat. Markdown syntax: • Bold: **bold** • Italic: __italic__ • Strike: --strike-- • Underline: ~~underline~~ • Code: ##code##
SendDocumentMessage	**aChatId:** Id of Chat which message will be sent **aFilePath:** full file path of document **aInlineKeyboard:** Optional Buttons (only bots).	Sends a Document to a Chat.
SendPhotoMessage	**aChatId:** Id of Chat which message will be sent **aFilePath:** full file path of photo **Width:** witdh of photo. **Height:** width of photo. **InlineKeyboard:** Optional Buttons (only bots).	Sends a Photo to a Chat.
SendVideoMessage	**aChatId:** Id of Chat which message will be sent **aFilePath:** full file path of video **aWidth:** witdh of video. **Height:** width of video. **aDuration:** duration of video in seconds. **aInlineKeyboard:** Optional Buttons (only bots).	Sends a Video to a Chat.
SendInvoiceMessage	**aChatId:** Id of Chat which message will be sent **aInvoice:** Text of Message. **aInlineKeyboard:** Optional Buttons (only bots).	Sends an Invoice (only available when is a Bot and in Private Channels).
EditTextMessage	**aChatId:** Id of Chat which message will be sent **aMessageId:** Id of Message to modify **Text:** Text of Message.	Edits the text of a message (or a text of a game message)

	InlineKeyboard: Optional Buttons (only bots). **ShowKeyboard:** Optional Buttons (only bots).	
AddChatMember	**aChatId:** Id of Chat which message will be sent **aUserId:** Identifier of the user. **aForwardLimit:** The number of earlier messages from the chat to be forwarded to the new member; up to 100. Ignored for supergroups and channels.	Adds a new member to a chat. Members can't be added to private or secret chats. Members will not be added until the chat state has been synchronized with the server.
AddChatMembers	**aChatId:** Id of Chat which message will be sent **aUserIds:** Identifiers of the users to be added to the chat.	Adds multiple new members to a chat. Currently this option is only available for supergroups and channels. This option can't be used to join a chat. Members can't be added to a channel if it has more than 200 members. Members will not be added until the chat state has been synchronized with the server.
GetChatMember	**aChatId:** Chat Identifier. **aUserId:** User Identifier.	Returns information about a single member of a chat.
GetBasicGroupFullInfo	**aGroupId:** Basic Group Identifier	Returns full information about a basic group by its identifier.
GetSupergroupMembers	**aSuperGroupId:** Identifier of the supergroup or channel. **aSupergroupMembersFilter:** The type of users to return. By default null **aOffset:** Number of users to skip. **aLimit:** The maximum number of users be returned; up to 200.	Returns information about members or banned users in a supergroup or channel.
JoinChatByInviteLink	**aLink:** Invite link to import;	Uses an invite link to add the current user to the chat if possible. The new member will not be added until the chat state has been synchronized with the server.
CreateNewSecretChat	**aUserId:** Identifier of the user.	Creates a new secret chat.
CreateNewBasicGroupChat	**aUserIds:** Identifiers of the users to be added to the chat. **aTitle:** Title of the new basic group	Creates a new basic group
CreateNewSupergroupChat	**aTitle:** Title of the new SuperGroup **aIsChannel:** True, if a channel chat should be created. **aDescription:** Chat Description.	Creates a new supergroup or channel.
CreatePrivateChat	**aUserId:** Identifier of the user. **aForce:** If true, the chat will be created without network request. In this case all information about the chat except its type, title and photo can be incorrect	Returns an existing chat corresponding to a given user
GetChats	**aOffsetOrder:** Chat order to return chats from **aOffsetChatId:** Chat identifier to return chats from **aLimit:** The maximum number of chats to be returned.	Returns an ordered list of chats. Chats are sorted by the pair (order, chat_id) in decreasing order (cannot be used is logged as Bot)
GetChat	**aChatId:** Chat identifier	Returns information about a chat by its identifier
GetChatHistory	**aChatId:** Chat identifier **aFromMessageId:** Identifier of the message starting from which history	Returns messages in a chat. The messages are returned in a reverse chronological order

	must be fetched; use 0 to get results from the last message. **aOffset:** Specify 0 to get results from exactly the from_message_id or a negative offset up to 99 to get additionally some newer messages. **aLimit:**The maximum number of messages to be returned	
GetUser	**aUserId:** User Identifier	Returns information about a user by their identifier.
AddProxyHTTP	**aServer:** Server name of proxy. **aPort:** Number of proxy port. **aUserName:** Username for logging in; may be empty. **aPassword:** Password for logging in; may be empty. **aHTTPOnly:** Pass true, if the proxy supports only HTTP requests and doesn't support transparent TCP connections via HTTP CONNECT method.	Adds a HTTP proxy server for network requests. Can be called before authorization.
AddProxyMTProto	**aServer:** Server name of proxy. **aPort:** Number of proxy port. **aSecret:** The proxy's secret in hexadecimal encoding.	Adds a MTProto proxy server for network requests. Can be called before authorization.
AddProxySocks5	**aServer:** Server name of proxy. **aPort:** Number of proxy port. **aUserName:** Username for logging in; may be empty. **aPassword:** Password for logging in; may be empty.	Adds a Socks5 proxy server for network requests. Can be called before authorization.
EnableProxy	**aId:** ID of proxy	Enables a proxy. Only one proxy can be enabled at a time. Can be called before authorization.
DisableProxy		Disables the currently enabled proxy. Can be called before authorization.
RemoveProxy	**aId:** ID of proxy	Removes a proxy server. Can be called before authorization.
GetProxies		Returns list of proxies that are currently set up. Can be called before authorization.
getChatSponsoredMessage	**aChatId:** ID of the chat	Returns sponsored message to be shown in a chat; for channel chats only. Returns a 404 error if there is no sponsored message in the chat.
ViewMessage	**aSponsorChatId:** ID of the sponsor Chat **aMessageId:** ID of the message	Informs TDLib that messages are being viewed by the user. Many useful activities depend on whether the messages are currently being viewed or not
Logout		Logouts from Telegram.
TDLibSend	**aRequest:** JSON Request.	Send any Request in JSON protocol.

Example How to send a Text Message

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram.Telegram.API.ApiHash = "your api hash";
oTelegram.Telegram.API.ApiId = "your api id";
oTelegram.PhoneNumber = "your phone number";
oTelegram.Active = true;
...
oTelegram.SendTextMessage("1234", "My First Message from sgcWebSockets");
```

Example How to send a method not implemented

You can Send Any JSON message using TDLibSend method, example: join a telegram chat.

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram.Telegram.API.ApiHash = "your api hash";
oTelegram.Telegram.API.ApiId = "your api id";
oTelegram.PhoneNumber = "your phone number";
oTelegram.Active = true;
...
oTelegram.TDLibSend("{\"@type\": \"joinChat\", \"chat_id\": \"1234\"}");
```

Events

OnBeforeReadEvent
This event is called when JSON message is received by Telegram API component and is still not processed. Set Handled property to True if you process this event manually or don't want that event is processed by component. You can use this event to log all messages too.

OnMessageText
This event is called when a New Message Text has been received, read MessageText parameter to access to message text properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier.
- **Text:** Text of message.

OnMessageDocument
This event is called when a New Document Message is received. Access to MessageDocument to get access to Document properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier (read SenderChat and SenderUser from tdlib 1.7.+).
- **FileName:** Name of Document.
- **DocumentId:** Document Identifier.
- **LocalPath:** full path to local file if exists.
- **MimeType:** Mime-type of document.
- **Size:** Size of Document.
- **RemoteDocumentId:** Remote Document Identifier.

OnMessagePhoto
This event is called when a New Photo Message is received. Access to MessagePhoto to get access to Photo properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier (read SenderChat and SenderUser from tdlib 1.7.+).
- **PhotoId:** Photo Identifier.
- **LocalPath:** full path to local file if exists.
- **Size:** Size of Photo.
- **RemotePhotoId:** Remote Photo Identifier.

OnVideoPhoto
This event is called when a New Video Message is received. Access to MessageVideo to get access to Video properties.

- **ChatId:** Chat Identifier.
- **MessageId:** Message Identifier.
- **SenderUserId:** Sender Identifier (read SenderChat and SenderUser from tdlib 1.7.+).
- **VideoId:** Photo Identifier.

- **LocalPath:** full path to local file if exists.
- **Width:** width of video.
- **Height:** height of video.
- **Duration:** duration in seconds of video.
- **Size:** Size of Video.
- **RemoteVideoId:** Remote Photo Identifier.

OnMessageSponsored
This event is called when a New Sponsored Message has been received (after calling the method getChatSponsoredMessage)

- **SponsorChatId:** Sponsor Chat Identifier.
- **MessageId:** Message Identifier.
- **Text:** Text of message.

OnNewChat
This event is called when a new chat is received.

- **ChatId:** Chat Identifier.
- **ChatType:** Chat Type (chatTypeSupergroup, chatTypePrivate...)
- **Title:** Chat name.
- **SuperGroupId:** Group Id if is a SuperGroup.
- **IsChannel:** returns if is channel or not.

OnNewCallbackQuery
This event is called when a new incoming callback query is received; for bots only.

- **Id:** Unique query identifier.
- **SenderUserId:** Identifier of the user who sent the query.
- **ChatId:** Identifier of the chat, in which que query was sent.
- **MessageId:** Identifier of the message, from which the query originated.
- **ChatInstance:** Identifier that uniquely corresponds to the chat to which the message was sent.
- **PayloadData:** the payload from a general callback button.
    - **Data:** Data that was attached to the callback button.

OnEvent
This event is called when a new Event is received by API Component. Can be used to process some events not implemented by API Component.

- **Event:** Event name (events like: updateOption, updateUser...)
- **Text:** full JSON message

OnException
This event is called if there is any exception when processing Telegram API Data.

Properties

MyId: returns the User Identifier of current user.

Full Code Sample

Check the following code sample which shows how connect to Telegram API, ask user to introduce a Code (if required by Telegram API), send a message when connection is ready and Log Text Messages received.

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram.Telegram.API.ApiHash = "your api hash";
oTelegram.Telegram.API.ApiId = "your api id";
oTelegram.PhoneNumber = "your phone number";
oTelegram.ApplicationVersion = "1.0";
oTelegram.DeviceModel = "Desktop";
oTelegram.LanguageCode = "en";
oTelegram.SystemVersion = "Windows";
oTelegram.Active = true;
```

```
void OnAuthenticationCode(TObject Sender, ref string Code)
{
  Code = InputBox("Telegram Code", "Introduce code", "");
}

void OnMessageText(TObject Sender, TsgcTelegramMessageText MessageText)
{
  Log("Message Received: " + MessageText.Text);
}

void OnConnectionStatus(TObject Sender, const string Status)
{
  if (Status == "connectionStateReady")
  {
    oTelegram.SendTextMessage("1234", "Hello Telegram!");
  }
}
```

Telegram | Send Telegram Message With In-line Buttons

Telegram API allows you to send messages with inline buttons to select options as an answer (this option is only available for bots).

Before you send a message create an instance of the class **TsgcTelegramReplyMarkupInlineKeyboard** and call the method **AddButtonTypeCallback** or **AddButtonTypeUrl** for every button you want to create.

Example

Create a new message asking the user if likes or not the message and a link to answer a poll. Process the response using **OnNewCallbackQuery** event.

```
TsgcTelegramReplyMarkupInlineKeyboard oReplyMarkup = new TsgcTelegramReplyMarkupInlineKeyboard();
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);

void OnNewCallbackQuery(TObject Sender, TsgcTelegramCallbackQuery CallbackQuery)
{
  if (CallbackQuery.PayloadData.Data == "I like it") then
  {
    MessageBox.Show("yes")
  }
  else
  {
    MessageBox.Show("no");
  }
}
```

Telegram | Send Bot Message With Buttons

Telegram API allows you to send messages with buttons to request data from the user (this option is only available for bots).

Before you send a message create an instance of the class **TsgcTelegramReplyMarkupShowKeyboard** and call the method **AddButtonTypeRequestLocation, AddButtonTypeRequestPhoneNumber** or **AddButtonTypeText** for every button you want to create.

Example

Create a new message asking the user to provide the PhoneNumber

```
oReplyMarkup = new TsgcTelegramReplyMarkupShowKeyboard();
oReplyMarkup.AddButtonTypeRequestPhoneNumber("Give me your phone");
sgcTelegram.SendTextMessage("123456", "Please provide the information below", null, oReplyMarkup);
```

Telegram | Send Telegram Message Bold

You can highlight text messages using bold, italic and more styles. Use the method **SendRichTextMessage**, to send a Text message with style capabilities, this method parses the text message and adds the entities required automatically to the API Telegram.

Markdown Syntax

- Bold [*]

```
**This is Bold**
```

- Italic [_]

```
__This is Italic__
```

- Strike [-]

```
--This is Strike--
```

- Underline [~]

```
~~This is Underline~~
```

- Code [#]

```
##This is Monospace##
```

Telegram | Chat not found as Bot

When you **log as bot**, the GetChats method cannot be used, so you don't get All available chats. If it's the **first time you login as Bot** and you try to **send a message** to a **known Chat**, you will get this **error**:

```
{"@type":"error","code":5,"message":"Chat not found"}
```

The solution is to call the **GetChat** method before sending a telegram message and pass the **ChatId** as a parameter. Once you get the Chat data, you can send telegram messages as usual.

As a note, you **only** need to **call GetChat** the **FIRST TIME** before sending a message if you have never received any bot message from this chat. If you close the application and start again, there is no need to call GetChat first because the Chat is already saved in the telegram database.

Telegram | Sponsored Messages

Each time the user opens a channel, channels.getSponsoredMessages must be called to receive sponsored messages available for this channel. The result must be cached for 5 minutes.

Displaying sponsored messages

Sponsored messages must be displayed below all other posts in the channel, after the user scrolls further down, past the last message. The promoted channel or bot specified in the from_id field must be displayed as the author of the message. The message should also contain one of the following buttons at the bottom:

- **View Bot:** if a bot is being promoted. Tapping the button must open the chat with the bot. If start_param is specified, the app must use the deep linking mechanism to open the bot.
- **View Channel:** if a channel is being promoted. Tapping the button must open the channel.
- **View Post:** if a channel is being promoted and channel_post is specified. Tapping the button must open the particular channel post.

Once the entire text is shown on the screen (excluding the button), **ViewMessage** method must be called with the random_id of this sponsored message.

Get Sponsored Messages

Send a request to the channel asking if there are sponsored messages available, just call the method **GetChatSponsoredMessage**.

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram.Telegram.API.ApiHash = "ABCDEFGHIJKLMN";
oTelegram.Telegram.API.ApiId = "1234";
oTelegram.PhoneNumber = "008745744155";
oTelegram.Active = true;
oTelegram->getChatSponsoredMessage("100");
```

If the chat has sponsored messages, the event **OnMessageSponsored** is called with the content of the Sponsored message. If there are no messages, a 404 error is returned.

```
private void(TObject Sender, TsgcTelegramMessageSponsored MessageSponsored)
{
  DoLog(MessageSponsored.Text);
}
```

Call the method **ViewMethod** after the Sponsored Messages has been shown to the user.

```
oTelegram.ViewMessage("100", "54653256245");
```

Telegram | Send Telegram Invoice Message

If your bot supports inline mode, users can also send invoices to other chats via the bot, including to one-on-one chats with other users.

Invoice messages feature a photo and description of the product along with a prominent Pay button. Tapping this button opens a special payment interface in the Telegram app

The bots can send invoices as a message using the method **SendInvoiceMessage**.

```
private void SendInvoice()
{
  TsgcTelegramSendInvoice oInvoice = new TsgcTelegramSendInvoice();
  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);
}
```

Telegram | Get SuperGroup Members

Telegram API allows you to get information about members of a SuperGroup. Use the method **GetSuperGroup-Members** to get information about members or banned users in a supergroup or channel. Can be used only if SupergroupFullInfo.can_get_members is true; additionally, administrator privileges may be required for some filters.

By default the method returns All members of the group, but you can filter the members returned using the Filter parameter. There are the following parameters:

tsgmFilterNone
Default value, means members are not filtered.

tsgmFilterAdministrators
Returns the creator and administrators.

tsgmFilterBanned
Returns users banned from the supergroup or channel; can be used only by administrators.
You can use the argument aSuperGroupMembersQuery to search using a query.

tsgmFilterBots
Returns bot members of the supergroup or channel.

tsgmFilterContacts
Returns contacts of the user, which are members of the supergroup or channel.
You can use the argument aSuperGroupMembersQuery to search using a query.

tsgmFilterMention
Returns users which can be mentioned in the supergroup.

tsgmFilterRecent
Returns recently active users in reverse chronological order.

tsgmFilterRestricted
Returns restricted supergroup members; can be used only by administrators.
You can use the argument aSuperGroupMembersQuery to search using a query.

tsgmFilterSearch
Used to search for supergroup or channel members via a (string) query.
You can use the argument aSuperGroupMembersQuery to search using a query.

You can read the result of the result using OnEvent callback and filtering by event = "chatMembers".

```
Telegram.GetSupergroupMembers(1452979380);


private void OnTelegramEvent(TObject Sender, const string Event, const string Text)
{
  if (Event == "chatMembers")
  {
    ReadJSON(Text);
  }
}
```

Telegram | Add Telegram Proxy

Telegram Client can be configured to make use of a proxy. Currently, Telegram supports 3 types of proxies:

1. HTTP
2. MTProto
3. Socks5

Add Proxy

In order to configure a HTTP Proxy, first you must add the proxy to telegram configuration, to do this, just call **AddProxyHTTP** and if successful, a message will be returned with the new proxy added. Once the proxy has been added to the list, just call **EnableProxy** and pass the **ID of the proxy** received on the confirmation message.

```
Telegram.AddProxyHTTP("8.8.8.8", 8080, "", "", true);
// ... read the confirmation message and save the ID of the proxy.
Telegram.EnableProxy(2);
```

Remove Proxy

Call **RemoveProxy** method and pass the ID of the proxy you want to remove.

Telegram | Register Telegram User

The process to register a new user in Telegram is very simple, you need your API Id and API Hash, and the phone number of the new account.

Configure the telegram client:

- API Id
- API Hash
- Telephone Number of the new telegram account.

Start the client and a new code will be sent to the phone, the client will ask for the telegram code and if it's correct, the event OnRegisterUser will be called. In this event set the First Name and Last Name of the user and the registration will be completed.

```
TsgcTDLib_Telegram oTelegram = new TsgcTDLib_Telegram();
oTelegram.Telegram.API.ApiHash = "ABCDEFGHIJKLMN";
oTelegram.Telegram.API.ApiId = "1234";
oTelegram.PhoneNumber = "008745744155";
oTelegram.Active = true;

void OnTelegramAuthenticationCode(TObject Sender, ref string Code)
{
  Code = "code sent to phone";
}

void OnTelegramRegisterUser(TObject Sender, ref string FirstName, ref string LastName)
{
  FirstName = "first name";
  LastName = "last name";
}
```

RCON

RCON

The Source RCON Protocol is a TCP/IP-based communication protocol used by Source Dedicated Server, which allows console commands to be issued to the server via a "remote console", or RCON. The most common use of RCON is to allow server owners to control their game servers without direct access to the machine the server is running on.

Configuration

The **RCON_Options** allows you to configure the following properties:

- **Host:** server remote address.
- **Port:** server listening port.
- **Password:** is the secret string used to authenticate against the server

Connect

Use the property **Active** to Connect / Disconnect from server.
When Active is set to True, the client tries to connect to the server, if it can connect, it will try to authenticate using the provided password.
The server will send a response to an authentication request. The event **OnAuthenticate** will be called and you can read if authentication is successful or not using the Authenticate parameter.

Send Commands

Use the method **ExecCommand** to send commands to the server. The responses will be available **OnResponse** Event.

```
TsgcLib_RCON oRCON = new TsgcLib_RCON();
oRCON.RCON_Options.Host = "127.0.0.1";
oRCON.RCON_Options.Port = 25575;
oRCON.RCON_Options.Password = "test";
oRCON.Active = true;


void OnAuthenticate(TObject Sender, bool Authenticated, const TsgcRCON_Packet aPacket)
{
  if (Authenticated == true)
  {
    DoLog("#authenticated");
  }
  else
  {
    DoLog("#not authenticated");
  }
}


void OnResponse(Object Sender, const string aResponse, const TsgcRCON_Packet aPacket)
{
  DoLog(aResponse);
}
```

CryptoHopper

CryptoHopper

CryptoHopper is an automated crypto trading bot that allows you to automate trading and portfolio management for Bitcoin, Ethereum, Litecoin and more.

Configuration

Requires a **Developer Account** and once you have been approved you can start to create a new App. The API uses OAuth2 to authenticate, so you can retrieve the **client_id** and **client_secret** from your App.

```
TsgcHTTP_Cryptohopper oCryptoHopper = new TsgcHTTP_Cryptohopper();
oCryptoHopper.CryptoHopperOptions.OAuth2.ClientId = "client_id";
oCryptoHopper.CryptoHopperOptions.OAuth2.ClientSecret = "client_secret";
oCryptoHopper.CryptoHopperOptions.OAuth2.LocalIP = "127.0.0.1";
oCryptoHopper.CryptoHopperOptions.OAuth2.LocalPort = 8080;
oCryptoHopper.CryptoHopperOptions.OAuth2.Scope = "read,notifications,manage,trade";
```

Methods

CryptoHopper uses HTTPs as the protocol to send Requests to the API. Some methods require authentication (place orders, retrieve user data...) and some others are public (get exchange data for example).

The functions return the CryptoHopper response and if there is any error an exception will be raised.

Hoppers

Manage Basic Hopper Operations.

Method	Arguments	Description
GetHoppers		Get Hoppers of users.
CreateHopper	**aBody:** configuration json text.	Create a new Hopper.
GetHopper	**aId:** hopper id	Retrieve Hopper
DeleteHopper	**aId:** hopper id	Delete Hopper
UpdateHopper	**aId:** hopper id **aBody:** configuration json text.	Update Hopper

Orders

Manage the Orders of your Hopper.

Method	Arguments	Description
GetOpenOrders	**aId:** hopper id	Retrieve all of the open orders of the hopper.
CreateNewOrder	**aId:** hopper id **aOrder:** instance of TsgcHTTPCTHOrder	Create new buy or sell order. For sell, rather use the sell endpoint.
PlaceMarketOrder	**aId:** hopper id **aOrderSide:** cthosBuy or cthosSell. **aCoin:** coin name, example: EOS **aAmount:** order size.	Place a Market Order.
PlaceLimitOrder	**aId:** hopper id **aOrderSide:** cthosBuy or cthosSell. **aCoin:** coin name, example: EOS **aAmount:** order size. **aPrice:** limit price.	Place a Limit Order
DeleteOrder	**aId:** hopper id **aOrderId:** order id	Deletes order for selected hopper.
DeleteAllOrders	**aId:** hopper id	Deletes all open order for selected hopper.
GetOpenOrder	**aId:** hopper id **aOrderId:** order id	Get open order in hopper by id.
CancelOrder	**aId:** hopper id **aOrderId:** order id	Cancel an open order.

Position

Manage the Positions of your Hopper.

Method	Arguments	Description
GetPosition	**aId:** hopper id	Get open positions of hopper.

Trade

Trade History from your Hopper.

Method	Arguments	Description
GetTradeHistory		Get the trade history of the hopper.
GetTradeHistoryById	**aId:** hopper id **aTradeId:** trade id	Get a trade by id of the hopper.

Exchange

Get Information from available exchanges on CryptoHopper

Method	Arguments	Description
GetExchange		Get all available exchanges on Cryptohopper.
GetAllTickers	**aExchange:** exchange name	Get ticker for all pairs
GetMarketTicker	**aExchange:** exchange name **aPair:** pair name	Get ticker from market pair.
GetOrderBook	**aExchange:** exchange name **aPair:** pair name **aDepth:** order book depth	Gets the orderbook for the selected exchange, market and orderbook depth.

Webhooks

Trade History from your Hopper.

Method	Arguments	Description
CreateWebhook	**aURL:** webhook url **aMessageTypes:** message types separated by comma.	Update or create a Webhook

DeleteWebhook	**aURL:** webhook url	Delete an existing Webhook.

Signals

Send Signals to CryptoHopper API.

Method	Arguments	Description
SendSignal	**aSignal:** is the class with all the fields required to send a signal.	Sends a Signal
SendTestSignal	**aSignal:** is the class with all the fields required to send a signal.	Sends a Test Signal
GetSignalStats	**aSignalId:** id of the signal. **aExchange:** optional, name of the exchange.	Retrieve some of the signal statistics.

How to Update Cryptohopper Config

Use the UpdateHopper method to update the Hopper Configuration. The method is overloaded so you can pass the JSON string or use the object TsgcHTTPCTHopper and use the properties to enable or disable the Hopper Properties.

```
public string EnableHopper()
{
  TsgcHTTPCTHopper oHopper = new TsgcHTTPCTHopper();
  oHopper->Enabled = 1;
  return Cryptohopper->UpdateHopper("1234", oHopper);
}
```

How to Configure Webhook

Webhook allows you to receive notifications when something happens in a hopper. Webhooks require a public HTTPs Server which will listen in a URL address all messages sent by cryptohopper. The public server needs to be protected with a SSL certificate (self-signed certificates are not allowed).

First you must create a webhook, so configure the Webhook property of Cryptohopper client setting the Host and Port when the server will be listening. Then configure the certificate in SSLOptions property.

Example: The public IP address will be 1.1.1.1 and the listening port will be 443. The certificate is stored as PEM file with sgc.pem filename and without password.

```
/* OAuth2 */
cryptohopper.CryptohopperOptions.OAuth2.ClientId = "client_id";
cryptohopper.CryptohopperOptions.OAuth2.ClientSecret = "client_secret";
cryptohopper.CryptohopperOptions.OAuth2.LocalIP = "127.0.0.1";
cryptohopper.CryptohopperOptions.OAuth2.LocalPort = 8080;
/* Webhook */
cryptohopper.CryptohopperOptions.Webhook.Enabled = True;
cryptohopper.CryptohopperOptions.Webhook.Host = "1.1.1.1";
cryptohopper.CryptohopperOptions.Webhook.Port = 443;
cryptohopper.CryptohopperOptions.Webhook.ValidationCode = "1234";
cryptohopper.CryptohopperOptions.Webhook.SSLOptions.CertFile = "sgc.pem";
cryptohopper.CryptohopperOptions.Webhook.SSLOptions.KeyFile = "sgc.pem";
cryptohopper.CryptohopperOptions.Webhook.SSLOptions.RootCertFile = "sgc.pem";
cryptohopper.CryptohopperOptions.Webhook.SSLOptions.Password = "";
cryptohopper.StartWebhook();
```

RTCMultiConnection

RTCMultiConnection

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

Configuration

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

Server

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

IceServers

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

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

VideoResolution

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

HTMLDocuments

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

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

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

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

Applications

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

Video-Broadcast-ing	Multi-user (one-to-many) video broadcasting using star topology.

WebPush

RFC 8030
RFC 8291

The WebPush protocol is defined by the **RFC 8030** (Delivery using HTTP Push) and **RFC 8291** (Message Encryption).

Web Push is a **standardized protocol** for **delivering push notifications to web browsers**. It uses the Push API, which is a standard web API that enables websites to register and receive push messages. The Push API allows a website to send push messages to a user's browser, even when the user is not actively browsing the website.

To use Web Push, a website first needs to **obtain a push subscription from the user's browser**. The subscription consists of a unique endpoint URL and an encryption key. The endpoint URL is a URL that the website can use to send push messages to the user's browser, and the encryption key is used to encrypt and decrypt the push messages.

Once the website has **obtained a push subscription**, it can **send push messages** to the user's browser by making an HTTP request to the endpoint URL. The push message is sent in a special format called the Web Push Protocol Message, which consists of a set of headers and a payload. The headers contain information such as the encryption key and the TTL (time-to-live) of the message, while the payload contains the actual content of the message.

When the **user's browser receives a push message**, it **first decrypts the message** using the encryption key. It then **displays the notification to the user**, along with any additional actions that the user can take, such as dismissing the notification or opening the website.

To ensure the security and privacy of push messages, Web Push uses end-to-end encryption and requires that push subscriptions be obtained over a secure connection (e.g., HTTPS). Additionally, the protocol provides mechanisms for authenticating the sender of a push message and preventing abuse (e.g., by limiting the number of push messages that a website can send to a user).

Components

There are 2 components which support WebPush:

- **TsgcWSAPIServer_WebPush:** implements the WebPush protocol on the server side, allowing you to ask users for permission, register subscriptions, send notifications, and more. This component already encapsulates a WebPush client to send notifications.
- **TsgcWebPush_Client:** implements the WebPush protocol on the client side, allowing you to send notifications to users via desktop and mobile web. This is useful if you already have the keys and endpoint and only want to publish WebPush messages to the subscribed clients.

TsgcWSAPIServer_WebPush

TsgcWSServer_API_WebPush is a component that provides functionality for handling WebPush subscriptions. WebPush is a protocol for delivering real-time notifications to web applications that run in the browser. This component can be used to manage subscriptions and send notifications to subscribed clients. Find below the properties, events, and methods provided by TsgcWSServer_API_WebPush class, along with code examples that demonstrate how to use them.

Configuration

1. **Attach a TsgcWSServer_API_WebPush** to a WebSocket server using the Server property.
2. Configure the **public and private keys** in the **WebPush.VAPID** property. (Registered users can download an executable that generates the VAPID keys for windows).
3. Requires deploying the **OpenSSL 3.0.0 version**
4. In the **WebPush.Endpoints** property you can define your own endpoints to handle the WebPush subscriptions; by default, accessing the "/sgcWebPush.html" endpoint will show a simple webpage that allows you to subscribe to the WebPush notifications.
5. Start the server and access the endpoint configured to test it.

Properties

- **VAPID:** This property is used to set the VAPID (Voluntary Application Server Identification) details for sending WebPush notifications. VAPID is a method for identifying who is sending the push notifications. It is mandatory for all push notifications to have VAPID credentials. The TsgcHTTP_API_WebPush_VAPID_Options object has two properties, PublicKey and PrivateKey, which are used to identify the application server that sends the notification.
    - **DER:** the public and private keys in DER format
    - **PEM:** the private key in PEM PKCS8 format.
    - **Details:** currently only the mailto used for signing the HTTP request.
- **ClientOptions:** This property is used to set the client-side options for sending WebPush notifications.
    - **Log:** enable if you want to save the client HTTP requests to a text log.
    - **LogOptions:** here you can set the filename.
    - **TLSOptions:** currently only OpenSSL 3.0.0 supports sending WebPush notifications.
- **EndPoints:** This property is used to set the endpoints for various WebPush operations, such as subscription, unsubscription, and notification. The TsgcWSWebPushEndpoints_Options object has several properties, including Subscription, Unsubscription, ServiceWorker, Home, WebPush, and VAPIDPublicKey. Each of these properties is an instance of the TsgcWSWebPushEndpoint class, which contains the endpoint URL and other details.
    - **Home:** the default HTML page.
    - **WebPush:** the default webpush javascript code.
    - **ServiceWorker:** the javascript code that handles the push notifications.
    - **VAPIDublicKey:** the endpoint that returns the public key in DER format.
    - **Subscription:** the endpoint that notifies the webpush subscriptions.
    - **Unsubscription:** the endpoint that notifies the webpush unsubscriptions.

Methods

Find below the most important methods.

SendNotification

Use this method to send a notification given a subscription object. The subscription object is just a class with the following properties

- **Endpoint:** the url where the client must POST a message.
- **PublicKey:** the public key used to encrypt the message.
- **AuthSecret:** the secret used to encrypt the message.
- **RawText:** contains the full JSON string of the subscription.

The message can be a string or an object of TsgcWebPushMessage

```
void SendNotification(TsgcWebPushSubscription aSubscription)
{
    TsgcWebPushMessage oMessage = new TsgcWebPushMessage();
    oMessage.Title = "eSeGeCe Notification";
    oMessage.Body = "Subscription Successfully Registered!!!";
    oMessage.Icon = "https://www.esegece.com/images/esegece_logo_small.png";
    oMessage.Url = "https://www.esegece.com";
    sgcWSAPIServer_WebPush1.SendNotification(aSubscription, oMessage);
}
```

BroadcastNotification

Use this method to send a Notification to all the clients registered using the **Subscriptions** property (every time a new client is subscribed, it's added to an internal list. And when the client unsubscribed it's deleted). You can Add or Remove subscription manually using the method **Subscriptions.AddSubscription** and **Subscription.RemoveSubscription**.

```
void BroadcastNotification()
{
    TsgcWebPushMessage oMessage = new TsgcWebPushMessage();
    oMessage.Title = "eSeGeCe Notification";
    oMessage.Body = "New version released!!!";
    oMessage.Icon = "https://www.esegece.com/images/esegece_logo_small.png";
    oMessage.Url = "https://www.esegece.com";
    sgcWSAPIServer_WebPush1.BroadcastNotification(oMessage);
}
```

Events

OnWebPushSubscription

This event is fired when a client subscribes to WebPush notifications. The event handler can be used to store the subscription details on the server-side.

OnWebPushUnsubscription

This event is fired when a client unsubscribes from WebPush notifications. The event handler can be used to remove the subscription details from the server-side.

OnWebPushSendNotificationException

This event is fired when an exception occurs while sending a WebPush notification using the BroadcastNotification method. The event handler can be used to handle the exception and remove the subscription details if required.

TsgcWebPush_Client

The TsgcWebPush_Client is a class that allows you to send a notification once you obtain the subscription details.

Find below an example of using the WebPush client to send a notification given an endpoint, public key and authentication secret from a WebPush subscription.

```
public void SendWebPushNotification()
{
    var oSubscription = new TsgcHTTP_API_WebPush_PushSubscription();
    oSubscription.Endpoint = "endpoint";
    oSubscription.PublicKey = "public key";
    oSubscription.AuthSecret = "authentication secret";
    var oWebPush = new TsgcWebPush_Client();
    oWebPush.VAPID.PEM.PrivateKey.Text = "private_key_pem";
    oWebPush.VAPID.DER.PrivateKey = "private_key";
    oWebPush.VAPID.DER.PublicKey = "public_key";
    oWebPush.SendNotification(oSubscription, "{\"title\": \"eSeGeCe Notification\", \"body\": \"Hello from eSeGeC
}
```

Extensions

WebSocket protocol is designed to be extended. WebSocket Clients may request extensions and WebSocket Servers may accept some or all extensions requested by clients.

Extensions supported:

1. Deflate-Frame: compress WebSocket frames.

2. PerMessage-Deflate: compress WebSocket messages.

Extensions | PerMessage-Deflate

PerMessage is a WebSocket protocol extension, if the extension is supported by Server and Client, both can compress transmitted messages:

- Uses Deflate as the compression method.
- Compression only applies to Application data (control frames and headers are not affected).
- Server and client can select which messages will be compressed.

Max Window Bits

This extension allows customizing the server and client size of the sliding window used by the LZ77 algorithm (between 8 and 15). The greater this value, the more likely it is to find and eliminate duplicates, but it consumes more memory and CPU cycles. 15 is the default value.

No Context Take Over

By default, previous messages are used for compression and decompression. If messages are similar, this improves the compression ratio. If enabled, then each message is compressed using only its own message data. By default, it is disabled.

MemLevel

This value is not negotiated between server and client. When set to 1, it uses the least memory, but slows down the compression algorithm and reduces the compression ratio; when set to 9, it uses the most memory and delivers the best performance. By default, it is set to 1.

Extensions | Deflate-Frame

This is a WebSocket protocol extension that allows the compression of frames sent using the WebSocket protocol, supported by WebKit browsers like Chrome or Safari. This extension is supported on Server and Client components.

This extension has been deprecated.

MCP

MCP (Model Context Protocol) is a standardized protocol designed to define and manage the **contextual exchange of information** between language models and external systems.

It allows structured, secure, and dynamic communication that enables AI models to **understand**, **extend**, and **interact** with external tools or data sources beyond their native environment.

MCP provides a **uniform interface** for passing contextual metadata, enabling models to interpret instructions, retrieve data, and execute tasks in a predictable and interoperable way similar in spirit to how *WebAuthn* standardizes authentication flows.

Purpose

MCP was developed to solve the challenges of **context fragmentation** and **integration inconsistency** between models, clients, and tools.
With MCP, any compliant client can communicate with any MCP-compatible model or service through a shared, well-defined structure.

Typical use cases include:

- Attaching **context metadata** (user session, app state, permissions, etc.) to model queries.
- Exchanging **function or tool definitions** dynamically.
- Managing **stateful interactions** across model sessions.
- Establishing **secure interoperability** between LLMs, SDKs, and backend services.

Components

- **TsgcWSAPIServer_MCP**: The component provides a simple but powerful solution for implementing the MCP server, with features like Tools, Prompts and Resources requests, Authentication, Flow control using events and much more.
- **TsgcWSAPIClient_MCP**: The component provides a simple but powerful solution for implementing the MCP client, with features like Tools, Prompts and Resources requests and more.

MCP Server

TsgcWSAPIServer_MCP exposes the Model Context Protocol (MCP) over an sgcWebSockets HTTP server endpoint. The component bridges incoming HTTP requests with the TsgcAI_MCP_Server engine so that MCP-compatible clients can negotiate sessions, enumerate prompts/resources/tools, and invoke tools through JSON-RPC style calls.

Configuration

- **Place components:** The TsgcWSAPIServer_MCP component must be attached to an HTTP server, TsgcWebSocketHTTP_Server or TsgcWebSocketServer_HTTPAPI using the Server property.
- **Pick an endpoint:** Adjust EndpointOptions.Endpoint if the default /mcp path should change. Requests whose document equals this path are treated as MCP calls; others fall back to the parent API's resource handling.
- **Configure MCP settings:** Use **MCPOptions.ServerInfo** to name/version the server, **MCPOptions.SessionTimeout** to control session lifetime and **MCPOptions.AuthenticationOptions** to require an API key or a custom HTTP header before any JSON-RPC call is processed.
- **Register prompts, tools, resources and resource templates:** Populate the read-only **Tools**, **Prompts**, **Resources** and **ResourceTemplates** lists exposed by the component.
- **Handle events:** Wire the published events described below to plug in business logic for session lifecycle, prompt/resource resolution, or tool execution. Exceptions can be intercepted to customize HTTP response codes.

Properties

- **EndpointOptions:** configure here which endpoint will handle the MCP requests. It is mandatory to define one endpoint; by default it is "/mcp".
- **MCPOptions:** here you can configure the main MCP options.
  - **SessionTimeout:** after the defined interval set in milliseconds, if there is no request the session will be deleted.
  - **ServerInfo:** configure the Name and the version of the MCP server.
  - **AuthenticationOptions:** combine reusable security policies.
    - **CustomHeader:** enable it to reject requests that do not include the expected header/value pair (for example `X-MCP-Client: OperationsDesk`). Headers are checked on every HTTP round-trip so long-running streaming sessions stay protected.
    - **ApiKey:** flips on shared-secret validation through the `Authorization: Bearer <value>` header. Use it together with TLS to keep credentials private.
- **TransportOptions:** enable or disable which transports are supported by the MCP Server.
  - **Http.Enabled:** short-lived HTTP requests where every connection is closed when the response is received.
  - **HttpStreamable.Enabled:** allows a persistent HTTP channel between the server and the client avoiding the negotiation phase and increasing request/response throughput.
  - **HttpStreamable.ValidateOrigin:** when enabled, enforces that subsequent requests within the same session reuse the original `Origin` header to mitigate cross-site request forgery attempts.

Securing the endpoint

MCP sits on top of standard HTTP so the component validates authentication data before it reaches the core protocol handler. When a request does not match the configured credentials an HTTP 401 is returned automatically and the JSON-RPC payload is never parsed. Combine **CustomHeader** with **ApiKey** to require both signals at the same time.

Events

- **OnMCPException:** Receives the exception and an HTTP response code (default 500) that handlers can override before the JSON-RPC error is returned.
- **OnMCPHTTPRequest:** Handlers can inspect/modify the request and set `Handled := True` to supply a custom response, skipping the MCP engine entirely.
- **OnMCPHTTPResponse:** Use to inject additional headers or logging.
- **OnMCPInitialize:** Fires before the response is sent back to the client so that server capabilities can be customised.
- **OnMCPSessionNew:** Raised every time a new session is created.
- **OnMCPSessionEnd:** Triggered when a session is closed or expires.
- **OnMCPRequestTool:** When a client requests a tool you can populate the response payload.
- **OnMCPRequestPrompt:** Raised when the client requests prompt contents.
- **OnMCPRequestResource:** Ideal for streaming files, database records, or other context resources back to the client.
- **OnMCPResponseRootsList:** Delivers the `roots.list` response generated by **RequestRootsList**, enabling applications to tailor the set of shared roots per session.
- **OnMCPResponseSamplingCreateMessage:** Raised after **RequestSamplingCreateMessage** completes so message sampling workflows can stream partial generations to clients.
- **OnMCPResponseElicitationCreate:** Raised when an `elicitation.create` request concludes, providing access to the dynamically requested schema and collected values.
- **OnMCPRequestResourceTemplatesList:** Raised when the client requests the list of resource templates via `resources/templates/list`.
- **OnMCPResourceSubscribe:** Raised when the client subscribes to resource change notifications via `resources/subscribe`.
- **OnMCPResourceUnsubscribe:** Raised when the client unsubscribes from resource change notifications via `resources/unsubscribe`.
- **OnMCPLoggingSetLevel:** Raised when the client sets the server logging level via `logging/setLevel`.
- **OnMCPCompletionComplete:** Raised when the client requests autocomplete suggestions via `completion/complete`.
- **OnMCPProgress:** Raised when the client sends a progress notification via `notifications/progress`.
- **OnMCPCancelled:** Raised when the client cancels an in-flight request via `notifications/cancelled`.

Component surface

The MCP server component aggregates lower-level building blocks from the **sgcAI_MCP** units. Understanding the exposed properties and helper classes makes it easier to wire the component into existing REST APIs or background workers.

Runtime properties

- **Server:** link to the HTTP server instance (TsgcWebSocketHTTP_Server or TsgcWebSocketServer_HTTPAPI). The component registers its endpoint handlers once this property is assigned.
- **Tools / Prompts / Resources:** read-only references to **TsgcAI_MCP_ToolsList**, **TsgcAI_MCP_PromptsList** and **TsgcAI_MCP_ResourcesList**. They describe the capabilities advertised through `tools.list`, `prompts.list` and `resources.list` requests and are the main entry point to publish custom logic.
- **MCPOptions.ServerInfo:** configure the *name* and *version* strings returned in MCP discovery responses. Populate these fields to help clients display human-friendly information in their capability browsers.
- **MCPOptions.SessionTimeout:** expressed in milliseconds. When the timer elapses without additional traffic the in-memory session record is deleted and **OnMCPSessionEnd** fires. Tune this value to balance resource usage and session stickiness.
- **EndpointOptions.Endpoint:** defaults to `/mcp`. Changing it lets the component coexist with other HTTP APIs under the same server object.

Catalogue helper methods

The collections returned by the **Tools**, **Prompts** and **Resources** properties expose a compact public surface aimed at run-time configuration. The most frequently used members are:

- **TsgcAI_MCP_ToolsList**
  - **AddTool(const Name, Description):** creates or replaces a tool entry and returns the mutable **TsgcAI_MCP_Tool** so that schemas and defaults can be configured.
  - **Clear:** remove all registered tools, for instance when reloading configuration from disk.
  - **Count / Items[Index]:** expose the catalogue for diagnostics, dashboards or tests.
- **TsgcAI_MCP_PromptsList**
  - **AddPrompt(const Name, Description):** registers a new prompt template and returns the backing **TsgcAI_MCP_Prompt** so that arguments and default messages can be edited.
  - **Clear:** wipe all prompts before repopulating them at runtime.
  - **Count / Items[Index]:** iterate over available prompts when building custom administration UIs.
- **TsgcAI_MCP_ResourcesList**
  - **AddResource(Uri, Name, Title, Description, MimeType):** publish static or computed resources that can later be streamed from **OnMCPRequestResource**.
  - **Clear:** reset the resource catalogue when the backend configuration changes.
  - **Count / Items[Index]:** provide quick access to the registered resources, simplifying health checks or debugging tools.
- **TsgcAI_MCP_ResourceTemplatesList**
  - **AddResourceTemplate(UriTemplate, Name, Title, Description, MimeType):** register a URI template pattern for dynamic resource resolution.
  - **Clear:** reset the resource templates catalogue.
  - **Count / Items[Index]:** iterate over registered templates.

Server-initiated requests

Besides answering client calls, the server can proactively request additional information from the MCP client once a session is established. The following helpers send the JSON-RPC calls and trigger the matching response events.

- **RequestRootsList(Session):** asks the client for its available roots and raises **OnMCPResponseRootsList** when the reply is received.
- **RequestSamplingCreateMessage(Session, Request):** initiates a `sampling.createMessage` flow. The typed response is exposed through **OnMCPResponseSamplingCreateMessage**.
- **RequestElicitationCreate(Session, Request):** starts an `elicitation.create` exchange and surfaces the result via **OnMCPResponseElicitationCreate**.

Server-initiated notifications

The server can also send notifications to connected clients:

- **SendNotificationResourcesUpdated(Uri):** notifies subscribed clients that a specific resource has been updated.
- **SendLogMessage(Level, Logger, Data):** sends a logging message notification to all connected sessions whose logging level is at or above the specified level.

Event lifecycle

The MCP gateway surfaces the complete HTTP-to-MCP pipeline through events. They fire in a predictable order so that authentication, request handling and telemetry can be layered without interfering with each other.

- **OnMCPHTTPRequest:** raised as soon as an HTTP request matches **EndpointOptions.Endpoint**. Inspect *Request*/*Response* parameters and flip `Handled` to `True` to return a custom payload (for example to serve health checks or static diagnostics) before the MCP engine processes the call.
- **OnMCPInitialize:** triggered once the MCP JSON payload has been parsed but before the final response is sent. Use it to stamp headers, enrich the **TsgcAI_MCP_Response*** objects with metadata or attach request-scoped services to **TsgcAI_MCP_Session**.
- **OnMCPRequestTool / OnMCPRequestPrompt / OnMCPRequestResource:** the core business events. Each one receives the active **TsgcAI_MCP_Session**, the strongly typed request and a mutable response helper. Populate `Result` structures, mark `IsError` when validation fails and, for resources, add one or more streamed contents.
- **OnMCPResponseRootsList / OnMCPResponseSamplingCreateMessage / OnMCPResponseElicitationCreate:** receive the responses of server-initiated requests, letting you merge client-provided roots, streaming messages or elicited data into your business workflow.
- **OnMCPHTTPResponse:** fires after the response has been serialized. Ideal for logging, tracing and for appending extra HTTP headers such as caching hints.
- **OnMCPSessionNew / OnMCPSessionEnd:** session bookkeeping callbacks that delimit the lifetime of each HTTP interaction. Use them to allocate per-session caches, emit audit trails or clear temporary files.
- **OnMCPException:** invoked whenever the server needs to return an MCP error. The handler receives the original exception plus a mutable HTTP status code so you can downgrade errors (for example returning 429 on rate limiting scenarios).

Example: composing tools, prompts and resources

The following Delphi snippet shows how a single MCP server can expose a planning prompt, a resource backed by analytics data and a tool that orchestrates both. The example also logs session creation/termination to highlight the lifecycle events and enables header+API-key validation so only authorised MCP clients gain access.

```
private void FormCreate(object sender, EventArgs e)
{
  MCPServer.MCPOptions.AuthenticationOptions.CustomHeader.Enabled = true;
  MCPServer.MCPOptions.AuthenticationOptions.CustomHeader.Header = "X-MCP-Client";
  MCPServer.MCPOptions.AuthenticationOptions.CustomHeader.Value = "OperationsDesk";
  MCPServer.MCPOptions.AuthenticationOptions.ApiKey.Enabled = true;
  MCPServer.MCPOptions.AuthenticationOptions.ApiKey.Value = "super-secret-token";
  MCPServer.Tools.Clear();
  TsgcAI_MCP_Tool LTool = MCPServer.Tools.AddTool("ops.generate-plan", "Creates a remediation plan for failed dep
  LTool.InputSchema.Properties.AddProperty("deploymentId", true, aimcpjtString, "Identifier returned by CI/CD");
  MCPServer.Prompts.Clear();
  TsgcAI_MCP_Prompt LPrompt = MCPServer.Prompts.AddPrompt("ops.plan-template", "Guides the assistant through reme
  LPrompt.Arguments.AddArgument("deploymentId", "Deployment identifier to analyse", true);
  LPrompt.Messages.AddText("system", "You are an SRE helping to roll back failed deployments.");
  MCPServer.Resources.Clear();
  MCPServer.Resources.AddResource("metrics://deployments", "DeploymentMetrics", "Deployment metrics feed", "Expos
}
private void MCPServerMCPSessionNew(object sender, TsgcAI_MCP_Session aSession)
{
  MemoLog.Lines.Add("#session_new " + aSession.Id);
}
private void MCPServerMCPSessionEnd(object sender, TsgcAI_MCP_Session aSession)
{
  MemoLog.Lines.Add("#session_end " + aSession.Id);
}
private void MCPServerMCPRequestPrompt(object sender,
  TsgcAI_MCP_Session aSession, TsgcAI_MCP_Request_PromptsGet aRequest,
  TsgcAI_MCP_Response_PromptsGet aResponse)
{
  if (aRequest.Params.Name.Equals("ops.plan-template", StringComparison.OrdinalIgnoreCase)) {
    aResponse.Result.Description = "Step-by-step remediation checklist";
    aResponse.Result.Messages.Clear();
    aResponse.Result.Messages.AddText("assistant", "Review deployment " + aRequest.Params.Arguments.Node["deploym
```

```
    aResponse.Result.Messages.AddText("assistant", "Summarise blockers and propose mitigations.");
  }
}
private void MCPServerMCPRequestResource(object sender,
  TsgcAI_MCP_Session aSession, TsgcAI_MCP_Request_ResourcesRead aRequest,
  TsgcAI_MCP_Response_ResourcesRead aResponse)
{
  if (aRequest.Params.Uri.Equals("metrics://deployments", StringComparison.OrdinalIgnoreCase)) {
    aResponse.Result.Contents.Clear();
    aResponse.Result.Contents.AddContentText("metrics://deployments", "DeploymentMetrics", "Deployment metrics fe
      FetchDeploymentMetrics(aRequest.Params.Arguments.Node["deploymentId"].AsString));
  } else {
    aResponse.Result.IsError = true;
  }
}
private void MCPServerMCPRequestTool(object sender,
  TsgcAI_MCP_Session aSession, TsgcAI_MCP_Request_ToolsCall aRequest,
  TsgcAI_MCP_Response_ToolsCall aResponse)
{
  if (aRequest.Params.Name.Equals("ops.generate-plan", StringComparison.OrdinalIgnoreCase)) {
    aResponse.Result.Content.Clear();
    aResponse.Result.Content.AddText("Use prompt ops.plan-template to gather context about deployment " +
      aRequest.Params.Arguments.Node["deploymentId"].AsString);
    aResponse.Result.Content.AddText("Download metrics from metrics://deployments for the same identifier.");
    aResponse.Result.Content.AddText("Draft the remediation plan and include rollback or fix-forward options.");
  }
}
```

MCP Server Flow

- MCP Sessions
  - MCP Server Sessions
- MCP Server Requests
  - MCP Server Tools
  - MCP Server Prompts
  - MCP Server Resources

MCP Server | Sessions

After a successful initialization between the server and the client, a new session is created. The session is always sent in the request and response between the client and the server to identify which session is being used in a request/response flow. When using the HTTP transport, the lifetime of a session is a single HTTP request and response, meaning that once the response has been sent, the session is closed.

The session object is passed on every event and to know when a session has been created or deleted, use the following events:

- **OnMCPSessionNew:** every time a new session id is created this event is called.
- **OnMCPSessionEnd:** when a session has ended, this event is called.

OnMCPSessionNew

The event is called when a new session is created. You can access the Session parameter to get the ID of the session, when the session expires, and more.

OnMCPSessionEnd

The event is called when a session has finished.

```
public void OnMCPSessionEnd(object sender, TsgcAI_MCP_Session aSession)
{
    Console.WriteLine("#session_end: " + aSession.Id);
}
public void OnMCPSessionNew(object sender, TsgcAI_MCP_Session aSession)
{
    Console.WriteLine("#session_new: " + aSession.Id);
}
```

MCP Server | Tools

The **TsgcWSAPIServer_MCP** is the high-level component that exposes Model Context Protocol (MCP) server features over sgcWebSockets. When **MCP clients request tools** according to the MCP Server Tools specification, this component orchestrates the HTTP gateway, session lifecycle and JSON-RPC serialization so that your Delphi code can focus on implementing business logic.

Incoming HTTP requests that hit the MCP endpoint defined in **EndpointOptions.Endpoint** are intercepted by **TsgcWSServer_API_MCP**. The component creates (or resumes) an MCP session, forwards the request to the underlying TsgcAI_MCP_Server engine and finally writes the JSON-RPC response back to the HTTP client while preserving the mcp-session-id header required by the specification.

Tools List

The server keeps an in-memory catalogue of tools in **TsgcAI_MCP_ToolsList**. Tools are guaranteed to be unique by name, expose descriptions and provide a JSON-Schema-like input description that is emitted in the response to the specification's tools.list method. When a client invokes tools.list, TsgcWSAPIServer_MCP loads the request, serializes the current catalogue and sends it back with a 200 HTTP status code.

Example code that publishes a simple calculator tool:

```
public partial class MainForm : Form
{
    private void MainForm_Load(object sender, EventArgs e)
    {
        var oTool = MCPServer.Tools.AddTool("math.add", "Adds two numbers");
        // Define input schema properties
        oTool.InputSchema.Properties.AddProperty("a", true, AIMCPJsonType.Number, "Left operand");
        oTool.InputSchema.Properties.AddProperty("b", true, AIMCPJsonType.Number, "Right operand");
    }
}
```

A compliant tools.list response sent to the client will mirror the MCP JSON schema payload. For example:

```
{
  "jsonrpc": "2.0",
  "id": "42",
  "result": {
    "tools": [
      {
        "name": "math.add",
        "description": "Adds two numbers",
        "inputSchema": {
          "type": "object",
          "required": ["a", "b"],
          "properties": {
            "a": { "type": "number", "description": "Left operand" },
            "b": { "type": "number", "description": "Right operand" }
          }
        }
      }
    ]
  }
}
```

Tool Request

When a client issues a tools.call JSON-RPC request, **TsgcWSAPIServer_MCP** hydrates the strongly-typed request object (including tool name and the provided arguments) before raising the **OnMCPRequestTool** event. Your handler populates the response payload which is then serialized back to the client, alongside a success HTTP status code.

A typical handler looks like this:

```
private void MCPServer_MCPRequestTool(
    object sender,
    TsgcAI_MCP_Session session,
    TsgcAI_MCP_Request_ToolsCall request,
    TsgcAI_MCP_Response_ToolsCall response)
{
    double LA, LB;
    if (request.Params.Name == "math.add")
    {
        LA = request.Params.Arguments.Node["a"].AsNumber;
        LB = request.Params.Arguments.Node["b"].AsNumber;
        response.Result.Content.AddText(
            string.Format("Sum = {0:F2}", LA + LB)
        );
    }
    else
    {
        response.Result.IsError = true;
    }
}
```

The generated JSON-RPC response will follow the spec's tools.call schema, wrapping one or more content parts and optionally a structured payload. Because the component clears the response and sets the HTTP code for you, no additional plumbing is required.

Public API surface

Several helper classes surface the tool catalogue and the event response helpers. The following lists summarise the methods that are typically called from production code.

TsgcAI_MCP_ToolsList

- **AddTool(const Name, Description):** creates or replaces a tool entry and returns the underlying **TsgcAI_MCP_Tool** so you can configure schemas and defaults.
- **Clear:** removes all registered tools. Call this before rebuilding the catalogue when configuration is loaded from disk.
- **Count / Items[Index]:** expose the list contents for enumeration, diagnostics or custom serialisation.

TsgcAI_MCP_Tool

- **InputSchema.Properties.AddProperty(Name, Required, JsonType, Description):** documents the JSON input payload that will be advertised through **tools.list**.
- **InputSchema.Properties.Clear:** rebuild the schema on the fly when your application refreshes tool capabilities at runtime.
- **Description:** writable property that stores the human-readable summary shared with clients.

TsgcAI_MCP_Response_ToolsCall

- **Result.Content.AddText(Value):** appends text blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.AddImage(Data, MimeType):** appends image blocks to the tool response. Multiple calls produce multi-part responses
- **Result.Content.AddAudio(Data, MimeType):** appends audio blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.AddEmbeddedResource(Uri, Title, Text, MimeType):** appends embedded resource blocks to the tool response. Multiple calls produce multi-part responses.

- **Result.Content.AddResourceLink(Uri, Name, MimeType):** appends resource link blocks to the tool response. Multiple calls produce multi-part responses.
- **Result.Content.Clear:** resets any previously generated output before you append new content.
- **Result.IsError:** set this flag to **True** when the tool call cannot be fulfilled so the framework serialises an MCP error object.

Advanced example

The next snippet builds a geo lookup tool that validates inputs, returns multiple text blocks and flips **IsError** when the parameters are invalid.

```
private void MainForm_Load(object sender, EventArgs e)
{
    MCPServer.Tools.Clear();
    var tool = MCPServer.Tools.AddTool("geo.locate", "Returns coordinates for a city and country");
    tool.InputSchema.Properties.Clear();
    tool.InputSchema.Properties.AddProperty("city", true, AIMCPJsonType.String, "City to search");
    tool.InputSchema.Properties.AddProperty("country", false, AIMCPJsonType.String, "Optional ISO country code");
}
private void MCPServer_MCPRequestTool(
    object sender,
    TsgcAI_MCP_Session session,
    TsgcAI_MCP_Request_ToolsCall request,
    TsgcAI_MCP_Response_ToolsCall response)
{
    if (!string.Equals(request.Params.Name, "geo.locate", StringComparison.OrdinalIgnoreCase))
        return;
    response.Result.Content.Clear();
    var city = request.Params.Arguments.Node["city"].AsString();
    var country = request.Params.Arguments.Node["country"].AsString();
    if (string.IsNullOrWhiteSpace(city))
    {
        response.Result.Content.AddText("City cannot be empty.");
        response.Result.IsError = true;
        return;
    }
    response.Result.Content.AddText($"City: {city}");
    if (!string.IsNullOrWhiteSpace(country))
        response.Result.Content.AddText($"Country: {country}");
    response.Result.Content.AddText($"Coordinates: {LookupCityCoordinates(city, country)}");
}
```

MCP Server | Prompts

In MCP, *prompts* are reusable templates or workflows the server exposes. They are user-controlled (clients choose which prompt to invoke). The two main methods are:

- **prompts/list:** Requests the list of all available prompt definitions exposed by the server. It's handled internally by the TsgcWSAPIServer_MCP component.
- **prompts/get:** Retrieves a specific prompt's content and optional metadata, typically a message sequence describing how the client should interact. The event **OnMCPRequestPrompt** is called when a new request is received by the server.

Additionally, if the server sets prompts.listChanged = true in its declared capabilities, it may send notifications like notifications/prompts/list_changed when the prompt catalog updates.

When a client requests prompts/get, it sends arguments (if any) and expects a sequence of messages (user or assistant roles) with content blocks (usually text, but could embed resources or images).

If something is wrong (missing prompt, invalid args), the server will return JSON-RPC error codes such as -32602 (Invalid params) or -32603 (Internal error).

Prompts List

The server keeps an in-memory catalogue of prompts in **TsgcAI_MCP_PromptsList**. Prompts are guaranteed to be unique by name, expose descriptions and provide a JSON-Schema-like input description that is emitted in the response to the specification's prompts.list method. When a client invokes prompts.list, TsgcWSAPIServer_MCP loads the request, serializes the current catalogue and sends it back with a 200 HTTP status code.

Example code that publishes a code review prompt:

```
public partial class MainForm : Form
{
    private void MainForm_Load(object sender, EventArgs e)
    {
        // Clear existing prompts
        MCPServer.Prompts.Clear();
        // Create a new prompt named "CodeReview"
        var oPrompt = MCPServer.Prompts.AddPrompt(
            "CodeReview",
            "Asks the LLM to analyze code quality and suggest improvements"
        );
        // Add argument 'code'
        oPrompt.Arguments.AddArgument(
            "code",
            "The code to review",
            true // required
        );
    }
}
```

Prompt Request

When a client issues a prompts.get JSON-RPC request, **TsgcWSAPIServer_MCP** hydrates the strongly-typed request object (including the prompt name and the provided arguments) before raising the **OnMCPRequestPrompt** event. Your handler populates the response payload which is then serialized back to the client, alongside a success HTTP status code.

A typical handler looks like this:

```
private void MCPServer_MCPRequestPrompt(
    object sender,
    TsgcAI_MCP_Session session,
    TsgcAI_MCP_Request_PromptsGet request,
    TsgcAI_MCP_Response_PromptsGet response)
{
    if (request.Params.Name == "CodeReview")
    {
        response.Result.Description = "Code review prompt";
        response.Result.Messages.AddText(
            "user",
            "Please review this Delphi code: ShowMessage('Hello World');"
        );
    }
}
```

Public API surface

Prompts are modelled through a handful of helper classes. The following methods are used most frequently when curating prompt templates and when composing a **prompts/get** response.

TsgcAI_MCP_PromptsList

- **AddPrompt(const Name, Description):** registers a new prompt entry and returns a **TsgcAI_MCP_Prompt** object so that arguments and messages can be configured.
- **Clear:** wipes the catalogue, handy during start-up when prompts are loaded from configuration files.
- **Count / Items[Index]:** provide access to the list contents for diagnostics, UI visualisation or manual serialisation.

TsgcAI_MCP_Prompt

- **Arguments.AddArgument(Name, Description, Required):** declares the structured inputs the prompt accepts.
- **Arguments.Clear:** rebuilds the argument list when your prompt changes at runtime.
- **Messages.AddText(Role, Text):** pre-seeds conversational turns that are sent back to the client as part of the prompt definition.

TsgcAI_MCP_Response_PromptsGet

- **Result.Description:** short human-readable summary that accompanies the prompt payload.
- **Result.Messages.AddText(Role, Text):** injects additional messages dynamically when the prompt is fetched.
- **Result.Messages.AddImage(Data, MimeType):** appends image blocks to the prompt response. Multiple calls produce multi-part responses
- **Result.Messages.AddAudio(Data, MimeType):** appends audio blocks to the prompt response. Multiple calls produce multi-part responses.
- **Result.Messages.AddEmbeddedResource(Uri, Title, Text, MimeType):** appends embedded resource blocks to the prompt response. Multiple calls produce multi-part responses.
- **Result.Messages.Clear:** reset the response before reusing the object.

Advanced example

The next example publishes a prompt that guides an LLM through a triage workflow. It highlights optional arguments and how to return multi-step instructions in the handler.

```
private void MainForm_Load(object sender, EventArgs e)
{
    MCPServer.Prompts.Clear();
    var 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.");
}
private void MCPServer_MCPRequestPrompt(
    object sender,
    TsgcAI_MCP_Session session,
    TsgcAI_MCP_Request_PromptsGet request,
    TsgcAI_MCP_Response_PromptsGet response)
{
    if (!string.Equals(request.Params.Name, "IssueTriage", StringComparison.OrdinalIgnoreCase))
        return;
    response.Result.Description = "Guided checklist for support analysts.";
    response.Result.Messages.Clear();
    response.Result.Messages.AddText("user", "Review the ticket summary and ask clarifying questions.");
    response.Result.Messages.AddText("assistant", "Acknowledge the request and confirm the escalation path.");
    if (!string.IsNullOrWhiteSpace(request.Params.Arguments.Node["customerPriority"].AsString()))
        response.Result.Messages.AddText("assistant", "Adjust SLA checks based on the provided customer priority.
}
```

MCP Server | Resources

In MCP, **resources** represent addressable data objects that the server exposes such as files, database records, generated documents, or dynamic API outputs.

They are *client-controlled*, meaning the client can decide which resource to request and how to interpret the content.

The two main methods involved are:

- **resources/list:** Requests the list of all available resource definitions the server exposes. It's handled internally by the TsgcWSAPIServer_MCP component.
- **resources/read:** Retrieves the content and optional metadata of a specific resource. The event **OnMCPRequestResource** is called when a new request is received by the server. A resource typically includes:
  - **uri:** resource identifier.
  - **mimeType:** content type (text/plain, application/json, image/png, etc.).
  - **data or content:** the actual payload (text or binary).

Additionally, if the server sets resources.listChanged = true in its declared capabilities, it may send notifications like notifications/resources/list_changed when the resource catalog updates.

Resources List

The server keeps an in-memory catalogue of resources in **TsgcAI_MCP_ResourcesList**. Resources are guaranteed to be unique by uri, expose descriptions and provide a JSON-Schema-like input URI that is emitted in the response to the specification's resources.list method. When a client invokes resources.list, **TsgcWSAPIServer_MCP** loads the request, serializes the current catalogue and sends it back with a 200 HTTP status code.

Example code that publishes a file resource:

```
public partial class MainForm : Form
{
    private void MainForm_Load(object sender, EventArgs e)
    {
        // Clear existing resources
        MCPServer.Resources.Clear();
        // Register a Rust source file resource
        MCPServer.Resources.AddResource(
            "file:///project/src/main.rs",       // URI
            "main.rs",                           // Name
            "Rust Software Application Main File", // Title
            "Primary application entry point",     // Description
            "text/x-rust"                          // MIME type
        );
    }
}
```

Resources Request

When a client issues a resource.read JSON-RPC request, **TsgcWSAPIServer_MCP** hydrates the strongly-typed request object (including uri resource, name and the provided arguments) before raising the **OnMCPRequestResource** event. Your handler populates the response payload which is then serialized back to the client, alongside a success HTTP status code.

A typical handler looks like this:

```
private void MCPServer_MCPRequestResource(
```

```
    object sender,
    TsgcAI_MCP_Session session,
    TsgcAI_MCP_Request_ResourcesRead request,
    TsgcAI_MCP_Response_ResourcesRead response)
{
    if (request.Params.Uri == "file:///project/src/main.rs")
    {
        response.Result.Contents.AddContentText(
            "file:///project/src/main.rs",         // URI
            "main.rs",                             // Name
            "Rust Software Application Main File", // Description
            "text/x-rust",                         // MIME type
            "fn main() {\n    println!(\"Hello world!\");\n}" // Content
        );
    }
}
```

Public API surface

Resources combine a catalogue entry (**TsgcAI_MCP_Resource**) and the streaming helpers used inside **OnMCPRequestResource**. These are the methods you will reach for most often.

TsgcAI_MCP_ResourcesList

- **AddResource(Uri, Name, Title, Description, MimeType):** registers a static or dynamically generated resource.
- **Clear:** removes every resource definition, useful before rebuilding the list from configuration files.
- **Count / Items[Index]:** expose the catalogue for enumeration or diagnostic views.

TsgcAI_MCP_Resource

- **MimeType / Title / Description:** writable properties that describe the resource payload returned to the client.
- **Name / Uri:** use these properties to update identifiers without re-registering the resource entry.

TsgcAI_MCP_Response_ResourcesRead

- **Result.Contents.AddContentText(Uri, Name, Title, MimeType, Text):** sends a text payload back to the caller.
- **Result.Contents.AddContentBlob(Uri, Name, Title, MimeType, Text):** sends a text payload back to the caller.
- **Result.Contents.Clear:** resets the outgoing payload before reusing the response object.
- **Result.IsError:** flag the response as an error when the resource cannot be produced.

Advanced example

The example below publishes a JSON resource that returns aggregated metrics. It demonstrates how to validate the incoming URI and craft different responses depending on the request.

```
private void MainForm_Load(object sender, EventArgs e)
{
    MCPServer.Resources.Clear();
    MCPServer.Resources.AddResource(
        "metrics://daily",
        "DailyMetrics",
        "Daily KPI snapshot",
        "Summaries generated each night",
        "application/json");
}
```

```
private void MCPServer_MCPRequestResource(
    object sender,
    TsgcAI_MCP_Session session,
    TsgcAI_MCP_Request_ResourcesRead request,
    TsgcAI_MCP_Response_ResourcesRead response)
{
    response.Result.Contents.Clear();
    if (string.Equals(request.Params.Uri, "metrics://daily", StringComparison.OrdinalIgnoreCase))
    {
        response.Result.Contents.AddContentText(
            "metrics://daily",
            "DailyMetrics",
            "Daily KPI snapshot",
            "application/json",
            BuildMetricsPayload(DateTime.Today));
        return;
    }
    response.Result.IsError = true;
    response.Result.Contents.AddContentText(
        request.Params.Uri,
        "UnknownResource",
        "Not found",
        "text/plain",
        "Resource is not published by this server.");
}
```

MCP Server | Roots

Roots mark the filesystem locations a client allows the server to explore. Each entry is a file:// URI with an optional display name, letting MCP automations honour project boundaries while the user keeps control of what is shared. Servers query the catalogue by issuing roots/list and, when the client announces listChanged, receive notifications/roots/list_changed so they can refresh cached views.
Because roots are client-owned, the server always acts as the requester. The roots capability must be present in the client's initialise payload before you attempt a discovery call; otherwise RequestRootsList raises CapabilityNotSupported.

Requesting the root catalogue

TsgcWSAPIServer_MCP exposes a helper that serialises roots/list and routes the reply back to your code:

- **Request helper:** call RequestRootsList(Session) once the MCP session advertises the roots capability. The method assigns an identifier automatically (unless you provide one) and persists the pending request so the response can be correlated.
- **Response event:** handle OnMCPResponseRootsList(Sender, Session, Request, Response). The event fires when the client answers with the available directories. Inspect Response.Roots or log Response.Write to examine the JSON payload.
- **Change detection:** if the client later sends notifications/roots/list_changed, submit another RequestRootsList so your server side cache stays in sync.

Sample code

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

```
public partial class FRMMCPServer : Form
{
    public FRMMCPServer()
    {
        InitializeComponent();
        MCPServer.OnMCPSessionNew += MCPServer_SessionNew;
        MCPServer.OnMCPResponseRootsList += MCPServer_RootsList;
    }

    private void MCPServer_SessionNew(object sender, TsgcAI_MCP_Session session)
    {
        if (session.ClientCapabilities.Roots.Enabled)
        {
            var requestId = MCPServer.RequestRootsList(session);
            memoLog.AppendText($"roots/list sent id {requestId}\r\n");
        }
    }

    private void MCPServer_RootsList(
        object sender,
        TsgcAI_MCP_Session session,
        TsgcAI_MCP_Request_RootsList request,
        TsgcAI_MCP_Response_RootsList response)
    {
        memoLog.AppendText($"roots/list returned {response.Roots.Count} entries\r\n");
        memoLog.AppendText("roots/list payload " + response.Write() + "\r\n");
    }
}
```

MCP Server | Sampling

Sampling lets the server ask the client to run an LLM generation on its behalf. Requests describe a conversation history, optional system prompt, model preferences and token budgets. The client keeps control of which provider is used, performs any human-in-the-loop reviews and finally returns the approved completion.
Clients must declare the sampling capability during initialize; otherwise a sampling attempt is rejected with a capability error. The specification encourages user approval and allows text, image or audio content blocks inside the conversational history.

Issuing a sampling request

TsgcWSAPIServer_MCP streamlines the sampling/createMessage flow with a helper and a response event:

- **Request helper:** build a TsgcAI_MCP_Request_SamplingCreateMessage, populate Params.Messages, Params.ModelPreferences, Params.SystemPrompt and optionally Params.MaxTokens. Pass it to RequestSamplingCreateMessage(Session, Request); the method stamps an id (if needed) and records the pending call.
- **Response event:** handle OnMCPResponseSamplingCreateMessage(Sender, Session, Request, Response). Inspect Response.Role, Response.Content, Response.Model and Response.StopReason to continue your workflow.
- **Error handling:** declined or cancelled requests arrive as JSON-RPC errors, so wrap the helper in try/except and monitor OnMCPException for logging.

Sample code

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

```
public partial class FRMMCPServer : Form
{
    public FRMMCPServer()
    {
        InitializeComponent();
        MCPServer.OnMCPResponseSamplingCreateMessage += MCPServer_SamplingResponse;
    }
    private void RequestTicketSummary(TsgcAI_MCP_Session session, string ticketText)
    {
        if (!session.ClientCapabilities.Sampling.Enabled)
            return;
        var request = new TsgcAI_MCP_Request_SamplingCreateMessage();
        request.Params.Messages.AddTextMessage(
            "user",
            $"Summarise the following ticket: {ticketText}"
        );
        request.Params.SystemPrompt = "You are a concise support assistant.";
        request.Params.ModelPreferences.Hints.AddHint("claude-3-sonnet");
        request.Params.ModelPreferences.SpeedPriority = 0.6;
        request.Params.ModelPreferences.IntelligencePriority = 0.8;
        request.Params.HasMaxTokens = true;
        request.Params.MaxTokens = 200;
        var requestId = MCPServer.RequestSamplingCreateMessage(session, request);
        memoLog.AppendText($"sampling/createMessage id {requestId}\r\n");
    }
    private void MCPServer_SamplingResponse(
        object sender,
        TsgcAI_MCP_Session session,
        TsgcAI_MCP_Request_SamplingCreateMessage request,
        TsgcAI_MCP_Response_SamplingCreateMessage response)
    {
        if (response.Content is TsgcAI_MCP_Response_Result_Content_Text text)
        {
            memoLog.AppendText($"assistant: {text.Text}\r\n");
        }
        memoLog.AppendText($"model: {response.Model} reason: {response.StopReason}\r\n");
    }
}
```

MCP Server | Elicitation

Elicitation lets the server ask the client to collect structured input from the user mid-workflow. The request contains a human-friendly message plus a constrained JSON schema describing the fields that must be captured. The client renders an appropriate form, validates the answer locally, and returns whether the user accepted, declined, or cancelled the prompt.
The client must advertise the elicitation capability during initialize. Schemas are intentionally limited to flat objects with primitive properties (string, number, boolean or enum) to make rendering straightforward and to keep sensitive data out of scope.

Requesting user input

TsgcWSAPIServer_MCP provides a helper for elicitation/create and raises a response event when the user finishes interacting:

- **Request helper:** create TsgcAI_MCP_Request_ElicitationCreate, set Params._Message, and describe the expected payload through Params.RequestedSchema. Call RequestElicitationCreate(Session, Request) to send it to the client.
- **Response event:** handle OnMCPResponseElicitationCreate(Sender, Session, Request, Response). Inspect Response.Action (accept, decline or cancel) and parse Response.Content when the user accepted.
- **Validation:** rejected requests arrive as JSON-RPC errors. Always check Action before consuming any content and be prepared to ask again if the user cancelled.

Sample code

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

```
public partial class FRMMCPServer : Form
{
    public FRMMCPServer()
    {
        InitializeComponent();
        MCPServer.OnMCPResponseElicitationCreate += MCPServer_ElicitationResponse;
    }
    private void RequestContactDetails(TsgcAI_MCP_Session session)
    {
        if (!session.ClientCapabilities.Elicitation.Enabled)
            return;
        var 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.";
        var name = request.Params.RequestedSchema.Properties.AddProperty("name");
        name._Type = "string";
        name.Title = "Full name";
        name.Required = true;
        var email = request.Params.RequestedSchema.Properties.AddProperty("email");
        email._Type = "string";
        email.Title = "Email address";
        email.Format = "email";
        email.Required = true;
        var 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;
        var requestId = MCPServer.RequestElicitationCreate(session, request);
        memoLog.AppendText($"elicitation/create id {requestId}\r\n");
    }
    private void MCPServer_ElicitationResponse(
        object sender,
        TsgcAI_MCP_Session session,
        TsgcAI_MCP_Request_ElicitationCreate request,
        TsgcAI_MCP_Response_ElicitationCreate response)
    {
        memoLog.AppendText($"elicitation action: {response.Action}\r\n");
        if (string.Equals(response.Action, "accept", StringComparison.OrdinalIgnoreCase)
            && response.Content != null)
        {
            memoLog.AppendText("payload: " + response.Content.Text + "\r\n");
```

```
        }
    }
}
```

OpenAI

OpenAI is a private research laboratory that aims to develop and direct artificial intelligence (AI) in ways that benefit humanity as a whole. OpenAI has developed the following projects:

- **GPT-3:** This powerful language model serves as the basis for other OpenAI products. It analyzes human-generated text to learn to generate similar text on its own.
- **DALL-E and DALL-E 2:** These generative AI platforms can analyze text-based descriptions of images that users want them to produce and then generate those images exactly as described.
- **CLIP:** CLIP is a neural network that synthesizes visuals and text pertaining to them to predict the best possible captions that most accurately describe those visuals. Because of its ability to learn from more than one type of data (both images and text), it can be categorized as multimodal AI.
- **ChatGPT:** ChatGPT is currently the most advanced AI chatbot designed for generating humanlike text and producing answers to users' questions. Having been trained on large data sets, it can generate answers and responses the way a human would.
- **Codex:** Codex was trained on billions of lines of code in various programming languages to help software developers simplify coding processes. It's founded on GPT-3 technology, but instead of generating text, it generates code.
- **Whisper:** Whisper is an automatic speech recognition (ASR) tool. It has been trained on a multitude of audio data to recognize, transcribe, and translate speech in about 100 different languages, including technical language and different accents.

OpenAI API

The OpenAI API can be applied to virtually any task that involves understanding or generating natural language, code, or images. OpenAI offers a spectrum of models with different levels of power suitable for different tasks, as well as the ability to fine-tune your own custom models. These models can be used for everything from content generation to semantic search and classification.

Most common uses

- **Completion**
  - **OpenAI Completion Examples**
- **Chat**
  - **OpenAI Chat Examples**
- **Edit**
  - **OpenAI Edit Examples**
- **Audio**
  - **OpenAI Transcribe & Translate Examples**
- **Moderation**
  - **OpenAI Moderation Examples**
- **RealTime**
  - **OpenAI RealTime Examples**
- **Responses**
  - **OpenAI Responses Examples**
- **Speech**
  - **OpenAI Speech Examples**
- **Fine-Tuning**
  - **OpenAI Fine-Tuning Examples**
- **Batch**
  - **OpenAI Batch Examples**

- **Uploads**
  - **OpenAI Uploads Examples**

Configuration

OpenAI

The OpenAI API uses API keys for authentication. Visit your API Keys page to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code (browsers, apps). Production requests must be routed through your own backend server where your API key can be securely loaded from an environment variable or key management service.

This **API Key** must be configured in the **OpenAIOptions.ApiKey** property of the component. Optionally, for users who belong to multiple organizations, you can set your Organization in the property **OpenAIOptions.Organization** if your account belongs to an organization.

Once the API Key is configured, find below a list of available functions to interact with the OpenAI API.

Azure

The client supports Microsoft Azure OpenAI Services, so you can use your Azure account to interact with the Azure OpenAI API too. To configure the client to work with Azure, follow the steps below:

1. Configure the property **OpenAIOptions.Provider** = oapvAzure
2. Set the values of ResourceName and DeploymentId (these values can be located in your Azure Account)
    1. **OpenAIOptions.AzureOptions.ResourceName** = <your resource name>.
    2. **OpenAIOptions.AzureOptions.DeploymentId** = <your deployment id>.
3. Set the API Key of your Azure Account
    1. **OpenAIOptions.ApiKey** = <azure api key>.

Keep in mind that not all the OpenAI methods are supported by Azure, currently only the following methods are supported:

1. Completion
2. Chat Completion

Properties

OpenAIOptions

- **ApiKey:** The API key for authenticating with the OpenAI API.
- **Organization:** Optional organization ID for users belonging to multiple organizations.
- **Provider:** Select the provider: oapvOpenAI (default) or oapvAzure for Microsoft Azure OpenAI Services.
- **AzureOptions:** Configuration for Azure OpenAI Services.
    - **ResourceName:** The Azure resource name.
    - **DeploymentId:** The Azure deployment ID.
    - **APIVersion:** The Azure API version.
- **HttpOptions:** HTTP connection settings.
    - **ReadTimeout:** Timeout in milliseconds for reading HTTP responses. Default is 0 (no timeout).
- **LogOptions:** Logging configuration.
    - **Enabled:** When True, HTTP requests and responses are logged to a file.
    - **FileName:** The file path where log output is written.
- **RetryOptions:** Automatic retry configuration for failed API requests.
    - **Enabled:** When True, failed requests are automatically retried.
    - **Retries:** Maximum number of retry attempts.
    - **Wait:** Wait time in milliseconds between retry attempts.

Models

List and describe the various models available in the API.

- **GetModels:** Lists the currently available models, and provides basic information about each one such as the owner and availability.
- **GetModel:** Retrieves a model instance, providing basic information about the model such as the owner and permissioning.
    - **Model:** The ID of the model to use for this request

Completions

Given a prompt, the model will return one or more predicted completions, and can also return the probabilities of alternative tokens at each position.

- **CreateCompletion:** Creates a completion for the provided prompt and parameters
    - **Model:** ID of the model to use. You can use the List models API to see all of your available models, or see our Model overview for descriptions of them.
    - **Prompt:** The prompt to generate completions.

Chat

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

- **Model:** ID of the model to use. Call GetModels to get a list of all models supported by the Chat API.
- **Message:** The message to generate chat completions for.
- **Role:** by default user, other options are: system, assistant.

Edits

Given a prompt and an instruction, the model will return an edited version of the prompt.

- **CreateEdit:** Creates a new edit for the provided input, instruction, and parameters.
    - **Model:** ID of the model to use. You can use the text-davinci-edit-001 or code-davinci-edit-001 model with this endpoint.
    - **Instruction:** The instruction that tells the model how to edit the prompt.
    - **Input:** (optional) The input text to use as a starting point for the edit.

Images

Given a prompt and/or an input image, the model will generate a new image.

- **CreateImage:** Creates an image given a prompt.
    - **Prompt:** A text description of the desired image(s). The maximum length is 1000 characters.
- **CreateImageEdit:** Creates an edited or extended image given an original image and a prompt.
    - **Image:** The image to edit. Must be a valid PNG file, less than 4MB, and square. If mask is not provided, image must have transparency, which will be used as the mask.
    - **Prompt:** A text description of the desired image(s). The maximum length is 1000 characters.
- **CreateImageVariations:** Creates a variation of a given image.
    - **Image:** The image to use as the basis for the variation(s). Must be a valid PNG file, less than 4MB, and square.

Embeddings

Get a vector representation of a given input that can be easily consumed by machine learning models and algorithms.

- **CreateEmbeddings:** Creates an embedding vector representing the input text.
  - **Model:** ID of the model to use.
  - **Input:** Input text to get embeddings for.

Audio

Turn Audio into Text.

- **CreateTranscriptionFromFile:** Transcribes audio into the input language from a filename
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Filename:** The audio file to transcribe, in one of these formats: mp3, mp4, mpeg, mpga, m4a, wav, or webm.
- **CreateTranscription:** Records audio for X seconds and transcribes it.
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Time:** time in milliseconds, by default 10 seconds.
- **CreateTranslationFromFile:** Translates audio into English.
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Filename:** The audio file to translate, in one of these formats: mp3, mp4, mpeg, mpga, m4a, wav, or webm.
- **CreateTranslation:** Records audio for X seconds and translates it.
  - **Model:** ID of the model to use. Only whisper-1 is currently available.
  - **Time:** time in milliseconds, by default 10 seconds.

Files

Files are used to upload documents that can be used with features like Fine-tuning.

- **ListFiles:** Returns a list of files that belong to the user's organization.
- **UploadFile:** Upload a file that contains document(s) to be used across various endpoints/features. Currently, the size of all the files uploaded by one organization can be up to 1 GB.
  - **Filename:** Name of the JSON Lines file to be uploaded. If the purpose is set to "fine-tune", each line is a JSON record with "prompt" and "completion" fields representing your training examples.
  - **Purpose:** The intended purpose of the uploaded documents. Use "fine-tune" for Fine-tuning.
- **DeleteFile:** Delete a file.
  - **FileId:** The ID of the file to use for this request
- **RetrieveFile:** Returns information about a specific file.
  - **FileId:** The ID of the file to use for this request
- **RetrieveFileContent:** Returns the contents of the specified file
  - **FileId:** The ID of the file to use for this request.

Fine-Tunes

Manage fine-tuning jobs to tailor a model to your specific training data.

- **CreateFineTune:** Creates a job that fine-tunes a specified model from a given dataset. Response includes details of the enqueued job including job status and the name of the fine-tuned models once complete.
  - **TrainingFile:** The ID of an uploaded file that contains training data.
- **ListFineTunes:** List your organization's fine-tuning jobs
- **RetrieveFineTune:** Gets info about the fine-tune job.
  - **FineTuneId:** The ID of the fine-tune job
- **CancelFineTune:** Immediately cancel a fine-tune job.
  - **FineTuneId:** The ID of the fine-tune job
- **ListFineTuneEvents:** Get fine-grained status updates for a fine-tune job.
  - **FineTuneId:** The ID of the fine-tune job
- **DeleteFineTuneModel:** Delete a fine-tuned model. You must have the Owner role in your organization.
  - **Model:** The model to delete.

Moderations

Given an input text, outputs if the model classifies it as violating OpenAI's content policy.

- **CreateModeration:** Classifies if text violates OpenAI's Content Policy
  - **Input:** The input text to classify

RealTime

The OpenAI Realtime API enables low-latency, multimodal interactions including speech-to-speech conversational experiences and real-time transcription.

- **Transcription:** You can use the Realtime API for transcription-only use cases, either with input from a microphone or from a file. For example, you can use it to generate subtitles or transcripts in real-time. With the transcription-only mode, the model will not generate responses.

Assistants

Build AI assistants that can call models and use tools to perform tasks.

- **CreateAssistant:** Creates an assistant with a model and instructions.
- **ListAssistants:** Returns a list of assistants.
- **RetrieveAssistant:** Retrieves an assistant by ID.
- **ModifyAssistant:** Modifies an existing assistant.
- **DeleteAssistant:** Deletes an assistant.

Threads

Threads represent a conversation session. Messages are added to threads, which are then processed by runs.

- **CreateThread:** Creates a new thread.
- **RetrieveThread:** Retrieves a thread by ID.
- **ModifyThread:** Modifies a thread.
- **DeleteThread:** Deletes a thread.

Thread Messages

Messages are added to threads and contain the content of a conversation.

- **CreateMessage:** Creates a message within a thread.
  - **ThreadId:** The ID of the thread.
- **ListMessages:** Returns a list of messages for a given thread.
- **RetrieveMessage:** Retrieves a message by thread and message ID.
- **ModifyMessage:** Modifies a message.
- **DeleteMessage:** Deletes a message from a thread.

Runs

Runs represent an execution on a thread with an assistant. The assistant uses its configuration and the thread messages to perform tasks by calling models and tools.

- **CreateRun:** Creates a run for a thread with a specified assistant.
- **ListRuns:** Returns a list of runs belonging to a thread.
- **RetrieveRun:** Retrieves a run by thread and run ID.
- **ModifyRun:** Modifies a run.

- **SubmitToolOutputsToRun:** Submits tool outputs for a run that requires action.
- **CancelRun:** Cancels an in-progress run.

Run Steps

Run steps represent the individual steps taken during a run execution.

- **ListRunSteps:** Returns a list of run steps belonging to a run.
- **RetrieveRunSteps:** Retrieves a run step by thread, run, and step ID.

Vector Stores

Vector stores are used to store and search over files using embeddings for retrieval-augmented generation (RAG).

- **CreateVectorStore:** Creates a vector store.
- **ListVectorStores:** Returns a list of vector stores.
- **RetrieveVectorStore:** Retrieves a vector store by ID.
- **ModifyVectorStore:** Modifies a vector store.
- **DeleteVectorStore:** Deletes a vector store.

Vector Store Files

Manage files within vector stores.

- **CreateVectorStoreFile:** Attaches a file to a vector store.
- **ListVectorStoreFiles:** Returns a list of files in a vector store.
- **RetrieveVectorStoreFile:** Retrieves a vector store file.
- **DeleteVectorStoreFile:** Removes a file from a vector store.

Vector Store File Batches

Batch operations for adding files to vector stores.

- **CreateVectorStoreFileBatch:** Creates a batch to add multiple files to a vector store.
- **RetrieveVectorStoreFileBatch:** Retrieves a file batch by ID.
- **CancelVectorStoreFileBatch:** Cancels an in-progress file batch.
- **ListVectorStoreFilesBatch:** Lists files in a batch.

Speech

Generate spoken audio from text using text-to-speech models.

- **CreateSpeech:** Generates audio from the input text. Returns audio data as a string or writes to a response stream.
  - **Model:** The TTS model to use (e.g. tts-1, tts-1-hd).
  - **Input:** The text to generate audio for.
  - **Voice:** The voice to use (alloy, echo, fable, onyx, nova, shimmer).

Fine-Tuning Jobs

Manage fine-tuning jobs to create customized models. This is the newer fine-tuning API that replaces the legacy Fine-Tunes endpoint.

- **CreateFineTuningJob:** Creates a fine-tuning job from a training file.
- **ListFineTuningJobs:** Lists your organization's fine-tuning jobs.
- **RetrieveFineTuningJob:** Retrieves info about a fine-tuning job.
- **CancelFineTuningJob:** Cancels a fine-tuning job.
- **ListFineTuningJobEvents:** Lists status updates for a fine-tuning job.
- **ListFineTuningJobCheckpoints:** Lists checkpoints for a fine-tuning job.

Responses

Create and manage model responses. The Responses API supports multi-turn conversations, tool use, and structured outputs.

- **CreateResponse:** Creates a model response.

- **RetrieveResponse:** Retrieves a response by ID.
- **DeleteResponse:** Deletes a response.
- **CancelResponse:** Cancels an in-progress response.
- **ListInputItems:** Lists input items for a response.

Batches

Create and manage batch API requests for asynchronous processing at lower cost.

- **CreateBatch:** Creates a batch job.
- **RetrieveBatch:** Retrieves a batch by ID.
- **ListBatches:** Lists all batches.
- **CancelBatch:** Cancels an in-progress batch.

Uploads

Upload large files in parts. Useful for files that exceed the standard upload size limit.

- **CreateUpload:** Creates an upload session.
- **AddUploadPart:** Adds a part to an upload.
- **CompleteUpload:** Completes a multi-part upload and creates a file.
- **CancelUpload:** Cancels an in-progress upload.

OpenAI | Completion

Given a prompt, the model will return one or more predicted completions, and can also return the probabilities of alternative tokens at each position.

Simple Example

Use the text-davinci-003 model to get a predicted completion.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";
MessageBox.Show(OpenAI._CreateCompletion("text-davinci-003", "Say this is a test"));
```

Advanced Example

Use the text-davinci-003 model to get a predicted completion with more random output and generate 2 completions for each prompt.

```
TsgcHTTP_OpenAI_JSON OpenAI = new TsgcHTTP_OpenAI_JSON();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Completion oRequest = new TsgcOpenAIClass_Request_Completion();
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 (oResponse.Choices.Length > 0)
{
  MessageBox.Show(oResponse.Choices[0].Text);
}
```

OpenAI | Chat

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

Simple Example

Interact with ChatGPT by sending a Hello message.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";
MessageBox.Show(OpenAI._CreateChatCompletion("gpt-3.5-turbo", "Hello!"));
```

Advanced Example

Use the gpt-3-5 model to chat with more random output and generate 2 completions for each prompt.

```
TsgcHTTP_OpenAI_JSON OpenAI = new TsgcHTTP_OpenAI_JSON();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

TsgcOpenAIClass_Request_ChatCompletion oRequest = new TsgcOpenAIClass_Request_ChatCompletion();
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 (oResponse.Choices.Length > 0)
{
  MessageBox.Show(oResponse.Choices[0]._Message.Content);
}
```

OpenAI | Edit

Given a prompt and an instruction, the model will return an edited version of the prompt.

Simple Example

Tell OpenAI to fix the spelling mistakes of a prompt.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";
MessageBox.Show(OpenAI._CreateEdit("text-davinci-edit-001", "Fix the spelling mistakes", "What day of the wek is
```

Advanced Example

Tell OpenAI to fix the spelling mistakes of a prompt, with more random output and generate 2 completions for each prompt.

```
TsgcHTTP_OpenAI_JSON OpenAI = new TsgcHTTP_OpenAI_JSON();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Edit oRequest = new TsgcOpenAIClass_Request_Edit();
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 (oResponse.Choices.Length > 0)
{
  MessageBox.Show(oResponse.Choices[0].Text);
}
```

OpenAI | Audio

Create Transcription

Transcribes audio into the input language.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

MessageBox.Show(OpenAI._CreateTranscriptionFromFile("whisper-1", "c:\media\audio.mp3"));
```

Create Translation

Translates audio to English.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

MessageBox.Show(OpenAI._CreateTranslationFromFile("whisper-1", "c:\media\audio.mp3"));
```

OpenAI | Moderation

Given an input text, outputs if the model classifies it as violating OpenAI's content policy.

Simple Example

Moderate the following text

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

MessageBox.Show(OpenAI._CreateModeration("I want to kill them."));
```

Advanced Example

Moderate the following text choosing the model.

```
TsgcHTTP_OpenAI_JSON OpenAI = new TsgcHTTP_OpenAI_JSON();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

TsgcOpenAIClass_Request_Moderation oRequest = new TsgcOpenAIClass_Request_Moderation();
oRequest.Model = "text-moderation-latest";
oRequest.Input = "I want to kill them.";
TsgcOpenAIClass_Response_Moderation oResponse = OpenAI.CreateModeration(oRequest);

if (oResponse.results.Length > 0)
{
  MessageBox.Show(oResponse.results[0].flagged.ToString());
}
```

OpenAI | RealTime

You can use the Realtime API for transcription-only use cases, either with input from a microphone or from a file. For example, you can use it to generate subtitles or transcripts in real-time. With the transcription-only mode, the model will not generate responses.

To use the Realtime API for transcription, you need to create a transcription session, connecting via WebSockets. Use the component **TsgcWSAPI_OpenAI** and **TsgcWebSocketClient** to start a new transcription session.

Find below an example of real-time transcription using OpenAI API.

```
var wsClient = new TsgcWebSocketClient(null);
var audioRecorder = new TsgcAudioRecorderWave(null);
var openAI = new TsgcWSAPI_OpenAI(null);
openAI.Client = wsClient;
openAI.AudioRecorder = audioRecorder;
openAI.OpenAIOptions.APIKey = "your-api-key-here";
openAI.OpenAIOptions.Method = OpenAIMethod.Transcription;
openAI.OpenAIOptions.Provider = OpenAIProvider.OpenAI;
openAI.InputAudio.Language = "en";
openAI.InputAudio.Model = "whisper-1";
openAI.AudioTranscriptionCompleted += OnAudioTranscriptionCompleted;
private void OnAudioTranscriptionCompleted(object sender, TsgcWSOpenAIConversation_Item_Completed aItem)
{
    Console.WriteLine("#transcription_completed: " + aItem.Transcript);
}
```

Send Audio Manually

The component allows you to send audio files manually. You can use the method AppendInputAudioBuffer and pass the audio as a TStream. The audio format must be 24 kHz mono PCM (only a rate of 24000 is supported).

OpenAI | Responses

The Responses API is the core API for interacting with OpenAI models. It supports text and image inputs, tool use, streaming, and structured outputs.

Simple Example

Create a response using the Responses API.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";
MessageBox.Show(OpenAI._CreateResponse("gpt-4o", "What is the capital of France?"));
```

Methods

- **CreateResponse:** Creates a model response given a model and input.
  - **Model:** ID of the model to use (e.g. gpt-4o, gpt-4o-mini).
  - **Input:** Text or structured input for the model.
- **RetrieveResponse:** Retrieves a previously created response by its ID.
  - **ResponseId:** The ID of the response to retrieve.
- **DeleteResponse:** Deletes a response by its ID.
  - **ResponseId:** The ID of the response to delete.
- **CancelResponse:** Cancels an in-progress response.
  - **ResponseId:** The ID of the response to cancel.
- **ListInputItems:** Returns a list of input items for a given response.
  - **ResponseId:** The ID of the response.

OpenAI | Speech

Generate spoken audio from text using the Text-to-Speech (TTS) API. Supports multiple voices and output formats.

Simple Example

Generate speech from text and save to a file stream.

```
TsgcHTTP_API_OpenAI 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
{
  oStream.Dispose();
}
```

Methods

- **CreateSpeech:** Generates audio from the input text.
  - **Model:** ID of the model to use (tts-1 or tts-1-hd).
  - **Input:** The text to generate audio for. Maximum 4096 characters.
  - **Voice:** The voice to use (alloy, echo, fable, onyx, nova, shimmer).
  - **ResponseStream:** The stream where the audio data will be written.
  - **ResponseFormat:** (optional) The audio format: mp3 (default), opus, aac, or flac.
  - **Speed:** (optional) The speed of the generated audio (0.25 to 4.0, default 1.0).

OpenAI | Fine-Tuning

Manage fine-tuning jobs to tailor a model to your specific training data. Fine-tuning lets you get more out of the models by providing higher quality results, the ability to train on more examples, and cost savings.

Simple Example

Create a fine-tuning job and list existing jobs.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

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

// List fine-tuning jobs
MessageBox.Show(OpenAI._ListFineTuningJobs());
```

Methods

- **CreateFineTuningJob:** Creates a fine-tuning job which begins the process of creating a new model from a given dataset.
  - **Model:** The name of the model to fine-tune.
  - **TrainingFile:** The ID of an uploaded file that contains training data.
- **ListFineTuningJobs:** List your organization's fine-tuning jobs.
- **RetrieveFineTuningJob:** Get info about a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job.
- **CancelFineTuningJob:** Immediately cancel a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job to cancel.
- **ListFineTuningJobEvents:** Get status updates for a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job.
- **ListFineTuningJobCheckpoints:** List checkpoints for a fine-tuning job.
  - **JobId:** The ID of the fine-tuning job.

OpenAI | Batch

Create large batches of API requests to run asynchronously. The Batch API returns completions within 24 hours at a 50% discount.

Simple Example

Create a batch job and retrieve its status.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

// Create a batch
MessageBox.Show(OpenAI._CreateBatch("file-abc123", "/v1/chat/completions"));

// List batches
MessageBox.Show(OpenAI._ListBatches());
```

Methods

- **CreateBatch:** Creates and executes a batch from an uploaded file of requests.
  - **InputFileId:** The ID of an uploaded file that contains requests for the batch.
  - **Endpoint:** The endpoint to be used for all requests in the batch (e.g. /v1/chat/completions).
  - **CompletionWindow:** (optional) The time frame within which the batch should be processed. Default is 24h.
- **RetrieveBatch:** Retrieves a batch by its ID.
  - **BatchId:** The ID of the batch to retrieve.
- **ListBatches:** List your organization's batches.
- **CancelBatch:** Cancels an in-progress batch.
  - **BatchId:** The ID of the batch to cancel.

OpenAI | Uploads

Upload large files in parts. The Uploads API allows you to upload files larger than 512 MB by splitting them into parts and uploading each part separately.

Simple Example

Create an upload, add a part, and complete it.

```
TsgcHTTP_API_OpenAI OpenAI = new TsgcHTTP_API_OpenAI();
OpenAI.OpenAIOptions.ApiKey = "API_KEY";

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

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

// Complete the upload
MessageBox.Show(OpenAI._CompleteUpload("upload_abc123", "[\"part_abc123\"]"));
```

Methods

- **CreateUpload:** Creates an intermediate Upload object to begin a multi-part upload.
  - **Filename:** The name of the file to upload.
  - **Purpose:** The intended purpose of the uploaded file (e.g. fine-tune).
  - **Bytes:** The number of bytes in the file.
  - **MimeType:** The MIME type of the file (e.g. application/jsonl).
- **AddUploadPart:** Adds a part to an Upload object.
  - **UploadId:** The ID of the Upload.
  - **Filename:** The chunk of bytes for this part.
- **CompleteUpload:** Completes the Upload. The uploaded file is usable only after completion.
  - **UploadId:** The ID of the Upload.
  - **PartIds:** The ordered list of Part IDs as a JSON array string.
- **CancelUpload:** Cancels the Upload. No parts may be added after cancellation.
  - **UploadId:** The ID of the Upload to cancel.

OpenAI Audio

To use OpenAI APIs with voice commands, the following steps are required:

1. The microphone audio must be captured, so a speech-to-text system is needed to get the text that will be sent to OpenAI.
    1. Capturing the microphone audio is done using the component TsgcAudioRecorderMCI.
    2. Once we've captured the audio, it is sent to the OpenAI Whisper API to convert the audio file to text.
2. Once we get the speech-to-text result, we send the text to OpenAI using the ChatCompletion API.
3. The response from OpenAI must then be converted to speech using one of the following components:
    1. TsgcTextToSpeechSystem: (currently only for Windows) uses the Windows text-to-speech from the operating system.
    2. TsgcTextToSpeechGoogle: sends the response from OpenAI to the Google Cloud Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.
    3. TsgcTextToSpeechAmazon: sends the response from OpenAI to the Amazon AWS Servers and an mp3 file is returned which is played by the TsgcAudioPlayerMCI.

Components

The following components are used for capturing audio from the microphone, playing audio files, and converting text to speech.

- **TsgcAudioRecorderMCI:** (for windows only) this component allows you to access the microphone and convert the speech to a wave file.
- **TsgcAudioPlayerMCI:** (for windows only) this component allows you to play an mp3 file.
- **TsgcTextToSpeechSystem:** (for windows only) converts text to speech without the need of using an external mp3 file.
- **TsgcTextToSpeechGoogle:** converts text to speech using any of the Google Cloud voices available.
- **TsgcTextToSpeechAmazon:** converts text to speech using any of the Amazon AWS voices available.

TsgcAIChat - Unified AI Chat

TsgcAIChat is a single component that works with any AI provider. Switch providers by changing one property. The component includes built-in conversation history management, making it easy to build chat applications without manually tracking messages.

Supported Providers

- OpenAI
- Anthropic
- Gemini
- DeepSeek
- Ollama
- Grok
- Mistral

Properties

- **Provider:** TsgcAIChatProvider enum that selects the AI provider (aicpOpenAI, aicpAnthropic, aicpGemini, aicpDeepSeek, aicpOllama, aicpGrok, aicpMistral).
- **ChatOptions.ApiKey:** The API key for the selected provider.
- **ChatOptions.Model:** The model name to use (e.g. gpt-4o-mini, claude-sonnet-4-20250514, grok-3).
- **ChatOptions.MaxTokens:** Maximum number of tokens to generate (default 4096).
- **ChatOptions.Temperature:** Controls randomness in responses (0.0 to 2.0).
- **ChatOptions.BaseUrl:** Custom base URL for the API endpoint (useful for Ollama or custom deployments).
- **SystemMessage:** A persistent system prompt that is included with every request.
- **MaxHistoryMessages:** Maximum number of messages to keep in conversation history (default 50).

Methods

- **Chat(aMessage):** Sends a message and returns the complete response as a string.
- **ChatStream(aMessage):** Sends a message with streaming enabled. Response chunks are delivered through the OnChatStream event.
- **ChatWithSystem(aSystem, aMessage):** Sends a one-shot message with a custom system prompt, without affecting the persistent SystemMessage property.
- **ClearHistory:** Resets the conversation history, removing all stored messages.

Events

- **OnChatMessage(Sender, aRole, aContent):** Fired when a complete response is received. aRole indicates the message role (e.g. assistant), aContent contains the full response text.
- **OnChatStream(Sender, aChunk, Cancel):** Fired for each chunk during streaming. aChunk contains the partial text. Set Cancel to True to stop streaming.
- **OnChatError(Sender, aError):** Fired when an error occurs. aError contains the error description.

Simple Example

Create a chat component, configure a provider, and send a message.

```
TsgcAIChat Chat = new TsgcAIChat();
Chat.Provider = aicpOpenAI;
Chat.ChatOptions.ApiKey = "API_KEY";
Chat.ChatOptions.Model = "gpt-4o-mini";
MessageBox.Show(Chat.Chat("Hello!"));
```

Switch Provider Example

Change the provider to use a different AI backend with the same component.

```
Chat.Provider = aicpAnthropic;
Chat.ChatOptions.ApiKey = "ANTHROPIC_KEY";
Chat.ChatOptions.Model = "claude-sonnet-4-20250514";
MessageBox.Show(Chat.Chat("Hello from Claude!"));
```

Conversation History Example

The component automatically maintains conversation history, allowing the model to remember context from previous messages.

```
TsgcAIChat Chat = new TsgcAIChat();
Chat.Provider = aicpOpenAI;
Chat.ChatOptions.ApiKey = "API_KEY";
Chat.ChatOptions.Model = "gpt-4o-mini";
Chat.SystemMessage = "You are a helpful assistant.";
Chat.Chat("My name is John.");
MessageBox.Show(Chat.Chat("What is my name?")); // remembers context
Chat.ClearHistory();
```

Streaming Example

Use streaming to receive the response in real-time chunks.

```
TsgcAIChat Chat = new TsgcAIChat();
Chat.Provider = aicpGrok;
Chat.ChatOptions.ApiKey = "XAI_KEY";
Chat.ChatOptions.Model = "grok-3";
Chat.OnChatStream += OnStream;
Chat.ChatStream("Tell me a story.");

void OnStream(object Sender, string aChunk, ref bool Cancel)
{
  Memo1.Text = Memo1.Text + aChunk;
}
```

Ollama Local Example

Use Ollama to run models locally without an API key.

```
TsgcAIChat Chat = new TsgcAIChat();
Chat.Provider = aicpOllama;
Chat.ChatOptions.Model = "llama3";
MessageBox.Show(Chat.Chat("Hello!"));
```

Anthropic Claude

Anthropic is an AI safety company that builds reliable, interpretable, and steerable AI systems. Their flagship model family is Claude, which excels at thoughtful dialogue, content creation, complex reasoning, coding, and more.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Anthropic** to interact with the Anthropic Claude API.

Anthropic API

The Anthropic API provides access to Claude models for building AI-powered applications. The API supports text generation, vision (image understanding), tool use (function calling), extended thinking, document/PDF processing, prompt caching, citations, web search, streaming, token counting, and message batches.

Features

- **Messages**
  - **Anthropic Messages Examples**
- **Vision**
  - **Anthropic Vision Examples**
- **Tool Use**
  - **Anthropic Tool Use Examples**
- **Models**
  - **Anthropic Models Examples**
- **Batches**
  - **Anthropic Batches Examples**
- **Extended Thinking**
  - **Anthropic Extended Thinking Examples**
- **Documents**
  - **Anthropic Documents Examples**
- **Prompt Caching**
  - **Anthropic Prompt Caching Examples**
- **Citations**
  - **Anthropic Citations Examples**
- **Web Search**
  - **Anthropic Web Search Examples**
- **Structured Outputs**
  - **Anthropic Structured Outputs Examples**
- **Files**
  - **Anthropic Files API Examples**
- **MCP Connector**
  - **Anthropic MCP Connector Examples**

Configuration

The Anthropic API uses API keys for authentication. Visit your API Keys page in the Anthropic Console to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **AnthropicOptions.ApiKey** property of the component. The **AnthropicOptions.AnthropicVersion** property specifies the API version (default: 2023-06-01).

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "YOUR_API_KEY";
```

Properties

AnthropicOptions

- **ApiKey:** The API key for authenticating with the Anthropic API.

- **AnthropicVersion:** The API version string (default: 2023-06-01).
- **BetaHeaders:** Optional beta feature headers to enable pre-release API features (e.g. files-api-2025-04-14, mcp-client-2025-11-20).
- **HttpOptions:** HTTP connection settings.
  - **ReadTimeout:** Timeout in milliseconds for reading HTTP responses. Default is 0 (no timeout).
- **LogOptions:** Logging configuration.
  - **Enabled:** When True, HTTP requests and responses are logged to a file.
  - **FileName:** The file path where log output is written.
- **RetryOptions:** Automatic retry configuration for failed API requests.
  - **Enabled:** When True, failed requests are automatically retried.
  - **Retries:** Maximum number of retry attempts.
  - **Wait:** Wait time in milliseconds between retry attempts.

Messages

Send a structured list of input messages with text and/or image content, and the model will generate the next message in the conversation.

- **CreateMessage:** Creates a message with the specified model and parameters.
  - **Model:** The model to use (e.g. claude-sonnet-4-20250514).
  - **Message:** The user message content.
  - **MaxTokens:** The maximum number of tokens to generate (required, default 4096).
- **CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.
- **CreateMessageJSON:** Creates a message that returns structured JSON conforming to a provided JSON Schema.
  - **Schema:** A JSON Schema string defining the output format.
- **CreateMessageWithThinking:** Creates a message with extended thinking enabled.
  - **BudgetTokens:** Maximum token budget for thinking (minimum 1024).
- **CreateDocumentMessage:** Creates a message with a base64 document (PDF or text).
  - **DocumentBase64:** The base64-encoded document data.
  - **MediaType:** The MIME type (e.g. application/pdf).

Vision

Claude can understand images passed as base64-encoded content blocks within messages.

- **CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **Prompt:** The text prompt to accompany the image.

Tool Use

Claude can use tools (function calling) to interact with external systems. You define tools with their names, descriptions, and input schemas, and Claude will generate tool_use content blocks when it wants to call a tool.

Models

List and describe the available Claude models.

- **GetModels:** Lists all available models.
- **GetModel:** Retrieves information about a specific model.
  - **ModelId:** The ID of the model to retrieve.

Extended Thinking

Extended thinking enables Claude to reason step-by-step before responding, improving quality for complex tasks like math, coding, and analysis.

- **ThinkingType:** Set to 'enabled' to activate thinking, 'disabled' to turn it off.
- **ThinkingBudgetTokens:** Token budget for thinking (min 1024, must be less than MaxTokens).

- **CreateMessageWithAdaptiveThinking:** Creates a message with adaptive thinking, letting Claude decide the thinking depth automatically.

Documents

Claude can process PDF documents and text files sent as content blocks. Supports base64, text, and URL source types.

- **CreateDocumentMessage:** Sends a document with a text prompt.
  - **DocumentBase64:** The base64-encoded document data.
  - **MediaType:** The MIME type (application/pdf, text/plain).

Prompt Caching

Cache frequently used context (system prompts, content blocks, tool definitions) between API calls to reduce costs by up to 90% on cache reads.

- **SystemCacheControl:** Set to True on the request to cache the system prompt.
- **CacheControl:** Set to 'ephemeral' on content blocks or tool definitions.
- **BetaHeaders:** Optional beta feature headers in AnthropicOptions.

Citations

When documents are sent with citations enabled, Claude includes source references in its response. Citation types include char_location (text), page_location (PDF), content_block_location (custom content), and web_search_result_location (web search).

Web Search

Claude can search the web for real-time information using the built-in web_search tool. Other built-in tools include code_execution and computer use.

- **CreateMessageWithWebSearch:** Creates a message with web search enabled.
  - **Model:** The model to use.
  - **Message:** The user query.

Token Counting

Count the number of tokens in a message before sending it.

- **CountTokens:** Counts the number of input tokens for a message.
  - **Model:** The model to use for token counting.
  - **Message:** The message content to count tokens for.

Message Batches

The Message Batches API allows you to process large volumes of messages asynchronously.

- **ListBatches:** Lists all message batches.
- **GetBatch:** Retrieves a specific batch by ID.
- **CancelBatch:** Cancels a batch that is still processing.
- **GetBatchResults:** Retrieves the results of a completed batch.

Structured Outputs

Force Claude to return valid JSON conforming to a provided JSON Schema. Combine with the Effort parameter to control output quality vs. cost.

- **CreateMessageJSON:** Creates a message with JSON schema output.
  - **Schema:** A JSON Schema string defining the output format.
- **OutputFormatSchema:** (Request property) JSON Schema for structured output.
- **Effort:** (Request property) Controls quality vs. cost: 'low', 'medium', 'high', 'max'.
- **Strict:** (Tool property) When True, tool inputs are guaranteed to match the input_schema.

Files API

Upload, list, retrieve, download, and delete files. Uploaded files can be referenced in messages using document content blocks with file source type. Requires beta header `files-api-2025-04-14`.

- **UploadFile:** Uploads a local file. Returns file metadata.
- **ListFiles:** Lists uploaded files with pagination.
- **GetFile:** Retrieves metadata for a specific file.
- **DownloadFile:** Downloads file content.
- **DeleteFile:** Permanently deletes a file.

Request Parameters

Additional request parameters available on the Messages API.

- **ServiceTier:** Controls priority tier usage: 'auto' (default) or 'standard_only'.
- **InferenceGeo:** Controls inference geography: 'global' (default) or 'us'.
- **Container:** Container ID for reusing code execution environments across requests.
- **IsError:** (Content block property) Set True on tool_result blocks to indicate tool execution failure.
- **CacheTTL:** Extended cache time-to-live: '5m' (default) or '1h'.
- **ThinkingType:** Set to 'adaptive' to let Claude decide thinking depth automatically.

MCP Connector

Connect Claude to external MCP (Model Context Protocol) servers to access third-party tools. Requires beta header `mcp-client-2025-11-20`.

- **CreateMessageWithMCP:** Creates a message with an MCP server connection.
  - **MCPServerUrl:** HTTPS URL of the MCP server.
  - **MCPServerName:** Unique name for the server.
- **MCPServers:** (Request property) Array of MCP server definitions with ServerType, Url, Name, AuthorizationToken.
- **MCPServerName:** (Tool property) References an MCP server when ToolType is 'mcp_toolset'.

Anthropic | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to Claude.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._CreateMessage("claude-sonnet-4-20250514", "Hello!"));
```

System Prompt Example

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

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._CreateMessageWithSystem("claude-sonnet-4-20250514",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Advanced Example

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

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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!";
var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

TsgcAnthropicClass_Response_Messages oResponse = Anthropic.CreateMessage(oRequest);
if (oResponse.Content.Length > 0)
  MessageBox.Show(oResponse.Content[0].Text);
```

Streaming Example

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

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
Anthropic.OnHTTPAPISSE += OnSSEEvent;
Anthropic._CreateMessageStream("claude-sonnet-4-20250514", "Tell me a story.");

void OnSSEEvent(object Sender, string aEvent, string aData, ref bool Cancel)
{
  // aEvent contains the event type (e.g. content_block_delta)
  // aData contains the JSON data for this event
  Memo1.Lines.Add(aData);
```

```
}
```

Anthropic | Vision

Claude can understand and analyze images. You can send images as base64-encoded data within content blocks.

Supported Image Formats

- **image/jpeg** - JPEG images
- **image/png** - PNG images
- **image/gif** - GIF images
- **image/webp** - WebP images

Simple Example

Send an image with a prompt asking Claude to describe it.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

// Load image and encode to base64
byte[] fileBytes = System.IO.File.ReadAllBytes("photo.png");
string vBase64 = Convert.ToBase64String(fileBytes);

MessageBox.Show(Anthropic._CreateVisionMessage("claude-sonnet-4-20250514",
  "What is in this image?", vBase64, "image/png"));
```

Advanced Example

Use content blocks for more control over the image message.

```
TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
oRequest.Model = "claude-sonnet-4-20250514";
oRequest.MaxTokens = 4096;

TsgcAnthropicClass_Request_Message oMessage = new TsgcAnthropicClass_Request_Message();
oMessage.Role = "user";

// Image content block
TsgcAnthropicClass_Request_Content_Block oImageBlock = new TsgcAnthropicClass_Request_Content_Block();
oImageBlock.ContentType = "image";
oImageBlock.MediaType = "image/jpeg";
oImageBlock.Data = vBase64;

// Text content block
TsgcAnthropicClass_Request_Content_Block oTextBlock = new TsgcAnthropicClass_Request_Content_Block();
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;

var oResponse = Anthropic.CreateMessage(oRequest);
if (oResponse.Content.Length > 0)
  MessageBox.Show(oResponse.Content[0].Text);
```

Anthropic | Tool Use

Claude supports tool use (function calling), allowing you to define tools that Claude can invoke during a conversation. When Claude decides to use a tool, it returns a **tool_use** content block. You then execute the tool and send back the result as a **tool_result** content block.

Tool Use Flow

1. Define tools with name, description, and input_schema (JSON Schema).
2. Send a message with tools defined.
3. Claude responds with a **tool_use** content block (stop_reason = 'tool_use').
4. Execute the tool with the provided input.
5. Send a new message with a **tool_result** content block containing the output.
6. Claude responds with the final answer.

Example

Define a weather tool and handle the tool use loop.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

// Step 1: Create request with tools
TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
oRequest.Model = "claude-sonnet-4-20250514";
oRequest.MaxTokens = 4096;

// Define user message
TsgcAnthropicClass_Request_Message oMessage = new TsgcAnthropicClass_Request_Message();
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();
oTool.Name = "get_weather";
oTool.Description = "Get the current weather in a given location";
oTool.InputSchema =
  "{\"type\":\"object\",\"properties\":{\"location\":{\"type\":\"string\"," +
  "\"description\":\"The city and state\"}},\"required\":[\"location\"]}";
SetLength(oTools, 1);
oTools[0] = oTool;
oRequest.Tools = oTools;

// Step 2: Send request
var oResponse = Anthropic.CreateMessage(oRequest);
// 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 < oResponse.Content.Length; 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)
}
```

Anthropic | Models

List and retrieve information about available Claude models.

List Models

Lists all available Claude models.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._GetModels());
```

Get Model

Retrieves information about a specific model.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._GetModel("claude-sonnet-4-20250514"));
```

Typed Response

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

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Response_Models oModels = Anthropic.GetModels();
for (int i = 0; i < oModels.Data.Length; i++)
  MessageBox.Show(oModels.Data[i].Id + " - " + oModels.Data[i].DisplayName);
```

Anthropic | Message Batches

The Message Batches API allows you to process large volumes of messages asynchronously. This is ideal for tasks that don't require immediate responses, such as bulk content generation, data analysis, or batch processing workflows.

List Batches

Lists all message batches for your organization.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._ListBatches());
```

Get Batch

Retrieves information about a specific batch.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._GetBatch("batch_id_here"));
```

Cancel Batch

Cancels a batch that is still processing.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._CancelBatch("batch_id_here"));
```

Get Batch Results

Retrieves the results of a completed batch.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._GetBatchResults("batch_id_here"));
```

Typed Response

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

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

var oBatches = Anthropic.ListBatches();
for (int i = 0; i < oBatches.Batches.Length; i++)
  MessageBox.Show(oBatches.Batches[i].Id + " - " + oBatches.Batches[i].ProcessingStatus);
```

Anthropic | Extended Thinking

Extended thinking gives Claude the ability to think through complex problems step-by-step before providing a response. When enabled, Claude creates internal reasoning (thinking blocks) that improve the quality of responses for math, coding, analysis, and other complex tasks.

Simple Example

Send a message with extended thinking enabled using the convenience method. The temperature is automatically set to 1.0 (required by the API when thinking is enabled).

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._CreateMessageWithThinking("claude-sonnet-4-20250514",
  "How many r's are in the word strawberry?", 10000));
```

Advanced Example

Use the typed request/response classes for full control. Set ThinkingType to 'enabled' and ThinkingBudgetTokens to the desired token budget (minimum 1024). The response will contain thinking and text content blocks.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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.";
var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

TsgcAnthropicClass_Response_Messages oResponse = Anthropic.CreateMessage(oRequest);
for (int i = 0; i < oResponse.Content.Length; i++)
{
  if (oResponse.Content[i].ContentType == "thinking")
    MessageBox.Show("Thinking: " + oResponse.Content[i].Thinking);
  else if (oResponse.Content[i].ContentType == "text")
    MessageBox.Show("Response: " + oResponse.Content[i].Text);
}
```

Multi-turn with Thinking

When using extended thinking in multi-turn conversations, thinking and redacted_thinking blocks from previous responses must be passed back in the conversation. Use the ContentBlocks array to include these blocks.

```
// Pass back thinking blocks from a previous response
TsgcAnthropicClass_Request_Content_Block oBlock = new TsgcAnthropicClass_Request_Content_Block();
oBlock.ContentType = "thinking";
oBlock.Text = oPrevThinkingBlock.Thinking;   // thinking text
oBlock.Signature = oPrevThinkingBlock.Signature; // signature string

// Pass back redacted thinking blocks
TsgcAnthropicClass_Request_Content_Block oBlock2 = new TsgcAnthropicClass_Request_Content_Block();
oBlock2.ContentType = "redacted_thinking";
oBlock2.Data = oPrevRedactedBlock.Data;
```

Properties

- **ThinkingType:** Set to 'enabled' to activate extended thinking, or 'disabled' to turn it off.
- **ThinkingBudgetTokens:** The maximum number of tokens for thinking (minimum 1024). Must be less than MaxTokens.

Notes

- Temperature must be 1.0 when thinking is enabled (the convenience method sets this automatically).
- TopK cannot be used when thinking is enabled.
- BudgetTokens must be at least 1024 and less than MaxTokens.
- Response content blocks may include: thinking, redacted_thinking, and text types.

Anthropic | Documents

Claude can process PDF documents and text files sent as content blocks within messages. Documents are sent as base64-encoded data or via URL, and Claude can analyze, summarize, and answer questions about their content.

Simple Example

Send a PDF document with a question using the convenience method.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

// Load PDF file and encode to base64
string vBase64 = sgcBase64Encode(LoadFileToBytes("document.pdf"));
MessageBox.Show(Anthropic._CreateDocumentMessage("claude-sonnet-4-20250514",
  "Summarize this document.", vBase64, "application/pdf"));
```

Advanced Example with Citations

Use the typed classes to send a document with citations enabled. When citations are enabled, Claude's response will include references to specific parts of the source document.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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";
var oBlocks = oMessage.ContentBlocks;
SetLength(oBlocks, 2);
oBlocks[0] = oDocBlock;
oBlocks[1] = oTextBlock;
oMessage.ContentBlocks = oBlocks;

var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

TsgcAnthropicClass_Response_Messages oResponse = Anthropic.CreateMessage(oRequest);
if (oResponse.Content.Length > 0)
  MessageBox.Show(oResponse.Content[0].Text);
```

Properties

- **ContentType:** Set to 'document' for document content blocks.
- **SourceType:** The source type: 'base64' (default), 'text', or 'url'.
- **MediaType:** The MIME type (e.g. application/pdf, text/plain).
- **Data:** The base64-encoded document data, or URL when SourceType is 'url'.
- **Title:** Optional document title for reference.

- **Context:** Optional context or metadata about the document.
- **CitationsEnabled:** Set to True to enable source citations in the response.

Supported Formats

- **PDF:** application/pdf (max 32 MB, max 100 pages per request)
- **Plain Text:** text/plain

Anthropic | Prompt Caching

Prompt caching allows you to cache frequently used context between API calls, reducing costs by up to 90% on cache reads and improving latency. Cached content is marked with a cache_control parameter and is reused across requests within the cache TTL window.

System Prompt Caching

Cache a long system prompt to avoid re-processing it on every request. Set SystemCacheControl to True to automatically wrap the system prompt with cache_control.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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();
oMessage.Role = "user";
oMessage.Content = "Analyze this contract clause.";
var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

var oResponse = Anthropic.CreateMessage(oRequest);
// Check cache usage in response
MessageBox.Show("Cache created: " +
  oResponse.Usage.CacheCreationInputTokens.ToString());
MessageBox.Show("Cache read: " +
  oResponse.Usage.CacheReadInputTokens.ToString());
MessageBox.Show(oResponse.Content[0].Text);
```

Content Block Caching

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

```
TsgcAnthropicClass_Request_Content_Block oBlock = new TsgcAnthropicClass_Request_Content_Block();
oBlock.ContentType = "text";
oBlock.Text = "(large reference text to cache)";
oBlock.CacheControl = "ephemeral";  // Mark for caching
```

Tool Definition Caching

Cache tool definitions to avoid re-processing them on every request. This is useful when you have many tool definitions that remain constant across requests.

```
TsgcAnthropicClass_Request_Tool oTool = new TsgcAnthropicClass_Request_Tool();
oTool.Name = "search_database";
oTool.Description = "Search the database for records.";
oTool.InputSchema = "{\"type\":\"object\",\"properties\":{\"query\":{\"type\":\"string\"}}}";
oTool.CacheControl = "ephemeral";  // Cache this tool definition
```

Properties

- **SystemCacheControl:** Set to True on the request to cache the system prompt with ephemeral cache_control.
- **CacheControl:** Set to 'ephemeral' on content blocks or tool definitions to mark them for caching.
- **CacheCreationInputTokens:** (Response) Number of tokens used to create cache entries.

- **CacheReadInputTokens:** (Response) Number of tokens read from cache (cost savings).

Pricing

- **Cache writes:** 1.25x the base input token price (5-minute TTL).
- **Cache reads:** 0.1x the base input token price (90% savings).
- **Cache TTL:** 5 minutes by default. Subsequent requests refresh the TTL.

Anthropic | Citations

When documents are sent with citations enabled, Claude's response includes references back to specific parts of the source documents. This lets you verify claims and trace information to its origin.

Enabling Citations

Set CitationsEnabled to True on document content blocks. Citations must be enabled on ALL or NONE of the documents in a request.

```
TsgcAnthropicClass_Request_Content_Block oDocBlock = new TsgcAnthropicClass_Request_Content_Block();
oDocBlock.ContentType = "document";
oDocBlock.SourceType = "base64";
oDocBlock.MediaType = "application/pdf";
oDocBlock.Data = sgcBase64Encode(LoadFileToBytes("report.pdf"));
oDocBlock.CitationsEnabled = true;  // Enable citations
```

Reading Citations from Response

Text content blocks in the response may contain a Citations array with source references.

```
var oResponse = Anthropic.CreateMessage(oRequest);
for (int i = 0; i < oResponse.Content.Length; i++) {
  if (oResponse.Content[i].ContentType == "text") {
    MessageBox.Show(oResponse.Content[i].Text);

    // Process citations
    for (int j = 0; j < oResponse.Content[i].Citations.Length; j++) {
      var oCitation = oResponse.Content[i].Citations[j];
      MessageBox.Show(string.Format("  Citation [{0}]: \"{1}\"",
        oCitation.CitationType, oCitation.CitedText));

      if (oCitation.CitationType == "page_location")
        MessageBox.Show(string.Format("    Pages {0}-{1} of \"{2}\"",
          oCitation.StartPageNumber, oCitation.EndPageNumber,
          oCitation.DocumentTitle));
    }
  }
}
```

Citation Types

- **char_location:** Character-level reference in plain text documents. Fields: DocumentIndex, DocumentTitle, StartCharIndex, EndCharIndex.
- **page_location:** Page-level reference in PDF documents. Fields: DocumentIndex, DocumentTitle, StartPageNumber, EndPageNumber.
- **content_block_location:** Block-level reference in custom content documents. Fields: DocumentIndex, DocumentTitle, StartBlockIndex, EndBlockIndex.
- **web_search_result_location:** Reference to web search results. Fields: Url, Title, EncryptedIndex.

Anthropic | Web Search

The Web Search tool enables Claude to search the web for real-time information during a conversation. This is a built-in server-side tool that Anthropic hosts and executes automatically.

Simple Example

Use the convenience method to create a message with web search enabled.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
MessageBox.Show(Anthropic._CreateMessageWithWebSearch("claude-sonnet-4-20250514",
  "What are the latest news about Delphi programming?"));
```

Advanced Example

Use the typed classes for full control over the web search tool parameters such as MaxUses.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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
var 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.";
var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

TsgcAnthropicClass_Response_Messages oResponse = Anthropic.CreateMessage(oRequest);
for (int i = 0; i < oResponse.Content.Length; i++)
{
  if (oResponse.Content[i].ContentType == "text")
    MessageBox.Show(oResponse.Content[i].Text);
}
```

Built-in Tool Types

- **web_search_20250305:** Server-side web search. Properties: Name ('web_search'), MaxUses.
- **code_execution_20250825:** Sandboxed code execution. Properties: Name ('code_execution'), MaxUses.
- **computer_20250124:** Computer use (mouse, keyboard, screenshots). Properties: Name ('computer'), DisplayWidthPx, DisplayHeightPx.

Response Content Types

When built-in tools are used, the response may contain additional content block types:

- **server_tool_use:** Indicates Claude invoked a built-in tool. Fields: Id, Name, Input.
- **web_search_tool_result:** Contains web search results. Fields: ToolUseId, Data (raw JSON content).

Anthropic | Structured Outputs

Structured Outputs force Claude to return responses conforming to a JSON schema. This guarantees valid, parseable JSON output matching your schema definition.

Simple Example

Use the convenience method to create a message with JSON schema output.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

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

MessageBox.Show(Anthropic._CreateMessageJSON("claude-sonnet-4-20250514",
  "Extract the name and age: John is 30 years old.", vSchema));
```

Advanced Example

Use the typed classes to combine structured output with the effort parameter.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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!";
var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

TsgcAnthropicClass_Response_Messages oResponse = Anthropic.CreateMessage(oRequest);
for (int i = 0; i < oResponse.Content.Length; i++)
{
  if (oResponse.Content[i].ContentType == "text")
    MessageBox.Show(oResponse.Content[i].Text);
}
```

Strict Tool Use

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

```
TsgcAnthropicClass_Request_Tool oTool = new TsgcAnthropicClass_Request_Tool();
oTool.Name = "get_weather";
oTool.Description = "Get the current weather for a location";
oTool.Strict = true;  // Guarantee schema conformance
oTool.InputSchema = "{\"type\":\"object\",\"properties\":{\"location\":" +
  "{\"type\":\"string\"}},\"required\":[\"location\"],\"additionalProperties\":false}";
```

Properties

- **OutputFormatSchema:** A JSON Schema string. Claude's response text will be valid JSON conforming to this schema.
- **Effort:** Controls output quality vs. cost. Values: 'low', 'medium', 'high' (default), 'max' (Opus 4.6 only).
- **Strict:** (on tools) When True, tool inputs are guaranteed to conform to the input_schema.

Anthropic | Files API

The Files API allows you to upload, list, retrieve, download, and delete files. Uploaded files can be referenced in messages using document content blocks with file source type.
Note: The Files API requires the beta header `files-api-2025-04-14`. Set this in the **AnthropicOptions.BetaHeaders** property.

Upload a File

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
Anthropic.AnthropicOptions.BetaHeaders = "files-api-2025-04-14";

TsgcAnthropicClass_Response_File oFile = Anthropic.UploadFile("C:\\documents\\report.pdf");
MessageBox.Show("File ID: " + oFile.Id);
MessageBox.Show("Filename: " + oFile.Filename);
MessageBox.Show("Size: " + oFile.SizeBytes.ToString() + " bytes");
MessageBox.Show("MIME Type: " + oFile.MimeType);
```

List Files

```
TsgcAnthropicClass_Response_FileList oList = Anthropic.ListFiles(50);
for (int i = 0; i < oList.Data.Length; i++)
  MessageBox.Show(oList.Data[i].Id + " - " + oList.Data[i].Filename +
    " (" + oList.Data[i].SizeBytes.ToString() + " bytes)");
MessageBox.Show("Has more: " + oList.HasMore.ToString());
```

Use File in Messages

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

```
TsgcAnthropicClass_Request_Content_Block oDocBlock = new TsgcAnthropicClass_Request_Content_Block();
oDocBlock.ContentType = "document";
oDocBlock.SourceType = "file";
oDocBlock.FileId = "file_abc123";  // ID from UploadFile

TsgcAnthropicClass_Request_Content_Block oTextBlock = new TsgcAnthropicClass_Request_Content_Block();
oTextBlock.ContentType = "text";
oTextBlock.Text = "Summarize this document.";
```

Delete a File

```
TsgcAnthropicClass_Response_File oDeleted = Anthropic.DeleteFile("file_abc123");
MessageBox.Show("Deleted: " + oDeleted.Id);
```

API Methods

- **UploadFile:** Uploads a file from a local path. Returns file metadata (Id, Filename, MimeType, SizeBytes).
- **ListFiles:** Lists all uploaded files with pagination support (Limit, AfterId, BeforeId).
- **GetFile:** Retrieves metadata for a specific file by ID.
- **DownloadFile:** Downloads the content of a file (only files created by code execution tool are downloadable).
- **DeleteFile:** Permanently deletes a file by ID.

Anthropic | MCP Connector

The MCP (Model Context Protocol) connector allows Claude to access tools from external MCP servers. This enables integration with third-party services and custom tool providers.
Note: MCP connector requires the beta header `mcp-client-2025-11-20`. Set this in the **AnthropicOptions.BetaHeaders** property.

Simple Example

Use the convenience method to create a message with an MCP server.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
Anthropic.AnthropicOptions.BetaHeaders = "mcp-client-2025-11-20";

MessageBox.Show(Anthropic._CreateMessageWithMCP("claude-sonnet-4-20250514",
  "What tools are available?",
  "https://my-mcp-server.example.com/sse",
  "my-mcp-server"));
```

Advanced Example

Use the typed classes for full control over MCP server configuration, including authentication.

```
TsgcHTTP_API_Anthropic Anthropic = new TsgcHTTP_API_Anthropic();
Anthropic.AnthropicOptions.ApiKey = "API_KEY";
Anthropic.AnthropicOptions.BetaHeaders = "mcp-client-2025-11-20";

TsgcAnthropicClass_Request_Messages oRequest = new TsgcAnthropicClass_Request_Messages();
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
var 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";
var 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.";
var oMessages = oRequest.Messages;
SetLength(oMessages, 1);
oMessages[0] = oMessage;
oRequest.Messages = oMessages;

TsgcAnthropicClass_Response_Messages oResponse = Anthropic.CreateMessage(oRequest);
for (int i = 0; i < oResponse.Content.Length; i++)
{
  if (oResponse.Content[i].ContentType == "text")
    MessageBox.Show(oResponse.Content[i].Text);
  else if (oResponse.Content[i].ContentType == "mcp_tool_use")
    MessageBox.Show("MCP tool call: " + oResponse.Content[i].Name +
      " on " + oResponse.Content[i].ServerName);
}
```

MCP Server Properties

- **ServerType:** Currently only 'url' is supported.
- **Url:** The HTTPS URL of the MCP server.
- **Name:** Unique identifier for this server. Must match the MCPServerName in the toolset.
- **AuthorizationToken:** Optional OAuth Bearer token for authenticated servers.

Response Content Types

- **mcp_tool_use:** Claude invoked an MCP tool. Fields: Id, Name, ServerName, Input.
- **mcp_tool_result:** Result from an MCP tool execution. Fields: ToolUseId, IsError, Data (content).

Gemini

Google Gemini is a family of multimodal AI models developed by Google DeepMind. Gemini models support text generation, vision, structured outputs, embeddings, and tool use, offering powerful capabilities for building AI-powered applications.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Gemini** to interact with the Gemini API.

Gemini API

The Gemini API provides access to Google Gemini models for building AI-powered applications. The API supports content generation, vision (image understanding), structured JSON outputs, streaming, token counting, embeddings, tool use (function calling), and model listing.

Features

- **Messages**
  - **Gemini Messages Examples**
- **Vision**
  - **Gemini Vision Examples**
- **Models**
  - **Gemini Models Examples**
- **Structured Outputs**
  - **Gemini Structured Outputs Examples**
- **Token Counting**
  - **Gemini Token Counting Examples**
- **Embeddings**
  - **Gemini Embeddings Examples**
- **Tool Use**
  - **Gemini Tool Use Examples**

Configuration

The Gemini API uses API keys for authentication. Visit your API Keys page in Google AI Studio to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **GeminiOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "YOUR_API_KEY";
```

Messages

Send content to a Gemini model and receive generated responses. The model generates the next message based on the provided input.

- **_CreateContent:** Creates content with the specified model and user prompt.
  - **Model:** The model to use (e.g. gemini-2.0-flash).
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentWithSystem:** Creates content with a system instruction.
  - **Model:** The model to use.
  - **System:** System instruction that sets the behavior of the model.
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentStream:** Creates content with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

Gemini models can understand images passed as base64-encoded content along with text prompts.

- **_CreateVisionContent:** Sends an image with a text prompt.
  - **Model:** The model to use.
  - **Prompt:** The text prompt to accompany the image.
  - **Base64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Structured Outputs

Generate structured JSON output from a Gemini model by providing a JSON schema that defines the expected response format.

- **_CreateContentJSON:** Creates content with structured JSON output.
  - **Model:** The model to use.
  - **Message:** The user message content.
  - **Schema:** A JSON schema defining the expected output structure.
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Models

List and retrieve details about available Gemini models.

- **_GetModels:** Lists all available models.
- **_GetModel:** Gets details for a specific model.
  - **ModelId:** The identifier of the model to retrieve.

Token Counting

Count the number of tokens in a message before sending it to a model.

- **_CountTokens:** Counts tokens for a message.
  - **Model:** The model to use for tokenization.
  - **Message:** The text content to count tokens for.

Embeddings

Generate vector embeddings for text content using Gemini models.

- **_EmbedContent:** Generates embeddings for text.
  - **Model:** The model to use for embedding generation.
  - **Text:** The text content to generate embeddings for.

DeepSeek

DeepSeek is a Chinese AI company focused on building powerful open-source language models. Their models excel at coding, reasoning, and general-purpose tasks, offering strong performance at competitive pricing.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_DeepSeek** to interact with the DeepSeek API.

DeepSeek API

The DeepSeek API provides access to DeepSeek models for building AI-powered applications. The API supports text generation, vision (image understanding), streaming, and model listing. The API follows an OpenAI-compatible format, making it easy to integrate.

Features

- **Messages**
  - **DeepSeek Messages Examples**
- **Vision**
  - **DeepSeek Vision Examples**
- **Models**
  - **DeepSeek Models Examples**

Configuration

The DeepSeek API uses API keys for authentication. Visit your API Keys page in the DeepSeek Platform to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **DeepSeekOptions.ApiKey** property of the component.

```
TsgcHTTP_API_DeepSeek DeepSeek = new TsgcHTTP_API_DeepSeek();
DeepSeek.DeepSeekOptions.ApiKey = "YOUR_API_KEY";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. deepseek-chat).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

DeepSeek can understand images passed as base64-encoded content within messages.

- **_CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **Prompt:** The text prompt to accompany the image.

Models

List the available DeepSeek models.

- **_GetModels:** Lists all available models.

DeepSeek | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to DeepSeek.

```
TsgcHTTP_API_DeepSeek DeepSeek = new TsgcHTTP_API_DeepSeek();
DeepSeek.DeepSeekOptions.ApiKey = "API_KEY";
MessageBox.Show(DeepSeek._CreateMessage("deepseek-chat", "Hello!"));
```

System Prompt Example

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

```
TsgcHTTP_API_DeepSeek DeepSeek = new TsgcHTTP_API_DeepSeek();
DeepSeek.DeepSeekOptions.ApiKey = "API_KEY";
MessageBox.Show(DeepSeek._CreateMessageWithSystem("deepseek-chat",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

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

```
TsgcHTTP_API_DeepSeek DeepSeek = new TsgcHTTP_API_DeepSeek();
DeepSeek.DeepSeekOptions.ApiKey = "API_KEY";
DeepSeek.OnHTTPAPISSE += OnSSEEvent;
DeepSeek._CreateMessageStream("deepseek-chat", "Tell me a story.");

void OnSSEEvent(object Sender, string aEvent, string aData, ref bool Cancel)
{
  // aEvent contains the event type
  // aData contains the JSON data for this event
  Memo1.Lines.Add(aData);
}
```

DeepSeek | Vision

DeepSeek can understand and analyze images. You can send images as base64-encoded data within content blocks.

Vision Example

Send an image with a prompt asking DeepSeek to describe it.

```
TsgcHTTP_API_DeepSeek DeepSeek = new TsgcHTTP_API_DeepSeek();
DeepSeek.DeepSeekOptions.ApiKey = "API_KEY";

// Load image and encode to base64
byte[] fileBytes = System.IO.File.ReadAllBytes("photo.jpg");
string vImageBase64 = Convert.ToBase64String(fileBytes);

MessageBox.Show(DeepSeek._CreateVisionMessage("deepseek-chat",
  "Describe this image", vImageBase64, "image/jpeg"));
```

DeepSeek | Models

List the available DeepSeek models.

List Models

Lists all available DeepSeek models.

```
TsgcHTTP_API_DeepSeek DeepSeek = new TsgcHTTP_API_DeepSeek();
DeepSeek.DeepSeekOptions.ApiKey = "API_KEY";
MessageBox.Show(DeepSeek._GetModels());
```

Gemini

Google Gemini is a family of multimodal AI models developed by Google DeepMind. Gemini models support text generation, vision, structured outputs, embeddings, and tool use, offering powerful capabilities for building AI-powered applications.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Gemini** to interact with the Gemini API.

Gemini API

The Gemini API provides access to Google Gemini models for building AI-powered applications. The API supports content generation, vision (image understanding), structured JSON outputs, streaming, token counting, embeddings, tool use (function calling), and model listing.

Features

- **Messages**
  - **Gemini Messages Examples**
- **Vision**
  - **Gemini Vision Examples**
- **Models**
  - **Gemini Models Examples**
- **Structured Outputs**
  - **Gemini Structured Outputs Examples**
- **Token Counting**
  - **Gemini Token Counting Examples**
- **Embeddings**
  - **Gemini Embeddings Examples**
- **Tool Use**
  - **Gemini Tool Use Examples**

Configuration

The Gemini API uses API keys for authentication. Visit your API Keys page in Google AI Studio to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **GeminiOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "YOUR_API_KEY";
```

Messages

Send content to a Gemini model and receive generated responses. The model generates the next message based on the provided input.

- **_CreateContent:** Creates content with the specified model and user prompt.
  - **Model:** The model to use (e.g. gemini-2.0-flash).
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentWithSystem:** Creates content with a system instruction.
  - **Model:** The model to use.
  - **System:** System instruction that sets the behavior of the model.
  - **Message:** The user message content.
  - **MaxOutputTokens:** Maximum number of tokens to generate.
- **_CreateContentStream:** Creates content with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

Gemini models can understand images passed as base64-encoded content along with text prompts.

- **_CreateVisionContent:** Sends an image with a text prompt.
  - **Model:** The model to use.
  - **Prompt:** The text prompt to accompany the image.
  - **Base64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Structured Outputs

Generate structured JSON output from a Gemini model by providing a JSON schema that defines the expected response format.

- **_CreateContentJSON:** Creates content with structured JSON output.
  - **Model:** The model to use.
  - **Message:** The user message content.
  - **Schema:** A JSON schema defining the expected output structure.
  - **MaxOutputTokens:** Maximum number of tokens to generate.

Models

List and retrieve details about available Gemini models.

- **_GetModels:** Lists all available models.
- **_GetModel:** Gets details for a specific model.
  - **ModelId:** The identifier of the model to retrieve.

Token Counting

Count the number of tokens in a message before sending it to a model.

- **_CountTokens:** Counts tokens for a message.
  - **Model:** The model to use for tokenization.
  - **Message:** The text content to count tokens for.

Embeddings

Generate vector embeddings for text content using Gemini models.

- **_EmbedContent:** Generates embeddings for text.
  - **Model:** The model to use for embedding generation.
  - **Text:** The text content to generate embeddings for.

Gemini | Messages

Send a prompt to a Gemini model and receive generated content. The Gemini API uses the generateContent endpoint to process text input and return model responses.

Simple Example

Send a Hello message to Gemini.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
MessageBox.Show(Gemini._CreateContent("gemini-2.0-flash", "Hello!"));
```

System Instruction Example

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

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
MessageBox.Show(Gemini._CreateContentWithSystem("gemini-2.0-flash",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

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

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
Gemini.OnHTTPAPISSE += OnSSEEvent;
Gemini._CreateContentStream("gemini-2.0-flash", "Tell me a story.");

void OnSSEEvent(object Sender, string aEvent, string aData, ref bool Cancel)
{
  Memo1.Lines.Add(aData);
}
```

Advanced Example

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

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

TsgcGeminiClass_Request_GenerateContent oRequest = new TsgcGeminiClass_Request_GenerateContent();
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;

var oResponse = Gemini.CreateContent(oRequest);
if (oResponse.Candidates.Length > 0)
  if (oResponse.Candidates[0].Parts.Length > 0)
```

```
    MessageBox.Show(oResponse.Candidates[0].Parts[0].Text);
```

Gemini | Vision

Gemini can understand and analyze images. You can send images as base64-encoded data within content parts.

Vision Example

Send a base64-encoded image to Gemini for analysis.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

// Load image and encode to base64
byte[] fileBytes = System.IO.File.ReadAllBytes("photo.jpg");
string vImageBase64 = Convert.ToBase64String(fileBytes);

MessageBox.Show(Gemini._CreateVisionContent("gemini-2.0-flash",
  "Describe this image", vImageBase64, "image/jpeg"));
```

Gemini | Models

List and retrieve information about available Gemini models.

List Models

Lists all available models.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
MessageBox.Show(Gemini._GetModels());
```

Get Model

Get details for a specific model.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
MessageBox.Show(Gemini._GetModel("gemini-2.0-flash"));
```

Advanced Example

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

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

TsgcGeminiClass_Response_Models oModels = Gemini.GetModels();
for (int i = 0; i < oModels.Models.Length; i++)
  MessageBox.Show(oModels.Models[i].DisplayName + " - " + oModels.Models[i].Description);
```

Gemini | Structured Outputs

Gemini can return responses in structured JSON format conforming to a schema you define. This guarantees valid, parseable JSON output matching your schema definition.

Simple Example

Use the convenience method to create content with JSON schema output.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

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

MessageBox.Show(Gemini._CreateContentJSON("gemini-2.0-flash",
  "Extract the name and age: John is 30 years old.", vSchema));
```

Advanced Example

Use the typed classes to configure structured output with ResponseMimeType and ResponseSchema.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

TsgcGeminiClass_Request_GenerateContent oRequest = new TsgcGeminiClass_Request_GenerateContent();
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;

var oResponse = Gemini.CreateContent(oRequest);
if (oResponse.Candidates.Length > 0)
  if (oResponse.Candidates[0].Parts.Length > 0)
    MessageBox.Show(oResponse.Candidates[0].Parts[0].Text);
```

Properties

- **ResponseMimeType:** Set to 'application/json' to enable structured JSON output.
- **ResponseSchema:** A JSON Schema string defining the expected output structure.

Gemini | Token Counting

Count the number of tokens in a message before sending it to the API. This helps estimate costs and ensure messages fit within model limits.

Simple Example

Count tokens for a message.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
MessageBox.Show(Gemini._CountTokens("gemini-2.0-flash", "Hello, how are you?"));
```

Advanced Example

Use the typed request/response classes for full control over token counting parameters.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

TsgcGeminiClass_Request_CountTokens oRequest = new TsgcGeminiClass_Request_CountTokens();
oRequest.Model = "gemini-2.0-flash";
var oContents = new TsgcGeminiClass_Request_Content[1];
oContents[0] = new TsgcGeminiClass_Request_Content();
oContents[0].Role = "user";
var oParts = new TsgcGeminiClass_Request_Part[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);
MessageBox.Show("Total tokens: " + oResponse.TotalTokens.ToString());
```

Gemini | Embeddings

Generate vector embeddings for text content. Embeddings capture the semantic meaning of text and can be used for similarity search, clustering, and other NLP tasks.

Simple Example

Generate embeddings for text.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";
MessageBox.Show(Gemini._EmbedContent("text-embedding-004", "Hello world"));
```

Advanced Example

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

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

TsgcGeminiClass_Response_Embedding oEmbedding = Gemini.EmbedContent("text-embedding-004", "Hello world");
MessageBox.Show("Dimensions: " + oEmbedding.Values.Length.ToString());
for (int i = 0; i < oEmbedding.Values.Length; i++)
  MessageBox.Show(oEmbedding.Values[i].ToString());
```

Gemini | Tool Use

Gemini supports function calling (tool use), allowing you to define functions that the model can invoke during a conversation. When Gemini decides to call a function, it returns a functionCall part in its response. You then execute the function and send the result back.

Function Calling Flow

1. Define function declarations with name, description, and parameters (JSON Schema).
2. Send a content generation request with function declarations.
3. Gemini responds with a **functionCall** part containing the function name and arguments.
4. Execute the function with the provided arguments.
5. Send a new request with a **functionResponse** part containing the result.
6. Gemini responds with the final answer.

Example

Define a weather function and handle function calling.

```
TsgcHTTP_API_Gemini Gemini = new TsgcHTTP_API_Gemini();
Gemini.GeminiOptions.ApiKey = "API_KEY";

// Step 1: Create request with function declarations
TsgcGeminiClass_Request_GenerateContent oRequest = new TsgcGeminiClass_Request_GenerateContent();
oRequest.Model = "gemini-2.0-flash";
oRequest.MaxOutputTokens = 4096;

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

// Define function
SetLength(oFunctions, 1);
oFunctions[0] = new TsgcGeminiClass_Request_FunctionDeclaration();
oFunctions[0].Name = "get_weather";
oFunctions[0].Description = "Get the current weather in a given location";
oFunctions[0].Parameters =
  "{\"type\":\"object\",\"properties\":{\"location\":{\"type\":\"string\"," +
  "\"description\":\"The city and state\"}},\"required\":[\"location\"]}";
oRequest.FunctionDeclarations = oFunctions;

// Step 2: Send request
var oResponse = Gemini.CreateContent(oRequest);
// Step 3: Check for function call
if (oResponse.Candidates.Length > 0) {
  if (oResponse.Candidates[0].Parts.Length > 0) {
    if (oResponse.Candidates[0].Parts[0].FunctionCallName != "") {
      var vFunctionName = oResponse.Candidates[0].Parts[0].FunctionCallName;
      var vFunctionArgs = oResponse.Candidates[0].Parts[0].FunctionCallArgs;
      // Step 4: Execute function and get result
      var vResult = "72 degrees and sunny";
    }
  }
}
```

Properties

- **FunctionDeclarations:** Array of TsgcGeminiClass_Request_FunctionDeclaration defining available functions.
  - **Name:** The function name.
  - **Description:** A description of what the function does.
  - **Parameters:** JSON Schema string defining the function parameters.

- **ToolChoice:** Controls function calling behavior. Set the mode (e.g. 'AUTO', 'ANY', 'NONE').

Ollama

Ollama is an open-source tool for running large language models locally. It supports a wide range of models including Llama, Mistral, Gemma, Phi, and many others, enabling local AI inference without requiring cloud API access. The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Ollama** to interact with the Ollama API.

Ollama API

The Ollama API provides access to locally running models for AI-powered applications. The API supports text generation, streaming, model management, and embeddings. No API key is required by default since models run locally. The host and port are configurable.

Features

- **Messages**
  - **Ollama Messages Examples**
- **Models**
  - **Ollama Models Examples**
- **Embeddings**
  - **Ollama Embeddings Examples**

Configuration

Ollama runs locally and by default listens on **http://localhost:11434**. Configure the host in the **OllamaOptions.Host** property. An API key is optional and only needed if you have configured authentication on your Ollama instance.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. llama3).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Models

Manage and query locally available models.

- **_GetModels:** Lists all available models.
- **_GetTags:** Lists all locally available model tags.
- **_ShowModel:** Retrieves detailed information about a specific model.
  - **Model:** The name of the model to query.
- **_PullModel:** Downloads a model from the Ollama model library.
  - **Model:** The name of the model to pull.
- **_DeleteModel:** Deletes a locally available model.
  - **Model:** The name of the model to delete.

Embeddings

Generate vector embeddings from text input using locally running models.

- **_CreateEmbeddings:** Generates embeddings for the given text.
  - **Model:** The model to use for embeddings (e.g. llama3).
  - **Input:** The text to generate embeddings for.

Ollama | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to a local Ollama model.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._CreateMessage("llama3", "Hello!"));
```

System Prompt Example

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

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._CreateMessageWithSystem("llama3",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

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

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

void OnSSEEvent(object Sender, string aEvent, string aData, ref bool Cancel)
{
  // aEvent contains the event type
  // aData contains the JSON data for this event
  Memo1.Lines.Add(aData);
}
```

Ollama | Models

Manage and query locally available Ollama models.

List Models

Lists all available models.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._GetModels());
```

Get Tags

Lists all locally available model tags.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._GetTags());
```

Show Model

Retrieves detailed information about a specific model.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._ShowModel("llama3"));
```

Pull Model

Downloads a model from the Ollama model library.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._PullModel("llama3"));
```

Ollama | Embeddings

Generate vector embeddings from text input using locally running Ollama models. Embeddings can be used for semantic search, clustering, and other NLP tasks.

Create Embeddings

Generates embeddings for the given text input.

```
TsgcHTTP_API_Ollama Ollama = new TsgcHTTP_API_Ollama();
Ollama.OllamaOptions.Host = "http://localhost:11434";
MessageBox.Show(Ollama._CreateEmbeddings("llama3", "Hello world"));
```

Grok

Grok is a conversational AI assistant developed by xAI, designed to provide helpful and informative responses with real-time knowledge. Grok models are known for their strong reasoning capabilities and up-to-date information access.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Grok** to interact with the Grok API.

Grok API

The Grok API provides access to Grok models for building AI-powered applications. The API supports text generation, vision (image understanding), streaming, and model listing.

Features

- **Messages**
  - **Grok Messages Examples**
- **Vision**
  - **Grok Vision Examples**
- **Models**
  - **Grok Models Examples**

Configuration

The Grok API uses API keys for authentication. Visit your xAI Console to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **GrokOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Grok Grok = new TsgcHTTP_API_Grok();
Grok.GrokOptions.ApiKey = "YOUR_API_KEY";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. grok-3).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.

Vision

Grok can understand images passed as base64-encoded content within messages.

- **_CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).
  - **Prompt:** The text prompt to accompany the image.

Models

List the available Grok models.

- **_GetModels:** Lists all available models.

Grok | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to Grok.

```
TsgcHTTP_API_Grok Grok = new TsgcHTTP_API_Grok();
Grok.GrokOptions.ApiKey = "API_KEY";
MessageBox.Show(Grok._CreateMessage("grok-3", "Hello!"));
```

System Prompt Example

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

```
TsgcHTTP_API_Grok Grok = new TsgcHTTP_API_Grok();
Grok.GrokOptions.ApiKey = "API_KEY";
MessageBox.Show(Grok._CreateMessageWithSystem("grok-3",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

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

```
TsgcHTTP_API_Grok Grok = new TsgcHTTP_API_Grok();
Grok.GrokOptions.ApiKey = "API_KEY";
Grok.OnHTTPAPISSE += OnSSEEvent;
Grok._CreateMessageStream("grok-3", "Tell me a story.");

void OnSSEEvent(object Sender, string aEvent, string aData, ref bool Cancel)
{
  Memo1.Lines.Add(aData);
}
```

Grok | Vision

Grok can understand images passed as base64-encoded content within messages. Use the vision endpoint to send an image along with a text prompt and receive a description or analysis.

Vision Example

Send a base64-encoded image to Grok for analysis.

```
TsgcHTTP_API_Grok Grok = new TsgcHTTP_API_Grok();
Grok.GrokOptions.ApiKey = "API_KEY";
MessageBox.Show(Grok._CreateVisionMessage("grok-2-vision",
  "Describe this image.", vBase64, "image/jpeg"));
```

Grok | Models

List the available Grok models and their basic information.

List Models Example

Retrieve the list of all available Grok models.

```
TsgcHTTP_API_Grok Grok = new TsgcHTTP_API_Grok();
Grok.GrokOptions.ApiKey = "API_KEY";
MessageBox.Show(Grok._GetModels);
```

Mistral AI

Mistral AI is a French artificial intelligence company that develops efficient and powerful language models. Their models are known for strong performance across reasoning, coding, and multilingual tasks, while maintaining competitive efficiency.
The sgcWebSockets library provides a Delphi component **TsgcHTTP_API_Mistral** to interact with the Mistral AI API.

Mistral API

The Mistral API provides access to Mistral models for building AI-powered applications. The API supports text generation, vision (image understanding), streaming, embeddings, JSON mode, and model listing.

Features

- **Messages**
  - **Mistral Messages Examples**
- **Vision**
  - **Mistral Vision Examples**
- **Models**
  - **Mistral Models Examples**
- **Embeddings**
  - **Mistral Embeddings Examples**

Configuration

The Mistral API uses API keys for authentication. Visit your API Keys page in the Mistral Console to retrieve the API key you'll use in your requests.
Remember that your API key is a secret! Do not share it with others or expose it in any client-side code.
This **API Key** must be configured in the **MistralOptions.ApiKey** property of the component.

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "YOUR_API_KEY";
```

Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

- **_CreateMessage:** Creates a message with the specified model and user prompt.
  - **Model:** The model to use (e.g. mistral-large-latest).
  - **Message:** The user message content.
- **_CreateMessageWithSystem:** Creates a message with a system prompt.
  - **System:** System prompt that sets the behavior of the assistant.
- **_CreateMessageStream:** Creates a message with streaming (SSE) enabled. Events are delivered through the OnHTTPAPISSE event handler.
- **_CreateMessageJSON:** Creates a message with JSON mode enabled. The model will return a valid JSON response.
  - **Model:** The model to use.
  - **Message:** The user message content.

Vision

Mistral can understand images passed as base64-encoded content within messages.

- **_CreateVisionMessage:** Sends an image with a text prompt.
  - **ImageBase64:** The base64-encoded image data.
  - **MediaType:** The MIME type (image/jpeg, image/png, image/gif, image/webp).

- **Prompt:** The text prompt to accompany the image.

Models

List the available Mistral models.

- **_GetModels:** Lists all available models.

Embeddings

Get a vector representation of a given input that can be used for semantic search, clustering, and other machine learning tasks.

- **_CreateEmbeddings:** Creates an embedding vector representing the input text.
  - **Model:** The model to use (e.g. mistral-embed).
  - **Input:** Input text to get embeddings for.

Mistral | Messages

Send a structured list of input messages with text content, and the model will generate the next message in the conversation.

Simple Example

Send a Hello message to Mistral.

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
MessageBox.Show(Mistral._CreateMessage("mistral-large-latest", "Hello!"));
```

System Prompt Example

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

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
MessageBox.Show(Mistral._CreateMessageWithSystem("mistral-large-latest",
  "You are a helpful assistant that responds in Spanish.",
  "What is the capital of France?"));
```

Streaming Example

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

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
Mistral.OnHTTPAPISSE += OnSSEEvent;
Mistral._CreateMessageStream("mistral-large-latest", "Tell me a story.");

void OnSSEEvent(object Sender, string aEvent, string aData, ref bool Cancel)
{
  Memo1.Lines.Add(aData);
}
```

JSON Mode Example

Request a JSON-formatted response from Mistral.

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
MessageBox.Show(Mistral._CreateMessageJSON("mistral-large-latest",
  "List 3 colors as JSON"));
```

Mistral | Vision

Mistral can understand images passed as base64-encoded content within messages. Use the vision endpoint to send an image along with a text prompt and receive a description or analysis.

Vision Example

Send a base64-encoded image to Mistral for analysis.

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
MessageBox.Show(Mistral._CreateVisionMessage("mistral-large-latest",
  "Describe this image.", vBase64, "image/jpeg"));
```

Mistral | Models

List the available Mistral models and their basic information.

List Models Example

Retrieve the list of all available Mistral models.

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
MessageBox.Show(Mistral._GetModels);
```

Mistral | Embeddings

Get a vector representation of a given input that can be used for semantic search, clustering, and other machine learning tasks.

Embeddings Example

Create an embedding vector for a text input.

```
TsgcHTTP_API_Mistral Mistral = new TsgcHTTP_API_Mistral();
Mistral.MistralOptions.ApiKey = "API_KEY";
MessageBox.Show(Mistral._CreateEmbeddings("mistral-embed", "Hello world"));
```

IoT

The Internet of things (IoT) refers to the concept of extending Internet connectivity beyond conventional computing platforms such as personal computers and mobile devices, and into any range of traditionally "dumb" or non-internet-enabled physical devices and everyday objects. Embedded with electronics, Internet connectivity, and other forms of hardware (such as sensors), these devices can communicate and interact with others over the Internet, and they can be remotely monitored and controlled.

sgcWebSockets package implements the following IoT clients:

1. Amazon AWS IoT: AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. This enables you to collect telemetry data from multiple devices, and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

2. Azure IoT Hub: IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and secure communications between millions of IoT devices and a cloud-hosted solution backend.

IoT Amazon MQTT Client

What Is AWS IoT?

AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. This enables you to collect telemetry data from multiple devices, and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

Message broker

Provides a secure mechanism for devices and AWS IoT applications to publish and receive messages from each other. You can use either the MQTT protocol directly or MQTT over WebSocket to publish and subscribe.

The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like Sensor/temp/room1.

The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing. The act of registering to receive messages for a topic filter is referred to as subscribing.

The topic namespace is isolated for each AWS account and region pair. For example, the Sensor/temp/room1 topic for an AWS account is independent from the Sensor/temp/room1 topic for another AWS account. This is true of regions, too. The Sensor/temp/room1 topic in the same AWS account in us-east-1 is independent from the same topic in us-east-2. AWS IoT does not support sending and receiving messages across AWS accounts and regions.

The message broker maintains a list of all client sessions and the subscriptions for each session. When a message is published on a topic, the broker checks for sessions with subscriptions that map to the topic. The broker then forwards the publish message to all sessions that have a currently connected client.

MQTT Client

TsgcIoTAmazon_MQTT_Client is the component used to connect to AWS IoT. One client can connect to only one device. The client connects using the plain MQTT protocol and authenticates using an X.509 Client Certificate.

In order to connect to AWS IoT, the client needs the following properties:

Amazon.ClientId: identification of client, optional.
Amazon.Endpoint: server name where MQTT client will connect.
Amazon.Port: by default uses port 8883. If port is 443, uses ALPN automatically to connect (Requires custom Indy version).

AWS IoT Core supports devices and clients that use the MQTT and the MQTT over WebSocket Secure (WSS) protocols to publish and subscribe to messages. The following table lists the protocols that the AWS IoT device endpoints support and the authentication methods and ports they use.

Protocol	Authentication	Port	ALPN Protocol Name
MQTT over WebSocket	Signature Version 4	443	
MQTT over WebSocket	Custom Authentication	443	
MQTT	X.509 client certificate	443	x-amzn-mqtt-ca

MQTT	X.509 client certificate	8883	
MQTT	Custom Authentication	443	mqtt

Certificates Authentication

You need to create certificates in your Amazon AWS console and set the path where they are stored.

Using **OpenSSL** as IOHandler you must set the certificate in the following paths

Certificate.Enabled: set to True if you want to use certificates.
Certificate.CertFile: path to X.509 client certificate.
Certificate.KeyFile: path to X.509 client key file.

Using **SChannel** as IOHandler, first convert the PEM Certificate + Key to a PFX certificate. This requires OpenSSL binaries:

```
openssl pkcs12 -inkey 884ccf73ff-private.pem.key -in 884ccf73ff-certificate.pem.crt -export -out 884ccf73ff-certi
```

Then set the following paths (there is no need to set the key file because it is already included in the certificate).

Certificate.Enabled: set to True if you want to use certificates.
Certificate.CertFile: path to PFX certificate

SignatureV4 Authentication

You need to create a user in your Amazon AWS console and save the Access and Secret keys, which will be used to sign the WebSocket request.

SignatureV4.Enabled: set to True if you want to use this type of authentication.
SignatureV4.Region: the region where your device is located (example: us-east-1).
SignatureV4.AccessKey: the access key created in your Amazon console or obtained as a temporary credential.
SignatureV4.SecretKey: the secret key created in your amazon console or get as temporary credential
SignatureV4.SessionToken: (conditional) if you are using Temporary Security Credentials, set here the security token.
OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used.
  - **osIAPI_1_0:** uses API 1.0 OpenSSL, it's latest supported by Indy
  - **osIAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **osIAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where are located the openSSL libraries
  - **osIpNone:** this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
  - **osIpDefaultFolder:** sets automatically the openSSL path where the libraries should be located for all IDE personalities.
  - **osIpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = osIpCustomFolder define here the full path where are located the openSSL libraries.
- **UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
  - **osIsSymLinksDefault:** by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).

osIsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.
osIsSymLinksLoad: Load SymLinks after trying to load the version libraries.
osIsSymLinksDontLoad: don't load the SymLinks.

**SignatureV4 requires Indy 10.5.7+*

Custom Authentication

Custom authentication enables you to define how to authenticate and authorize clients by using authorizer resources. The device passes credentials in either the request's header fields or query parameters (for MQTT over WebSockets protocols) or in the user name and password field of the MQTT CONNECT message (for the MQTT and MQTT over WebSockets protocols).

CustomAuthentication.Enabled: set to True if you want to use this type of Authentication.
CustomAuthentication.Parameters: set here the query parameters which will be passed to the server (by default is /mqtt)
CustomAuthentication.Headers: here you can put the custom header fields.
CustomAuthentication.WebSockets: if set to true, the connection will work over WebSocket protocol, otherwise will work over plain TCP.

MQTTAuthentication.Enabled: if you need to pass the username/password in the mqtt connection, enable this property
MQTTAuthentication.Username: username of the mqtt connection
M**QTTAuthentication.Password:** secret of the mqtt connection.

Client can send optionally a ClientId to identifiy client connection, then other clients can subscribe to receive a notification every time this client has connected, subscribed, disconnected...

Authorization

If you can't connect using port 8883 and use TCP as transport (which is the default), amazon takes "AWS IoT Core policy" to provide or not authorization to clients and subscriptions. Most probably you must authorize your client id. Enter in your Amazon AWS console, go to IoT Core and access the menu "Secure/Policies", there select the policy attached to your IoT Thing and check at the end how connection is configured. Example:

```
{
  "Effect": "Allow",
  "Action": [
    "iot:Connect"
  ],
  "Resource": [
    "arn:aws:iot:us-east-1:222178873557:client/sdk-java",
    "arn:aws:iot:us-east-1:222178873557:client/basicPubSub",
    "arn:aws:iot:us-east-1:222178873557:client/sdk-nodejs-*"
  ]
}
```

This configuration means that only clients with ID: sdk-java, basicPubSub and sdk-nodejs-* will be allowed to connect. Change accordingly and try again.
If it still doesn't work, enable log and check in cloudwatch the reason why you can't connect.

Other properties

MQTTHeartBeat: if enabled attempts to keep alive MQTT connection sending a ping every x seconds.

Interval: number of seconds between each ping.

MQTTAuthentication: if enabled includes in MQTT connection the username and password

UserName: name of the user
Password: secret string

WatchDog: if enabled, when an unexpected disconnection is detected, tries to reconnect to the server automatically.

Interval: seconds before reconnection attempts.

Attempts: maximum number of reconnection attempts; zero means unlimited.

LogFile: if enabled, saves socket messages to a log file (useful for debugging). The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

Implementation

Amazon MQTT implementation is based on MQTT version 3.1.1 but it deviates from the specification as follows:

- In AWS IoT, subscribing to a topic with Quality of Service (QoS) 0 means a message is delivered zero or more times. A message might be delivered more than once. Messages delivered more than once might be sent with a different packet ID. In these cases, the DUP flag is not set.
- AWS IoT does not support publishing and subscribing with QoS 2. The AWS IoT message broker does not send a PUBACK or SUBACK when QoS 2 is requested.
- When responding to a connection request, the message broker sends a CONNACK message. This message contains a flag to indicate if the connection is resuming a previous session. The value of this flag might be incorrect if two MQTT clients connect with the same client ID simultaneously.
- When a client subscribes to a topic, there might be a delay between the time the message broker sends a SUBACK and the time the client starts receiving new matching messages.
- The MQTT specification provides a provision for the publisher to request that the broker retain the last message sent to a topic and send it to all future topic subscribers. AWS IoT does not support retained messages. If a request is made to retain messages, the connection is disconnected.
- The message broker uses the client ID to identify each client. The client ID is passed in from the client to the message broker as part of the MQTT payload. Two clients with the same client ID are not allowed to be connected concurrently to the message broker. When a client connects to the message broker using a client ID that another client is using, a CONNACK message is sent to both clients and the currently connected client is disconnected.
- On rare occasions, the message broker might resend the same logical PUBLISH message with a different packet ID.
- The message broker does not guarantee the order in which messages and ACK are received.

Connect to AWS IoT

First, you must sign in your AWS console, register a new device and create a X.509 certificate for this device. Once is done, you can create a new TsgcIoTAmazon_MQTT_Client and connect to AWS IoT Server. For example:

Topics

The message broker uses topics to route messages from publishing clients to subscribing clients. The forward slash (/) is used to separate topic hierarchy. The following table lists the wildcards that can be used in the topic filter when you subscribe. # Must be the last character in the topic to which you are subscribing. Works as a wildcard by matching the current tree and all subtrees.
For example, a subscription to Sensor/# receives messages published to Sensor/, Sensor/temp, Sensor/temp/room1, but not the messages published to Sensor.

+ Matches exactly one item in the topic hierarchy. For example, a subscription to Sensor/+/room1 receives messages published to Sensor/temp/room1, Sensor/moisture/room1, and so on.

Reserved Topics

Following methods are used to subscribe / publish to reserved topics.

Subscribe_ClientConnected(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID connects to AWS IoT
Subscribe_ClientDisconnected(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID disconnects to AWS IoT
Subscribe_ClientSubscribed(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID subscribes to an MQTT topic
Subscribe_ClientUnSubscribed(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID unsubscribes to an MQTT topic

Publish_Rule(const aRuleName, aText: String): A device or an application publishes to this topic to trigger rules directly

Publish_DeleteShadow(const aThingName, aText: String): A device or an application publishes to this topic to delete a shadow
Subscribe_DeleteShadow(const aThingName: String): A device or an application subscribe to this topic to delete a shadow
Subscribe_ShadowDeleted(const aThingName: String): The Device Shadow service sends messages to this topic when a shadow is deleted
Subscribe_ShadowRejected(const aThingName: String): The Device Shadow service sends messages to this topic when a request to delete a shadow is rejected
Publish_ShadowGet(const aThingName, aText: String): An application or a thing publishes an empty message to this topic to get a shadow
Subscribe_ShadowGet(const aThingName: String): An application or a thing subscribe to this topic to get a shadow
Subscribe_ShadowGetAccepted(const aThingName: String): The Device Shadow service sends messages to this topic when a request for a shadow is made successfully
Subscribe_ShadowGetRejected(const aThingName: String): The Device Shadow service sends messages to this topic when a request for a shadow is rejected
Publish_ShadowUpdate(const aThingName, aText: String): A thing or application publishes to this topic to update a shadow
Subscribe_ShadowUpdateAccepted(const aThingName: String): The Device Shadow service sends messages to this topic when an update is successfully made to a shadow
Subscribe_ShadowUpdateRejected(const aThingName: String): The Device Shadow service sends messages to this topic when an update to a shadow is rejected
Subscribe_ShadowUpdateDelta(const aThingName: String): The Device Shadow service sends messages to this topic when a difference is detected between the reported and desired sections of a shadow
Subscribe_ShadowUpdateDocuments(const aThingName: String): AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed

Persistent Sessions

A persistent session represents an ongoing connection to an MQTT message broker. When a client connects to the AWS IoT message broker using a persistent session, the message broker saves all subscriptions the client makes during the connection. When the client disconnects, the message broker stores unacknowledged QoS 1 messages and new QoS 1 messages published to topics to which the client is subscribed. When the client reconnects to the persistent session, all subscriptions are reinstated and all stored messages are sent to the client at a maximum rate of 10 messages per second.

You create an MQTT persistent session setting the **cleanSession** parameter to False **OnMQTTBeforeConnect** event. If no session exists for the client, a new persistent session is created. If a session already exists for the client, it is resumed.

Devices need to look at the **Session** attribute in the **OnMQTTConnect** event to determine if a persistent session is present. If **Session is True**, a persistent session is present and stored messages are delivered to the client. If **Session is False**, no persistent session is present and the client must re-subscribe to its topic filters.

Persistent sessions have a default expiry period of 1 hour. The expiry period begins when the message broker detects that a client disconnects (MQTT disconnect or timeout). The persistent session expiry period can be increased through the standard limit increase process. If a client has not resumed its session within the expiry period, the session is terminated and any associated stored messages are discarded. The expiry period is approximate, sessions might be persisted for up to 30 minutes longer (but not less) than the configured duration.

Temporary Credentials

AWS IoT Core can work with Temporary Credentials obtained through Identity Pools, there are 2 types of Identities:

- **UnAuthenticated:** only requires to set the policy type in the IAM
- **Authenticated:** requires to set the policy type in IAM and AWS IoT Core policies

Unauthenticated

If you are using Unauthenticated credentials, just attach the policy in the UnAuthenticated Role automatically created in the IAM menu. Then configure the client setting the Access, Secret Key and Token returned by Cognito service.
Find below a code in .NET to get unauthenticated credentials

```
CognitoAWSCredentials credentials = new CognitoAWSCredentials(
    "us-east-1:cc3c9c48-646d-44ef-bfd5-0c5fb2f0882f", // Identity pool ID
    Amazon.RegionEndpoint.USEast1 // Region
);

var identityPoolId = credentials.GetCredentialsAsync();

AmazonCognitoIdentityClient cognitoClient = new AmazonCognitoIdentityClient(
    credentials, // the anonymous credentials
    Amazon.RegionEndpoint.USEast1 // the Amazon Cognito region
);

GetIdRequest idRequest = new GetIdRequest();
idRequest.AccountId = "222178873557";
idRequest.IdentityPoolId = "us-east-1:cc3c9c48-646d-44ef-bfd5-0c5fb2f0882f";

GetIdResponse idResp = cognitoClient.GetId(idRequest);

string AccessKey = identityPoolId.Result.AccessKey;
string SecretKey = identityPoolId.Result.SecretKey;
string SessionToken = identityPoolId.Result.Token;

string IdentityId = idResp.IdentityId;
```

Authenticated

Authenticated credentials, requires attaching the policy in the Authenticated Role automatically created in the IAM menu and attach the policy of the user in AWS IoT Core policies.
So create a new policy in the IoT Core policies menu and every time a new user authenticates, attach this policy to this user.
You can use the following command of AWS to attach a policy or create a lambda function.

```
aws iot attach-policy --policy-name PolicyName --target us-east-1:XXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

Device Provisioning

The Fleet Provisioning service supports the following MQTT API operations:

- **CreateCertificateFromCsr:** Creates a certificate from a certificate signing request (CSR). AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a PENDING_ACTIVATION status. When you call RegisterThing to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.
- **CreateKeysAndCertificate:** Creates new keys and a certificate. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). The new certificate has a PENDING_ACTIVATION status. When you call RegisterThing to provision a thing with this certificate, the certificate status changes to ACTIVE or INACTIVE as described in the template.
- **RegisterThing:** Provisions a thing using a pre-defined template.

CreateCertificateFromCsr

Use the method CreateCertificateFromCsr passing the CertificateSigningRequest as a parameter to create the certificate. In order to receive the response to this request, first subscribe to the following methods: SubscribeCreateCertificateFromCsrResponse and SubscribeCreateCertificateFromCsrError

CreateKeysAndCertificate

Use the method CreateKeysAndCertificate to create a new certificate and keys. In order to receive the response to this request, first subscribe to the following methods SubscribeCreateKeysAndCertificateResponse and SubscribeCreateKeysAndCertificateError

RegisterThing

Use the method RegisterThing to register a new thing passing as a parameter the Template Name and the Payload in JSON format. In order to receive the response to this request, first subscribe to the following methods SubscribeRegisterThingResponse and SubscribeRegisterThingError.

IoT Azure MQTT Client

What is Azure IoT Hub?

IoT Hub is a managed service, hosted in the cloud, that acts as a central message hub for bi-directional communication between your IoT application and the devices it manages. You can use Azure IoT Hub to build IoT solutions with reliable and secure communications between millions of IoT devices and a cloud-hosted solution backend. You can connect virtually any device to IoT Hub.

IoT Hub supports communications both from the device to the cloud and from the cloud to the device. IoT Hub supports multiple messaging patterns such as device-to-cloud telemetry, file upload from devices, and request-reply methods to control your devices from the cloud. IoT Hub monitoring helps you maintain the health of your solution by tracking events such as device creation, device failures, and device connections.

IoT Hub's capabilities help you build scalable, full-featured IoT solutions such as managing industrial equipment used in manufacturing, tracking valuable assets in healthcare, and monitoring office building usage.

Message broker

IoT Hub gives you a secure communication channel for your devices to send data. IoT Hub and the device SDKs support the following protocols for connecting devices:

- **MQTT**
- **MQTT** over **WebSockets**

Multiple authentication types support a variety of device capabilities:

- **SAS** token-based authentication to quickly get started with your IoT solution.
- Individual **X.509 certificate** authentication for secure, standards-based authentication.

MQTT Client

TsgcIoTAzure_MQTT_Client is the component used to connect to Azure IoT. One client can connect to only one device. The client connects using the plain MQTT protocol and authenticates using SAS / X.509 Client Certificate.

In order to connect to Azure IoT Hub, client needs the following properties:

Azure.IoTHub: server name where MQTT client will connect.
Azure.DeviceId: name of device in azure IoT Hub.

Azure allows multiple authentication types, by default uses SAS tokens.

SAS Authentication

SAS.Enabled: enable if authentication uses SAS.
SAS.SecretKey: the SAS Token from your Azure IoT Account.
SAS.KeyName: the Shared Access Key Name.
SAS.Expiry: set the number of minutes before SAS Token expires. Default value is 1440 (24 hours).

If you have a connection string, you can read the connection string values automatically using the method **ReadConnectionString**. Example:

```
ReadConnectionString('HostName=yourhub.azure-
devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=Yj7RRPnkSDTv+UCFLgwIP/
FrbDymZv4qVAIoTLHUFR8=');
```

X509 Certificates

Using **OpenSSL** as IOHandler

Certificate.Enabled: enable if authentication uses certificates.
Certificate.CertFile: path to X.509 client certificate.
Certificate.KeyFile: path to X.509 client key file.
Certificate.Password: if certificate has a password set here.
Version: TLS version, by default uses TLS 1.0

Using **SChannel** as IOHandler

Certificate.Enabled: enable if authentication uses certificates.
Certificate.CertFile: path to PFX certificate (first the certificate must be converted to PFX). Read More.
Certificate.Password: if certificate has a password set here.
Version: TLS version, by default uses TLS 1.0

Other properties:

MQTTHeartBeat: if enabled attempts to keep alive MQTT connection sending a ping every x seconds.

Interval: number of seconds between each ping.

WatchDog: if enabled, when an unexpected disconnection is detected, tries to reconnect to the server automatically.

Interval: seconds before reconnection attempts.

Attempts: maximum number of reconnection attempts; zero means unlimited.

LogFile: if enabled, saves socket messages to a log file (useful for debugging). The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

Azure MQTT implementation is based on MQTT version 3.1.1 but it deviates from the specification as follows:

- IoT Hub does not support QoS 2 messages. If a device app publishes a message with QoS 2, IoT Hub closes the network connection.
- IoT Hub does not persist Retain messages. If a device sends a message with the RETAIN flag set to 1, IoT Hub adds the x-opt-retain application property to the message. In this case, instead of persisting the retain message, IoT Hub passes it to the backend app.
- IoT Hub only supports one active MQTT connection per device. Any new MQTT connection on behalf of the same device ID causes IoT Hub to drop the existing connection.

Connect to Azure IoT Hub

First, you must sign in to your Azure account, register a new device, and create an authentication method for this device. Once that is done, you can create a new TsgcIoTAzure_MQTT_Client and connect to Azure IoT Hub.

For example:

Device To Cloud

When sending information from the device app to the solution back end, IoT Hub exposes the following options:

1. **Device-to-cloud messages** for time series telemetry and alerts.
You can send key-value properties using a TStringList, just fill the TStringList with the desired message properties and pass these as argument.

If you need to set the **ContentType** and **ContentEncoding** of the message, you must add these values to the Properties List. The names of these properties are defined by Azure.

Name	Value
$.ct	application/json
$.ce	utf-8

2. **Device twin's reported properties** for reporting device state information such as available capabilities, conditions, or the state of long-running workflows. For example, configuration and software updates.

Cloud To Device

IoT Hub provides three options for device apps to expose functionality to a back-end app:

1. **Direct methods** for communications that require immediate confirmation of the result. Direct methods are often used for interactive control of devices such as turning on a fan.
You can respond to public methods using the following method.
2. **Twin's desired properties** for long-running commands intended to put the device into a certain desired state. For example, set the telemetry send interval to 30 minutes. You can get properties using the following method.
3. **Cloud-to-device messages** for one-way notifications to the device app. To get messages, first you must subscribe.
Messages are received in the **OnMQTTPublish** event (text payload) or **OnMQTTPublishEx** event (payload as TsgcWSMQTTPublishData with Value, Bytes and Stream properties).

Upload Files

IoT hub facilitates file uploads from connected devices by providing them with shared access signature (SAS) URIs or X509 certificates.

If you select SAS, you must set the following properties:

- **Azure.IoTHub:** example youriothub.azure-devices.net
- **Azure.DeviceId:** example: myDevice
- **SAS.SecretKey:** example: Yj7RRPnkSDTv+UCFLgwIP/FrbDymZv4qVAIoTLHUFR8=
- **SAS.KeyName:** example: iothubowner
- **SAS.Enabled:** the value must be set to true.

If you select X509 certificates, you must set the following properties:

- **Certificate.CertFile:** the path to your PEM certificate.
- **Certificate.KeyFile:** the path to your PEM key file.
- **Certificate.Enabled:** the value must be set to true.

Use the method **UploadFile** to upload a file to the Azure Servers. If the Overwrite parameter is set to true, it will replace the existing file. If the Overwrite parameter is set to false, it will only upload if the file doesn't exist and will raise an error if it exists (this is the option by default).

```
void UploadFileToAzure()
{
  TOpenDialog oDialog = new TOpenDialog();
  if (oDialog.Execute())
  {
    AzureIoT.UploadFile(oDialog.FileName);
  }
}
```

Device Provisioning Service

Azure IoT allows you to register devices from code using DPS. Currently, the library supports registering a device passing the Scope Id and Registration Id as parameters.

```
TsgcIoTAzure_MQTT_Client oClient = new TsgcIoTAzure_MQTT_Client();
oClient.Certificate.CertFile = "cert.pem";
oClient.Certificate.KeyFile = "key.pem";
oClient.Certificate.Enabled = true;
TsgcIoT_Azure_OperationRegistrationState oResponse = new TsgcIoT_Azure_OperationRegistrationState();
if (oClient.ProvisioningDeviceClient_Register("scope_id", "registration_id", oResponse))
  MessageBox.Show("#Provisioning Register OK: " + oResponse.Status);
else
  MessageBox.Show("#Provisioning Register Error: " + oResponse.Status);
```

Azure IoT Explorer

You can use the **Azure IoT Explorer** application to interact with devices connected to your IoT Hub. You can see the telemetry messages received, the devices registered, and more. The application is free and can be downloaded from:

https://github.com/Azure/azure-iot-explorer/releases

HTTP

The HTTP protocol allows you to fetch resources from servers, such as images and HTML documents. It is a client-server protocol, which means that the client requests from the server the resources it needs.

When a client wants to connect to a server, it follows the next steps:

1. Open a new TCP connection
2. Send a message to the server with the requested data

```
GET / HTTP/1.1
Host: server.com
Accept-Language: en-us
```

3. Read the response sent by the server

```
HTTP/1.1 200 OK
Server: Apache
Content-Length: 120
Content-Type: text/html

...
```

Components

- **HTTP/2:** HTTP/2 (or h2) is a binary protocol that brings push, multiplexing streams and frame control to the web.
- **HTTP/1 Client:** a non-visual component that inherits from the TIdHTTP client component.
- **OAuth2:** OAuth2 allows third-party applications to receive limited access to an HTTP service.
- **JWT:** JWT allows creating data with optional signature and/or encryption whose payload holds JSON that asserts some number of claims.
- **Amazon SQS:** a fully managed message queuing service for microservices, distributed systems, and serverless applications.
- **Google Cloud Pub/Sub:** provides messaging between applications and is designed to provide reliable, many-to-many, asynchronous messaging between applications.
- **Google Calendar:** allows you to use Google Calendar API V3: get Calendars, events, synchronize with your own calendar...
- **Google FCM:** sends notifications using Firebase Cloud Messaging.

HTTP/2

HTTP/2 is an evolution of the HTTP 1.1 protocol, it basically tries to be more efficient when using networks. The semantics are the same, so it's designed to be compatible with old protocols.

HTTP 1.1 Limitations

HTTP 1.1 is limited to processing one request per connection, so usually clients use more than one connection to request files from servers. But this raises a problem, because when there are too many open TCP connections, there is a race between clients to use the server resources and performance is lower and lower as more clients connect to servers.

Main features

- HTTP/2 is a binary protocol (remember that HTTP 1.1 is a text protocol).
- HTTP/2 works over TLS and ALPN.
- It's multiplexed (allows you to send more than one request over a single TCP Connection).
- Server can push responses to clients.
- Reduces Round Trip Times, so clients can load faster.

HTTP/2 introduces other improvements, more details: RFC7540.

HTTP/2 requires our custom Indy version because it requires ALPN protocol.

Components

- **TsgcHTTP2Client:** client component that fully supports HTTP/2 protocol (sgcWebSockets 100% Pascal code, without external libraries).
- **TsgcWebSocketHTTPServer:** server component that fully supports HTTP/2 protocol (sgcWebSockets 100% Pascal code, without external libraries). By default HTTP/2 is disabled, you can enable using HTTPOptions property and set Enable = true.
- **TsgcWebSocketServer_HTTPAPI:** server component that supports HTTP/2 protocol (Microsoft implementation and Requires Windows 2016+ or Windows 10+).
- DataSnap Servers: datasnap server can support HTTP/2 protocol too.

APIs

- **Apple Push Notifications:** push user-facing notifications to the user's device from a server provider.

TsgcHTTP2Client

TsgcHTTP2Client implements Client HTTP/2 Component and can connect to HTTP/2 servers. Follow the steps below to configure this component:

1. Create a new instance of **TsgcHTTP2Client** component.

2. Send the request to server and process the response using OnHTTP2Response event. example:

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2Response += OnHTTP2ResponseEvent;
oClient.Get("https://www.google.com");

void OnHTTP2ResponseEvent(object Sender,
    TsgcHTTP2ConnectionClient Connection, TsgcHTTP2RequestProperty Request,
        TsgcHTTP2ResponseProperty Response)
{
  MessageBox.Show(Response.DataString);
}
```

Most common uses

- **Requests**
  - Request HTTP/2 Method
  - HTTP/2 Server Push
  - Download File
  - HTTP/2 Partial Responses
  - HTTP/2 Headers
- **Connection**
  - Client Close Connection
  - Client Keep Connection Active
  - HTTP/2 Reason Disconnection
  - Client Pending Requests
- **Authentication**
  - Client Authentication
  - HTTP/2 and OAuth2
- **Classes**
  - TsgcHTTP2ConnectionClient
  - TsgcHTTP2RequestProperty
  - TsgcHTTP2ResponseProperty

Methods

The following HTTP methods are supported:

GET: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

HEAD: The HEAD method asks for a response identical to that of a GET request, but without the response body.

POST: The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

PUT: The PUT method replaces all current representations of the target resource with the request payload.

DELETE: The DELETE method deletes the specified resource.

CONNECT: The CONNECT method establishes a tunnel to the server identified by the target resource.

OPTIONS: The OPTIONS method is used to describe the communication options for the target resource.

TRACE: The TRACE method performs a message loop-back test along the path to the target resource.

PATCH: The PATCH method is used to apply partial modifications to a resource.

HTTP/2 client component also implements the following methods:

Ping: sends a ping to a Server.

Close: sends a message to server that the connection will be closed.

Disconnect: disconnects the socket connection.

Properties

Authentication: allows you to authenticate against OAuth2 before sending an HTTP/2 request.

Token

OAuth: assign here a TsgcHTTP_OAuth_Client component to get OAuth2 credentials. Read more about OAuth2.

JWT: assign here a TsgcHTTP_JWT_Client component to get JWT credentials. Read more about JWT.

Request: Specifies the header values to send to the HTTP/2 server.

Settings: Specifies the header values to send to the HTTP/2 server.

EnablePush: by default enabled, this setting can be used to avoid server push content to client.

HeaderTableSize: Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block. The initial value is 4,096 octets.

InitialWindowSize: Indicates the sender's initial window size (in octets) for stream-level flow control. The initial value is 65,535 octets. This setting affects the window size of all streams.

MaxConcurrentStreams: Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially, there is no limit to this value.

MaxFrameSize: Indicates the size of the largest frame payload that the sender is willing to receive, in octets. The initial value is 16,384 octets.

MaxHeaderListSize: This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

FragmentedData: this property allows you to configure how to handle the fragments received.

h2fdOnlyBuffer: it's the default option, the response is dispatched only when has been received the latest packet.

h2fdAll: the response is dispatched for every packet received (one or more) on the event OnHTTP2ResponseFragment and on the event OnHTTP2Response when the latest packet has been received.

h2fdOnlyFragmented: the response is only dispatched in the event OnHTTP2ResponseFragment for every packet received (one response can be compound of 1 or multiple packets).

ReadTimeout: max time in milliseconds to wait for a synchronous HTTP/2 response (e.g. Get, Post). Default is 60000 (60 seconds). Set to 0 for no timeout (infinite wait until the response is fully received or the connection is closed). For large file transfers (1 GB+), set this to 0 or a sufficiently large value.

Host: IP or DNS name of the server.

HeartBeat: if enabled attempts to keep alive HTTP/2 connection sending a ping every x seconds.

Interval: number of seconds between each ping.

HeartBeatType: allows customizing how the HeartBeat works

- **hbtAlways:** sends a ping every x seconds defined in the Interval.
- **hbtOnlyIfNoMsgRcvInterval:** sends a ping every x seconds only if no messages have been received during the latest x seconds defined in the Interval property.

TCPKeepAlive: if enabled, uses keep-alive at TCP socket level, in Windows will enable SIO_KEEPALIVE_VALS if supported and if not will use keepalive. By default is disabled. Read about Dropped Disconnections.

Time: if after X time socket doesn't send anything, it will send a packet to keep-alive connection (value in milliseconds).

Interval: after sends a keep-alive packet, if not received a response after interval, it will send another packet (value in milliseconds).

ConnectTimeout: max time in milliseconds before a connection is ready.

ReadTimeout: max time in milliseconds to read messages.

WriteTimeOut: max time in milliseconds sending data to other peer, 0 by default (only works under Windows OS).

Port: Port used to connect to the host.

LogFile: if enabled, saves socket messages to a log file (useful for debugging). The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by socket it will be saved on a file.

FileName: full path to the filename.

NotifyEvents: defines which mode to notify WebSocket events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access to controls that are not thread-safe, you need to implement your own synchronization methods.

Proxy: here you can define if you want to connect through a HTTP Proxy Server. If you need to connect to SOCKS proxies, just enable SOCKS.Enable property too.

WatchDog: if enabled, when an unexpected disconnection is detected, tries to reconnect to the server automatically.

Interval: seconds before reconnection attempts.

Attempts: maximum number of reconnection attempts; zero means unlimited.

Throttle: used to limit bits per second sent or received.

TLS: enables a secure connection.

TLSOptions: if TLS enabled, here you can customize some TLS properties.

ALPNProtocols: list of the ALPN protocols which will be sent to server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if certificate is secured with a password, set here.
VerifyCertificate: if certificate must be verified, enable this property.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default uses TLS 1.0, if server requires a higher TLS version, here can be selected.
IOHandler: select which library you will use to connect using TLS.
- **iohOpenSSL:** uses OpenSSL library and is the default for Indy components. Requires to deploy openssl libraries for win32/win64.
- **iohSChannel:** uses Secure Channel which is a security protocol implemented by Microsoft for Windows, doesn't require to deploy openssl libraries. Only works in Windows 32/64 bits.

OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used.
  - **osIAPI_1_0:** uses API 1.0 OpenSSL, it's latest supported by Indy
  - **osIAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **osIAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where are located the openSSL libraries
  - **osIpNone:** this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
  - **osIpDefaultFolder:** sets automatically the openSSL path where the libraries should be located for all IDE personalities.
  - **osIpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = osIpCustomFolder define here the full path where are located the openSSL libraries.
- **UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
  - **osIsSymLinksDefault:** by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
  - **osIsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.
  - **osIsSymLinksLoad:** Load SymLinks after trying to load the version libraries.
  - **osIsSymLinksDontLoad:** don't load the SymLinks.

SChannel_Options: allows you to use a certificate from Windows Certificate Store.
- **CertHash:** is the certificate Hash. You can find the certificate Hash running a dir command in powershell.
- **CipherList:** here you can set which Ciphers will be used (separated by ":"). Example: CALG_AES_256:CALG_AES_128
- **CertStoreName:** the store name where is stored the certificate. Select one of below:
  - **scsnMY** (the default)
  - **scsnCA**
  - **scsnRoot**
  - **scsnTrust**
- **CertStorePath:** the store path where is stored the certificate. Select one of below:
  - **scspStoreCurrentUser** (the default)
  - **scspStoreLocalMachine**
- **UseLegacyCredentials:** force the use of SCHANNEL_CRED.

Events

OnHTTP2Response

This event is called when client receives a Response from Server. Access to Response object to get full information about Server Response.

Response.Headers: HTTP/2 headers
Response.Data: Raw body response.
Response.DataString: body response as string.
Response.DataUTF8: body response as UTF-8 string.

```
void OnHTTP2ResponseEvent(object Sender, TsgcHTTP2ConnectionClient Connection,
  TsgcHTTP2RequestProperty Request, TsgcHTTP2ResponseProperty Response)
{
  MessageBox.Show(Response.Headers.Text + "\r\n" + Response.DataString);
}
```

OnHTTP2ResponseFragment

This event is called when client receives a fragment response from Server, so means that this stream will receive more updates.

OnHTTP2Authorization

In this event you can set the UserName and Password when Authentication is Basic, or the Token for OAuth2 Authentications.

OnHTTP2BeforeRequest

This event is called before client sends Headers Request to server. You can add or modify the headers before they are sent to HTTP/2 server.

OnHTTP2Connect

This event is called just after client connects successfully to server.

OnHTTP2Disconnect

This event is called when connection is closed.

OnHTTP2Exception

If there is any exception while client is connected to server, here you can catch the Exception.

OnHTTP2GoAway

This event is raised when client receives a GoAway message from server.

OnHTTP2PendingRequests

After a disconnection, if there are pending requests to be sent or received, here you can set if you want reconnect and/or clear pending requests.

OnHTTP2PushPromise

When server sends a PushPromise to client, client can accept or not the PushPromise packets.

OnHTTP2RSTStream

When server resets a stream, this event is called.

TsgcHTTP2Client | Request HTTP/2 Method

HTTP/2 Client can work in blocking and non-blocking mode, internally the component works in a secondary thread and requests are processed asynchronously, but you can call a request and wait till this request is completed.

Find below an example of how client can request an HTML page to a HTTP/2 Server and how can work in both modes.

Asynchronous Mode

Get the following url: https://www.google.com and be notified when client receives the full response. After you call **GETASYNC** method, the process continues and OnHTTP2Response event is called when response is received.

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2Response += OnHTTP2ResponseEvent;
oClient.GetAsync("https://www.gooogle.com");
void OnHTTP2ResponseEvent(object Sender, TsgcHTTP2ConnectionClient Connection,
  TsgcHTTP2RequestProperty Request, TsgcHTTP2ResponseProperty Response)
{
  MessageBox.Show(Response.Headers.Text + #13#10 + Response.DataString);
}
```

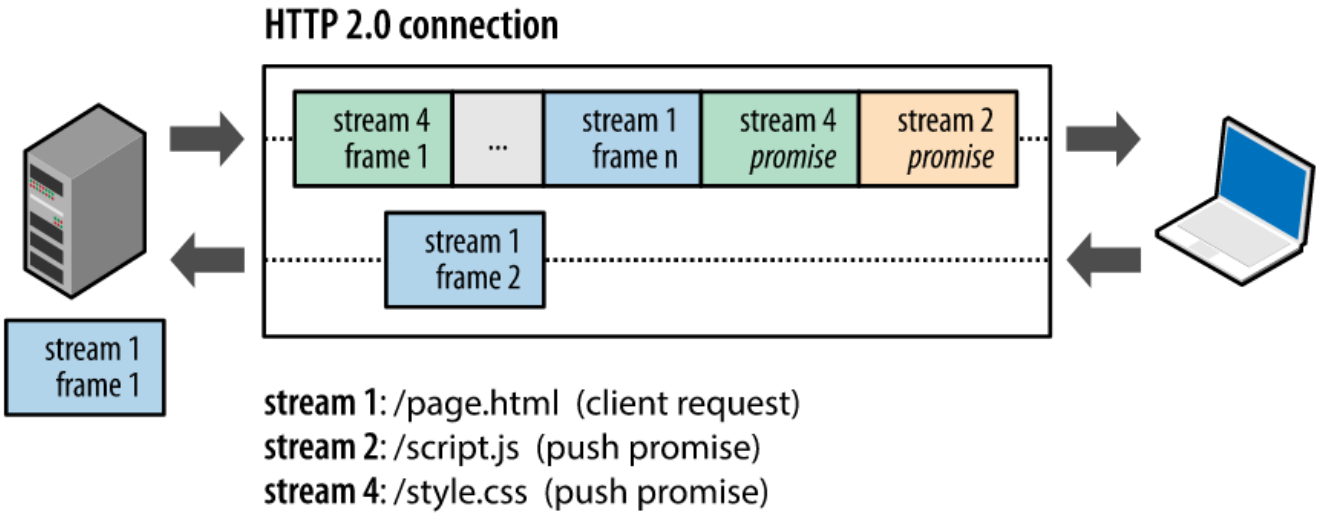
Blocking Mode

Get the following url: https://www.google.com and wait till client receives the full response. After you call **GET** method, the process waits till response is received or time out is reached.
You can access to the Raw Response data, using Response property of HTTP/2 client. Here you can access to Raw Headers, Status response code, Charset and more.

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
MessageBox.Show(oClient.Get("https://www.gooogle.com"));
```

Requests | HTTP/2 Server Push

Server Push is the ability of the server to send multiple responses for a single client request. That is, in addition to the response to the original request, the server can push additional resources to the client, without the client having to request each one explicitly.



Every time server sends to client a PushPromise message, OnHTTP2PushPromise event is called. When the client receives a PushPromise, it means that the server will send this resource in the next packets, so the client can accept or reject it.

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2PushPromise += OnHTTP2PushPromiseEvent;
oClient.Get("https://http2.golang.org/serverpush");
...
void OnHTTP2PushPromiseEvent(object Sender, TsgcHTTP2ConnectionClient Connection,
  TsgcHTTP2_Frame_PushPromise PushPromise, ref bool Cancel)
{
  if (PushPromise.URL == "/serverpush/static/godocs.js")
  {
    Cancel = true;
  }
  else
  {
    Cancel = false;
  }
}
```

TsgcHTTP2Client | HTTP/2 Download File

When the client requests a file from the server, use OnHTTP2Response event to load the stream response.

Large File Downloads

When downloading large files (hundreds of MB or more), set **HTTP2Options.ReadTimeout** to **0** (no timeout) to ensure the transfer completes without being interrupted by the default 60-second timeout:

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.HTTP2Options.ReadTimeout = 0; // no timeout for large files
oClient.Get("https://server/largefile", oStream);
```

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2Response += OnHTTP2ResponseEvent;
oClient.Get("https://http2.golang.org/file/gopher.png");
...
void OnHTTP2ResponseEventEvent(object Sender, TsgcHTTP2ConnectionClient Connection,
  TsgcHTTP2RequestProperty Request, TsgcHTTP2ResponseProperty Response)
{
  using (FileStream oStream = new FileStream("file", FileMode.Create, FileAccess.Write))
  {
    Response.Data.CopyTo(oStream);
  }
}
```

TsgcHTTP2Client | HTTP/2 Partial Responses

Usually when you send an HTTP Request, server sends a response with the file requested, sometimes, instead of sending a single response, the server can send multiple responses like a stream, in these cases you can use **OnHTTP2ResponseFragment** event to capture these responses and show to user.

Example: send a request to https://http2.golang.org/clockstream and server will send a stream response every second.

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2ResponseFragment += OnHTTP2ResponseFragmentEvent;
oClient.Get("https://http2.golang.org/clockstream");
...
void OnHTTP2ResponseFragmentEvent(object Sender, TsgcHTTP2ConnectionClient Connection,
  TsgcHTTP2RequestProperty Request, TsgcHTTP2ResponseFragmentProperty Fragment)
{
  MessageBox.Show(Fragment.DataString);
}
```

TsgcHTTP2Client | HTTP/2 Headers

TsgcHTTP2Client allows customizing Headers sent to server when client connects
Example: if you need to add this HTTP Header "Client: sgcWebSockets"

```
void OnHTTP2BeforeRequest(object Sender, TsgcHTTP2ConnectionClient Connection,
  ref string Headers)
{
  Headers = Headers + Environment.NewLine + "Client: sgcWebSockets";
}
```

You can use Request.CustomHeaders to add your customized headers too.

TsgcHTTP2Client | Client Close Connection

Connection can be closed using Active property or using Close/Disconnect methods.

Active property

When connection is active and you set Active := False, the connection will be closed immediately without sending any message to server about the disconnection.

Disconnect

You can use Disconnect method (from TsgcHTTP2Client or TsgcHTTP2ConnectionClient) to disconnect the socket.

Close

This method, sends a message to server informing that connection will be closed and you can send optionally some info about the reason of the disconnection. It is a clean way to close an HTTP/2 connection.
Close method can be called from TsgcHTTP2Client or TsgcHTTP2ConnectionClient objects.

The following error reasons can be sent:

- no error
- protocol error
- internal error
- flow control error
- settings timeout
- stream closed
- frame size error
- refused stream
- cancel
- compression error
- connect error
- enhance your calm
- inadequate security
- required

TsgcHTTP2Client | Client Keep Connection Active

Once your client has connected to server, sometimes connection can be closed due to poor signal, connection errors... there are 2 properties which help to keep connection active.

HeartBeat

HeartBeat property allows you to **send a Ping** every X **seconds** to **maintain connection alive**. Some servers, close TCP connections if there is no data exchanged between peers. HeartBeat solves this problem, sending a ping every a specific interval. Usually this is enough to maintain a connection active.

The property HeartBeatType allows customizing how the HeartBeat works:

1. **hbtAlways:** sends a ping every x seconds defined in the Interval.
2. **hbtOnlyIfNoMsgRcvInterval:** sends a ping every x seconds only if no messages have been received during the latest x seconds defined in the Interval property.

Example: send a ping every 30 seconds

```
oClient = new TsgcHTTP2Client();
oClient.HeartBeat.Interval = 30;
oClient.HeartBeat.Enabled = true;
oClient.Active = true;
```

WatchDog

If WatchDog is enabled, when client detects a disconnection, WatchDog try to reconnect again every X seconds until connection is active again.

Example: reconnect every 10 seconds after a disconnection with unlimited attempts.

```
oClient = new TsgcHTTP2Client();
oClient.WatchDog.Interval = 10;
oClient.WatchDog.Attempts = 0;
oClient.WatchDog.Enabled = true;
oClient.Active = true;
```

TsgcHTTP2Client | HTTP/2 Reason Disconnection

HTTP/2 Server can disconnect a client for several reasons, when the server wants to inform the client of the reason why it is disconnecting, it sends a **GoAway** message to client with information about disconnection.

Use OnHTTP2GoAway event to catch the reason why server has disconnected (if client wants to close a connection, can use the method close to send the reason why is closing the connection).

TsgcHTTP2GoAwayProperty Object contains the information about disconnection

- **LastStreamId:** is the last stream processed by server.
- **ErrorCode:** integer which identifies the error code.
- **ErrorDescription:** description of the error, one of the following:
    - no error
    - protocol error
    - internal error
    - flow control error
    - settings timeout
    - stream closed
    - frame size error
    - refused stream
    - cancel
    - compression error
    - connect error
    - enhance your calm
    - inadequate security
    - required
- **AdditionalDebugData:** optional string which offers more information about disconnection.

TsgcHTTP2Client | Client Pending Requests

When client sends several requests, these are processed in a secondary thread, sometimes connection can be closed for any reason, and there are still requests pending. Use OnHTTP2PendingRequests event to handle these pending requests. This event is called when client detects a disconnection and there are still pending requests. This event has 2 parameters:

1. **Reconnect:** by default disable, if you set to true, client will reconnect automatically.
2. **Clear:** by default enabled, if you set to false, when client connects again, it will try to resend pending requests to server.

TsgcHTTP2Client | Client Authentication

HTTP/2 client supports 2 authentication types: Basic Authentication and OAuth2 Authentication.

Use **OnHTTP2Authorization** event to handle both types of authentication.

Basic Authentication

If server returns a header requesting Basic Authentication, set OnHTTP2Authorization the username and password.

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2Authorization += OnHTTP2AuthorizationEvent;
...
void OnHTTP2AuthorizationEvent(object Sender, TsgcHTTP2ConnectionClient Connection, string AuthType, string AuthD
{
  if (AuthType == "Basic")
  {
    UserName = "user";
        Password = "secret";
  }
}
```

Bearer Token

If server returns a header requesting Bearer Token Authentication, set OnHTTP2Authorization the token.

```
TsgcHTTP2Client oClient = new TsgcHTTP2Client();
oClient.OnHTTP2Authorization += OnHTTP2AuthorizationEvent;
...
void OnHTTP2AuthorizationEvent(object Sender, TsgcHTTP2ConnectionClient Connection, string AuthType, string AuthD
{
  if (AuthType == "Bearer")
  {
    aToken = "bearer token";
  }
}
```

Bearer value from Third-party

If you already know the Bearer Value, because you have obtained using another method, you can pass the Bearer value as an HTTP header using the following properties of the request, just set before calling any HTTP Request method:

```
TsgcHTTP2Client.Request.BearerAuthentication
 = true


TsgcHTTP2Client.Request.BearerToken = "< value of the token >"
```

OAuth2

Read the following article if you want to use our OAuth2 component with HTTP/2 client.

TsgcHTTP2Client | HTTP/2 and OAuth2

OAuth2 is a common authorization method used by several companies like Google. When you want to authenticate against Google servers to use any of their APIs, usually requires an Authentication using OAuth2.

sgcWebSockets supports OAuth2 under HTTP/2 client, there is a property called **Authentication.Token.OAuth** where you must assign an instance of TsgcHTTP_OAuth2.

How connect to GMail Google API

In order to connect to Google APIs, we will need to create an instance of TsgcHTTP_OAuth2 and fill the following data:

```
TsgcHTTP_OAuth1.AuthorizationServerOptions.AuthURL := 'https://accounts.google.com/o/oauth2/auth';
TsgcHTTP_OAuth1.AuthorizationServerOptions.TokenURL := 'https://accounts.google.com/o/oauth2/token';


TsgcHTTP_OAuth1.LocalServerOptions.IP := '127.0.0.1';
TsgcHTTP_OAuth1.LocalServerOptions.Port := 8080;


TsgcHTTP_OAuth1.OAuth2Options.ClientId := 'your client id';
TsgcHTTP_OAuth1.OAuth2Options.ClientSecret := 'your client secret';
```

After fill the OAuth2 client component, create a new instance of TsgcHTTP2Client and Assign the OAuth2 component to the HTTP/2 client.

```
 TsgcHTTP2Client1.Authentication.Token.OAuth := TsgcHTTP_OAuth1;
```

Finally, do a request to get a list of messages of account yourname@gmail.com

```
oStream := TStringStream.Create('');
Try
  TsgcHTTP2Client1.Get('https://gmail.googleapis.com/gmail/v1/users/yourname@gmail.com/messages', oStream);
  ShowMessage(oStream.DataString);
Finally
  oStream.Free;
End;
```

TsgcHTTP2ConnectionClient

TsgcHTTP2ConnectionClient is a wrapper of client HTTP/2 connections, you can access to this object on Client Events.

Methods

- **Ping:** sends a ping to server to maintain connection alive.
- **Close:** sends a message to server with information about why is disconnecting.
- **Disconnect:** closes the connection without sending any informational message to then server.
- **HTTP/2 Methods:**

  GET: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

  HEAD: The HEAD method asks for a response identical to that of a GET request, but without the response body.

  POST: The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

  PUT: The PUT method replaces all current representations of the target resource with the request payload.

  DELETE: The HEAD method asks for a response identical to that of a GET request, but without the response body.

  CONNECT: The CONNECT method establishes a tunnel to the server identified by the target resource.

  OPTIONS: The OPTIONS method is used to describe the communication options for the target resource.

  TRACE: The TRACE method performs a message loop-back test along the path to the target resource.

  PATCH: The PATCH method is used to apply partial modifications to a resource.

TsgcHTTP2RequestProperty

This object is received as an argument OnHTTP2Response event, it allows to know the original request of the response send by the server.

Properties

- **Method:** identifies the HTTP/2 method (GET, POST...)
- **URL:** is the URL requested.
- **Request:** contains the fields of the request.

TsgcHTTP2ResponseProperty

This object is received as an argument OnHTTP2Response event, it allows to know the response sent by the server to the client.

Properties

- **Headers:** contains a list of raw headers received from server.
- **Data:** contains the raw body sent by the server as response to request.
- **DataString:** is the conversion to string of Data.
- **DataUTF8:** is the conversion to UTF8 string of Data.
- **PushPromise:** if assigned, contains the PushPromise object sent by the server to client (means that this response object has not been requested by client).

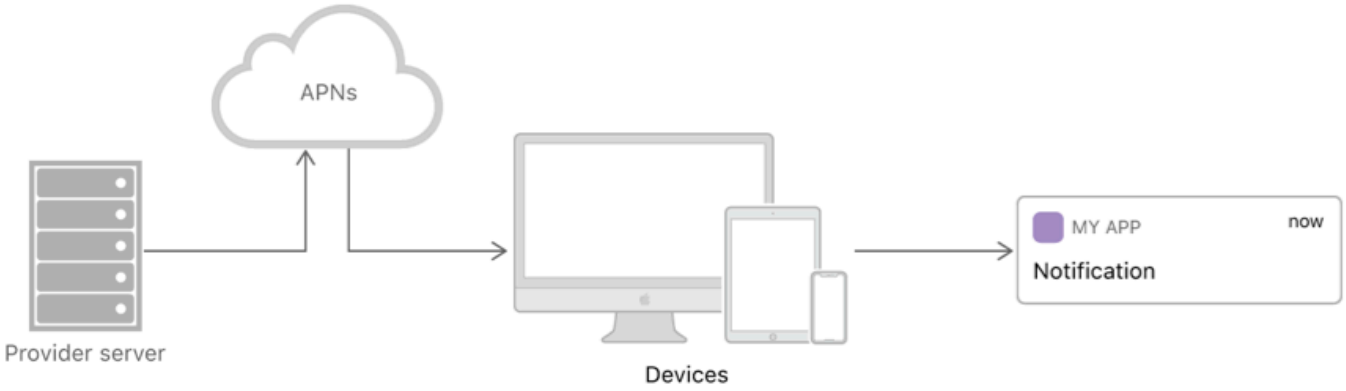
HTTP2 | Apple Push Notifications

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server

Apple allows sending push notifications to Apple devices using the Apple Push Notification Service (APNs).

When you want to send a notification to a device, the provider must send a HTTP/2 POST to APNs including the following information:

- JSON Payload with the information you want to send.
- A Device Token that identifies the user's device.
- Some HTTP Headers about how deliver the notification.
- A SSL Certificate or a JWT Token to Authenticate your request against APNs



What's required to Send Notifications

In order to send notifications to your device using Rad Studio, you must follow the next steps

- Register your APP with APNs
- Generate a Remote Notification
- Sending Notification Requests to APNs
    - Token-Based Connection to APNs
    - Certificate-Based Connection to APNs

APN | Generate a Remote Notification APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/generating_a_remote_notification

The Apple Notifications use a JSON payload to send the notification object. The maximum size of the payload is 4096 bytes.

JSON Payload Samples

Simple alert message

```
{
  "aps":{
    "alert":"Alert from sgcWebSockets!"
  }
}
```

Alert with title and subtitle.

```
{
   "aps" : {
      "alert" : {
         "title" : "Game Request",
         "subtitle" : "Five Card Draw"
         "body" : "Bob wants to play poker",
      },
      "category" : "GAME_INVITATION"
   },
   "gameID" : "12345678"
}
```

APN | Sending Notification Requests to APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server

Send your remote notification payload and device token information to Apple Push Notification service (APNs).

How to Connect to APNs

You must use HTTP/2 protocol and at least TLS 1.2 or later to establish a successful connection between your Server Provider and one of the following servers:

Development Server: https://api.sandbox.push.apple
Production Server: https://api.push.apple

Sample Code

Create a new instance of TsgcHTTP2Client and call the method POST to send a notification to APNs.

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();
// ... requires authorization code
oHTTP.Post("https://api.push.apple/3/device/device_token", "{\"aps\":{\"alert\":\"Alert from sgcWebSockets!\"}}")
if (oHTTP.Response.Status == 200)
{
  MessageBox.Show("Notification Sent Successfully");
}
else
{
  MessageBox.Show("Notification error");
}
```

To send notifications, you must establish either **token-based** or **certificate-based** trust with APNs using HTTP/2 protocol and TLS 1.2 or later.

- Token-Based Connection to APNs
- Certificate-Based Connection to APNs

APNs Trusted | Token-Based Connection to APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/establishing_a_token-based_connection_to_apns

Secure your communications with Apple Push Notification service (APNs) by using stateless authentication Tokens.

First you must obtain an **Encryption Key** and a **Key ID** from Apple Developer Account. Once you complete a successful registration, you will obtain a 10-Character string with the Key ID and an Authentication Token signing key as a .p8 file extension.

You must use the sgcWebSockets JWT Client to generate a JWT using **ES256** as algorithm. The token must not be generated for every HTTP/2 request, the token must not be refreshed before 20 minutes and not after 60 minutes.

Configure JWT Client

Configure the JWT Client with the following values:

- **JWTOptions.Header.Algorithm:** is the encryption algorithm you used to encrypt the token. APNs supports only the ES256 algorithm.
- **JWTOptions.Header.kid:** is the 10-character Key ID obtained from your developer account.
- **JWTOptions.Payload.iss:** the value for which is the 10-character Team ID you use for developing your company's apps. Obtain this value from your developer account.
- **JWTOptions.Payload.iat:** The "issued at" time, whose value indicates the time at which this JSON token was generated. Specify the value as the number of seconds since Epoch, in UTC. The value must be no more than one hour from the current time.
- **JWTOptions.RefreshTokenAfter:** set the value in seconds to 40 minutes (60*40).

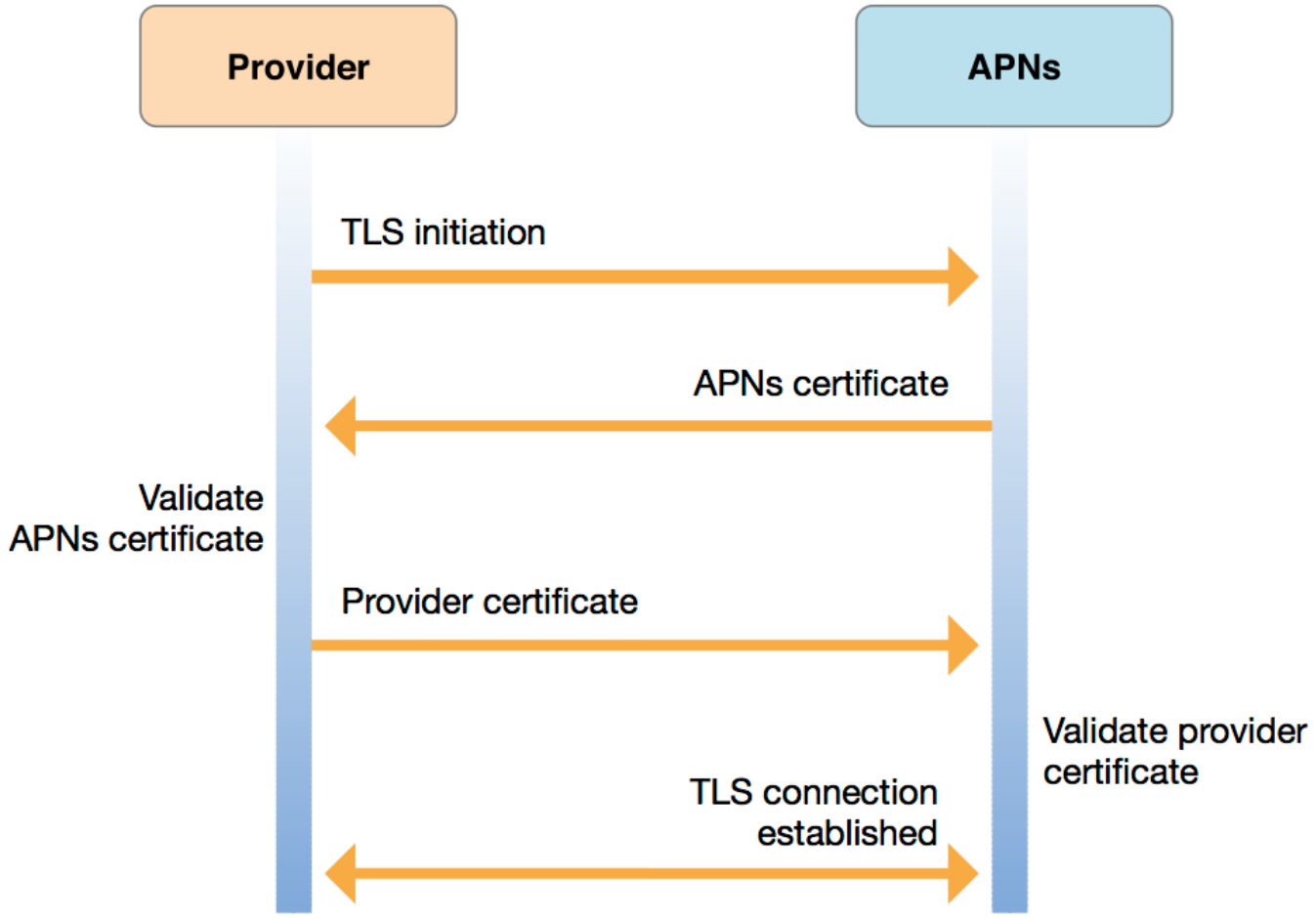
Using Token-Based connections requires sending the **apns-topic** with the value of your app's bundle ID/app id (example: com.example.application).

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();
oHTTP.TLSOptions.IOHandler = TwsTLSIOHandler.iohOpenSSL;


TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oHTTP.Authentication.Token.JWT = oJWT;
oJWT.JWTOptions.Header.alg = TsgcHTTP_JWT_Algorithm.jwtES256;
oJWT.JWTOptions.Header.kid = "apple key id";
oJWT.JWTOptions.Payload.iss = "issuer";
oJWT.JWTOptions.Payload.iat = unix time;
oJWT.JWTOptions.Algorithms.ES.PrivateKey.LoadFromFile("AuthKey_**.p8");
oJWT.JWTOptions.RefreshTokenAfter = 60*40;


oHTTP.Request.CustomHeaders = "apns-topic: com.example.application";
```

Certificate-Based Connection to APNs

https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/establishing_a_certificate-based_connection_to_apns

You can secure your communications with Apple Push Notification service (APNs) using a certificate obtained from Apple.



First enter in your developer account and **create a new certificate** for Apple Push Notification service

Once you have downloaded your certificate, the sgcWebSockets HTTP/2 client allows you to use 2 security IOHandlers (only for windows, for other personalities only openSSL is supported).

- OpenSSL
- SChannel (only for windows)

OpenSSL

If you use OpenSSL, you must deploy the OpenSSL libraries with your application. Before setting the certificate with the TsgcHTTP2Client, this certificate must first be converted to PEM format because OpenSSL doesn't allow importing P12 certificates directly.

Use the following commands to convert a single P12 certificate to a certificate in PEM format and a private key file

create PEM certificate file

```
openssl pkcs12 -in INFILE.p12 -out OUTFILE.crt -nokeys
```

Create Private Key file

```
openssl pkcs12 -in INFILE.p12 -out OUTFILE.key -nodes -nocerts
```

Once you have your certificate and private key in PEM format, you can configure the TsgcHTTP2Client as follows.

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();
oHTTP.TLSOptions.IOHandler = TwsTLSIOHandler.iohOpenSSL;
oHTTP.TLSOptions.CertFile = "certificate_file.pem";
oHTTP.TLSOptions.KeyFile = "private_key.pem";
oHTTP.TLSOptions.Password = "certificate password";
oHTTP.TLSOptions.Version = TwsTLSVersions.tls1_2;
```

SChannel

If you use SChannel there is no need to deploy any libraries and the certificate downloaded from Apple can be directly imported without the need of a previous conversion to PEM format.
Set the property UseLegacyCredentials to true when using SChannel as IOHandler.

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();
oHTTP.TLSOptions.IOHandler = TwsTLSIOHandler.iohSChannel;
oHTTP.TLSOptions.IOHandler.SChannel_Options.UseLegacyCredentials = true;
oHTTP.TLSOptions.CertFile = "certificate_file.p12";
oHTTP.TLSOptions.Password = "certificate password";
oHTTP.TLSOptions.Version = TwsTLSVersions.tls1_2;
```

Errors

If you get the error **"missing topic"** most probably you are using an universal certificate (certificates that can be used for push notifications, voip...) which requires to set the topic name with the value of your app's bundle ID/app id (example: com.example.application). Just set the apns-topic header with the correct value in the Request property of the HTTP/2 client.

```
oHTTP.Request.CustomHeaders = "apns-topic: com.example.application";
```

HTTP/1

TsgcHTTP1Client is a non-visual component that inherits from TIdHTTP indy component and adds some new properties.
This component is located in sgcHTTP unit.

TLSOptions

Allows you to configure how to connect to secure SSL/TLS servers using the HTTP/1 protocol.

ALPNProtocols: list of the ALPN protocols which will be sent to the server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if the certificate is secured with a password, set it here.
VerifyCertificate: if the certificate must be verified, enable this property. Use the event **OnSSLVerifyPeer** to customize the SSL verification.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default uses TLS 1.0. If the server requires a higher TLS version, it can be selected here.
Proxy: here you can define if you want to connect through a Proxy Server, you can connect to the following proxy servers:
- **pxyHTTP:** HTTP Proxy Server.
- **pxySocks4:** SOCKS4 Proxy Server.
- **pxySocks4A:** SOCKS4A Proxy Server.
- **pxySocks5:** SOCKS5 Proxy Server.

IOHandler: select which library you will use to connect using TLS.
- **iohOpenSSL:** uses OpenSSL library and is the default for Indy components. Requires deploying OpenSSL libraries for win32/win64.
- **iohSChannel:** uses Secure Channel, which is a security protocol implemented by Microsoft for Windows. It does not require deploying OpenSSL libraries. Only works on Windows 32/64 bits.

OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used.
  - **oslAPI_1_0:** uses API 1.0 OpenSSL, it's latest supported by Indy
  - **oslAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **oslAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where the OpenSSL libraries are located
  - **oslpNone:** this is the default. The OpenSSL libraries should be in the same folder as the binary or in a known path.
  - **oslpDefaultFolder:** automatically sets the OpenSSL path where the libraries should be located for all IDE personalities.
  - **oslpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where the OpenSSL libraries are located.
- **UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
  - **oslsSymLinksDefault:** by default, symlinks are enabled except under OSX64 (macOS Monterey and later fail when trying to load the library without a version).
  - **oslsSymLinksLoadFirst:** load symlinks first, before trying to load the versioned libraries.
  - **oslsSymLinksLoad:** load symlinks after trying to load the versioned libraries.
  - **oslsSymLinksDontLoad:** don't load the SymLinks.
- **MinVersion:** set here the minimum version that will use the client to connect to a secure server. By default, the value is tlsUndefined which means the minimum version is the same which has been set in the Version property. Example: if you want to set the Client to only connect using TLS 1.2 or TLS 1.3 set the following values.

```
SSLOptions.Version := tls1_3;
SSLOptions.OpenSSL_Options.MinVersion := tls1_2;
```

X509Checks: use this property to enable additional X509 certificate validations:
Mode: select which options will be validated
oslx509chHostName: verifies the hostname certificate.
oslx509chIPAddress: verifies the ip address of the certificate.
HostName: set the hostname if it's different from the request.
IPAddress: set the ip address if it's different from the request.

SChannel_Options: allows you to use a certificate from Windows Certificate Store.
CertHash: is the certificate Hash. You can find the certificate Hash running a dir command in power-shell.
CipherList: here you can set which Ciphers will be used (separated by ":"). Example: CALG_AES_256:CALG_AES_128
CertStoreName: the store name where the certificate is stored. Select one of the following:
scsnMY (the default)
scsnCA
scsnRoot
scsnTrust
CertStorePath: the store path where the certificate is stored. Select one of the following:
scspStoreCurrentUser (the default)
scspStoreLocalMachine

Log

If the Log property is enabled, it saves socket messages to a specified log file, useful for debugging.

LogOptions.FileName: full path to the filename.

Authentication

Allows you to authenticate using **OAuth2** or **JWT**.

Asynchronous Requests

By default, the HTTP1Client uses blocking requests, so after calling an HTTP request method, the client waits for the response from the server. Alternatively, you can use asynchronous methods to execute these HTTP requests in a secondary thread, avoiding blocking the thread where the request is called. The following asynchronous methods are implemented:

- GetAsync
- PostAsync
- PutAsync
- OptionsAsync
- DeleteAsync

After calling these methods, instead of waiting for the response, the code continues to the next line, and the response can be handled using the event **OnAsyncResponse**.

```
void OnAsyncResultEvent(object Sender, TsgcHTTPAsyncRequest aRequest, TIdHTTPResponse aResponse)
```

If there is any error while processing the Asynchronous request, the exception will be raised in the event **OnAsyncException**.

Examples

Request a **GET** method to HTTPs server and using **TLS 1.2**

```
TsgcHTTP1Client oHTTP = new TsgcHTTP1Client();
oHTTP.TLSOptions.Version = tls1_2;
MessageBox.Show(oHTTP.Get("https://www.google.es"));
```

Request a **GET** method to HTTPs server using **openSSL 1.1** and **TLS 1.3**

```
TsgcHTTP1Client oHTTP = new TsgcHTTP1Client();
oHTTP.TLSOptions.OpenSSL_Options.APIVersion = oslAPI_1_1;
oHTTP.TLSOptions.Version = tls1_3;
MessageBox.Show(oHTTP.Get("https://www.google.es"));
```

Request an **Asynchronous POST** method and read the response using the **OnAsyncResultEvent**.

```
void OnAsyncExceptionEvent(object Sender, TsgcThread aThread, Exception E)
{
    Log(E.Message);
}
void OnAsyncResultEvent(object Sender, TsgcHTTPAsyncRequest aRequest, TIdHTTPResponse aResponse)
{
    if (aResponse.ResponseCode == 200)
        Log("ok", aRequest.Response);
    else
        Log("error", aRequest.Response);
}
TsgcHTTP1Client oHTTP = new TsgcHTTP1Client();
oHTTP.OnAsyncResult += OnAsyncResultEvent;
oHTTP.OnAsyncException += OnAsyncExceptionEvent;
TStringStream oRequest = new TStringStream("body");
TStringStream oResponse = new TStringStream("");
oHTTP.PostAsync("https://localhost/test", oRequest, oResponse);
```

Request a GET method to HTTPs server using **SChannel for Windows**.

```
TsgcHTTP1Client oHTTP = new TsgcHTTP1Client();
oHTTP.TLSOptions.IOHandler = iohSChannel;
oHTTP.TLSOptions.Version = tls1_2;
MessageBox.Show(oHTTP.Get("https://www.google.es"));
```

Request **SSE** method to get data events

Events

OnSSEMessage

The event is called when a new SSE message is received.

OnSSLVerifyPeer

If verify certificate is enabled, in this event you can verify and decide whether to accept the server certificate.

OnSSLGetHandler

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

OnSSLAfterCreateHandler

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

OnAsyncResult

The event is called after requesting an Async method (using GetAsync, PutAsync... methods). Use the Response parameter to know the result of the request.

OnAsyncException

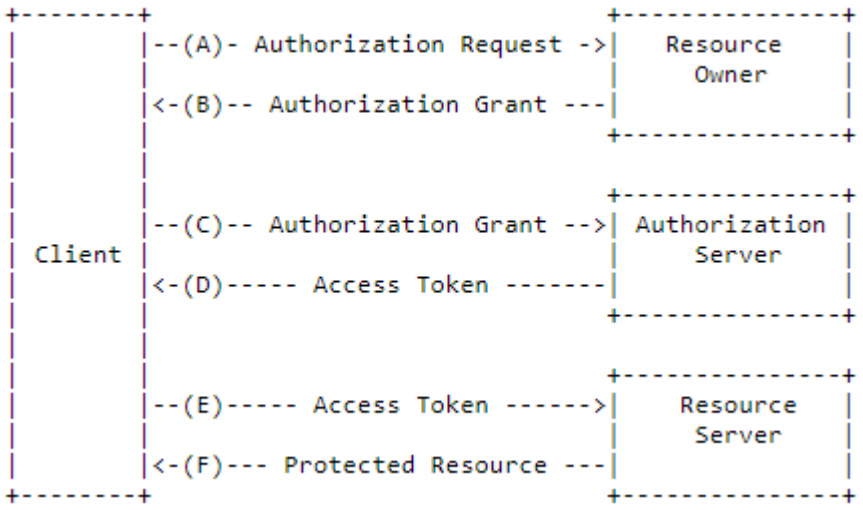
If there is any error while processing an async request, this event is called with the exception raised.

HTTP | OAuth2

OAuth2 allows third-party applications to receive limited access to an HTTP service, either on behalf of a resource owner or by allowing a third-party application to obtain access on its own behalf. Thanks to OAuth2, service providers and consumer applications can interact with each other in a secure way.

In OAuth2, there are 4 roles:

- **Resource Owner:** the user.
- **Resource Server:** the server that hosts the protected resources and provides access to it based on the access token.
- **Client:** the external application that seeks permission.
- **Authorization Server:** issues the access token after having authenticated the user.



Components

- **TsgcHTTP_OAuth2_Client:** is a client with support for OAuth2, so it can connect to OAuth2 servers to request an authentication like Google, Facebook...
- **TsgcHTTP_OAuth2_Server:** is the server implementation of OAuth2 protocol, allows you to protect the resources of the Server.

Server and **Client** OAuth2 components support **PKCE** (Proof Key for Code Exchange), which is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks (RFC 7636).

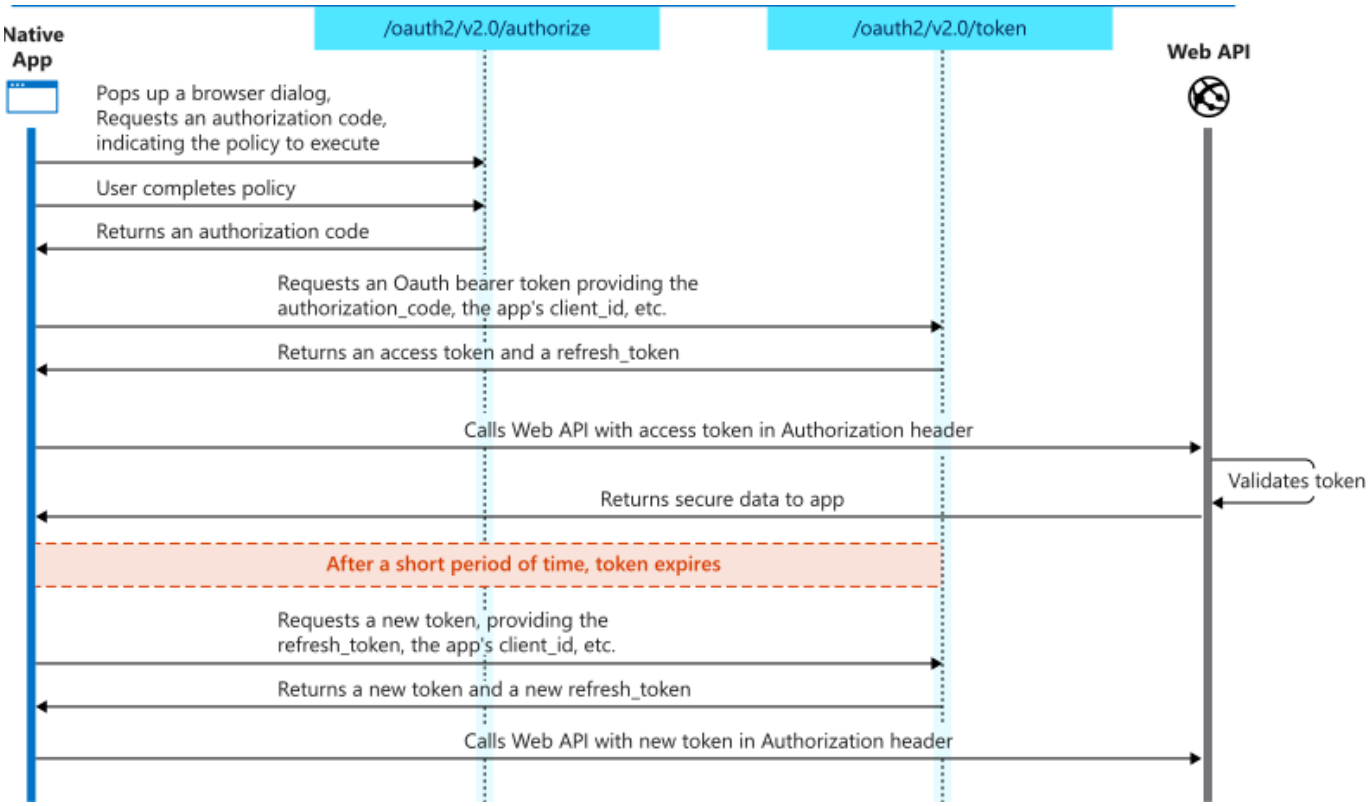
OAuth2 | TsgcHTTP_OAuth2_Client

This component allows you to handle flow between client and the other roles, basically, when you set Active := True, opens a new Web Browser and requests user grant authorization, if successful, authorization server sends a token to application which is processed and with this token, client can connect to resource server. This component, starts a simple HTTP server which handles authorization server responses and uses an HTTP client to request Access Tokens.

GrantType

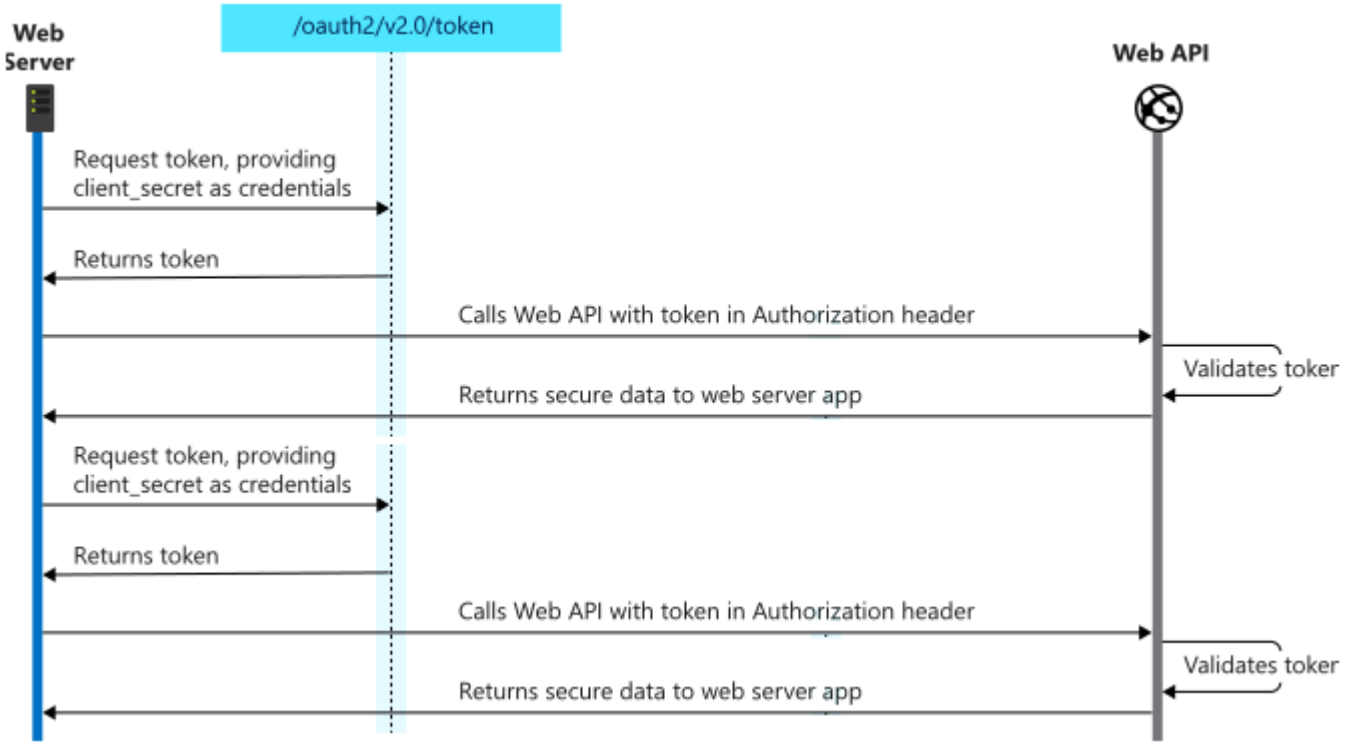
Client supports the following types of Authorization:

auth2Code: It's used to perform authentication and authorization in the majority of application types, including single page applications, web applications, and natively installed applications. The flow enables apps to securely acquire access_tokens that can be used to access resources secured, as well as refresh tokens to get additional access_tokens, and ID tokens for the signed in user.



auth2CodePKCE: it's the same authentication flow than auth2Code with PKCE enabled. PKCE (Proof Key for Code Exchange) is a security extension for OAuth 2.0, designed to enhance the security of authorization flows for native and single-page applications. It mitigates the risk of interception attacks, especially in public clients where the authorization code might be exposed to interception in transit. Usually this option is used in native and mobile apps.

auth2ClientCredentials: This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as daemons or service accounts.



auth2DeviceCode: Device Authorization Grant (RFC 8628) for input-constrained devices such as smart TVs, media consoles, and IoT devices that lack a browser or have limited input capabilities. The device displays a user code and verification URI; the user visits the URI on a secondary device (phone or computer) and enters the code to authorize.

LocalServerOptions

When a client needs a new Access Token, automatically starts an HTTP server to process response from Authorization server. This server is transparent for user and usually works in localhost. By default uses port 8080 but you can change if needed.

- **IP:** IP server listening, example: 127.0.0.1
- **Port:** by default 8080. When using GrantType = auth2CodePKCE (for desktop and native application), you can set the value of the port to zero and the server will choose a random port.
- **RedirectURL:** (optional) allows customizing the redirect URL, example: http://localhost:8080/oauth/.
- **SSL:** enable this property if local server runs on a secure port (*only supported by Professional and Enterprise Editions).
- **SSLOptions:** allows customizing the SSL properties of server (*only supported by Professional and Enterprise Editions).
- **LogOptions:** allows you to save the log of the Requests/Responses received and sent by the HTTP Internal Server (*Only for Professional and Enterprise Editions).
    - **Enabled:** set to True to enable the log to file.
    - **FileName:** set the file name to store the log file.

AuthorizationServerOptions

Here you must set URL for Authorization and Acces Token, usually these are provided in API specification. Scope is a list of all scopes requested by client. Example:

- **AuthURL:** https://accounts.google.com/o/oauth2/auth
- **TokenURL:** https://accounts.google.com/o/oauth2/token
- **Scope:** https://mail.google.com/
- **RevocationURL:** URL for token revocation endpoint (required for Revoke method). Example: https://accounts.google.com/o/oauth2/revoke
- **IntrospectionURL:** URL for token introspection endpoint (required for Introspect method). Example: https://accounts.google.com/o/oauth2/introspect

OAuth2Options

ClientId is a mandatory field which informs server which is the identification of client. Check your API specification to know how get a ClientId. The same applies for client secret.
Sometimes, server requires a user and password to connect using Basic Authentication, if this is the case, you can setup this in Username/Password fields. Example:

- **GrantType:** Authorization flow type
  - **auth2Code:** trusted apps, like a web-server.
  - **auth2CodePKCE:** untrusted native or mobile apps.
  - **auth2ClientCredentials:** automated apps without user interaction.
  - **auth2ResourceOwnerPassword:** allows an application to sign in the user by directly handling their password
  - **auth2DeviceCode:** Device Authorization Grant (RFC 8628) for input-constrained devices without a browser or with limited input capabilities.
- **ClientId:** 180803918307-eqjtm20gqfhcs6gjklbbrreng022mqqc.apps.googleusercontent.com
- **ClientSecret:** _by1iYYrvVHxC2Z8TbtNEYJN
- **Username:**
- **Password:**

HTTPClientOptions

Here you can customize the Client Options when connects to HTTP Server to request a new token.

TLSOptions: if TLS enabled, here you can customize some TLS properties.

ALPNProtocols: list of the ALPN protocols which will be sent to server.
RootCertFile: path to root certificate file.
CertFile: path to certificate file.
KeyFile: path to certificate key file.
Password: if certificate is secured with a password, set here.
VerifyCertificate: if certificate must be verified, enable this property.
VerifyDepth: is an Integer property that represents the maximum number of links permitted when verification is performed for the X.509 certificate.
Version: by default uses TLS 1.0, if server requires a higher TLS version, here can be selected.
IOHandler: select which library you will use to connect using TLS.
iohOpenSSL: uses OpenSSL library and is the default for Indy components. Requires to deploy openssl libraries for win32/win64.
iohSChannel: uses Secure Channel which is a security protocol implemented by Microsoft for Windows, doesn't require to deploy openssl libraries. Only works in Windows 32/64 bits.
OpenSSL_Options: allows defining which OpenSSL API will be used.
APIVersion: allows defining which OpenSSL API will be used.
oslAPI_1_0: uses API 1.0 OpenSSL, it's latest supported by Indy
oslAPI_1_1: uses API 1.1 OpenSSL, requires our custom Indy library and allows using OpenSSL 1.1.1 libraries (with TLS 1.3 support).
oslAPI_3_0: uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).
LibPath: here you can configure where are located the openSSL libraries
oslpNone: this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
oslpDefaultFolder: sets automatically the openSSL path where the libraries should be located for all IDE personalities.
oslpCustomFolder: if this is the option selected, define the full path in the property LibPathCustom.
LibPathCustom: when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.
UnixSymLinks: enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
oslsSymLinksDefault: by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
oslsSymLinksLoadFirst: Load SymLinks and do before trying to load the version libraries.
oslsSymLinksLoad: Load SymLinks after trying to load the version libraries.
oslsSymLinksDontLoad: don't load the SymLinks.
SChannel_Options: allows you to use a certificate from Windows Certificate Store.
CertHash: is the certificate Hash. You can find the certificate Hash running a dir command in powershell.
CertStoreName: the store name where is stored the certificate. Select one of below:

scsnMY (the default)
scsnCA
scsnRoot
scsnTrust

CertStorePath: the store path where is stored the certificate. Select one of below:
scspStoreCurrentUser (the default)
scspStoreLocalMachine

LogOptions: if a filename is set, it will save a log of HTTP requests/responses of the HTTP client

Properties

The following read-only properties provide access to the current token state:

- **AccessToken:** String (read-only). Returns the current access token obtained from the authorization server.
- **TokenType:** String (read-only). Returns the token type (e.g., 'Bearer').
- **CurrentExpiresIn:** Integer (read-only). Returns the number of seconds until the current access token expires.
- **CurrentRefreshToken:** String (read-only). Returns the current refresh token.

Methods

Revoke

Revokes an access or refresh token per RFC 7009. This method sends a revocation request to the authorization server to invalidate a previously obtained token.

Requires **AuthorizationServerOptions.RevocationURL** to be set before calling this method.

```
// Revoke the current access token
OAuth2Client.Revoke(OAuth2Client.AccessToken, "access_token");
// Revoke a refresh token
OAuth2Client.Revoke(OAuth2Client.CurrentRefreshToken, "refresh_token");
```

Introspect

Introspects a token per RFC 7662. This method queries the authorization server to determine the active state and metadata of a token.

Requires **AuthorizationServerOptions.IntrospectionURL** to be set before calling this method.

```
// Introspect the current access token
OAuth2Client.Introspect(OAuth2Client.AccessToken, "access_token");
```

Events

OnBeforeAuthorizeCode

This is the first event, it's called before client opens a new Web Browser session. URL parameter can be modified if needed (usually not necessary).

```
void OnOAuth2BeforeAuthorizeCode(TObject Sender, ref string URL, ref bool Handled)
{
  DoLog("BeforeAuthorizeCode: " + URL);
}
```

OnAfterAuthorizeCode

After a successful Authorization, server redirects the response to internal HTTP server, this response informs to client about Authorization code (which will be use later to get Access Token), state, scope...

```
void OnOAuth2AfterAuthorizeCode(TObject Sender, const string Code, const string State, const string Scope,
  const string RawParams, ref bool Handled)
{
  DoLog("AfterAuthorizeCode: " + Code);
}
```

OnErrorAuthorizeCode

If there is an error, this event will be raised with information about error.

```
void OnOAuth2ErrorAuthorizeCode(TObject Sender, const string Error, const string Error_Description,
  const string Error_URI, const string State, const string RawParams)
{
  DoLog("ErrorAuthorizeCode: " + Error + " " + Error_Description);
}
```

OnBeforeAccessToken

After get an Authorization Code, client connects to Authorization Server to request a new Access Token. Before client connects, this event is called where you can modify URL and parameters (usually not needed).

```
void OnOAuth2BeforeAccessToken(TObject Sender, ref string URL, ref string Parameters,
  ref bool Handled);
{
  DoLog("BeforeAccesToken: " + URL + " " + Parameters);
}
```

OnAfterAccessToken

If server accepts client requests, it releases a new Access Token which will be used by client to get access to resources server.

```
void OnOAuth2AfterAccessToken(TObject Sender, const string Access_Token, const string Token_Type,
  const string Expires_In, const string Refresh_Token, const string Scope, const string RawParams, ref bool Handl
{
  DoLog("AfterAccessToken: " + Access_Token + " " + Refresh_Token + " " + Expires_In);
}
```

OnErrorAccessToken

If there is an error, this event will be raised with information about error.

```
void OnOAuth2ErrorAccessToken(TObject Sender, const string Error, const string Error_Description,
  const string Error_URI, const string RawParams)
{
  DoLog("ErrorAccessToken: " + Error + " " + Error_Description);
}
```

OnBeforeRefreshToken

Access token expire after some certain time. If Authorization server releases a refresh token plus access token, client can connect after token has expires with a refresh token to request a new access token without the need of user Authenticates again with own credentials. This event is called before client requests a new access token.

```
void ONOAuth2BeforeRefreshToken(TObject Sender, ref string URL, ref string Parameters, ref bool Handled)
{
  DoLog("BeforeRefreshToken: " + URL + " " + Parameters);
}
```

OnAfterRefreshToken

If server accepts client requests, it releases a new Access Token which will be used by client to get access to resources server.

```
void OnOAuth2AfterRefreshToken(TObject Sender, const string Access_Token, const string Token_Type,
  const string Expires_In, const string Refresh_Token, const string Scope, const string RawParams, ref bool Handl
{
  DoLog("AfterRefreshToken: " + Access_Token + " " + Refresh_Token + " " + Expires_In)
}
```

OnErrorRefreshToken

If there is an error, this event will be raised with information about error.

```
void OnOAuth2ErrorRefreshToken(TObject Sender, const string Error, const string Error_Description,
  const string Error_URI, const string RawParams)
{
  DoLog("ErrorRefreshToken: " + Error + " " + Error_Description);
}
```

OnBeforeRevokeToken

Called before the client sends a token revocation request to the authorization server.

```
void OnOAuth2BeforeRevokeToken(TObject Sender, ref string URL, ref string Parameters, ref bool Handled)
{
  DoLog("BeforeRevokeToken: " + URL + " " + Parameters);
}
```

OnAfterRevokeToken

Called after the token revocation request completes successfully.

```
void OnOAuth2AfterRevokeToken(TObject Sender, const string RawParams, ref bool Handled)
{
  DoLog("AfterRevokeToken: " + RawParams);
}
```

OnErrorRevokeToken

Called when the token revocation request fails.

```
void OnOAuth2ErrorRevokeToken(TObject Sender, const string Error, const string Error_Description,
  const string Error_URI, const string RawParams)
{
  DoLog("ErrorRevokeToken: " + Error + " " + Error_Description);
}
```

OnBeforeIntrospectToken

Called before the client sends a token introspection request to the authorization server.

```
void OnOAuth2BeforeIntrospectToken(TObject Sender, ref string URL, ref string Parameters, ref bool Handled)
{
  DoLog("BeforeIntrospectToken: " + URL + " " + Parameters);
}
```

OnAfterIntrospectToken

Called after the token introspection request completes successfully. The response contains token metadata such as active state, scope, client_id, and expiration.

```
void OnOAuth2AfterIntrospectToken(TObject Sender, const string RawParams, ref bool Handled)
{
  DoLog("AfterIntrospectToken: " + RawParams);
}
```

OnErrorIntrospectToken

Called when the token introspection request fails.

```
void OnOAuth2ErrorIntrospectToken(TObject Sender, const string Error, const string Error_Description,
  const string Error_URI, const string RawParams)
{
  DoLog("ErrorIntrospectToken: " + Error + " " + Error_Description);
}
```

OnDeviceCode

Fired when a device code is received from the authorization server (Device Authorization Grant flow). Provides the UserCode that the user must enter and the VerificationURI where the user must navigate to authorize the device.

```
void OnOAuth2DeviceCode(TObject Sender, const string UserCode, const string VerificationURI)
{
  DoLog("DeviceCode - UserCode: " + UserCode + " VerificationURI: " + VerificationURI);
  // Display the UserCode and VerificationURI to the user
}
```

OnDeviceCodeExpired

Fired when the device code expires before the user completes authorization. The application should request a new device code.

```
void OnOAuth2DeviceCodeExpired(TObject Sender)
{
  DoLog("DeviceCode expired, request a new one");
}
```

OnHTTPResponse

This event is called before HTTP response is sent after a successful Access Token.

```
void OnOAuth2HTTPResponse(TObject Sender, ref int Code, ref string Text, ref bool Handled)
{
  Code = 200;
  Text = "Successful Authorization";
}
```

OAuth2 Code Example

Example of use to connect to Google Gmail API using OAuth2.

```
oAuth2 = new TsgcHTTP2_OAuth2.Create();
oAuth2.LocalServerOptions.Host = "127.0.0.1";
oAuth2.LocalServerOptions.Port = 8080;
oAuth2.AuthorizationServerOptions.AuthURL = "https://accounts.google.com/o/oauth2/auth";
oAuth2.AuthorizationServerOptions.Scope = "https://mail.google.com/";
oAuth2.AuthorizationServerOptions.TokenURL = "https://accounts.google.com/o/oauth2/token";
oAuth2.OAuth2Options->ClientId = "180803918357-eqjtn20gqfhcs6gjkebbrrenh022mqqc.apps.googleusercontent.com";
oAuth2.OAuth2Options->ClientSecret = "_by0iYYrvVHxC2Z8TbtNEYQN";
void OnOAuth2AfterAccessToken(TObject Sender, const string Access_Token, const string Token_Type,
  const string Expires_In, const string Refresh_Token, const string Scope, const string RawParams, ref bool Handl
{
// write your code here
}
oAuth2->OnAfterAccessToken = OnOAuth2AfterAccessToken;
oAuth2->Start();
```

Using TWebBrowser

You can use a TWebBrowser (if the webpage supports it) instead of regular WebBrowser like Chrome, Firefox or Edge.
Use the event **OnBeforeAuthorizeCode** to avoid opening a new WebBrowser session and use a TWebBrowser.

```
void OnBeforeAuthorizeCode(object Sender, ref string URL, ref bool Handled)
{
  Handled = true;
  WebBrowser1.Navigate(URL);
}
```

OAuth2 Client for Web Applications

When the OAuth2 Client must to get an Authorized Token connecting to a Web Application, the Local Server (used to get the authorized token) must be configured with the Web Application Parameters. Set the following **LocalServerOptions** properties:

- **IP:** Address IP previously configured in the OAuth2 Configuration.
- **Port:** Listening Port previously configured in the OAuth2 Configuration.
- **SSL:** if the server is using a secure connection, enable this option.

OAuth2 Client for Desktop Applications

When the OAuth2 Client is a Desktop or native application, the Local Server (used to get the authorized token) can be listening on a local ip address (example: 127.0.0.1) and the port can be chosen randomly. So, the **LocalServerOptions** should be configured as follows

- **IP:** 127.0.0.1
- **Port:** set to zero, and the server will choose automatically a random port.

When creating an OAuth2 Client for Desktop or Native applications, set the **OAuth2Options.GrantTypе** to **auth2CodePKCE** to add an extra security.

OAuth2 | TsgcHTTP_OAuth2_Client_Google

This component lets you login with your Google Account in an easy way.

Configuration

The module requires first **configure your OAuth2 Application** in your Google Account, once are configure just add a couple of lines in your application to allow users login with any Google Account.

The **Local Server** used to read the response from Google, by default listens on **IP Address 127.0.0.1** and **port 0** (random port). So you must configure the CallBack URL in the Google Application. Of course, you can modify the IP Address and port.

Once configured the OAuth2 Application in the Google Account, just create an instance of **TsgcHTTP_OAuth2_Client_Google** and call the method **Authenticate** passing as parameters the **Client_Id** and **Client_Secret**. This **method waits** (by default up to 60 seconds) till the user has **login successfully**. Returns an object where you can check if the user has authenticated or not, the Name, Id... and more data from the user profile.

Example

```
void GoogleSignIn()
{
  TsgcHTTP_OAuth2_Client_Google oClient = new TsgcHTTP_OAuth2_Client_Google();
  TsgcOAuth2_Google_Data oData = oClient.Authenticate("client_id", "client_secret");
  if (oData.Authenticated)
    MessageBox.Show(oData.UserProfile._Name);
}
```

TsgcHTTP_OAuth2_Client_Microsoft

This component lets you login with your Microsoft Account in an easy way.

Configuration

The module requires first **configure your OAuth2 Application** in your Microsoft Account, once are configure just add a couple of lines in your application to allow users login with any Microsoft Account.

The **Local Server** used to read the response from Microsoft, by default listens on **IP Address 127.0.0.1** and **port 8080** and uses SSL. So you must configure the CallBack URL as **https://localhost:8080** (Microsoft only allows localhost as a local IP Address) in the Microsoft Application. Of course, you can modify the IP Address and port.

Once configured the OAuth2 Application in the Microsoft Account, just create an instance of **TsgcHTTP_OAuth2_Client_Microsoft** and call the method **Authenticate** passing as parameters the **TenantId**, **Client_Id** This **method waits** (by default up to 60 seconds) till the user has **login successfully**. Returns an object where you can check if the user has authenticated or not, the Name, Id... and more data from the user profile.

Example

```
void MicrosoftSignIn()
{
  TsgcHTTPComponentClient_OAuth2_Microsoft oClient = new TsgcHTTPComponentClient_OAuth2_Microsoft();
  TsgcOAuth2_Microsoft_Data oData = oClient.Authenticate("tenant_id", "client_id", "client_secret");
  if (oData.Authenticated)
    MessageBox.Show(oData.UserProfile.DisplayName);
}
```

OAuth2 | Authorization Code Grant (RFC 6749)

Overview

The Authorization Code grant is the most common OAuth2 flow. It is designed for web applications, desktop applications, and mobile applications where the client can securely store a client secret. The flow involves redirecting the user to the authorization server, where they authenticate and grant permission. The server then returns an authorization code to a redirect URI, and the client exchanges that code for an access token.

Flow

1. Client redirects the user to the **AuthURL** with client_id, redirect_uri, response_type=code, scope, and state parameters.
2. User authenticates with the authorization server and grants permission to the client.
3. Authorization server redirects the user back to the redirect URI with a `code` parameter.
4. Client POSTs the authorization code to the **TokenURL** along with client_id, client_secret, and redirect_uri to exchange it for an access token.
5. Server returns `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2Code**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	The client secret issued by the authorization server.
`AuthorizationServerOptions.AuthURL`	The authorization endpoint URL where the user is redirected to authenticate.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL where the authorization code is exchanged for an access token.
`AuthorizationServerOptions.Scope`	The scope of the access request (e.g., openid, profile, email).
`LocalServerOptions.IP`	The IP address of the local redirect server (e.g., 127.0.0.1).
`LocalServerOptions.Port`	The port of the local redirect server (e.g., 8080).

Events

Event	Description
`OnBeforeAuthorizeCode`	Fired before the client redirects the user to the authorization endpoint. Allows customization of the authorization request.
`OnAfterAuthorizeCode`	Fired after the authorization code is received from the server.

`OnErrorAuthorizeCode`	Fired when an error occurs during the authorization code step.
`OnBeforeAccessToken`	Fired before the client exchanges the authorization code for an access token.
`OnAfterAccessToken`	Fired after the access token is received. Use this event to retrieve the token value.
`OnErrorAccessToken`	Fired when an error occurs during the token exchange step.

Example

```
OAuth2.OAuth2Options.GrantType = TsgcOAuth2GrantType.auth2Code;
OAuth2.OAuth2Options.ClientId = "your-client-id";
OAuth2.OAuth2Options.ClientSecret = "your-client-secret";
OAuth2.AuthorizationServerOptions.AuthURL = "https://provider.com/oauth2/authorize";
OAuth2.AuthorizationServerOptions.TokenURL = "https://provider.com/oauth2/token";
OAuth2.AuthorizationServerOptions.Scope.Text = "openid profile";
OAuth2.LocalServerOptions.IP = "127.0.0.1";
OAuth2.LocalServerOptions.Port = 8080;
OAuth2.Start();
```

Token Refresh

When an access token expires, you can use the refresh token to obtain a new one without requiring user interaction. Call the `Refresh` method with the refresh token value:
The `OnAfterAccessToken` event will fire with the new access token. If the refresh fails, the `OnErrorAccessToken` event will fire.

When to Use

Use the Authorization Code grant for:

- Web applications with a server-side backend that can securely store the client secret.
- Server-side rendered applications (e.g., ASP.NET, PHP, Java).
- Desktop applications where the client secret can be reasonably protected.

For public clients (native apps, SPAs) where the client secret cannot be stored securely, consider using Authorization Code with PKCE instead.

OAuth2 | Authorization Code with PKCE (RFC 7636)

Overview

The Authorization Code with PKCE (Proof Key for Code Exchange) grant extends the standard Authorization Code flow with an additional security layer. It is designed for native applications, mobile applications, and single-page applications (SPAs) where the client secret cannot be stored securely. Instead of relying on a client secret, the client generates a cryptographic code verifier and challenge that prove the entity exchanging the authorization code is the same entity that initiated the flow.

How PKCE Works

1. Client generates a random **code_verifier** (32 bytes of cryptographically random data, Base64URL encoded).
2. Client computes the **code_challenge** as the SHA-256 hash of the code_verifier, Base64URL encoded.
3. Client sends the `code_challenge` and `code_challenge_method=S256` with the authorization request.
4. After user authorization, the client sends the original `code_verifier` with the token exchange request.
5. The authorization server verifies that `SHA256(code_verifier) == code_challenge` before issuing the token.

This prevents authorization code interception attacks because an attacker who intercepts the code cannot exchange it without the original code_verifier.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2CodePKCE**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	Optional. Some providers require it even with PKCE; others do not.
`AuthorizationServerOptions.AuthURL`	The authorization endpoint URL where the user is redirected to authenticate.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL where the authorization code is exchanged for an access token.
`AuthorizationServerOptions.Scope`	The scope of the access request.
`LocalServerOptions.IP`	The IP address of the local redirect server (e.g., 127.0.0.1).
`LocalServerOptions.Port`	The port of the local redirect server. Set to **0** for a random available port.

Example

```
OAuth2.OAuth2Options.GrantType = TsgcOAuth2GrantType.auth2CodePKCE;
OAuth2.OAuth2Options.ClientId = "your-client-id";
OAuth2.AuthorizationServerOptions.AuthURL = "https://provider.com/oauth2/authorize";
OAuth2.AuthorizationServerOptions.TokenURL = "https://provider.com/oauth2/token";
OAuth2.AuthorizationServerOptions.Scope.Text = "openid profile";
OAuth2.LocalServerOptions.IP = "127.0.0.1";
```

```
OAuth2.LocalServerOptions.Port = 0;
OAuth2.Start();
```

Random Port

When `LocalServerOptions.Port` is set to **0**, the component automatically selects a random available port. This is recommended for desktop and mobile applications because it avoids port conflicts. The selected port is included in the redirect URI sent to the authorization server.

Token Refresh

PKCE flows typically return a refresh token. Use the `Refresh` method to obtain a new access token:

When to Use

Use the Authorization Code with PKCE grant for:

- Desktop applications (native Windows, macOS, Linux).
- Mobile applications (iOS, Android).
- Single-page applications (SPAs) running in the browser.
- Any public client that cannot securely store a client secret.
- As a more secure alternative to the standard Authorization Code grant, even for confidential clients.

OAuth2 | Client Credentials Grant (RFC 6749)

Overview

The Client Credentials grant is used for machine-to-machine (M2M) authentication where no user interaction is required. The client authenticates directly with the authorization server using its own credentials (client_id and client_secret) and receives an access token. There is no authorization code step and no user login.
This grant type is appropriate when the client is acting on its own behalf rather than on behalf of a user.

Flow

1. Client POSTs `client_id`, `client_secret`, `grant_type=client_credentials`, and `scope` to the token endpoint.
2. Authorization server validates the client credentials.
3. Server returns `access_token`, `token_type`, and `expires_in`.

Note that refresh tokens are typically not issued with this grant type, since the client can simply request a new access token using its credentials.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2ClientCredentials**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	The client secret issued by the authorization server.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL. No AuthURL is needed since there is no user authorization step.
`AuthorizationServerOptions.Scope`	The scope of the access request.

Example

```
OAuth2.OAuth2Options.GrantType = TsgcOAuth2GrantType.auth2ClientCredentials;
OAuth2.OAuth2Options.ClientId = "your-client-id";
OAuth2.OAuth2Options.ClientSecret = "your-client-secret";
OAuth2.AuthorizationServerOptions.TokenURL = "https://provider.com/oauth2/token";
OAuth2.AuthorizationServerOptions.Scope.Text = "api.read api.write";
OAuth2.Start();
```

When to Use

Use the Client Credentials grant for:

- Backend services and daemons that run without user interaction.
- Server-to-server API communication.
- Scheduled jobs, batch processes, and cron tasks.
- Microservice-to-microservice authentication.
- Any scenario where the application acts on its own behalf, not on behalf of a user.

OAuth2 | Resource Owner Password Credentials Grant (RFC 6749)

Overview

The Resource Owner Password Credentials (ROPC) grant allows the client to collect the user's username and password directly and exchange them for an access token. The user's credentials are sent to the token endpoint, and the server returns an access token if they are valid.
This grant type should only be used when there is a high degree of trust between the resource owner and the client, such as when the client is a first-party application developed by the same organization that operates the authorization server.

Security Warning

This grant type is considered less secure than other OAuth2 flows. It exposes the user's credentials directly to the client application, which violates the principle of delegated authorization. The OAuth 2.1 specification has deprecated this grant type. Use Authorization Code with PKCE instead whenever possible.

Flow

1. User provides username and password directly to the client application.
2. Client POSTs `username`, `password`, `client_id`, `client_secret`, `grant_type=password`, and `scope` to the token endpoint.
3. Authorization server validates the credentials.
4. Server returns `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2ResourceOwnerPassword**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	The client secret issued by the authorization server.
`OAuth2Options.UserName`	The resource owner's username.
`OAuth2Options.Password`	The resource owner's password.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL. No AuthURL is needed since user credentials are provided directly.
`AuthorizationServerOptions.Scope`	The scope of the access request.

Example

```
OAuth2.OAuth2Options.GrantType = TsgcOAuth2GrantType.auth2ResourceOwnerPassword;
OAuth2.OAuth2Options.ClientId = "your-client-id";
OAuth2.OAuth2Options.ClientSecret = "your-client-secret";
OAuth2.OAuth2Options.UserName = "user@example.com";
OAuth2.OAuth2Options.Password = "user-password";
```

```
OAuth2.AuthorizationServerOptions.TokenURL = "https://provider.com/oauth2/token";
OAuth2.AuthorizationServerOptions.Scope.Text = "openid profile";
OAuth2.Start();
```

Token Refresh

If the server returns a refresh token, you can use it to obtain a new access token without re-entering the user's credentials:

When to Use

Use the Resource Owner Password Credentials grant only for:

- Legacy applications that cannot be migrated to a more secure flow.
- First-party applications where you control both the client and the authorization server.
- Migration scenarios where you are transitioning from direct username/password authentication to OAuth2.

For all other cases, use Authorization Code with PKCE or Authorization Code instead.

OAuth2 | Device Authorization Grant (RFC 8628)

Overview

The Device Authorization grant (also known as Device Code flow) is designed for input-constrained devices that cannot easily display a browser or accept keyboard input. Examples include smart TVs, IoT devices, media players, CLI tools, and kiosks. The device displays a short code that the user enters on another device (such as a phone or computer) to complete the authorization.

Flow

1. Client POSTs `client_id` and `scope` to the **DeviceAuthorizationURL**.
2. Server returns `device_code`, `user_code`, `verification_uri`, `expires_in`, and `interval`.
3. Client fires the **OnDeviceCode** event with the user code and verification URI. The application displays these to the user (e.g., "Go to https://provider.com/device and enter code: ABCD-1234").
4. User navigates to the verification URI on another device and enters the user code.
5. User authenticates and grants permission on the secondary device.
6. Meanwhile, the client automatically polls the token endpoint every `interval` seconds using the device code.
7. When the user completes authorization, the next poll returns the `access_token`.
8. If the device code expires before the user authorizes, the **OnDeviceCodeExpired** event fires.

Configuration

Property	Description
`OAuth2Options.GrantType`	Set to **auth2DeviceCode**.
`OAuth2Options.ClientId`	The client identifier issued by the authorization server.
`OAuth2Options.ClientSecret`	Optional. Some providers require it, others do not.
`AuthorizationServerOptions.TokenURL`	The token endpoint URL used for polling.
`AuthorizationServerOptions.DeviceAuthorizationURL`	The device authorization endpoint where the client requests a device code.
`AuthorizationServerOptions.Scope`	The scope of the access request.

Events

Event	Description
`OnDeviceCode`	Fired when the device code and user code are received from the server. Display the user code and verification URI to the user.
`OnDeviceCode-Expired`	Fired when the device code expires before the user completes authorization. The application should prompt the user to restart the flow.

`OnAfterAccessToken`	Fired when the user completes authorization and the access token is received.
`OnErrorAccessToken`	Fired when a non-recoverable error occurs during the device flow.

Example

Handling the Device Code Event

```
OAuth2.OAuth2Options.GrantType = TsgcOAuth2GrantType.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.OnDeviceCode += (sender, userCode, verificationUri) =>
{
    Console.WriteLine("Go to " + verificationUri + " and enter code: " + userCode);
};
OAuth2.Start();
```

Polling Behavior

After calling `Start`, the component automatically polls the token endpoint at the interval specified by the server (typically 5 seconds). If the server responds with a `slow_down` error, the component increases the polling interval by 5 seconds. Polling continues until one of the following occurs:

- The user completes authorization and the token is returned.
- The device code expires (`OnDeviceCodeExpired` fires).
- An unrecoverable error occurs (`OnErrorAccessToken` fires).

When to Use

Use the Device Authorization grant for:

- Smart TVs and streaming devices.
- Command-line interface (CLI) tools.
- IoT devices with limited input capability.
- Kiosks and digital signage.
- Any device where typing a full URL or credentials is impractical.

OAuth2 | DPoP - Demonstrating Proof of Possession (RFC 9449)

Overview

DPoP (Demonstrating Proof of Possession) is a security mechanism that binds access tokens to a specific client by requiring proof of possession of a private key. Unlike bearer tokens, which can be used by anyone who obtains them, DPoP-bound tokens are useless without the corresponding private key. This provides protection against token theft and replay attacks.
DPoP is not a grant type itself but a security enhancement that can be used with any OAuth2 grant type (Authorization Code, Client Credentials, etc.).

How It Works

1. Client generates an asymmetric key pair (ES256 or RS256).
2. Client creates a **DPoP proof JWT** with the following structure:
   - Header: `typ: dpop+jwt`, `alg: ES256`, `jwk: {public key}`
   - Payload: `htm` (HTTP method), `htu` (HTTP URI), `iat` (issued at), `jti` (unique identifier)
3. Client sends the DPoP proof as an HTTP header (`DPoP: <proof>`) with the token request.
4. Authorization server binds the issued token to the JWK thumbprint of the client's public key.
5. Server returns `token_type: DPoP` instead of `token_type: Bearer`.
6. For subsequent API calls, the client sends both the `Authorization: DPoP <token>` header and a new `DPoP: <proof>` header specific to that request.

Key Generation

You can generate an ES256 key pair directly in Delphi using the **GenerateDPoPKeyPair** method. This uses OpenSSL internally to create the key pair and automatically sets both `PrivateKey` and `PublicKeyJWK`:

```
OAuth2.DPoPOptions.Enabled = true;
OAuth2.DPoPOptions.Algorithm = TsgcHTTPOAuth2_DPoP_Algorithm.dpopES256;
OAuth2.GenerateDPoPKeyPair(); // Generates PrivateKey + PublicKeyJWK automatically
OAuth2.Start();
```

PublicKeyToJWK — Convert Any Key or Certificate to JWK

If you already have a public key, private key, or X.509 certificate (PEM format), you can convert it to JWK using the **PublicKeyToJWK** class function. It accepts any of these inputs and returns the correct JWK JSON:

Input	PEM Header
EC or RSA public key	`-----BEGIN PUBLIC KEY-----`
EC private key	`-----BEGIN EC PRIVATE KEY-----`
RSA private key	`-----BEGIN RSA PRIVATE KEY-----`
X.509 certificate	`-----BEGIN CERTIFICATE-----`

From a PEM key file

```
// From a public key file
OAuth2.DPoPOptions.PublicKeyJWK =
  OAuth2.PublicKeyToJWK(
```

```
    File.ReadAllText("public_key.pem"));

// From a private key file (extracts the public part)
OAuth2.DPoPOptions.PublicKeyJWK =
  OAuth2.PublicKeyToJWK(
    File.ReadAllText("private_key.pem"));
```

From a smart card certificate

Smart cards and HSMs protect the private key — it cannot be read or exported. However, the **certificate** (which contains the public key) is always exportable. Export the certificate as PEM and pass it to `PublicKeyToJWK`:

```
// Export the certificate from the smart card
string certPEM = mySmartCard.ExportCertificatePEM();

// Convert the certificate to JWK
OAuth2.DPoPOptions.PublicKeyJWK =
  OAuth2.PublicKeyToJWK(certPEM);

// No PrivateKey needed - use OnDPoPSign for signing via the card
OAuth2.DPoPOptions.Enabled = true;
OAuth2.OnDPoPSign += MyDPoPSignHandler;
```

This approach works with any card or HSM that can export its certificate: PKCS#11 tokens, Windows CNG smart cards, USB security keys (YubiKey, Feitian), and cloud HSMs (Azure Key Vault, AWS CloudHSM).

Using OpenSSL command-line tools

You can also generate a key pair externally:

DPoP Options

Property	Description
`DPoPOptions.Enabled`	Set to **True** to enable DPoP proof generation.
`DPoPOptions.Algorithm`	The signing algorithm. Supported values: **ES256** (recommended), **RS256**.
`DPoPOptions.PrivateKey`	The PEM-encoded private key used to sign the DPoP proof JWT.
`DPoPOptions.PublicKeyJWK`	The public key in JWK (JSON Web Key) format, included in the DPoP proof header.

Methods

Method	Description
`GenerateDPoPKeyPair`	Generates an ES256 key pair using OpenSSL and sets both `DPoPOptions.PrivateKey` (PEM) and `DPoPOptions.PublicKeyJWK` (JSON) automatically.
`GetDPoPProof(Method, URL, AccessToken)`	Generates a DPoP proof JWT for the given HTTP method, URL, and access token. Returns the signed JWT string. The proof includes the `ath` claim (access token hash) to bind the proof to the token.
`GetDPoPJWKThumbprint`	Returns the RFC 7638 JWK thumbprint (SHA-256) of the public key, used by the server to bind the token.

Nonce Handling

Some authorization servers require a server-provided nonce in the DPoP proof. When the server responds with a `DPoP-Nonce` header and a 400 or 401 status, the component automatically:

1. Extracts the nonce from the `DPoP-Nonce` response header.
2. Includes the nonce in the `nonce` claim of the next DPoP proof JWT.
3. Retries the request with the updated proof.

Custom Signing with OnDPoPSign (HSM / Smart Card)

In some environments, the private key is stored on a hardware security module (HSM) or smart card (e.g., via PKCS#11 or Windows CNG) and **cannot be extracted**. The `OnDPoPSign` event allows you to perform the signing externally while the component handles everything else (proof structure, nonce, headers).
When `OnDPoPSign` is assigned, the component calls it before attempting internal signing. If you set `Handled := True` and provide the `Signature`, the component uses your signature and skips the internal signing. The `DPoPOptions.PrivateKey` property is not needed in this case.

Event Signature

Parameter	Direction	Description
`SigningInput`	in	The data to sign: `Base64URL(header) + "." + Base64URL(payload)` . This is the standard JWS signing input.
`Algorithm`	in	The algorithm name: `"ES256"` or `"RS256"`.
`Signature`	out	Set this to the Base64URL-encoded signature bytes produced by your signing device.
`Handled`	out	Set to `True` to use your custom signature. If `False`, the component falls back to internal signing using `PrivateKey`.

Smart Card Example

The following example shows how to use DPoP with a signing card. The private key never leaves the card — only the hash is sent to the card for signing.

```
// Configure DPoP - no PrivateKey needed when using OnDPoPSign
OAuth2.DPoPOptions.Enabled = true;
OAuth2.DPoPOptions.Algorithm = TsgcHTTPOAuth2_DPoP_Algorithm.dpopES256;
OAuth2.DPoPOptions.PublicKeyJWK =
  "{\"kty\":\"EC\",\"crv\":\"P-256\",\"x\":\"...\",\"y\":\"...\"}";
OAuth2.OnDPoPSign += OnDPoPSignHandler;
OAuth2.Start();

private void OnDPoPSignHandler(TsgcHTTP_OAuth2_Client Sender,
  string SigningInput, string Algorithm,
  ref string Signature, ref bool Handled)
{
  // Hash and sign with smart card
  byte[] hash = SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(SigningInput));
  byte[] sig = mySmartCard.SignHash(hash);
  Signature = Base64UrlEncode(sig);
  Handled = true;
}
```

This approach works with any signing backend: PKCS#11 tokens, Windows CNG (CryptoAPI Next Generation), Azure Key Vault, AWS CloudHSM, or any custom hardware that can produce ECDSA/RSA signatures.

Example

```
// Configure the grant type
OAuth2.OAuth2Options.GrantType = TsgcOAuth2GrantType.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 = File.ReadAllText("dpop_private.pem");
OAuth2.DPoPOptions.PublicKeyJWK = "{\"kty\":\"EC\",\"crv\":\"P-256\",\"x\":\"...\",\"y\":\"...\"}";
OAuth2.Start();
// Generate DPoP proof for API calls
string proof = OAuth2.GetDPoPProof("GET", "https://api.provider.com/resource");
// Use: Authorization: DPoP <access_token>
// Use: DPoP: <proof>
```

OAuth2 | TsgcHTTP_OAuth2_Server

This component provides the OAuth2 protocol implementation in Server Side Components.
The server components have a property called Authorization.OAuth.OAuth2 where you can assign an instance of TsgcHTTP_OAuth2_Server, so if Authentication is enabled and OAuh2 property is attached to OAuth2 Server Component, the WebSocket and HTTP Requests require a Bearer Token to be processed, if not the connection will be closed automatically.

```
OAuth2 = new TsgcHTTP_OAuth2_Server();
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2;
```

The server supports the following authorization types:

- **auth2Code:** It's used to perform authentication and authorization in the majority of application types, including single page applications, web applications, and natively installed applications. The flow enables apps to securely acquire access_tokens that can be used to access resources secured, as well as refresh tokens to get additional access_tokens, and ID tokens for the signed in user.
- **auth2ClientCredentials:** This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of applications are often referred to as daemons or service accounts.
- **password** (Resource Owner Password Credentials): Allows an application to sign in the user by directly handling their credentials. The client sends the user's username and password to the token endpoint.
- **urn:ietf:params:oauth:grant-type:device_code** (Device Code): Device Authorization Grant per RFC 8628. Enables input-constrained devices (smart TVs, IoT devices) to obtain user authorization by having the user authorize on a secondary device.

The Authorization type can be customized when registering the App, by default, all authorization types are supported.

EndPoints

By default, the component is configured with the following endpoints to handle Authorization and Token request
Authorization: /sgc/oauth2/auth
Token: /sgc/oauth2/token
Revocation: /sgc/oauth2/revoke
Introspection: /sgc/oauth2/introspect
Device Authorization: /sgc/oauth2/device
Device Verification: /sgc/oauth2/device/verify
So if server is listening on port 443 and domain is www.esegece.com, the EndPoints will be:
Authorization: https://www.esegece.com/sgc/oauth2/auth
Token: https://www.esegece.com/sgc/oauth2/token
Revocation: https://www.esegece.com/sgc/oauth2/revoke
Introspection: https://www.esegece.com/sgc/oauth2/introspect
Device Authorization: https://www.esegece.com/sgc/oauth2/device
Device Verification: https://www.esegece.com/sgc/oauth2/device/verify
The endpoints can be configured in OAuth2Options property.
By default, **PKCE** (is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks) is enabled.

Configuration

Before you can begin the OAuth2 process, you must register which Apps will be available, this is done using Apps property of OAuth2 server component.

Register App

Use Apps.AddApp to add a new Application to OAuth2 server, you must set the following parameters:

- **App Name:** is the name of the Application. Example: MyApp
- **RedirectURI:** is where the responses will be redirected. Example: http://127.0.0.1:8080

- **ClientId:** is public information and is the ID of the client.
- **ClientSecret:** must be kept confidential.

Optionally you can set the following parameters:

- **ExpiresIn:** by default is 3600 seconds, so the token will expire in 1 hour, you can set a greater value if you need.
- **RefreshToken:** by default refresh tokens are supported, if not, set this parameter to false.
- **AllowedGrantTypes:** by default all grant types are supported (auth2Code and auth2ClientCredentials), but the server can be configured to only allow Code Authorization or only Client Credentials.

Delete App

Use Apps.RemoveApp to delete an existing App.

AddToken

If the server has been restarted while there were some token issued, you can recover these tokens using the method AddToken before starting the OAuth2 Server and after registering the Apps

- **AppName:** the name of the application.
- **Token:** access token.
- **Expires:** when the token expires.
- **RefreshToken:** refresh token.

RemoveToken

Removes an already issued Token.

OAuth2Options

The OAuth2Options property allows configuring the server endpoints and optional features.

Revocation

Token revocation per RFC 7009. When enabled, clients can revoke previously issued access or refresh tokens.

- **OAuth2Options.Revocation.Enabled:** set to True to enable the revocation endpoint.
- **OAuth2Options.Revocation.URL:** the endpoint URL path. Default: /sgc/oauth2/revoke

Introspection

Token introspection per RFC 7662. When enabled, resource servers can query the authorization server to determine the active state and metadata of a token.

- **OAuth2Options.Introspection.Enabled:** set to True to enable the introspection endpoint.
- **OAuth2Options.Introspection.URL:** the endpoint URL path. Default: /sgc/oauth2/introspect

DeviceAuthorization

Device Authorization Grant per RFC 8628. When enabled, input-constrained devices can request authorization by having the user authorize on a secondary device.

- **OAuth2Options.DeviceAuthorization.Enabled:** set to True to enable the device authorization endpoint.
- **OAuth2Options.DeviceAuthorization.URL:** the endpoint URL path for device code requests. Default: /sgc/oauth2/device
- **OAuth2Options.DeviceAuthorization.VerificationURL:** the endpoint URL path for the user verification page. Default: /sgc/oauth2/device/verify
- **OAuth2Options.DeviceAuthorization.ExpiresIn:** the lifetime in seconds of the device code. Default: 600 (10 minutes).
- **OAuth2Options.DeviceAuthorization.Interval:** the minimum polling interval in seconds that the client should use when polling the token endpoint. Default: 5.

Most common uses

- **QuickStart**
  - OAuth2 Server Example
  - OAuth2 Customize Sign-in HTML
  - OAuth2 Server Endpoints
  - OAuth2 Register Apps
  - OAuth2 Recover Access Tokens
- **Authenticate**
  - OAuth2 Server Autentication
  - OAuth2 None Authenticate some URLs

Connections

While OAuth2 is enabled on Server-side, if a websocket client tries to connect without providing a valid Token, the connection will be closed automatically. The same applies to HTTP requests.
TsgcWebSocketClient can be configured to request a OAuth2 token and sent when connects to server. You have 2 options in order to send a Bearer Token:
1. Use Authentication.Token property, this is usefull when you have a valid token obtained from an external third-party and you only want to pass as a connection header to get Access to server.

```
Authorization.Enabled = true;
Authorization.Token.Enabled = true;
Authorization.Token.AuthName = "Bearer";
Authorization.Token.AuthToken = "your token here";
```

2. Attach a TsgcHTTP_OAuth2_Client and let the client request an Access Token and send it automatically when websocket client connects to server.

Events

Some events are provided to handle the OAuth2 Flow Control.

OnOAuth2BeforeRequest

This event is called when a new HTTP connection is established with server and before checks if the connection request is trying to do an Authorization or request a new token. If you don't need that this request is processed by OAuth2 server, set Cancel parameter to true.
The event is called too when checks if the Token is valid.

OnOAuth2BeforeDispatchPage

The event is called before the Authorization web-page is showed to user, allows customizing the HTML code shown to user.

OnOAuth2Authentication

When a client request Authorization, server shows a page were user can allow connection and requires to login to server. This is the event where you can read the User/Password set by user and accept or not the connection.

OnOAuth2AfterAccessToken

After the server process successfully the Access Token, this event is called. Useful for log purposes.

OnOAuth2AfterRefreshToken

After the server process successfully the Refresh Token, this event is called. Useful for log purposes.

OnOAuth2AfterValidateAccessToken

When a client do a request with a Token, this token is processed by server to check if it's valid or not, if the token is valid and not expired, this event is called. Useful for log purposes.

OnOAuth2Unauthorized

This event is called before the connection is closed because there is no authorization token or is invalid, by default, the Disconnect parameter is true, you can set to false if you still want to accept the connection. This event can configure which endpoints must implement OAuth2 Authorization or not.

OnOAuth2AfterRevokeToken

Called after a token revocation attempt. This event is fired when a client sends a request to the revocation endpoint to invalidate a previously issued token. Useful for logging revocation activity.

OnOAuth2AfterIntrospectToken

Called after a token introspection request. This event is fired when a resource server queries the introspection endpoint to check the active state and metadata of a token. Useful for logging and auditing token validation.

OnOAuth2DeviceAuthorization

Called when a device authorization request is received at the device authorization endpoint. The device is requesting a device code and user code pair. This event allows customizing the response or logging the request.

OnOAuth2DeviceCodeVerification

Called when a user submits a verification code on the device verification page. This event allows the server to validate the user code entered by the user and authorize or deny the device.

DPoP (RFC 9449) - Server-Side Support

DPoP (Demonstrating Proof-of-Possession) per RFC 9449 enables the server to require sender-constrained tokens, ensuring that access tokens can only be used by the client that originally obtained them.

OAuth2Options.DPoP

- **OAuth2Options.DPoP:** Boolean. When set to True, enables DPoP-bound token validation. The server will require a valid DPoP proof header on resource requests that use DPoP-bound tokens.

OnOAuth2ValidateDPoP

This event is fired when a resource request includes a DPoP proof header and the server needs to validate it. Use this event to implement custom DPoP proof validation logic, such as verifying the proof signature, checking the token binding (jkt claim), and validating the proof claims (htm, htu, iat, ath).

```
void OnOAuth2ValidateDPoP(TObject Sender, const string DPoPProof, const string AccessToken,
  const string Method, const string URL, ref bool Valid)
{
  // Custom DPoP proof validation
  Valid = true; // Set to false to reject the request
}
```

Bug Fixes

- Fixed **SetOAuth2Options** memory leak that could occur when reassigning OAuth2 options.

OAuth2 | Server Example

Let's do a simple OAuth2 server example, using a TsgcWebSocketHTTPServer.

First, create a new TsgcWebSocketHTTPServer listening on port 443 and using a self-signed certificate in sgc.pem file.

```
oServer = new TsgcWebSocketHTTPServer();
oServer.Port = 80;
oServer.SSLOptions.Port = 443;
oServer.SSLOptions.CertFile = "sgc.pem";
oServer.SSLOptions.KeyFile = "sgc.pem";
oServer.SSLOptions.RootCertFile = "sgc.pem";
oServer.SSL = true;
```

Then create a new instance of TsgcHTTP_OAuth2_Server and assign to previously created server.
Register a new Application with the following values:

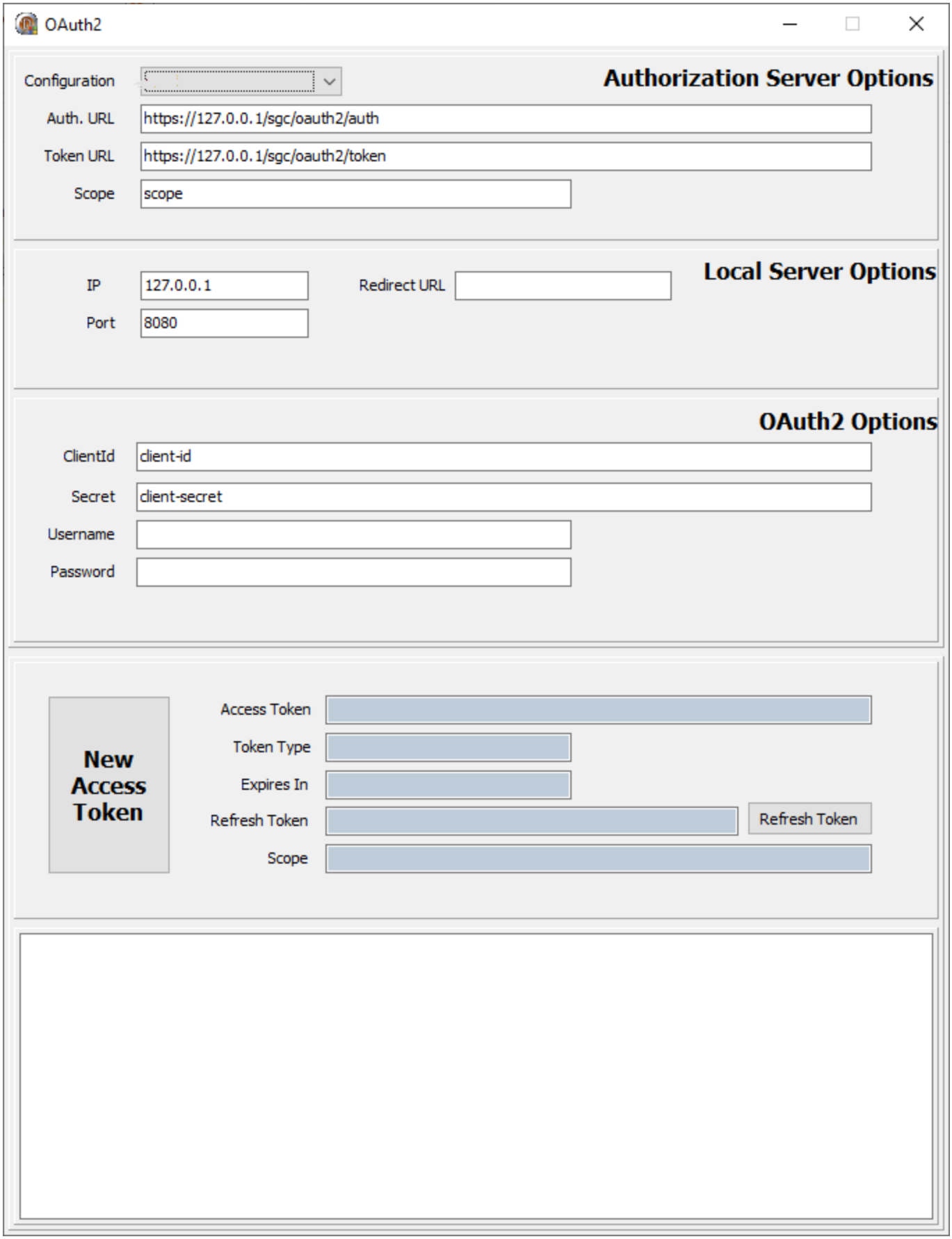
Name: MyApp
RedirectURI: http://127.0.0.1:8080
ClientId: client-id
ClientSecret: client-secret

```
OAuth2 = new TsgcHTTP_OAuth2_Server.Create();
OAuth2.Apps.AddApp("MyApp", "http://127.0.0.1:8080", "client-id", "client-secret");
oServer.Authentication.Enabled = true;
oServer.Authentication.OAuth.OAuth2 = OAuth2;
```

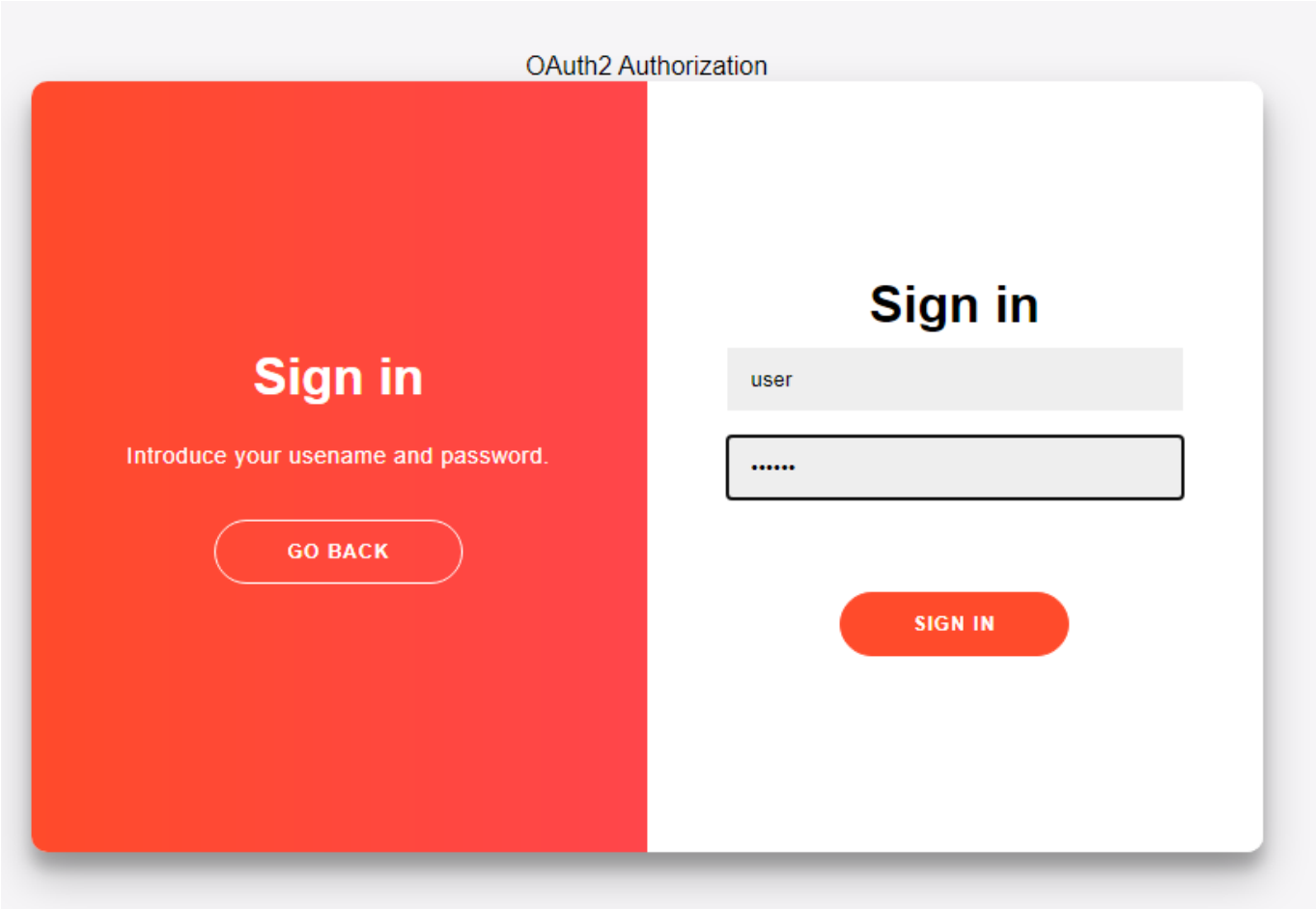
Then handle OnOAuth2Authentication event of OAuth2 server component and implement your own method to login users. I will use the pair "user/secret" to accept a login.

```
void OnOAuth2Authentication(TsgcWSConnection Connection, TsgcHTTPOAuth2Request OAuth2, string aUser,
  string aPassword, ref bool Authenticated)
{
  if ((aUser == "user") and (aPassword == "secret"))
  {
    Authenticated = true;
  }
}
```

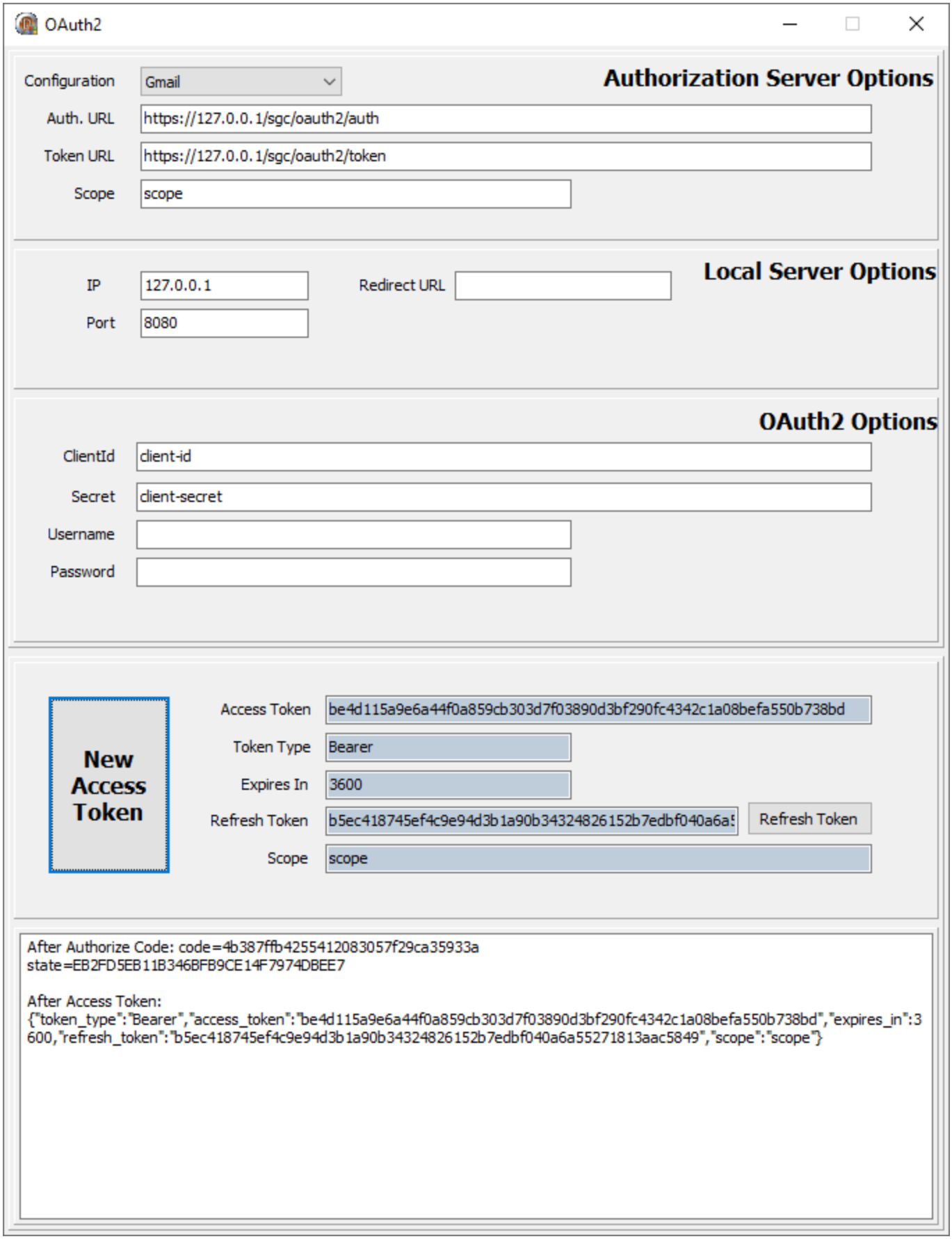
Finally start the server and use a OAuth2 client to login, example you can use the TsgcHTTP_OAuth2_Client included with sgcWebSockets library.



Request a New Access Token, a new Web Browser session will be shown and user must Allow connection and then login.



If login is successful a new Token will be returned to the client. Then all the requests must include this bearer token in the HTTP Headers.


OAuth2
Configuration
Gmail
Authorization Server Options
Auth. URL
https://127.0.0.1/sgc/oauth2/auth
Token URL
https://127.0.0.1/sgc/oauth2/token
Scope
scope
Local Server Options
IP
127.0.0.1
Redirect URL
Port
8080
OAuth2 Options
ClientId
client-id
Secret
client-secret
Username
Password
New Access Token
Access Token
be4d115a9e6a44f0a859cb303d7f03890d3bf290fc4342c1a08befa550b738bd
Token Type
Bearer
Expires In
3600
Refresh Token
b5ec418745ef4c9e94d3b1a90b34324826152b7edbf040a6a5
Refresh Token
Scope
scope
After Authorize Code: code=4b387ffb4255412083057f29ca35933a
state=EB2FD5EB11B346BFB9CE14F7974DBEE7
After Access Token:
{"token_type":"Bearer","access_token":"be4d115a9e6a44f0a859cb303d7f03890d3bf290fc4342c1a08befa550b738bd","expires_in":3600,"refresh_token":"b5ec418745ef4c9e94d3b1a90b34324826152b7edbf040a6a55271813aac5849","scope":"scope"}

OAuth2 | Customize Sign-In HTML

When an OAuth2 client do a request to get a new Access Token, a Web-Page is shown in a web-browser to Allow this connection and login with an User and Password.
The HTML page is included by default in Server component, but this code can be customized using **OnAuth2BeforeDispatchPage** event.

```
void OnOAuth2BeforeDispatchPage(TObject Sender; TsgcHTTPOAuth2Request OAuth2; ref string HTML)
{
  HTML = "your custom html";
}
```

If you customize your HTML with a completely new HTML code, at least you must maintain the form where the Username and password are sent:

```
<form action="">
<input type="hidden" name="request_type" value="sign-in" />
<input type="username" name="username" placeholder="Username" />
<input type="password" name="password" placeholder="Password" />
<input type="hidden" name="id" value="" />
<p></p>
<button>Sign In</button>
</form>
```

The id parameter, which is hidden, must maintain the same value of the original form to allow server identify the request.

OAuth2 | Server Endpoints

By default, the OAuth2 Server uses the following Endpoints:
Authorization: /sgc/oauth2/auth
Token: /sgc/oauth2/token
Revocation: /sgc/oauth2/revoke (POST) - Revokes tokens per RFC 7009
Introspection: /sgc/oauth2/introspect (POST) - Returns token metadata per RFC 7662
Device Authorization: /sgc/oauth2/device (POST) - Issues device codes per RFC 8628
Device Verification: /sgc/oauth2/device/verify (GET/POST) - User verification page
Which means that if your server listens on IP 80.54.41.30 and port 8443, the full OAuth2 Endpoints will be:
Authorization: https://80.54.41.30:8443/sgc/oauth2/auth
Token: https://80.54.41.30:8443/sgc/oauth2/token
Revocation: https://80.54.41.30:8443/sgc/oauth2/revoke
Introspection: https://80.54.41.30:8443/sgc/oauth2/introspect
Device Authorization: https://80.54.41.30:8443/sgc/oauth2/device
Device Verification: https://80.54.41.30:8443/sgc/oauth2/device/verify
This Endpoints can be modified easily, just access to OAuth2Options property of component and modify Authorization and Token URLs.
Example: if your endpoints must be
Authorization: https://80.54.41.30:8443/authentication/auth
Token: https://80.54.41.30:8443/authentication/token
Set the OAuth2Options property with the following values:
OAuth2Options.Authorization.URL = /authentication/auth
OAuth2Options.Token.URL = /authentication/token
The same approach applies to the other endpoints:
OAuth2Options.Revocation.URL = /authentication/revoke
OAuth2Options.Introspection.URL = /authentication/introspect
OAuth2Options.DeviceAuthorization.URL = /authentication/device
OAuth2Options.DeviceAuthorization.VerificationURL = /authentication/device/verify

OAuth2 | Register Apps

Before a new OAuth2 is requested by a client, the App must be registered in the server.
Register a new App requires the following information:

- **App Name:** is the name of the Application. Example: MyApp
- **RedirectURI:** is where the responses will be redirected. Example: http://127.0.0.1:8080
- **ClientId:** is public information and is the ID of the client.
- **ClientSecret:** must be kept confidential.

Optionally you can set the following parameters:

- **ExpiresIn:** by default is 3600 seconds, so the token will expire in 1 hour, you can set a greater value if you need.
- **RefreshToken:** by default refresh tokens are supported, if not, set this parameter to false.

If a new client wants to authenticate using OAuth2, first the App requires to be registered in the server, you can use:

1. RegisterApp
This method requires the App Name and RedirectURI, and will return a ClientId and ClientSecret.

2. Apps.AddApp
This method requires AppName, RedirectURI, ClientId and ClientSecret. Usually you can use this method when a server has some already created Apps and you want to load them before is started.

Both methods do the same, register the Application in the server, but first is most useful when the App is registered the first time and second method when you want to load all registered apps before start the server (because are saved on database for example).

OAuth2 | Recover Access Tokens

If the OAuth2 Server is destroyed (because it's restarted) and there are valid Access Tokens issued, these are lost by default. You can recover these Access Tokens using the method **AddToken**. This method stores the tokens in the OAuth2 Server.

Add a Token requires the following information:

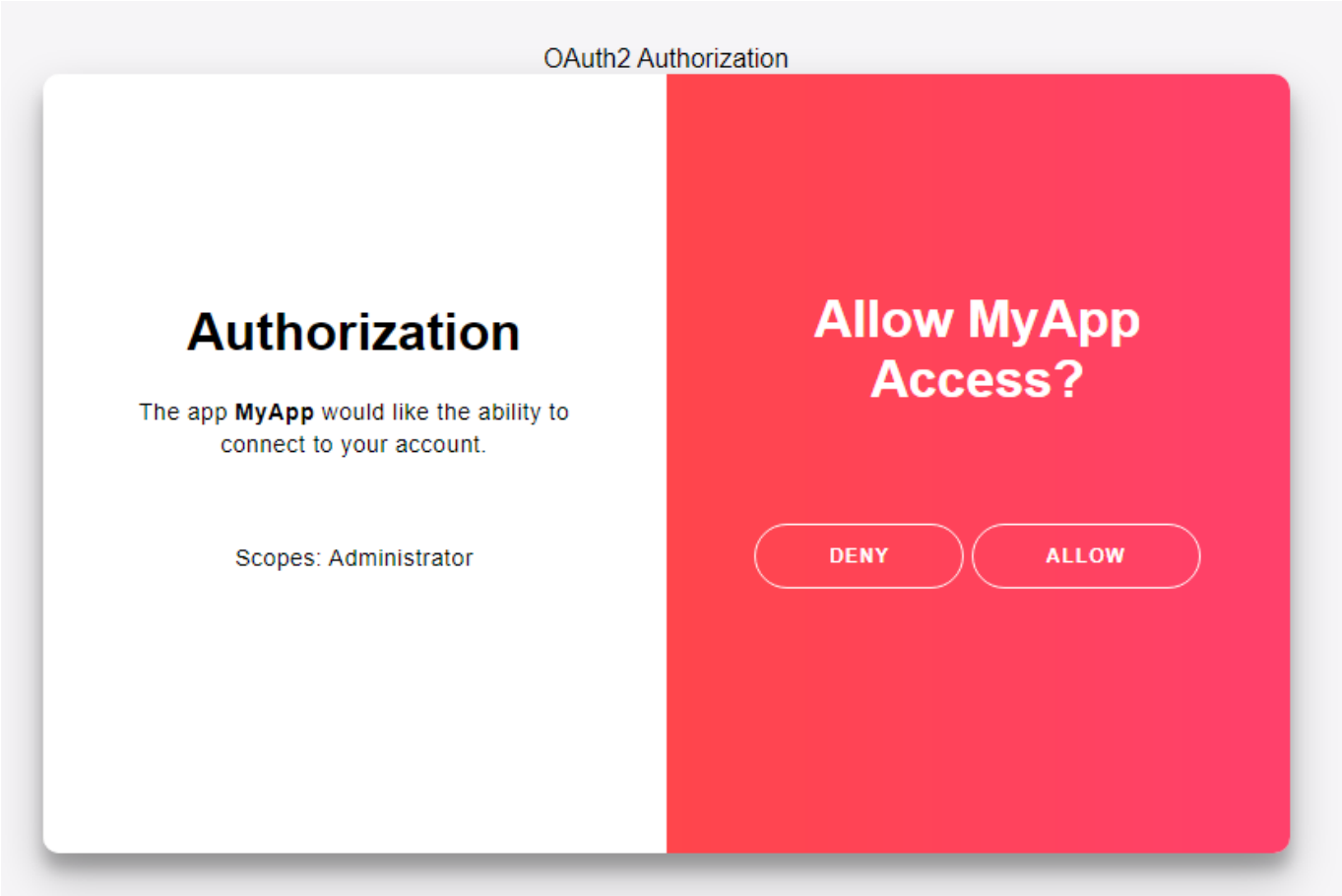
- **AppName:** the name of the app.
- **Token:** access token.
- **Expires:** when the token expires.
- **RefreshToken:** refresh token.

You can save the issued tokens handling the **OAuth2AfterAccessToken** event.

```
private void OnOAuth2AfterAccessToken(TObject Sender, TsgcWSConnection Connection, TsgcHTTPOAuth2Request OAuth2,
  string aResponse)
{
  // ... store OAuth2 Token data
}

OAuth2 = new TsgcHTTP_OAuth2_Server.Create();
OAuth2.Apps.AddApp("MyApp", "http://127.0.0.1:8080", "client-id", "client-secret");
OAuth2.AddToken("MyApp", "12146ce12b0e4813987f2794f768905cefc39da6fbd54f6d9b38387489280608", 1622796714,
  "ef3e3dfa56ec44109c3d345b1541f08e539ce21432d9433099b48a3d08d34bc0");
oServer.Authentication.Enabled = true;
oServer.Authentication.OAuth.OAuth2 = OAuth2;
```

OAuth2 | Server Authentication

When an OAuth2 client requests a new Authorization, the server shows a web page where the user must allow the connection and then login. This page is provided by sgcWebSockets library and is dispatched automatically when a client requests an Authorization.



If the user Allows the access, a login form will be shown where the user must set the Username and Password. This data will be received OnOAuth2Authentication event, so you must validate than the user/password is correct and if it is, then set Authenticated parameter to true.

```
void OnOAuth2Authentication(TsgcWSConnection Connection, TsgcHTTPOAuth2Request OAuth2, string aUser,
  string aPassword, ref bool Authenticated)
{
  if ((aUser == "user") and (aPassword == "secret"))
  {
    Authenticated = true;
  }
}
```

OAuth2 | None Authenticate URLs

By default, when OAuth2 is enabled on Server Side, all the HTTP Requests require Authentication using Bearer Tokens.
If you want allow some URLs to be accessed without the need of use a Bearer Token, you can use the event **OnOAuth2BeforeRequest**

```
procedure OnOAuth2BeforeRequest(TObject Sender; TsgcWSConnection aConnection; TStringList aHeaders;
  ref bool Cancel)
{
  if (DecodeGETFullPath(aHeaders) == "/Public.html")
  {
    Cancel = true;
  }
}
```

OAuth2 Server | Authorization Code Grant

Overview

The Authorization Code grant is the most common OAuth2 flow for server-side applications. The TsgcHTTP_OAuth2_Server provides two endpoints to handle this flow: an Authorization endpoint that displays a sign-in page to the user, and a Token endpoint that exchanges the authorization code for an access token.
This flow is suitable for web applications, desktop applications, and mobile applications where the client can interact with a browser.

Flow

1. Client redirects the user to `/sgc/oauth2/auth?response_type=code&client_id=...&redirect_uri=...&scope=...&state=...`
2. Server fires **OnOAuth2BeforeDispatchPage** to allow customization of the HTML sign-in page.
3. User submits credentials. Server fires **OnOAuth2Authentication** to validate the username and password.
4. If authenticated, the server redirects the user to the `redirect_uri` with `?code=...&state=...` appended.
5. Client POSTs to `/sgc/oauth2/token` with `grant_type=authorization_code&code=...&redirect_uri=...&client_id=...&client_secret=...`
6. Server validates the authorization code and client credentials, fires **OnOAuth2AfterAccessToken**, and returns a JSON response containing the `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.

Configuration

Property	Description
`OAuth2Options.Authorization.Enabled`	Set to True to enable the authorization endpoint. Default: True.
`OAuth2Options.Authorization.URL`	The authorization endpoint URL path. Default: /sgc/oauth2/auth
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token
`OAuth2Options.PKCE`	When True, the server requires PKCE (code_challenge and code_verifier) for authorization code requests. Default: True.

App Registration

Before clients can use the Authorization Code flow, you must register the application on the server. Use `Apps.AddApp` and ensure the `AllowedGrantTypes` includes `auth2Code`.

```
OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });
```

Events

Event	Description

`OnOAuth2BeforeDispatchPage`	Fired before the sign-in HTML page is sent to the user. Allows customization of the HTML content.
`OnOAuth2Authentication`	Fired when the user submits credentials. Validate the username and password here and set `Authenticated` to True or False.
`OnOAuth2AfterAccessToken`	Fired after the server successfully issues an access token. Useful for logging.
`OnOAuth2AfterValidateAccessToken`	Fired when a client makes a request with a valid, non-expired token. Useful for logging and auditing.

Example

The following example shows a complete server setup with the Authorization Code grant, including app registration and event handlers.

```
// Create and configure OAuth2 server
var OAuth2Server = new TsgcHTTP_OAuth2_Server();

// Register an application
OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });

// Attach to HTTP server
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;

OAuth2Server.OnOAuth2Authentication += (sender, user, password, ref authenticated) =>
{
  authenticated = (user == "admin") && (password == "secret");
};

OAuth2Server.OnOAuth2AfterAccessToken += (sender, token, refreshToken, expiresIn) =>
{
  Console.WriteLine("Access token issued: " + token);
};
```

PKCE Support

When `OAuth2Options.PKCE` is set to True (the default), the server enforces Proof Key for Code Exchange. The client must include a `code_challenge` parameter in the authorization request and a corresponding `code_verifier` in the token request. The server validates that the code_verifier matches the original code_challenge before issuing a token.
PKCE is recommended for all clients, especially public clients (native apps and SPAs) that cannot securely store a client secret.

OAuth2 Server | Client Credentials Grant

Overview

The Client Credentials grant is used for machine-to-machine (M2M) communication where no user interaction is required. The client authenticates directly with the TsgcHTTP_OAuth2_Server using its own credentials (client_id and client_secret) and receives an access token. There is no authorization code step, no redirect, and no user login page.
This grant type is appropriate when the client application is acting on its own behalf rather than on behalf of a specific user.

Flow

1. Client POSTs to the token endpoint with `grant_type=client_credentials&client_id=...&client_secret=...&scope=...`
2. Server validates the client credentials against the registered application.
3. Server fires **OnOAuth2AfterAccessToken** and returns a JSON response with `access_token`, `token_type`, and `expires_in`.

Note that no `OnOAuth2Authentication` event is fired for this grant type, since there is no user to authenticate. The server validates the client_id and client_secret against the registered app automatically.

Configuration

Register the application with `AllowedGrantTypes` that includes `auth2ClientCredentials`. Only the token endpoint is needed for this flow.

Property	Description
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token

App Registration

```
OAuth2Server.Apps.AddApp("MyServiceApp", "",
  "service-client-id", "service-client-secret", 3600, false,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2ClientCredentials });
```

The `RedirectURI` can be left empty since no redirect is involved. `RefreshToken` is typically set to False because the client can request a new token at any time using its credentials.

Events

Event	Description
`OnOAuth2AfterAccessToken`	Fired after the server successfully issues an access token. Useful for logging.
`OnOAuth2AfterValidateAccessToken`	Fired when a client makes a request with a valid token. Useful for auditing.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();
OAuth2Server.Apps.AddApp("BackendService", "",
  "service-client-id", "service-client-secret", 7200, false,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2ClientCredentials });
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;
OAuth2Server.OnOAuth2AfterAccessToken += (sender, token, refreshToken, expiresIn) =>
{
  Console.WriteLine("Service token issued: " + token);
};
```

When to Use

Use the Client Credentials grant on the server for:

- Backend services and daemons that authenticate without user interaction.
- Server-to-server API communication.
- Microservice-to-microservice authentication.
- Automated processes, scheduled jobs, and batch operations.

OAuth2 Server | Resource Owner Password Credentials Grant

Overview

The Resource Owner Password Credentials (ROPC) grant allows the client to send the user's username and password directly to the token endpoint. The TsgcHTTP_OAuth2_Server validates the credentials via the **OnOAuth2Authentication** event and issues an access token if authentication succeeds.
This flow does not involve a browser redirect or a sign-in page. The client collects credentials directly from the user and submits them to the server.

Security Warning

This grant type is deprecated in OAuth 2.1. It should only be used when other flows (such as Authorization Code with PKCE) are not feasible, for example in legacy applications or highly trusted first-party clients. The Authorization Code grant with PKCE is the recommended alternative.

Flow

1. Client POSTs to the token endpoint with `grant_type=password&username=...&password=...&client_id=...&client_secret=...&scope=...`
2. Server fires **OnOAuth2Authentication** to validate the username and password.
3. If authenticated, the server fires **OnOAuth2AfterAccessToken** and returns a JSON response with `access_token`, `token_type`, `expires_in`, and optionally a `refresh_token`.
4. If authentication fails, the server returns an error response.

Configuration

Register the application with `AllowedGrantTypes` that includes `auth2ResourceOwnerPassword`.

Property	Description
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token

App Registration

```
OAuth2Server.Apps.AddApp("MyApp", "",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2ResourceOwnerPassword });
```

Events

Event	Description
`OnOAuth2Authentication`	Fired when the server receives the username and password. Validate credentials here and set `Authenticated` to True or False.

`OnOAuth2AfterAccessToken`	Fired after the server successfully issues an access token. Useful for logging.
`OnOAuth2AfterValidateAccessToken`	Fired when a client makes a request with a valid token.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();
OAuth2Server.Apps.AddApp("MyApp", "",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2ResourceOwnerPassword });
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;
OAuth2Server.OnOAuth2Authentication += (sender, user, password, ref authenticated) =>
{
  authenticated = ValidateUser(user, password);
};
OAuth2Server.OnOAuth2AfterAccessToken += (sender, token, refreshToken, expiresIn) =>
{
  Console.WriteLine("Token issued for user: " + token);
};
```

When to Use

Use the Resource Owner Password Credentials grant only when:

- The client is a highly trusted first-party application.
- Other grant types (Authorization Code with PKCE) are not feasible.
- Migrating legacy applications that already collect user credentials directly.

For new applications, always prefer the Authorization Code grant with PKCE.

OAuth2 Server | Refresh Token Grant

Overview

The Refresh Token grant allows a client to exchange a previously issued refresh token for a new access token without requiring user interaction. The TsgcHTTP_OAuth2_Server validates the refresh token and issues a new access token (and optionally a new refresh token).
This is useful for long-lived sessions where the access token has a short expiration but the client needs continued access to protected resources.

Flow

1. Client POSTs to the token endpoint with `grant_type=refresh_token&refresh_token=...&client_id=...&client_secret=...&scope=...`
2. Server validates the refresh token against its stored tokens.
3. Server fires **OnOAuth2AfterRefreshToken** and returns a JSON response with a new `access_token`, `token_type`, `expires_in`, and optionally a new `refresh_token`.

Configuration

Refresh tokens are enabled per application when registering the app. Set the `RefreshToken` parameter to True in `Apps.AddApp`.

Property	Description
`OAuth2Options.Token.Enabled`	Set to True to enable the token endpoint. Default: True.
`OAuth2Options.Token.URL`	The token endpoint URL path. Default: /sgc/oauth2/token
`RefreshToken (AddApp parameter)`	Set to True to issue refresh tokens for this application. Default: True.

App Registration

When registering the app, set the `RefreshToken` parameter to True so that the server issues refresh tokens alongside access tokens.

```
// RefreshToken = true (6th parameter)
OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });
```

Events

Event	Description
`OnOAuth2AfterRefreshToken`	Fired after the server successfully issues a new access token from a refresh token. Useful for logging token rotation.
`OnOAuth2AfterAccessToken`	Fired after the new access token is issued.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();
OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;
OAuth2Server.OnOAuth2AfterRefreshToken += (sender, oldToken, newToken, newRefreshToken, expiresIn) =>
{
  Console.WriteLine("Token refreshed. New token: " + newToken);
};
```

Token Rotation

When a refresh token is used, the server may issue a new refresh token alongside the new access token. The old refresh token is invalidated. This is known as refresh token rotation and helps prevent token replay attacks.

Recovering Tokens After Restart

If the server is restarted while tokens are still valid, you can recover them using `Apps.AddToken` before starting the server. See OAuth2 Recover Access Tokens for details.

OAuth2 Server | Device Authorization Grant (RFC 8628)

Overview

The Device Authorization Grant is designed for input-constrained devices (smart TVs, IoT devices, CLI tools) that cannot easily display a browser or accept keyboard input. The TsgcHTTP_OAuth2_Server provides a device authorization endpoint and a verification page so that users can authorize the device from a secondary device such as a phone or computer.

Flow

1. Device POSTs to `/sgc/oauth2/device` with `client_id=...&scope=...`
2. Server fires **OnOAuth2DeviceAuthorization** and returns a JSON response with `device_code`, `user_code`, `verification_uri`, `expires_in`, and `interval`.
3. Device displays the user_code and verification_uri to the user (e.g., "Go to https://example.com/sgc/oauth2/device/verify and enter code: ABCD-1234").
4. User navigates to the verification URI on a secondary device and enters the user_code.
5. Server fires **OnOAuth2DeviceCodeVerification** to validate the user_code and authorize the device.
6. Meanwhile, the device polls the token endpoint with `grant_type=urn:ietf:params:oauth:grant-type:device_code&device_code=...&client_id=...`
7. Server returns `authorization_pending` until the user authorizes. Once authorized, it returns the access token.

Configuration

Property	Description
`OAuth2Options.DeviceAuthorization.Enabled`	Set to True to enable the device authorization endpoint. Default: False.
`OAuth2Options.DeviceAuthorization.URL`	The device authorization endpoint URL path. Default: /sgc/oauth2/device
`OAuth2Options.DeviceAuthorization.VerificationURL`	The verification page URL path where the user enters the code. Default: /sgc/oauth2/device/verify
`OAuth2Options.DeviceAuthorization.ExpiresIn`	The lifetime in seconds of the device code. Default: 600 (10 minutes).
`OAuth2Options.DeviceAuthorization.Interval`	The minimum polling interval in seconds that the device should use. Default: 5.

Events

Event	Description
`OnOAuth2DeviceAuthorization`	Fired when a device requests a device code. Allows customizing the response or logging the request.
`OnOAuth2DeviceCodeVerification`	Fired when a user submits a verification code on the device verification page. Validate the user_code and authorize or deny the device.

`OnOAuth2AfterAccessToken`	Fired after the server issues the access token to the device.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();
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,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;
OAuth2Server.OnOAuth2DeviceCodeVerification += (sender, userCode, ref authorized) =>
{
  authorized = true;
};
OAuth2Server.OnOAuth2AfterAccessToken += (sender, token, refreshToken, expiresIn) =>
{
  Console.WriteLine("Device authorized. Token: " + token);
};
```

Polling Behavior

While the user has not yet authorized the device, the device polls the token endpoint at the configured interval. The server responds with:

- `authorization_pending` -- the user has not yet completed authorization.
- `slow_down` -- the device is polling too frequently. The device should increase the interval by 5 seconds.
- `expired_token` -- the device code has expired. The device must restart the flow.
- `access_denied` -- the user denied authorization.

Once the user authorizes the device, the next poll returns the access token.

OAuth2 Server | Token Revocation (RFC 7009)

Overview

Token revocation allows clients to notify the TsgcHTTP_OAuth2_Server that a previously issued access token or refresh token is no longer needed. The server invalidates the token so it can no longer be used to access protected resources.
This is useful when a user logs out, when an application is uninstalled, or when a token may have been compromised.

Endpoint

POST to `/sgc/oauth2/revoke` with the following parameters:

Parameter	Description
`token`	The token to revoke (required).
`token_type_hint`	Optional hint about the token type: `access_token` or `refresh_token`.

Behavior

Per RFC 7009, the server always returns HTTP 200 OK regardless of whether the token was found or successfully revoked. This prevents token existence leakage -- a client cannot determine whether a token was valid by observing the response status.
When a refresh token is revoked, the associated access token may also be invalidated (implementation-dependent). When an access token is revoked, the associated refresh token remains valid unless explicitly revoked.

Configuration

Property	Description
`OAuth2Options.Revocation.Enabled`	Set to True to enable the revocation endpoint. Default: False.
`OAuth2Options.Revocation.URL`	The revocation endpoint URL path. Default: /sgc/oauth2/revoke

Events

Event	Description
`OnOAuth2AfterRevokeToken`	Fired after a token revocation attempt. Provides the token value, token_type_hint, and a `Revoked` parameter indicating whether the token was successfully invalidated. Useful for logging revocation activity.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();

OAuth2Server.OAuth2Options.Revocation.Enabled = true;

OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });

Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;

OAuth2Server.OnOAuth2AfterRevokeToken += (sender, token, tokenTypeHint, ref revoked) =>
{
  Console.WriteLine("Token revoked: " + token + " (type: " + tokenTypeHint + ")");
  revoked = true;
};
```

Client Request Example

A client revokes a token by sending a POST request to the revocation endpoint:

```
// Client-side: revoke an access token
HTTPClient.Post("https://example.com/sgc/oauth2/revoke",
  "token=eyJhbGciOi...&token_type_hint=access_token");
```

OAuth2 Server | Token Introspection (RFC 7662)

Overview

Token introspection allows resource servers to query the TsgcHTTP_OAuth2_Server to determine the active state and metadata of an access token or refresh token. This is useful when a resource server needs to validate a token without having to decode it locally.

Endpoint

POST to `/sgc/oauth2/introspect` with the following parameters:

Parameter	Description
`token`	The token to introspect (required).
`token_type_hint`	Optional hint about the token type: `access_token` or `refresh_token`.

Response

The introspection endpoint returns a JSON object with the following fields:

Field	Description
`active`	Boolean. True if the token is currently active (valid and not expired or revoked).
`scope`	The scope associated with the token.
`client_id`	The client identifier for the application that requested the token.
`token_type`	The type of the token (e.g., Bearer).
`exp`	The expiration time of the token (Unix timestamp).
`username`	The username associated with the token (if applicable).

If the token is invalid, expired, or revoked, the response contains only `{"active": false}`.

Configuration

Property	Description
`OAuth2Options.Introspection.Enabled`	Set to True to enable the introspection endpoint. Default: False.
`OAuth2Options.Introspection.URL`	The introspection endpoint URL path. Default: /sgc/oauth2/introspect

Events

Event	Description
`OnOAuth2AfterIntrospectToken`	Fired after a token introspection request. Provides the token value and a `IsActive` parameter that can be modified to override the default validation result. Useful for custom validation logic and auditing.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();

OAuth2Server.OAuth2Options.Introspection.Enabled = true;

OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });

Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;

OAuth2Server.OnOAuth2AfterIntrospectToken += (sender, token, ref isActive) =>
{
  Console.WriteLine("Token introspected: " + token + " Active: " + isActive);
};
```

Resource Server Request Example

A resource server validates a token by sending a POST request to the introspection endpoint:

```
// Resource server: introspect a token
string response = HTTPClient.Post("https://example.com/sgc/oauth2/introspect",
  "token=eyJhbGciOi...&token_type_hint=access_token");
// Response: {"active":true,"scope":"read write","client_id":"my-client-id",...}
```

OAuth2 Server | DPoP Validation (RFC 9449)

Overview

DPoP (Demonstrating Proof of Possession) per RFC 9449 enables the TsgcHTTP_OAuth2_Server to require sender-constrained tokens. When DPoP is enabled, the server validates DPoP proof JWTs, binds tokens to the client's JWK thumbprint, and returns `token_type: DPoP` instead of `Bearer`.
This ensures that access tokens can only be used by the client that originally obtained them, preventing token theft and replay attacks.

How It Works

1. Client includes a `DPoP` header with a signed JWT proof in token requests.
2. Server validates the proof JWT:
   - `typ` must be `dpop+jwt`
   - Algorithm (`alg`) must be asymmetric (e.g., ES256, RS256)
   - `htm` (HTTP method) and `htu` (HTTP URI) must match the request
   - `iat` (issued at) must be recent (freshness check)
   - Signature is verified against the JWK embedded in the proof header
3. Server extracts the JWK thumbprint and binds it to the issued token.
4. Token response includes `token_type: DPoP` instead of `Bearer`.
5. On subsequent resource requests, the server validates the DPoP proof including the `ath` claim (access token hash) to ensure the proof is bound to the correct token.

Configuration

Property	Description
`OAuth2Options.DPoP`	Boolean. Set to True to enable DPoP-bound token validation. Default: False.

Signature Verification

The server reconstructs the EC or RSA public key from the JWK included in the DPoP proof header and verifies the JWT signature. Only asymmetric algorithms are accepted (ES256, ES384, ES512, RS256, RS384, RS512, PS256, PS384, PS512). Symmetric algorithms (HS256, etc.) are rejected.

Nonce Support

The server can issue a `DPoP-Nonce` header in token responses. When present, the client must include the nonce value in the `nonce` claim of subsequent DPoP proof JWTs. This provides an additional layer of replay protection.

Events

Event	Description
`OnOAuth2ValidateDPoP`	Fired after standard DPoP validation completes. Allows custom validation logic such as checking additional claims or enforcing policy. Set `Valid` to False to reject the proof.

Example

```
var OAuth2Server = new TsgcHTTP_OAuth2_Server();
OAuth2Server.OAuth2Options.DPoP = true;
OAuth2Server.Apps.AddApp("MyApp", "http://127.0.0.1:8080",
  "my-client-id", "my-client-secret", 3600, true,
  new TsgcOAuth2GrantType[] { TsgcOAuth2GrantType.auth2Code });
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2 = OAuth2Server;
OAuth2Server.OnOAuth2ValidateDPoP += (sender, dpopProof, accessToken, method, url, ref valid) =>
{
  Console.WriteLine("DPoP proof validated for: " + method + " " + url);
  valid = true;
};
```

Invalid Proof Rejection

When DPoP proof validation fails, the server returns HTTP 400 with an error response:

```
{"error": "invalid_dpop_proof", "error_description": "..."}
```

Common reasons for rejection include:

- Missing or malformed DPoP header.
- Invalid JWT signature.
- Symmetric algorithm used (only asymmetric algorithms are allowed).
- Mismatched `htm` or `htu` claims.
- Expired or future `iat` claim.
- Mismatched `ath` (access token hash) on resource requests.
- Invalid or missing `nonce` when the server requires it.

OAuth2 | TsgcHTTP_OAuth2_Server_Provider

This component allows you to integrate External OAuth2 Providers (like Azure AD, Google, Facebook...) in your server component (like an HTP server), so an user can login using the Azure AD credentials and if the authentication is successful, the HTTP server can provide access to protected resources.

The server components have a property called Authorization.OAuth.OAuth2Provider where you can assign an instance of TsgcHTTP_OAuth2_Server_Provider, so if Authentication is enabled and OAuh2Provider property is attached to OAuth2 Provider Server Component, the WebSocket and HTTP Requests require a Cookie / Bearer Token to be processed, if not the connection will be closed automatically.

```
OAuth2Provider = new TsgcHTTP_OAuth2_ServerProvider();
Server.Authentication.Enabled = true;
Server.Authentication.OAuth.OAuth2Provider = OAuth2Provider;
```

Register OAuth2 Provider

Before the server is started, you must configure the OAuth2 Providers that the server will use to authenticate. Use the method **RegisterProvider** to configure the OAuth2 Providers, this method has the following parameters:

- **Name:** is the name of the provider, it can be any name, is just to identify the provider later.
- **ClientId:** is the public client Id, this value is provided by the OAuth2 Provider.
- **ClientSecret:** is the private client secret (must be keep confidential), this value is provided by the OAuth2 Provider.
- **AuthorizeURL:** is the URL where the OAuth2 client will redirect to login (the connection is using a web browser).
- **TokenURL:** is the URL the server will use to validate the token provided after a successful authorization (the connection is server to server).
- **Scope:** is the value of the scope/s.
- **URL:** is the URL of the HTTP Server that will be used to redirect to the Authorization URL.
- **CallbackURL:** is the URL configured in the OAuth2 Provider that will process the response sent from the OAuth2 server after a successful Authorization.

Example: to configure Azure AD, it requires a tenant-id which is added to the OAuth2 URLs, ClientId, ClientSecret, Scope and a CallbackURL.

```
RegisterProvider(
  'azure',
  '90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x',
  'PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token',
  'user.read',
  '/login',
  'https://localhost/callback'
  );
```

To delete an existing Provider, use the method **UnRegisterProvider**.

Properties

The following properties can be configured in the OAuth2Options property.

- **HTTPClientOptions:** when the server receives the response from the OAuth2 provider after a successful authorization, uses a connection from the HTTP server to the OAuth2 provider to validate the code received is valid. This connection can be configured using this property.
- **Cookies:** when the server receives a successful Token Access, if this property is enabled, a server cookie is created to store a public ID that it's linked to the private Token Access. Here you can configure the cookies values.

Most common uses

- **QuickStart**
  - OAuth2 Provider Azure AD
- **Authenticate**
  - OAuth2 Provider Private Endpoints
  - OAuth2 Provider Authentication
  - OAuth2 Provider Requests

OAuth2 Provider | Azure AD

Azure AD uses the following OAuth2 Authorization URLs

Authorization: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/authorize
Token: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token

The **<tenant-id>** must be replaced by your own values.

When you create the OAuth2 configuration, you must configure a server callback url, this url will be used by Azure to send a response to your server after a successful authorization.

Example: find below a simple example of how register Azure AD provider.

Values provided by Azure AD

ClientId: 90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x
ClientSecret: PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq
tenant: a0ca2055-5dd1-467f-bf13-291f6fd715c6
scope: user.read
CallbackURL: https://localhost/callback

How Register Azure AD

```
RegisterProvider(
  'azure',
  '90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x',
  'PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token',
  'user.read',
  '/login',
  'https://localhost/callback'
  );
```

OAuth2 Provider | Private Endpoints

Every time the Server receives a HTTP Request, the event **OnOAuth2IsPrivateEndpoint** is called to ask if the Endpoint is private or not. By default, is not private.

```
void OnOAuth2IsPrivateEndpoint(TObject *ender, const string aEndpoint, ref bool IsPrivate)
{
  if (aEndpoint == "/private")
  {
    IsPrivate = True;
  }
}
```

OAuth2 Provider | Authentication

The OAuth2 Provider Server Component allows you to authenticate using an external OAuth2 provider (like Azure AD or Google) to access the protected resources of your server. **Example:** you can configure your HTTP server to allow users to log in using Azure credentials; if the login is successful, those users will be allowed to access the protected resources of your server.

The Authentication process is done from the server side and the OAuth2 tokens are not shared with the clients, this means that when the user logins using Azure for example, if the authentication is successful, Azure returns an Access Token that allows you to send requests to the Azure server to get some information (depending of the scope) about the user profile, emails... This Access Token **IS NOT SHARED** with the client (example a web-browser), instead of returning the Access token to the client, the server creates a random ID that it's linked internally with the Access Token, so every time the Client (Web Browser) wants to do a call to the OAuth2 Server, uses the public ID and the server uses this ID to get the OAuth2 Access Token to proxy the HTTP Requests.

Find below an example of how the OAuth2 Authentication works. The example will use the Azure AD configuration described in the following link OAuth2 Provider Azure AD.

Start the Server

The server starts listening on localhost and port 443. The sgcWebSockets HTTP Server is linked to the OAuth2 Server Provider Component and the Authentication property is enabled.
Before the server is started, the Azure OAuth2 Provider is registered using the following method call.

```
RegisterProvider(
  'azure',
  '90945b8d-f6b7-4b97-b2bd-21c3c90b5f3x',
  'PN67Q~5m06c-~X_GMyMf9zMntmm5l2dt~3jVq',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize',
  'https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token',
  'user.read',
  '/login',
  'https://localhost/callback'
  );
```

User Logins

The user opens a new web browser and go to '/login' endpoint.
The server detects that the '/login' endpoint is used to login using the Azure provider so redirects to

https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/authorize

And the OAuth2 authentication Flow Starts.

OAuth2 Authentication

The user is redirected to the OAuth2 Server Authentication Endpoint, now he must login using the credentials and accept the terms of the OAuth2 Application.

If the authorization is successful, Azure AD sends a Code to the url

https://localhost/callback

Validate the OAuth2 Code

Now, the server has received a code from Azure and it will do an internal connection to Azure (from server to server) to validate this token is correct (and avoid someone is trying to hack the server).

The server connects to

https://login.microsoftonline.com/a0ca2055-5dd1-467f-bf13-291f6fd715c6/oauth2/v2.0/token

Passing some paramenters like the code received and the clientsecret, if the validation is successful, Azure returns the Access Token that can be used to access the Azure Protected Resources like read the profile, email...

Successful Access Token

When the server receives a success full AccessToken, the event **OnOAuth2ProviderTokenValid** is called, so here you can configure how the AccessToken is stored (if it is) accessing to the parameter class TsgcHTTPOAuth2ProviderToken

AccesToken: is the OAuth2 Token returned by Azure
ID: is the public identifier stored as a cookie.

In this event you can configure what to do after a successful authentication, example: if you want to redirect the user to the private url, use the following

Response.Redirect.URL := 'https://localhost/private';

Send Requests to Azure

Now, you can send requests to the Azure server using the Public ID stored as a cookie.
Example: if you want to read the profile data, use the following method.

Get('ID', 'https://graph.microsoft.com/v1.0/me');

Where ID is the public ID identifier.

OAuth2 Provider | Requests

Once the Authentication has been successful, you can send requests to the OAuth2 Protected Server using the Public ID Token stored as a cookie.

The OAuth2 Provider Server Component, has several methods to send HTTP Requests: GET, POST, DELETE...

You can pass the Token as a parameter or pass the RequestInfo class if you are using the Indy Server components.

```
void OnCommandGet(TIdHTTPRequestInfo ARequestInfo, TIdHTTPResponseInfo AResponseInfo)
{
  if (ARequestInfo.Document == "/private"
  {
    // return OAuth2 profile data
    AResponseInfo.ContentText = OAuth2Provider.Get("ID Token", "https://graph.microsoft.com/v1.0/me");
    AResponseInfo.ContentType = "application/json";
    AResponseInfo.ResponseNo = 200;
  }
  else
  {
    AResponseInfo.ResponseNo = 404;
  }
}
```

HTTP | JWT

JWT (JSON Web Token) typically consists of a header + payload + signature.

Header

Contains the metadata information about the JWT.

- **alg:** the algorithm used to sign the token.
- **typ:** the type of the token, always JWT.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

You can find more headers, but the previous ones will always be present.

Payload

The payload contains the claims of the JWT. The standard headers are the following:

- **iss:** the issuer, the entity that generates and issues the JWT.
- **sub:** the subject, the entity identified by this token.
- **aud:** the audience, the target audience for this JWT.
- **exp:** the expiry, the timestamp in UNIX format after which the token should not be accepted.
- **iat:** issued at, specifies the date when the token was issued.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Signature

The signature is created using the Encoded Header, Encoded Payload, a Secret and a Cryptographic Algorithm.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0
NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5M
```

```
DIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Algorithms supported

The following algorithms are supported by both Client and Server JWT components.

- HS256
- HS384
- HS512
- RS256
- RS384
- RS512
- ES256
- ES384
- ES512

OpenSSL libraries are required to sign and verify the JWT.

Components

- **TsgcHTTP_JWT_Client:** JWT client which allows you to encode and sign JWT and send as an Authorization Header in **HTTP** and **WebSocket** protocols.
- **TsgcHTTP_JWT_Server:** JWT server which allows you to decode and validate JWT received as an Authorization Header in **HTTP** and **WebSocket** protocols.

JWT | TsgcHTTP_JWT_Client

The TsgcHTTP_JWT_Client component allows you to encode and sign JWT Tokens, attached to a WebSocket Client or HTTP/2 client, the token will be sent automatically as an Authorization Bearer Token Header.

Configuration

You can configure the JWT values in the **JWTOptions** properties, there are 2 main properties: **Header** and **Payload**, just set the values for every required property.

If the Signature is encrypted using a Private Key (RS and ES algorithms), set the value in the **PrivateKey** property of the Algorithm.
If the Signature is encrypted using a Secret (HS algorithms), set the value in the **Secret** property of the Algorithm.

OpenSSL Options

Configure which openSSL libraries you will use when using JWT client.

OpenSSL_Options: configuration of the openSSL libraries.
- **APIVersion:** allows defining which OpenSSL API will be used.
  - **oslAPI_1_0:** uses API 1.0 OpenSSL, it's latest supported by Indy
  - **oslAPI_1_1:** uses API 1.1 OpenSSL, requires our custom Indy library and allows you to use OpenSSL 1.1.1 libraries (with TLS 1.3 support).
  - **oslAPI_3_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows you to use OpenSSL 3.0.0 libraries (with TLS 1.3 support).
- **LibPath:** here you can configure where are located the openSSL libraries
  - **oslpNone:** this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.
  - **oslpDefaultFolder:** sets automatically the openSSL path where the libraries should be located for all IDE personalities.
  - **oslpCustomFolder:** if this is the option selected, define the full path in the property LibPathCustom.
- **LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.
- **UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):
  - **oslsSymLinksDefault:** by default are enabled except under OSX64 (after MacOS Monterey fails trying to load the library without version.).
  - **oslsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.
  - **oslsSymLinksLoad:** Load SymLinks after trying to load the version libraries.
  - **oslsSymLinksDontLoad:** don't load the SymLinks.

Custom Headers

The Header and Payload properties contains the most common headers used to generate a JWT, but you can add more headers calling the method **AddKeyValue** and passing the Key and Value as parameters.
Example: if you want add a new record in the JWT Header with your name, use the following method

```
Header.AddKeyValue("name", "John Smith");
```

After configuring the properties, to generate the JWT, just call the method **Sign** and will return the value of the JWT.

WebSocket Client and JWT

TsgcWebSocketClient allows the use of JWT when connecting to WebSocket servers, just create a new JWT client and assign to **Authentication.Token.JWT** property.

```
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
oClient.URL = "ws://www.esegece.com:2052";


TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oJWT.JWTOptions.Header.alg = jwtRS256;
oJWT.JWTOptions.Payload.sub = "1234567890";
oJWT.JWTOptions.Payload.iat = 1516239022;


oClient.Authentication->Enabled = true;
oClient.Authentication.URL.Enabled = false;
oClient.Authentication.Token.Enabled = true;
oClient.Authentication.Token.JWT = oJWT;
oClient.Active = true;
```

HTTP Clients and JWT

TsgcHTTP2Client and TsgcHTTP1Client allows the use of JWT when connecting to HTTP/2 servers, just create a new JWT client and assign to **Authentication.Token.JWT** property.

```
TsgcHTTP2Client oHTTP = new TsgcHTTP2Client();


TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
oJWT.JWTOptions.Header.alg = jwtRS256;
oJWT.JWTOptions.Payload.sub = "1234567890";
oJWT.JWTOptions.Payload.iat = 1516239022;


oHTTP.Authentication.Token.JWT = oHTTP;
oHTTP.Get("https://your.api.com");
```

Expiration

The Authorization Token can be **re-created every time** you send an HTTP request using an HTTP client or can be **reused several times till it expires**.
Example: calling Apple APNs using Tokens, requires that the token is reused at least during 20 minutes and at a maximum of 1 hour. Use the Property **RefreshTokenAfter** to set the seconds when the token will expire, for example after 30 minutes.

```
RefreshTokenAfter = 60 * 40.
```

Create JWT Signature

You can **create JWT Signatures manually** to use on applications that doesn't make use of WebSocket or HTTP Protocol, or if you are using components from third-parties applications and you only need the JWT Token.

In order to obtain the JWT Signature, just create a new instance of the JWT Client and fill the properties manually, when all properties are set, call the method **Sign** and it will return the JWT Token.

```
TsgcHTTP_JWT_Client oJWT = new TsgcHTTP_JWT_Client();
// ... 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
MessageBox.Show(oJWT.Sign());
```

JWT | TsgcHTTP_JWT_Server

The TsgcHTTP_JWT_Server component allows you to **decode** and **validate** JWT tokens received in **WebSocket Handshake** when using WebSocket protocol or as **HTTP Header** when using HTTP protocol.

Configuration

You can configure the following properties in the **JWTOptions** property of the component:

If the Signature is validated using a Public Key (RS and ES algorithms), set the value in the **PublicKey** property of the Algorithm.
If the Signature is validated using a Secret (HS algorithms), set the value in the **Secret** property of the Algorithm.

To validate JWT tokens, just **attach a TsgcHTTP_JWT_Server** instance to **Authentication.JWT.JWT** property of the WebSocket/HTTP Server.

```
TsgcWebSocketHTTPServer oServer = new TsgcWebSocketHTTPServer();
oServer.Port = 80;
TsgcHTTP_JWT_Server oJWT = new TsgcHTTP_JWT_Server();
oJWT.JWTOptions.Algorithms.RS.PublicKey = "public key here";
oServer.Authorization.Enabled = true;
oServer.Authorization.JWT.JWT = oJWT;
oServer.Active = true;
```

Checks property allows you to enable some checks in the Payload of JWT, by default checks if the issued dates are valid.

Events

Use the following events to control the flow of JWT Validating Token.

OnJWTBeforeRequest

The event is called when a **new HTTP / WebSocket connection** is detected and **before any validation is done**. This connection can contain or not a JWT Token.
If you don't want to process this Connection using JWT Validation, just set the Cancel parameter to True (means that this connection will bypass JWT validations).
By default, all connections continue the process of JWT validation.

OnJWTBeforeValidateToken

The event is called when the **connection contains an Authorization token** and **before is validated**.
If you don't want to validate this token, just set the Cancel parameter to True (means that this connection will bypass JWT validations).
By default, all connections continue the process of JWT validation.

OnJWTBeforeValidateSignature

This event is called after the **token has been decoded**, so using Header and Payload parameters you have access to the content of JWT and before the signature is validated.
The parameter **Secret** is the secret that will be used to validate the signature and uses the PublicKey or Secret of the JWTOptions property. If this Token must be validated with another secret, the new value can be set to Secret parameter.
By default, all signatures are validated

OnJWTAfterValidateToken

The event is called after the signature has been validated, the parameter Valid shows if the signature is correct or not. If it's not correct the connection will be closed, otherwise the connection will continue.
You can access to the content of Header and Payload of JWT using the arguments provided.
If there is any error validating the JWT, will be informed in the Error argument.

OnJWTException

If there is any exception while processing the JWT Decoding and Validation, this event will be called with the content of error.

OnJWTUnauthorized

This event is called before the connection is closed because there is no authorization token or is invalid, by default, the Disconnect parameter is true, you can set to false if you still want to accept the connection. This event can configure which endpoints must implement JWT Authorization or not.

The error "Access to XmlHttpRequest at X from origin X has been blocked X by CORS policy: Response to preflight request doesn't pass access control check" means the Web-browser is trying to send a Preflight request but the request is not authorized by your server. To allow do a Preflight request, check if the request is CORS and if true don't disconnect it, find below an example:

```
public void OnJWTUnauthorized(object sender, TsgcWSConnection connection, ref bool disconnect)
{
    if (IsCorsHeader(((TsgcWSConnectionServer)connection).HeadersRequest))
    {
        disconnect = false;
    }
    else
    {
        disconnect = true;
    }
}
```

OnJWTResponseError

This event is called before the response error is sent to the client, allows customizing the Response Code, Text and Headers of the HTTP response. By default the Response Code Error is "401" and the Response Text is "Unauthorized".

WebAuthn

WebAuthn (Web Authentication) is a web standard developed by the World Wide Web Consortium (W3C) and FIDO Alliance to enable **secure**, **passwordless authentication on the web**. It is part of the broader FIDO2 framework and aims to **reduce reliance on traditional passwords**, which are often vulnerable to phishing, credential stuffing, and other attacks.

At its core, **WebAuthn allows users to authenticate using public-key cryptography. Instead of a username and password, users register a unique public-private key pair with a web application** (the Relying Party). The private key is securely stored on an authenticator—such as a hardware security key, smartphone, or built-in biometric device—while the public key is stored on the server.

During authentication, the server issues a challenge that must be signed by the user's private key. The signed challenge is returned and verified using the stored public key, ensuring both the integrity and origin of the response. This approach prevents credentials from being intercepted or reused.

WebAuthn supports a range of authenticators and devices, making it flexible for both developers and users. It also enables multi-factor authentication (MFA) when combined with other factors like PINs or biometrics, significantly improving security without sacrificing usability.

Server

- **TsgcWSAPIServer_WebAuthn:** The component provides a simple but powerful solution to implement the WebAuthn Relying Party server, enabling passwordless authentication in your web application. The server has passed the full **FIDO Conformance Test** using the Conformance Self-Validation Testing tool.

Client

- **Javascript Client:** act as the bridge between the user, browser, and authenticator during registration and authentication

TsgcWSAPIServer_WebAuthn

The **TsgcWSAPIServer_WebAuthn** component provides a simple but powerful solution to implement the WebAuthn Relying Party server, enabling passwordless authentication in your web application. A WebAuthn application consists of a WebAuthn server that handles the server-side registration and authentication and a client-side application that usually is a javascript application.

WebAuthn requires the use of secure connections (SSL/TLS), so the OpenSSL libraries must be deployed and configured with the server.

Only the OpenSSL 3.0.0+ API is supported, so previous OpenSSL versions may not work.

Configuration

The **TsgcWSAPIServer_WebAuthn** must be attached to an HTTP server, TsgcWebSocketHTTP_Server or TsgcWebSocketServer_HTTPAPI using the Server property. You can configure the server endpoints that will handle the registration and authentication options, and the WebAuthn options like supported algorithms, origins, and more.

Endpoints Options

Here you can configure the server endpoints that will handle the HTTP/JavaScript requests to use WebAuthn as an authenticator. The component is already configured with default endpoints, but you can change all of them to fit your needs.

- **AuthenticationOptions:** by default is /sgcWebAuthn/Authentication/Options
- **AuthenticationVerify:** by default is /sgcWebAuthn/Authentication/Verify
- **RegistrationOptions:** by default is /sgcWebAuthn/Registration/Options
- **RegistrationVerify:** by default is /sgcWebAuthn/Registration/Verify
- **Webauthn:** includes the javascript library used by default. You can disable this property and use your own webauthn library.
- **Test:** by default is disabled, only use to test the WebAuthn functionality.

Example: if your server is listening on domain www.test.com, the request to authentication options by default will be http://www.test.com/sgcWebAuthn/Authentication/Options

WebAuthn Options

In this property you can configure the main options of the WebAuthn Server Component.

- **RelyingParty:** a **mandatory** property where the **DNS name of the server** must be defined. Example: if the server is running on the domain www.test.com, set this property to "www.test.com".

WebAuthn uses origins to enforce same-origin policy constraints, which are essential for preventing phishing and cross-site attacks. During the WebAuthn registration and authentication processes, the origin is strictly validated by the browser and the authenticator.

- **Origins:** If the requests can come from different origins, use the property Origin to set the additional origins. Example: if the requests can come from login.test.com and www.test.co.uk, configure the Origins property with the values: https://login.test.com and https://www.test.co.uk

- **TopOrigins:** Normally, WebAuthn relies on the origin of the calling frame (i.e., the one invoking navigator.credentials.create() or navigator.credentials.get()). However, web pages can be embedded in iframes, which might come from a different origin than the top-level page. This opens up potential for abuse

or clickjacking-style attacks. To mitigate this, the WebAuthn Level 2 spec introduces TopOrigin where you can define the TopOrigins.

In WebAuthn, crossOrigin is a boolean parameter that indicates whether the WebAuthn operation is being performed from a cross-origin context, such as an iframe embedded from a different origin than the top-level browsing context.
This parameter was introduced to help browsers and authenticators safely handle authentication requests in embedded environments—a common scenario in modern web applications.

- **AllowCrossOrigins:** if true, indicates that requests made from a cross-origin iframe (e.g., an iframe at https://auth.example.com embedded in a page at https://app.example.org) are allowed. By default, it is disabled.

WebAuthn supports a variety of **cryptographic algorithms** for public key credential generation and verification. These algorithms are used during credential registration (with navigator.credentials.create()) and authentication (with navigator.credentials.get()), and they ensure secure signing and validation of challenges using asymmetric key pairs. The server is configured by default with the **ES256 and RS256** which are the most common algorithms. You can change at any time which algorithms are supported from the **Algorithms** property. The following algorithms are supported:

- ES256
- ES384
- ES512
- RS256
- RS384
- RS512
- PS256
- PS384
- PS512
- RS1
- EdDSA

In WebAuthn, **attestation is an optional mechanism** that allows the authenticator (e.g., device or security key) to provide information about its manufacturer, model, and security characteristics during credential creation. This information helps the Relying Party (RP) decide whether to trust the authenticator.

Different attestation formats define how this data is structured and verified. Three commonly used formats are android-key, packed, and others like fido-u2f, apple, or none. By default, all attestation formats are enabled. You can find below the list of supported attestation formats:

- **NoneAttestation:** in this case none attestation data is returned. Prioritizes user privacy by avoiding the exposure of device identifiers. Common in applications that don't care about device provenance.

- **PackedAttestation:** is a flexible, compact format used by many authenticators. The authenticator returns an attestation certificate and signature. Can be: **Full attestation:** Signed with a vendor-provided key and cert or **Self attestation:** Signed using the credential private key. Most widely used across different platforms (e.g., YubiKey, Windows Hello).

- **TPMAttestation:** Used by devices with a Trusted Platform Module (TPM). Attestation is signed using keys from the TPM and includes a certificate chain. Used by Enterprise desktops/laptops with TPM chips (e.g., Windows machines).

- **AndroidKeyAttestation:** Used by Android devices with the Android Keystore. The key is generated in hardware, and attestation includes information signed by a certificate chain issued by the device manufacturer. Used by Android phones with hardware-backed keystores (TEE or StrongBox).

- **AppleAttestation:** Used by Apple platform authenticators, such as Touch ID and Face ID. Attestation is generated by Apple's internal APIs and includes a special certificate format. Used on Safari using Apple biometrics.

- **FidoU2FAttestation:** Legacy attestation format used by FIDO U2F authenticators. Returns a U2F-compatible certificate and signature. Used by older security keys (e.g., early YubiKeys) that support FIDO U2F.

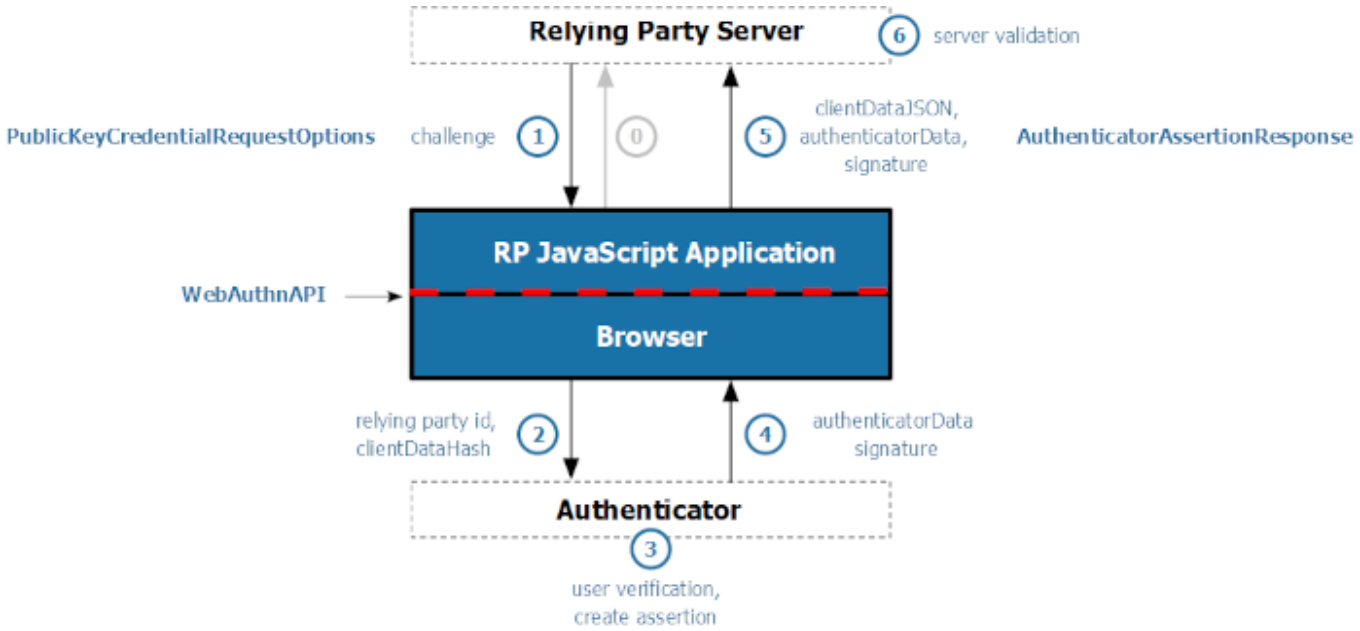
In the WebAuthn API, **AllowCredentials** is an **optional field** used during the authentication process (via navigator.credentials.get()). It **specifies a list of credential IDs that are permitted to authenticate the user** for a

particular Relying Party (RP). This mechanism lets the RP control which credentials are considered valid for a login attempt. The property credentials has the following fields:

- **AllowCredentials:** if enabled (false by default) specifies a list of credential IDs that are permitted to authenticate the user
- **ExcludeCredentials:** given a username, shows all existing credentials already stored in the server component.
- **Limit:** the max number of credentials that will be sent when ExcludeCredentials is true.

WebAuthn Protocol

- **WebAuthn Registration:** The server generates a challenge and sends it to the client, which uses an authenticator (e.g. security key or biometric device) to create a key pair. The public key is sent back and stored by the server for future authentication. Find below more info about the registration flow events:
  - **WebAuthn Registration Request**
  - **WebAuthn Registration Response**
  - **WebAuthn Registration Result**
- **WebAuthn Authentication:** The server sends a challenge to the client, which signs it using the previously registered private key stored in the authenticator. The signed response is verified by the server using the stored public key to confirm the user's identity. Find below more info about the authentication flow events:
  - **WebAuthn Authentication Request**
  - **WebAuthn Authentication Response**
  - **WebAuthn Authentication Result**
- **MDS:** The FIDO Alliance Metadata Service (MDS) is a centralized repository of Metadata Statements that is used by relying parties to validate authenticator attestation and prove the genuineness of the device model.



- **Authorization:** The client can request a bearer token from the server during the authentication flow. This token can be used later to open a new WebSocket or HTTP connection without the need to log in using passkeys.
  - **WebAuthn Authorization WebSocket**
  - **WebAuthn Authorization HTTP**

WebAuthn | Registration

WebAuthn Registration involves the **client** (browser), the **authenticator** (security device), and the **relying party** (RP; TsgcWSAPIServer_WebAuthn delphi/cbuilder server).

Registration Options

Creating a new user credential usually requires the client starting the registration flow using a new HTTP Request to the **Registration Options Endpoint** configured in the TsgcWSAPIServer_WebAuthn server. By default, the end-point is /sgcWebAuthn/Registration/Options, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Registration/Options.

The client sends a new request passing as a payload the following json (using test as username)

```
 {"username":"test","algorithms":[]}
```

The server reads the request and returns a response with a new challenge.

```
{
    "rp": {
        "name": "localhost",
        "id": "localhost"
    },
    "user": {
        "id": "36b9d6a84204487382fee62e7e67a80d",
        "name": "test",
        "displayName": "test"
    },
    "challenge": "6c6c468c99f24bf29a85a15b661f75f385654f97309c46bab2909c926e17ccbe",
    "pubKeyCredParams": [
        {
            "type": "public-key",
            "alg": "-7"
        },
        {
            "type": "public-key",
            "alg": "-257"
        }
    ],
    "timeout": 60000,
    "excludeCredentials": [],
    "authenticatorSelection": {
        "residentKey": "preferred",
        "requireResidentKey": false,
        "userVerification": "preferred"
    },
    "attestation": "direct",
    "hints": [],
    "extensions": {
        "credProps": true
    }
}
```

The response returns the following data:

- **challenge:** Randomized binary data, base64-encoded.
- **rp:** Relying Party info (your web app/site).
- **user:** User identifier and display name.
- **pubKeyCredParams:** Allowed algorithms (e.g., ES256, RS256).

- **authenticatorSelection:** Preferences for authenticator type (platform or cross-platform, user verification options).
- **timeout:** Suggested timeout value.
- **attestation:** Attestation conveyance preference ("none", "direct", "indirect").

Registration Verify

Now the client has the response from the server, reads the response and the authenticator returns the cryptographic data to the client web-browser. Now the client sends a new HTTP Request with the following data to the Registration Verify Endpoint configured in the TsgcWSAPIServer_WebAuthn server, by default is /sgcWebAuthn/Registration/Verify
, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Registration/Verify.

The authenticator responds back to the JavaScript in the browser with:

- **id:** Credential ID (unique ID for the authenticator-generated key pair).
- **rawId:** Binary form of Credential ID.
- **type:** Credential type ("public-key").
- **response:** Contains attestationObject and clientDataJSON.

Find below an example:

```
{
   "id": "yeA4BVRlrAfLG-KzqsL_rII4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
   "rawId": "yeA4BVRlrAfLG-KzqsL_rII4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
   "response": {
      "attestationObject": "o2NmbXRmcGFja2VkZ2F0dFN0....",
      "clientDataJSON": "eyJ0eXBlIjoid2ViYXV0aG4uY3Jl....",
      "transports": [
         "nfc",
         "usb"
      ],
      "publicKeyAlgorithm": -7,
      "publicKey": "MFkwEwYHKoZIzj0CAQYIKoZIzj....",
      "authenticatorData": "SZYN5YgOjGh0NBcPZHZgW4_k...."
   },
   "type": "public-key",
   "clientExtensionResults": {
      "credProps": {
         "rk": true
      }
   },
   "authenticatorAttachment": "cross-platform"
}
```

The server reads the JSON request from the client and decodes, verifies, and stores the public key and credential ID.

Decode attestationObject and clientDataJSON:

- Extract the public key from attestationObject.
- Validate the challenge in clientDataJSON matches the original one sent by the server.
- Ensure proper origin validation (myapp.example.com).

Store Credential:

- If the verification is successful, the event **OnWebAuthnRegistrationSuccessful** is called so you can store the credential ID and extracted public key securely in your database for future authentications.

- If there is any error, the event **OnWebAuthnRegistrationError** is called and you can obtain the details of the error.

Registration Flow

Get more info about the Registration Flow process using the following links:

- **WebAuthn Registration Request**
- **WebAuthn Registration Response**
- **WebAuthn Registration Result**

WebAuthn Registration | Request

This registration options request is essential to bootstrap WebAuthn registration securely. It asks the server to:

- Generate a challenge,
- Look up or provision the user ID,
- Return a valid PublicKeyCredentialCreationOptions object (with RP, user, algorithms, etc.).

The browser must obtain credential creation options from the server to initiate a secure WebAuthn credential registration. This is typically done via an HTTPS POST request from the browser to a server endpoint (e.g., /sgcWebAuthn/Registration/Options), which prepares parameters like the challenge, RP ID, and user info.

The request body contains data that identifies the user and possibly provides configuration preferences. It often looks like this:

```
{
  "username": "alice@example.com",
  "displayName": "Alice Smith",
  "authenticatorSelection": {
    "authenticatorAttachment": "platform",       // optional: platform or cross-platform
    "userVerification": "preferred"               // optional: required | preferred | discouraged
  },
  "attestation": "none",                          // or "direct", "indirect"
  "residentKey": "discouraged"                    // optional
}
```

The WebAuthn Server Component listens on the endpoint configured in the property **EndpointOptions.RegistrationOptions** the initial browser request to obtain a PublicKeyCredentialCreationOptions. When the server receives a new HTTP Request, the event **OnWebAuthnRegistrationOptionsRequest** is called and you can access the request sent by the client and cancel the request by setting the parameter Accept to False.

```
public void OnWebAuthnRegistrationOptionsRequest(object sender, TsgcWebAuthn_RegistrationOptions_Request aRequest
{
    if (aRequest.Username == "anonymous")
        accept = false;
}
```

WebAuthn Registration | Response

The server responds to the client's registration options request (e.g., POST /sgcWebAuthn/Registration/Options) with a JSON payload that looks like the following (after base64url-encoding binary fields):

```
{
  "publicKey": {
    "rp": {
      "name": "esegece software",
      "id": "esegece.com"
    },
    "user": {
      "id": "c3ViamVjdC1pZA",             // base64url-encoded ArrayBuffer (user han-
dle)
      "name": "webauthn@esegece.com",
      "displayName": "Delphi Developer"
    },
    "challenge": "Xz8x2K6nY3gZ...",          // base64url-encoded challenge
    "pubKeyCredParams": [
      { "type": "public-key", "alg": -7 }, // ES256
      { "type": "public-key", "alg": -257 } // RS256
    ],
    "timeout": 60000,
    "attestation": "none",
    "authenticatorSelection": {
      "authenticatorAttachment": "platform",
      "userVerification": "preferred",
      "residentKey": "discouraged"
    }
  }
}
```

Find below a description of the fields:

- **rp:** Relying Party info:
  - **name:** Friendly display name (e.g., "Example Inc.").
  - **id:** Relying Party ID (must match site origin or be a registrable suffix).
- **user:** Information identifying the user account:
  - **id:** Unique, stable byte array (e.g., user UUID).
  - **name:** Unique user identifier (username or email).
  - **displayName:** User-facing display name.
- **challenge:** A cryptographically random nonce (16–64 bytes), base64url-encoded.
- **pubKeyCredParams:** Algorithms the RP is willing to accept:
  - **Common:** -7 (ES256), -257 (RS256)
- **timeout:** Optional timeout for the creation operation (in ms).
- **attestation:** One of "none", "indirect", "direct".
- **authenticatorSelection:**
  - **authenticatorAttachment:** "platform" (built-in) or "cross-platform" (e.g., YubiKey).
  - **userVerification:** "required", "preferred", or "discouraged".
  - **residentKey:** "required", "preferred", or "discouraged".

Before the response is sent to the client, the event **OnWebAuthnRegistrationOptionsResponse** is called, allowing you to customize the response.

```
void OnWebAuthnRegistrationOptionsResponse(object sender,
    WebAuthnRegistrationOptionsRequest request,
    WebAuthnRegistrationOptionsResponse response)
{
```

```
    if (request.Username == "esegece.com")
    {
        response.ExcludeCredentials.AddCredentialRecordFromJSON("json1.txt");
        response.ExcludeCredentials.AddCredentialRecordFromJSON("json2.txt");
    }
}
```

WebAuthn Registration | Result

The goal is to verify the authenticity and integrity of the data returned by the client, ensure that the credential is bound to the expected user, and safely register a public key credential for future authentication.

The server must validate the client response following these steps:

- Parse the client response: The client sends a response like this to the server.

```
{
  "id": "base64url-encoded credential ID",
  "rawId": "base64url-encoded ID bytes",
  "response": {
    "clientDataJSON": "base64url",
    "attestationObject": "base64url"
  },
  "type": "public-key"
}
```

- The server validates the clientDataJSON and attestationObject
- Verifies the Attestation Statement (if defined)
- If everything is valid, it stores the credential and can trust the public key for future logins.

Registration Successful

If the response sent by the client is valid, the event **OnWebAuthnRegistrationSuccessful** is called and the Credential Record can be safely stored into a database for future logins validations.

```
void OnWebAuthnRegistrationSuccessful(
    object sender,
    WebAuthnRegistration registration,
    WebAuthnCredentialRecord credentialRecord,
    ref bool accept)
{
    // Store in a database
    using (var cmd = dbConnection.CreateCommand())
    {
        cmd.CommandText = "INSERT INTO Credentials (Credentials) VALUES (@json)";
        var param = cmd.CreateParameter();
        param.ParameterName = "@json";
        param.Value = credentialRecord.AsJson(); // or .AsJSON depending on naming
        cmd.Parameters.Add(param);
        cmd.ExecuteNonQuery();
    }
}
```

Registration Error

If there is any error while validating the client response, the event **OnWebAuthnRegistrationError** is called and you can access the reason for the error in the parameter aError.

```
void OnWebAuthnRegistrationError(
    object sender,
    WebAuthnRegistrationVerifyRequest request,
    WebAuthnRegistration registration,
    string error)
{
    Log("#webauthn_registration_error: " + error);
}
```

WebAuthn | Authentication

WebAuthn Authentication allows users to log in using previously registered public-key credentials. It involves validating a signed challenge using the user's stored public key from registration.

Authentication Options

Authenticating requires the client starting the authentication flow using a new HTTP Request to the **Authentication Options Endpoint** configured in the TsgcWSAPIServer_WebAuthn server. By default, the endpoint is /sgcWebAuthn/Authentication/Options, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Authentication/Options.

Client sends the assertion (authentication response) to the Server via POST

```
{"username":"test","user_verification":"preferred"}
```

The server Generates PublicKeyCredentialRequestOptions

```
{
    "challenge": "9d0d61edf30b45f8b88aef7087f9117716e2b7d8b0ee4460b06142f39dd0ec9f",
    "timeout": 60000,
    "rpId": "localhost",
    "allowCredentials": [
        {
            "id": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
            "type": "public-key",
            "transports": [
                "nfc",
                "usb"
            ]
        }
    ],
    "userVerification": "preferred",
    "hints": [],
    "attestation": "none",
    "attestationFormats": [],
    "extensions": {}
}
```

Authentication Verify

Authenticating requires the client starting the authentication flow using a new HTTP Request to the **Authentication Verify Endpoint** configured in the TsgcWSAPIServer_WebAuthn server. By default, the endpoint is /sgcWebAuthn/Authentication/Verify, so if your server is listening in the domain https://www.test.com, the client should make a new request to the url https://www.test.com/sgcWebAuthn/Authentication/Verify.

The browser prompts the user to use their authenticator (e.g., fingerprint, YubiKey). The authenticator signs the challenge with the private key linked to the credential ID. Returned credential includes:

- **rawId:** Credential ID (base64url).
- **response.authenticatorData:** Metadata from the authenticator (binary).
- **response.clientDataJSON:** Contains the original challenge, origin, and type.
- **response.signature:** Signature over authenticatorData + hash(clientDataJSON).

Find below a json example of the client request:

```
{
    "id": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
```

```
    "rawId": "yeA4BVRlrAfLG-KzqsL_rlI4ffhuKHK8uoEkVoab065UkS82Zqlh9VFQHIYwOuOo",
    "response": {
        "authenticatorData": "SZYN5YgOjGh0NBcPZHZgW4_krrmihjLHmVzzuoMdl2MFAAAABw",
        "clientDataJSON": "eyJ0eXBlIjoid2ViYXV0aG4uZ.....",
        "signature": "MEQCIAJRqvvys8....",
        "userHandle": "36b9d6a84204487382fee62e7e67a80d"
    },
    "type": "public-key",
    "clientExtensionResults": {},
    "authenticatorAttachment": "cross-platform"
}
```

The server reads the request from the client, validates that the credential is stored, and verifies the signature. If the signature is valid, the event **OnWebAuthnAuthenticationSuccessful** is called.

Authentication Flow

Get more info about the Authentication Flow process using the following links:

- **WebAuthn Authentication Request**
- **WebAuthn Authentication Response**
- **WebAuthn Authentication Result**

WebAuthn Authentication | Request

When a user attempts to log in, the browser sends a request to the server asking for the authentication options (also called "assertion options").

```
{
  "username": "alice@example.com"
}
```

This request allows the server to:

- Generate a challenge (a secure, random value).
- Look up which credentials (public key IDs) are valid for this user.
- Respond with cryptographic parameters that the browser will use in navigator.credentials.get().

When the WebAuthn Server Component receives this request, this is typically done via an HTTPS POST request from the browser to a server endpoint (e.g., /sgcWebAuthn/Authentication/Options), the event **OnWebAuthnAuthenticationOptionsRequest** is called, so you can add the credentials associated with the username (if any).

```
public void OnWebAuthnAuthenticationOptionsRequest(
    object sender,
    TsgcWebAuthn_AuthenticationOptions_Request aRequest,
    ref TsgcWebAuthn_CredentialRecords credentialRecords,
    ref bool accept)
{
    if (UserExistsInDB(aRequest.Username))
    {
        while (!EOF())
        {
            credentialRecords.AddCredentialRecordFromJSON(RecordFromDB());
            Next();
        }
    }
}
```

WebAuthn Authentication | Response

Once the browser (client) sends a request to the server with the user's username (or a similar identifier), the server replies with the authentication options that the client will use to begin the authentication process via the browser's navigator.credentials.get() API.

This response provides the parameters and constraints needed by the browser to generate a WebAuthn authentication assertion using the user's authenticator (e.g., security key, biometric device).

Example JSON Response:

```
{
  "challenge": "Z3lVbWV5YXBpbmdvZG90IQ",
  "timeout": 60000,
  "rpId": "example.com",
  "allowCredentials": [
    {
      "type": "public-key",
      "id": "dXNlckNyZWRJZA",
      "transports": ["usb", "nfc", "ble"]
    }
  ],
  "userVerification": "preferred"
}
```

Find below a description of the fields:

- **challenge:** A cryptographic challenge (randomly generated) to prevent replay attacks. Must be unique per request.
- **timeout:** Optional. How long the browser should wait (in ms) before canceling the operation.
- **rpId:** Relying Party Identifier — usually your domain name.
- **allowCredentials:** A list of acceptable credentials (credential IDs) that the user previously registered.
- **userVerification:** Informs the browser how to verify the user: "required", "preferred", or "discouraged".
- **extensions (optional):** Additional capabilities or data requested from the authenticator.

Before the response is sent to the client, the event **OnWebAuthnAuthenticationOptionsResponse** is called, allowing you to customize the response.

WebAuthn Authentication | Result

The goal is to validate the signed assertion provided by the browser, which proves the user owns the private key originally registered. This is what securely logs the user in. After the user interacts with their authenticator (e.g., fingerprint, security key), the browser sends a POST request back to the server with the authentication result. Find below a json example:

```
{
  "id": "credential-id",
  "rawId": "base64url-encoded-credential-id",
  "type": "public-key",
  "response": {
    "clientDataJSON": "base64url",
    "authenticatorData": "base64url",
    "signature": "base64url",
    "userHandle": "optional"
  }
}
```

When the server receives this request at the configured endpoint (e.g., /sgcWebAuthn/Authentication/Verify), it must validate the following steps:

- Retrieve Credential from DB: Use the credential-id to lookup the user's registered public key and metadata (e.g., signature counter).
- Decode & Parse clientDataJSON
- Validate Authenticator Data
- Verify Signature

If all validations are correct, the authentication is successful and the event **OnWebAuthnAuthenticationSuccessful** is called.

If any check fails, the event **OnWebAuthnAuthenticationError** is called with the reason for the error.

WebAuthn | MDS

The **Metadata Service (MDS)** is a **centralized service provided by the FIDO Alliance** that aggregates and publishes **Metadata Statements** about authenticators certified through FIDO certification programs. These statements contain detailed security, compliance, and operational information about the authenticators.

- The service endpoint is often referred to as **MDS3** (the third version of the Metadata Service protocol).
- Relying Parties (e.g., websites or applications implementing WebAuthn) **retrieve metadata statements** from MDS to make informed trust decisions about authenticators.

The MDS adds a crucial layer of trust and security validation for relying parties using WebAuthn:

- **Authenticator Validation:** Enables verification of authenticator compliance with FIDO standards and helps validate the AAGUID presented in a WebAuthn attestation.
- **Compromise & Revocation Detection:** Provides up-to-date information on compromised or revoked authenticators, allowing relying parties to block insecure devices.
- **Security Assurance:** Helps enforce security policies, such as only allowing authenticators that meet a certain FIDO certification level or user verification strength.
- **Interoperability:** Ensures consistent behavior and security expectations across different browsers, platforms, and devices using different authenticators.

Configuration

You can configure the use of the MDS using the property WebAuthnOptions.MDS, find below the main properties:

- **Enabled:** if true (the default value), the webauthn requests will be validated against the configured MDS file.
- **MDS_FileName:** the path where the MDS file is stored. It can be downloaded from the following URL: https://mds3.fidoalliance.org/
- **RootCert_FileName:** the path where the Root Certificate is stored. Must be defined to validate the certificate chain. It can be downloaded from https://valid.r3.roots.globalsign.com/
- **Leaf_CertificateCRL:** if true (by default is false), the leaf certificate of the blob will be validated against the CRL (Certificate Revocation List).
- **CRL_FileName:** the path where the CRL file is stored. If it's not defined and Leaf_CertificateCRL is enabled, it will try to download the CRL automatically.

WebAuthn Authorization

If you want the server to send a bearer token after a successful authentication that can be used to open a new WebSocket or HTTP connection, pass the parameter token = true. Example:

```
{
  "username": "alice@example.com", "token": true
}
```

After a successful Authentication, the server will send a response like this:

```
{
  "verified": "ok",
  "authentication": {
    "token": "C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0276176F9E6DE"
  }
}
```

The event **OnWebAuthnUnauthorized** is called when a request is not authorized and the connection will be disconnected. Here you can configure which endpoints require WebAuthn authentication and which do not.

WebAuthn Authorization | HTTP

Once you have a new token, just send an authorization header with this bearer token. You can use the Custom-Headers property of the TsgcHTTP1Client to configure the Bearer Token. Example:

```
public static async Task<string> GetHTTPRequestAsync(string aURL, string aToken)
{
    using (var client = new HttpClient())
    {
        // Add Authorization header
        client.DefaultRequestHeaders.Add("Authorization", "Bearer " + aToken);
        // Send GET request
        HttpResponseMessage response = await client.GetAsync(aURL);
        // Throw exception if request failed
        response.EnsureSuccessStatusCode();
        // Return the response content as string
        return await response.Content.ReadAsStringAsync();
    }
}
```

WebAuthn Authorization | WebSocket

Once you have a new token, just send an authorization header with this bearer token. You can use the event On-HandShake to add the Bearer Token to the connection request. Example:

```
public static async Task<ClientWebSocket> ConnectWithAuthAsync(Uri serverUri)
{
    var client = new ClientWebSocket();
    // Add custom Authorization header
    client.Options.SetRequestHeader("Authorization", "Bearer C760C1C39E3D4E829693A13F18F5CFDE537B516336FC48F7BAB0
    // Connect to WebSocket server
    await client.ConnectAsync(serverUri, CancellationToken.None);
    return client;
}
```

Webauthn | Javascript Client

WebAuthn (Web Authentication API) is a W3C standard that enables secure passwordless authentication using **public-key cryptography**. Instead of passwords, users register and authenticate using **hardware-based authenticators** (like fingerprint readers, Face ID, YubiKeys, etc.) or platform authenticators (built-in, like Touch ID).

Find below how to handle the Registration and Authentication using a Javascript client.

WebAuthn Registration

How WebAuthn Registration works

- **User Initiates Registration**:
  - The user provides a username and clicks **Register**.
- **Browser Requests Options from Server**:
  - The frontend makes a `POST` request to get registration options.
- **Browser Creates Credentials**:
  - Using the `navigator.credentials.create()` API via `SimpleWebAuthnBrowser.startRegistration()`, a credential is created.
- **Server Verifies Registration**:
  - The browser sends back the credential to the server.
  - The server verifies the registration and stores the public key for that user.

The **TsgcWSAPIServer_WebAuthn** component has an html file to test the WebAuthn protocol. This HTML file contains a minimal UI and JavaScript to interact with WebAuthn.

Walkthrough of sgcWebAuthn.html

Structure Overview:

- **Username Input**: Captures the user's identifier.
- **Buttons**:
  - `Register` – initiates WebAuthn registration.
  - `Authenticate` – initiates login (handled similarly).
- **Debug Console**: Shows real-time debug information (JSON from WebAuthn).

1. HTML UI for Input

```
<input type="text" id="username" name="username" autocomplete="username webauthn" />
<button id="btnRegBegin"><strong>Register</strong></button>
```

2. JavaScript: Button Click Handler

```
document.querySelector('#btnRegBegin').addEventListener('click', async () => {
  const username = document.getElementById("username").value;
  if (username == "") {
    document.getElementById('Error').innerText = 'Please enter a username to register';
    return;
}
  const resp = await fetch('<#webauthn_registration_options>', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ username, algorithms: [] })
});
  const options = await resp.json();
const attResp = await startRegistration(options);  // WebAuthn API
```

3. Server Response (Fake Endpoint in HTML)

```
fetch('/sgcWebAuthn/Registration/Options', ...)
fetch('/sgcWebAuthn/Registration/Verify', ...)
```

4. Finalizing Registration

```
const verificationResp = await fetch('/webauthn/register/verify', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(attResp)
});
const verificationJSON = await verificationResp.json();
if (verificationJSON && verificationJSON.verified) {
  document.getElementById('Success').innerHTML = `Authenticator registered!`;
}
```

WebAuthn Authentication

How WebAuthn Authentication works

- **User Initiates Authentication**:
  - The user provides a username and clicks **Register**.
- **Browser Requests Options from Server**:
  - The frontend makes a `POST` request to get authentication options.
- **Browser Creates Credentials**:
  - Using the `navigator.credentials.get()` API via `SimpleWebAuthnBrowser.startAuthentication()`.
- **Server Verifies Authentication**:
  - The browser sends back the credential to the server.
  - The server verifies the signed result.

1. HTML UI Setup

```
<div class="container">
<h1>WebAuthn Authentication Sample</h1>
  <section id="userdata">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" autocomplete="username webauthn" autofocus />
</section>
  <button id="btnAuthBegin"><strong>Authenticate</strong></button>
  <p id="Success" class="success"></p>
<p id="Error" class="error"></p>
  <details open>
    <summary>Console</summary>
    <textarea id="Debug"></textarea>
  </details>
</div>
```

2. Get Authentication Options

Before calling startAuthentication, you send the username to the server

```
const resp = await fetch('<#webauthn_authentication_options>', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    username: document.getElementById("username").value,
    user_verification: 'preferred'
  }),
});
```

The server responds with a JSON object that includes:

```
{
  "challenge": "base64url-encoded-random-string",
  "allowCredentials": [
    {
      "id": "base64url-credential-id",
      "type": "public-key"
    }
  ],
  "userVerification": "preferred",
  "rpId": "yourdomain.com"
}
```

This is called the PublicKeyCredentialRequestOptions.

3. Receive the Authenticator Response

The asseResp looks like this (simplified):

```
{
  "id": "credentialId",
  "rawId": "base64url-encoded-id",
  "response": {
    "authenticatorData": "...",
    "clientDataJSON": "...",
    "signature": "...",
    "userHandle": "..."
  },
  "type": "public-key",
  "clientExtensionResults": {}
}
```

This response proves that the user:

- Possesses the private key stored on the authenticator
- Signed the server's challenge
- Was physically present (if required)

4. Send the Signed Authentication Response to the Server

After the user interacts with their authenticator (via startAuthentication()), you get a response object in JavaScript.

```
const verificationResp = await fetch('/webauthn/authenticate/verify', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(asseResp), // this is the signed response
});
```

5. Get the Server's Response

The server will reply with a result like this:

```
{ "verified": true }
```

Or if something went wrong:

```
{ "verified": false, "error": "Invalid signature" }
```

And you handle it in your frontend code:

const result = await verificationResp.json();

```
if (result.verified) {
  document.getElementById('Success').textContent = 'User authenticated!';
} else {
  document.getElementById('Error').textContent = 'Authentication failed!';
}
```

Amazon AWS | SQS

What is Amazon SQS?

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. SQS eliminates the complexity and overhead associated with managing and operating message oriented middleware, and empowers developers to focus on differentiating work. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available.

Benefits

Eliminate administrative overhead

With SQS, there is no upfront cost, no need to acquire, install, and configure messaging software, and no time-consuming build-out and maintenance of supporting infrastructure.

Reliably deliver messages

SQS lets you decouple application components so that they run and fail independently, increasing the overall fault tolerance of the system.

Keep sensitive data secure

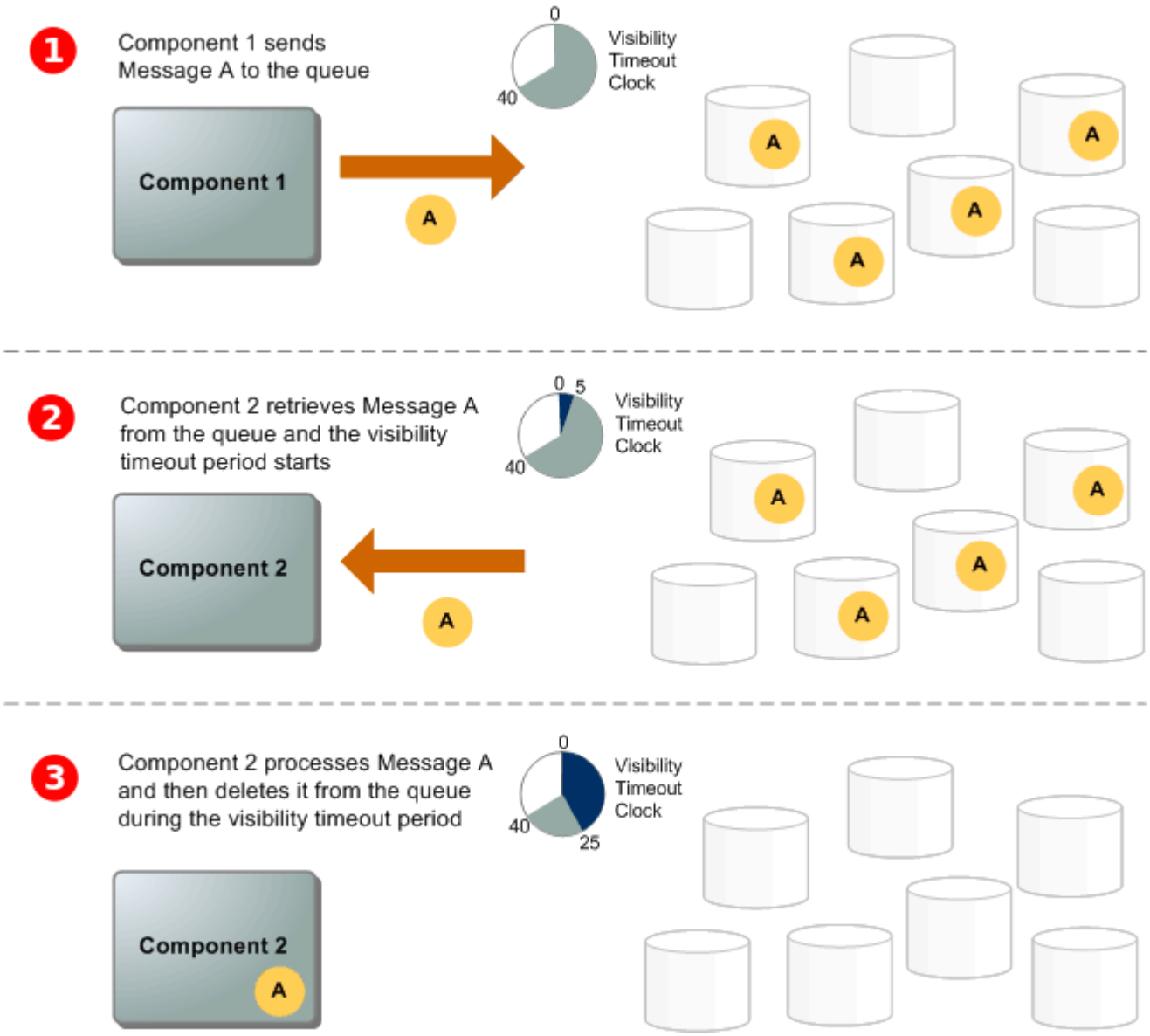
You can use Amazon SQS to exchange sensitive data between applications using server-side encryption (SSE) to encrypt each message body.

Scale elastically and cost-effectively

SQS scales elastically with your application so you don’t have to worry about capacity planning and pre-provisioning.

WorkFlow

The following scenario describes the lifecycle of an Amazon SQS message in a queue, from creation to deletion.



Getting Started with Amazon SQS

Before you begin, complete the steps in Setting Up Amazon SQS.

Step 1: Create a Queue

1. Sign in to the Amazon SQS console.
2. Choose Create New Queue.
3. On the Create New Queue page, ensure that you're in the correct region and then type the Queue Name.
4. Standard is selected by default. Choose FIFO.
5. To create your queue with the default parameters, choose Quick-Create Queue.

Your new queue is created and selected in the queue list.

Step 2: Send a Message

After you create your queue, you can send a message to it. The following example shows sending a message to an existing queue.

1. From the queue list, select the queue that you've created.
2. From Queue Actions, select Send a Message.
3. Your message is sent and the Send a Message to QueueName dialog box is displayed, showing the attributes of the sent message.

Step 3: Receive and Delete Your Message

After you send a message into a queue, you can consume it (retrieve it from the queue). When you request a message from a queue, you can't specify which message to get. Instead, you specify the maximum number of messages (up to 10) that you want to get.

Step 4: Delete Your Queue

If you don't use an Amazon SQS queue (and don't foresee using it in the near future), it is a best practice to delete it from Amazon SQS.

SQS Client

Events

OnSQSBeforeRequest

This event is called before sqs component does an HTTP request. You can get access to URL parameter and if Handled parameter is set to True, means component won't do an HTTP request.

OnSQSError

If there is any error when the component does a request, this event will be called with Error Code and Error Description.

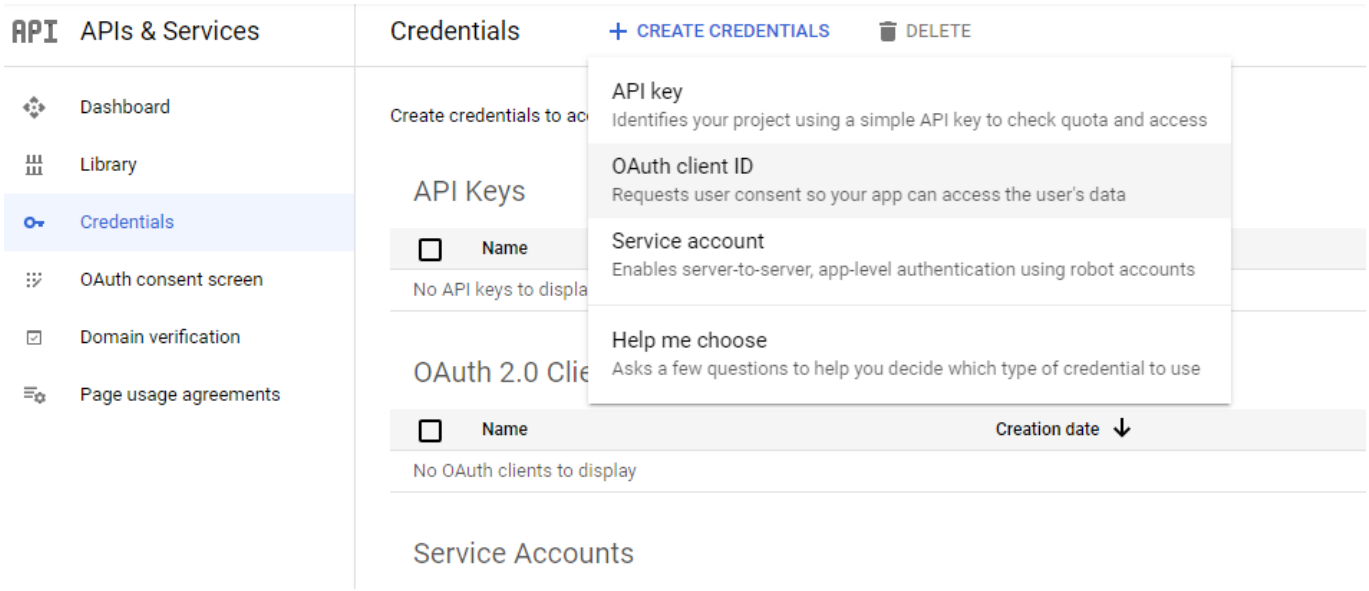
OnSQSResponse

This event is called after an HTTP request with raw response from server.

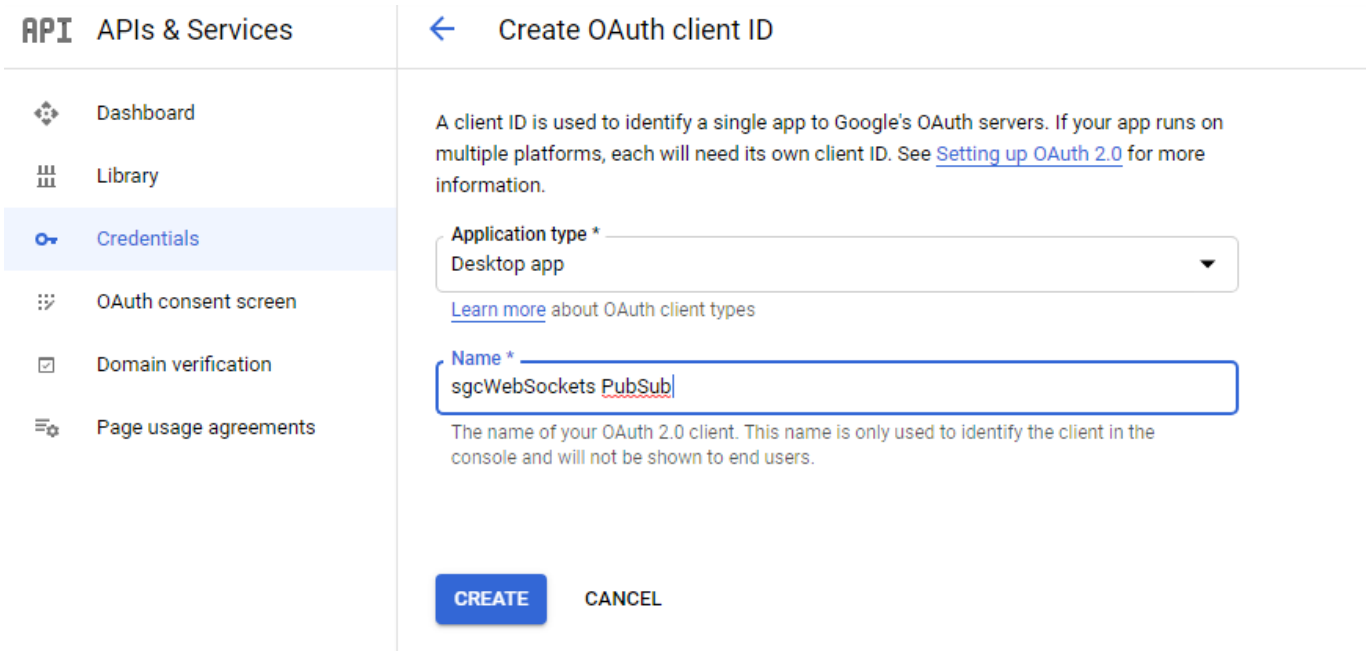
Google Cloud | Google OAuth2 Keys

In order to use the sgcWebSockets Google Cloud components and Authenticate using OAuth2, first you must obtain the OAuth2 Key from Google Cloud.
Find below the steps to get Google OAuth2 Keys and how to configure them in our PubSub sample application.

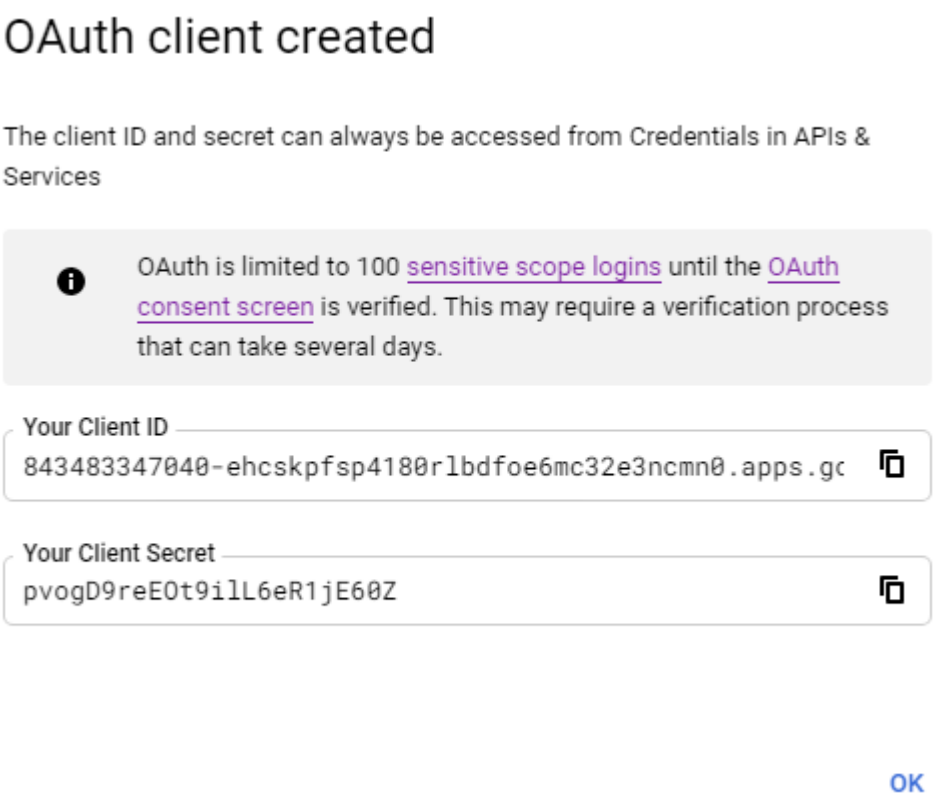
First **login** to your **Google Cloud Account** and use an existing project or create a new one.
After that, go to **Credentials** menu and press the button **CREATE CREDENTIALS**, select the option **OAuth Client ID**.



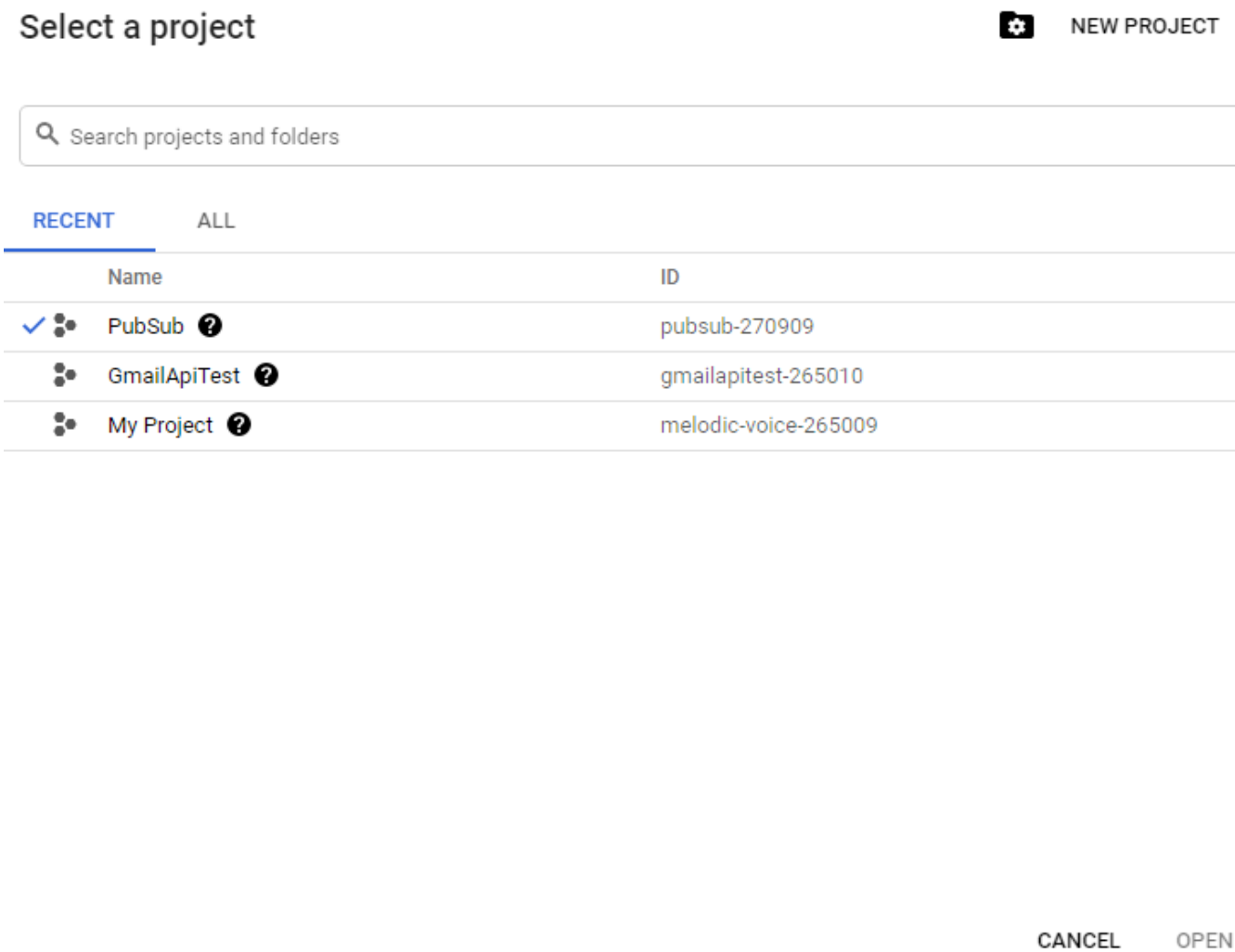
Select your application type and set a descriptive name.



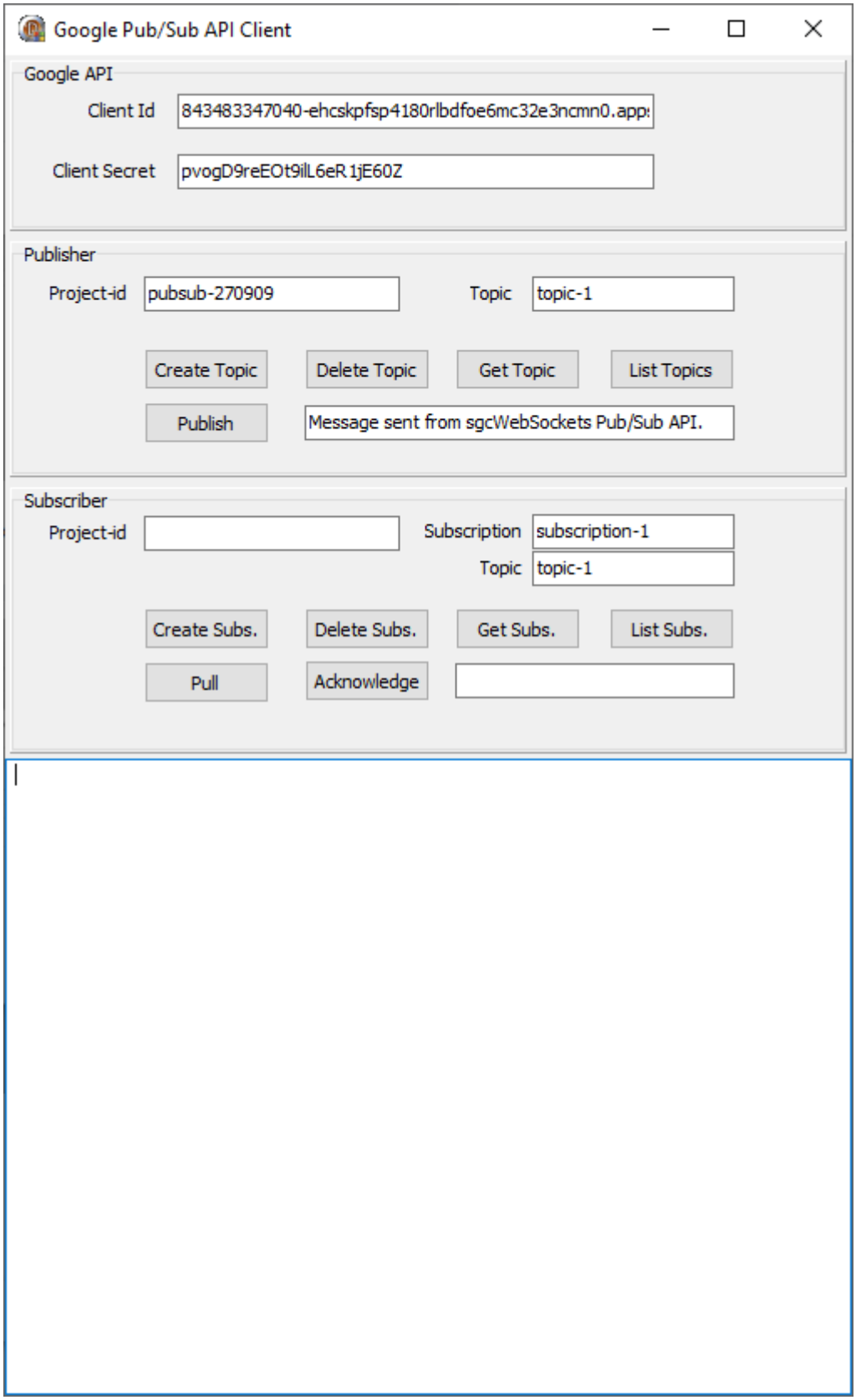
If successful, you will get your Client ID and Client Secret.



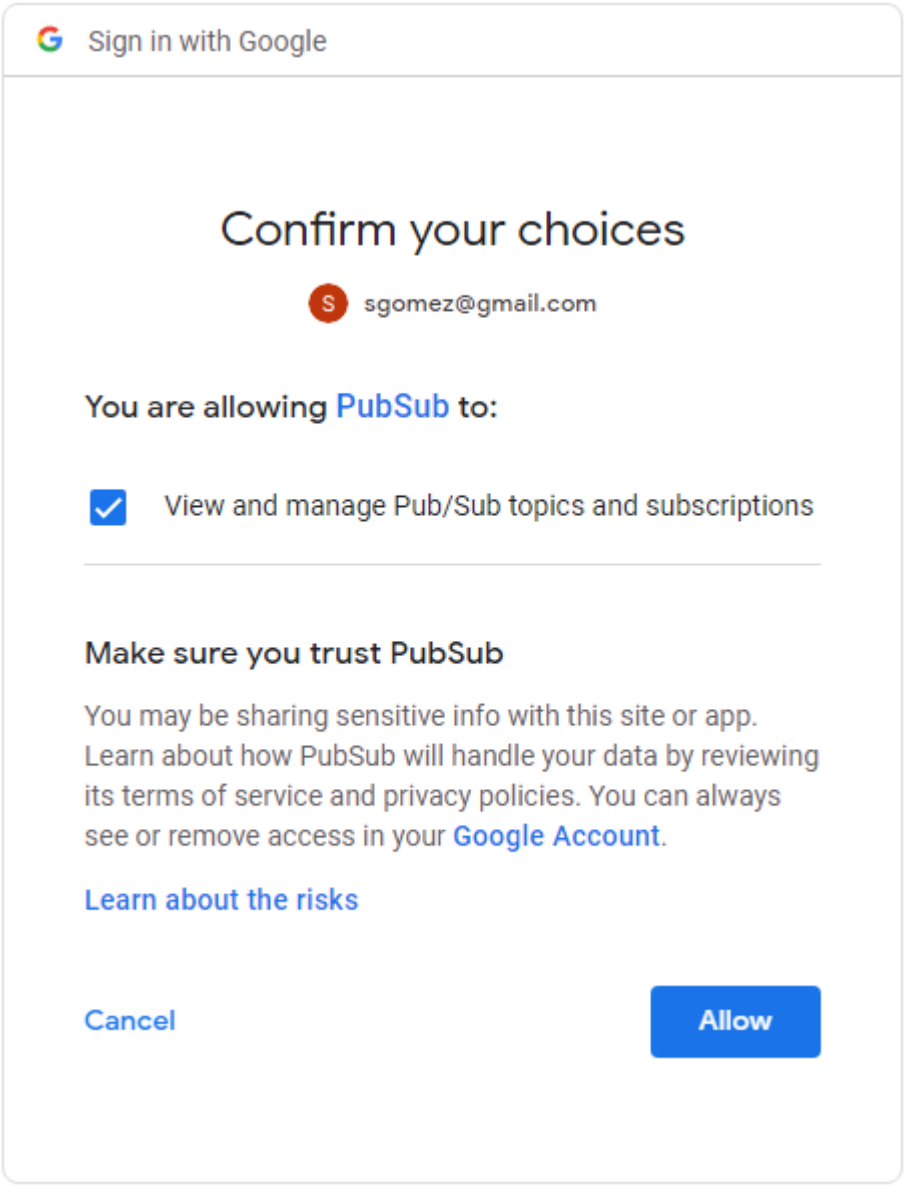
Don't share your OAuth2 data with anyone!
Now copy to the sgcWebSockets PubSub sample, and add the Project Id (NOT the project name)



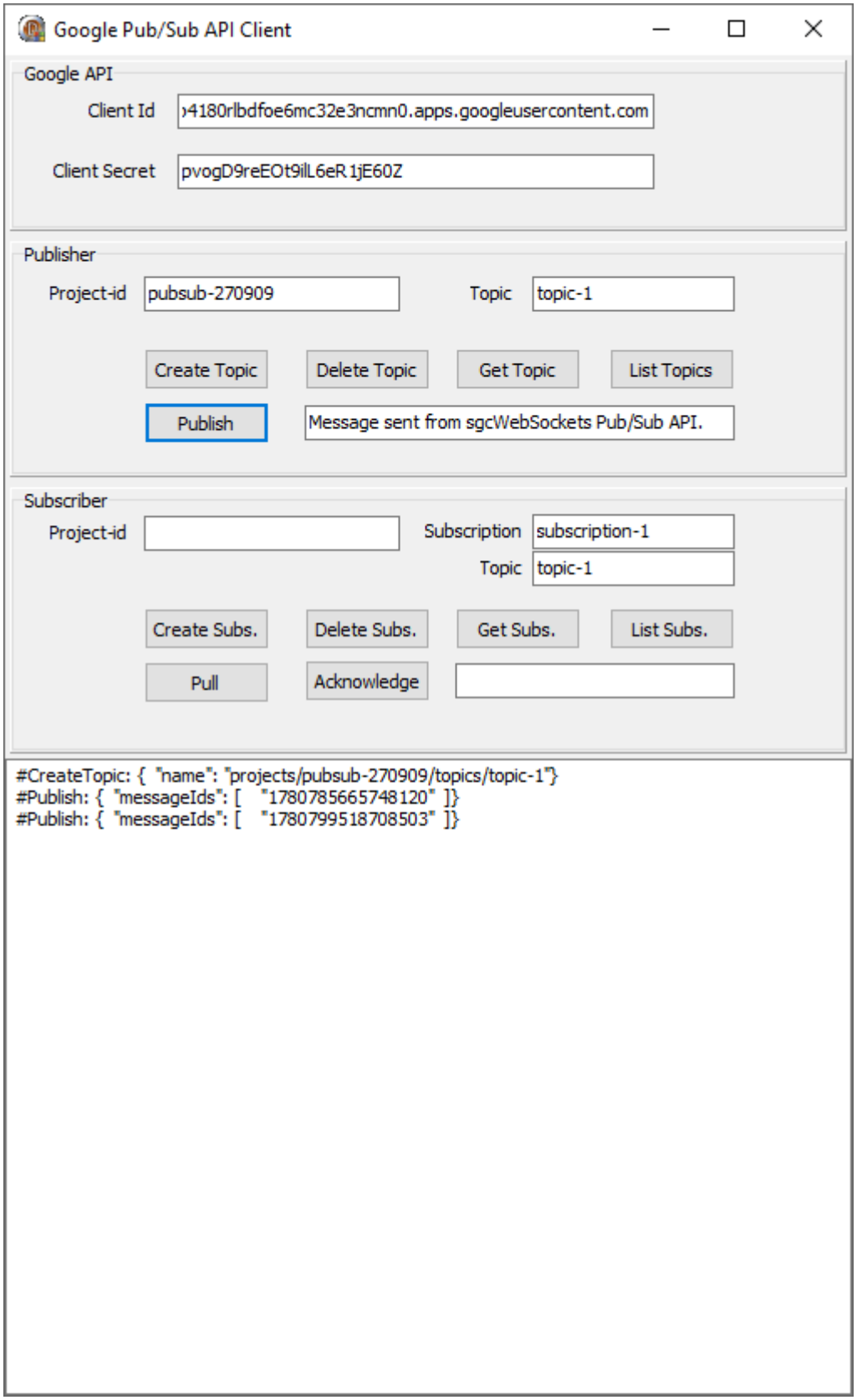
This is how it must be configured in the sgcWebSocket PubSub sample.



Then you can try to create a new topic, for example. The first time, you must authorize the OAuth2 connection, so a new web browser will be shown to request authorization to access your account with the OAuth2 credentials provided by Google.



Allow the connection, and if successful, you can start working with this API.


Google Pub/Sub API Client
Google API
Client Id
›4180rlbdfoe6mc32e3ncmn0.apps.googleusercontent.com
Client Secret
pvogD9reEOt9ilL6eR1jE60Z
Publisher
Project-id
pubsub-270909
Topic
topic-1
Create Topic
Delete Topic
Get Topic
List Topics
Publish
Message sent from sgcWebSockets Pub/Sub API.
Subscriber
Project-id
Subscription
subscription-1
Topic
topic-1
Create Subs.
Delete Subs.
Get Subs.
List Subs.
Pull
Acknowledge
#CreateTopic: { "name": "projects/pubsub-270909/topics/topic-1"}
#Publish: { "messageIds": ["1780785665748120"]}
#Publish: { "messageIds": ["1780799518708503"]}

Google Cloud | Pub/Sub

What is Google Cloud Pub/Sub?

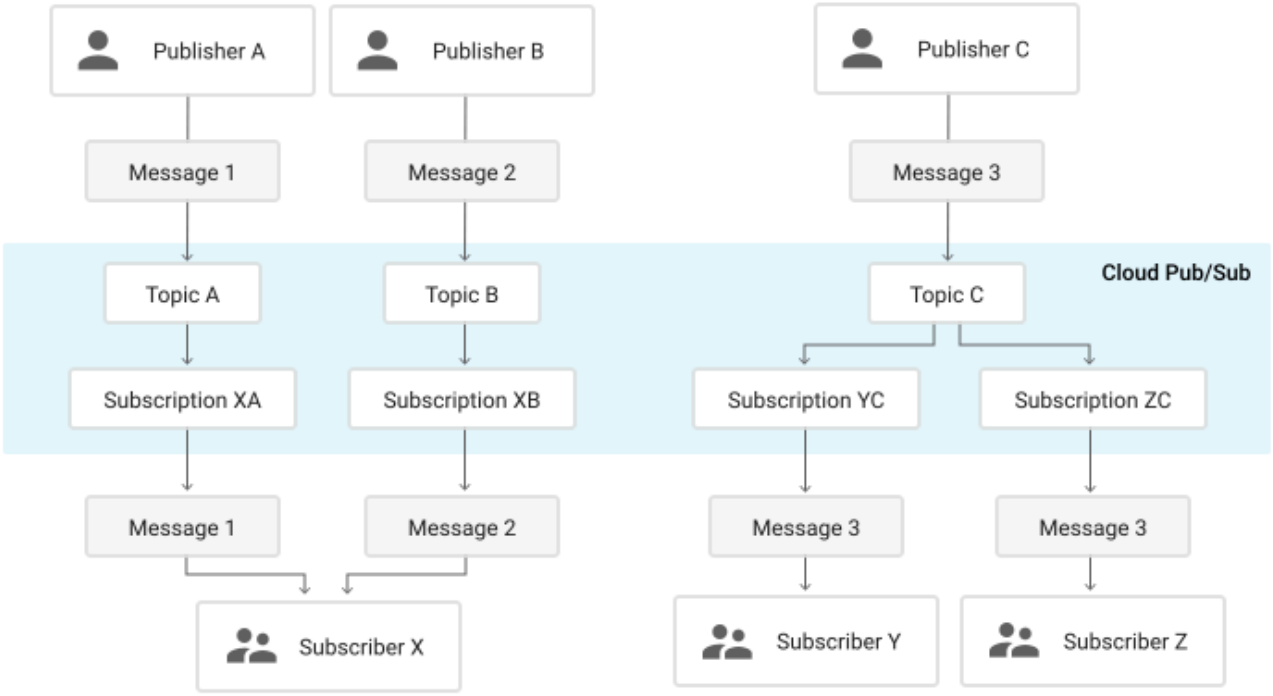
Pub/Sub brings the flexibility and reliability of enterprise message-oriented middleware to the cloud. At the same time, Pub/Sub is a scalable, durable event ingestion and delivery system that serves as a foundation for modern stream analytics pipelines. By providing many-to-many, asynchronous messaging that decouples senders and receivers, it allows for secure and highly available communication among independently written applications. Pub/Sub delivers low-latency, durable messaging that helps developers quickly integrate systems hosted on the Google Cloud Platform and externally.

Features

At-least-once delivery Synchronous, cross-zone message replication and per-message receipt tracking ensures at-least-once delivery at any scale.	**Open** Open APIs and client libraries in seven languages support cross-cloud and hybrid deployments.	**Exactly-once processing** Cloud Dataflow supports reliable, expressive, exactly-once processing of Cloud Pub/Sub streams.
Global by default Publish from anywhere in the world and consume from anywhere, with consistent latency. No replication necessary.	**No provisioning, auto-everything** Cloud Pub/Sub does not have shards or partitions. Just set your quota, publish, and consume.	**Compliance and security** Cloud Pub/Sub is a HIPAA-compliant service, offering fine-grained access controls and end-to-end encryption.
Integrated Take advantage of integrations with multiple services, such as Cloud Storage and Gmail update events and Cloud Functions for serverless event-driven computing.	**Seek and replay** Rewind your backlog to any point in time or a snapshot, giving the ability to reprocess the messages. Fast forward to discard outdated data.	

Publisher-subscriber relationships

A publisher application creates and sends messages to a topic. Subscriber applications create a subscription to a topic to receive messages from it. Communication can be one-to-many (fan-out), many-to-one (fan-in), and many-to-many.



Common use cases

- **Balancing workloads in network clusters.** For example, a large queue of tasks can be efficiently distributed among multiple workers, such as Google Compute Engine instances.
- **Implementing asynchronous workflows.** For example, an order processing application can place an order on a topic, from which it can be processed by one or more workers.
- **Distributing event notifications.** For example, a service that accepts user signups can send notifications whenever a new user registers, and downstream services can subscribe to receive notifications of the event.
- **Refreshing distributed caches.** For example, an application can publish invalidation events to update the IDs of objects that have changed.
- **Logging to multiple systems.** For example, a Google Compute Engine instance can write logs to the monitoring system, to a database for later querying, and so on.
- **Data streaming from various processes or devices.** For example, a residential sensor can stream data to backend servers hosted in the cloud.
- **Reliability improvement.** For example, a single-zone Compute Engine service can operate in additional zones by subscribing to a common topic, to recover from failures in a zone or region.

Authorization

Google Pub/Sub component client can login to Google Servers using the following methods:

- **gcaOAuth2:** OAuth2 protocol
- **gcaJWT:** JWT tokens.

OAuth2

The login is done using a web browser where the user logs in with their own account and authorizes the PubSub requests.

- **GoogleCloudOptions.OAuth2.ClientId:** is the ClientID provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.ClientSecret:** is the Client Secret string provided by Google to Authenticate through OAuth2 protocol.

- **GoogleCloudOptions.OAuth2.Scope:** is the scope of OAuth2, usually there is no need to modify the default value unless you need to get more access than default.
- **GoogleCloudOptions.OAuth2.LocalIP:** the OAuth2 protocol requires a local server listening for the response from the authentication server. This is the IP or DNS name. By default, it is 127.0.0.1.
- **GoogleCloudOptions.OAuth2.LocalPort:** Local server listening port.
- **GoogleCloudOptions.OAuth2.RedirectURL:** if you need to set a redirect URL different from LocalPort + LocalIP, you can set it in this property (example: http://127.0.0.1:8080/oauth2).

Service Accounts

The login is done by signing the requests using a private key provided by Google. This method is recommended for automated services or applications without user interaction.

- **GoogleCloudOptions.JWT.ClientEmail:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKeyId:** the Private Key ID provided by Google. "private_key_id" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKey:** the Private Key certificate provided by Google. "private_key" node in the JSON configuration file.

TLS Options

TLSOptions centralizes the configuration of the secure channel used to communicate with Google Cloud Pub/Sub.

- **TLSOptions.IOHandler:** pick the TLS implementation that fits your deployment (OpenSSL, SChannel...).
- **TLSOptions.Version:** restrict the negotiation to a given TLS version when your infrastructure requires it.
- **TLSOptions.VerifyCertificate:** control certificate validation, enabling stricter security policies.
- **TLSOptions.OpenSSL_Options.LibPath:** set the directory that contains the OpenSSL binaries shipped with your application.
- **TLSOptions.SChannel_Options.UseLegacyCredentials:** activates Windows legacy credential handling when SChannel requires it.

Example:

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.TLSOptions.IOHandler = TwsTLSIOHandler.iohOpenSSL;
oPubSub.TLSOptions.Version = TwsTLSVersions.tls1_3;
oPubSub.TLSOptions.VerifyCertificate = true;
oPubSub.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder;
```

When a new service account is created, you can download a JSON file with all configurations. This file can be processed by the PubSub component, just call the method **LoadSettingsFromFile** and pass the JSON filename as argument.

Most common uses

- **Configuration**
  - **Google OAuth2 Keys**
  - **Service Accounts**

Google Pub/Sub Client

OAuth2

In order to work with Google Pub/Sub API, sgcWebSockets Pub/Sub component uses OAuth2 as default authentication, so first you must set your **ClientId** and **ClientSecret** from your google account.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.GoogleCloudOptions.Authorization = gcaOAuth2;
```

```
oPubSub.GoogleCloudOptions.OAuth2.ClientId = "... your google client id...";
oPubSub.GoogleCloudOptions.OAuth2.ClientSecret = "... your google client secret...";
```

Service Accounts

Service Accounts require building a JWT and passing it as an authorization token.

```
TsgcHTTPGoogleCloud_PubSub_Client oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.GoogleCloudOptions.Authorization = gcaJWT;
oPubSub.GoogleCloudOptions.JWT.ClientEmail = "...google email...";
oPubSub.GoogleCloudOptions.JWT.PrivateKeyId = "...private key id...";
oPubSub.GoogleCloudOptions.JWT.PrivateKey = "...private key certificate...";
```

This is required in order to get an Authorization Token Key from Google which will be used for all Rest API calls.

All methods return a response, which may be successful or return an error.

Projects.Snapshots

Method	Parameters	Description	Example
CreateSnapshot	project, snapshot, subscription	Creates a snapshot from the requested subscription. Snapshots are used in subscriptions.seek operations, which allow you to manage message acknowledgments in bulk. That is, you can set the acknowledgment state of messages in an existing subscription to the state captured by a snapshot.	`CreateSnapshot('pubsub-270909', 'snapshot-1', 'subscription-1')`
DeleteSnapshot	project, snapshot	Removes an existing snapshot	`DeleteSnapshot('pubsub-270909', 'snapshot-1')`
ListSnapshots	project	Lists the existing snapshots	`ListSnapshots('pubsub-270909')`

Projects.Subscriptions

Method	Parameters	Description	Example
AcknowledgeSub-			

scrip-tion			
Create-Sub-scrip-tion	project, subscrip-tion, top-ic	Creates a subscrip-tion to a given topic. If the subscription already exists, re-turns ALREADY_EXISTS. If the corresponding topic doesn't exist, returns NOT_FOUND.	`CreateSubscription('pubsub-270909', 'subscription-1', 'topic-1')`
Delete-Sub-scrip-ton	project, subscrip-tion	Deletes an existing subscription. All messages retained in the subscription are immediately dropped.	`DeleteSubscription('pubsub-270909', 'subscription-1')`
Get-Sub-scrip-tion	project, subscrip-tion	Gets the configura-tion details of a sub-scription.	`GetSubscription('pubsub-270909', 'subscription-1')`
List-Sub-scrip-tions	project	Lists matching sub-scriptions.	`ListSubscriptions('pubsub-270909', 'subscription-1')`
Modify-Ack-Deadli-neSub-scrip-tion	project, subscrip-tion, Ack-Ids	Modifies the ack deadline for a spe-cific message. This method is useful to indicate that more time is needed to process a message by the subscriber, or to make the mes-sage available for redelivery if the pro-cessing was inter-rupted. Note that this does not modify the subscription-lev-el ackDeadlineSec-onds used for sub-sequent messages.	
Modify-Push-Config-Sub-scrip-tion	project, subscrip-tion	Modifies the Push-Config for a speci-fied subscription.This may be used to change a push sub-scription to a pull one (signified by an empty PushConfig) or vice versa, or change the end-point URL and other attributes of a push subscription. Mes-sages will accumu-late for delivery con-tinuously through the call regardless of changes to the PushConfig.	

Pull	project, subscription	Pulls messages from the server. The server may return UNAVAILABLE if there are too many concurrent pull requests pending for the given subscription.	`pull('pubsub-270909', 'subscription-1')`
Seek	project, subscription, timeUTC, snapshot	Seeks an existing subscription to a point in time or to a given snapshot, whichever is provided in the request. Snapshots are used in subscriptions.seek operations, which allow you to manage message acknowledgments in bulk. That is, you can set the acknowledgment state of messages in an existing subscription to the state captured by a snapshot. Note that both the subscription and the snapshot must be on the same topic.	

Projects.Topics

Method	Parameters	Description	Example
CreateTopic	project, topic	Creates the given topic with the given name	`CreateTopic('pubsub-270909', 'topic-1')`
DeleteTopic	project, topic	Deletes the topic with the given name. Returns NOT_FOUND if the topic does not exist. After a topic is deleted, a new topic may be created with the same name; this is an entirely new topic with none of the old configuration or subscriptions.	`DeleteTopic('pubsub-270909', 'topic-1')`
GetTopic	project, topic	Gets the configuration of a topic.	`GetTopic('pubsub-270909', 'topic-1')`
ListTopics	project	Lists matching topics.	`ListTopics('pubsub-270909')`

Publish	project, topic, message	Adds one or more messages to the topic. Returns NOT_FOUND if the topic does not exist.	`Publish('pubsub-270909', 'topic-1', 'My First PubSub Message.')`

Projects.Topics.Subscriptions

Method	Parameters	Description	Example
ListTopicSubscriptions	project, topic	Lists the names of the subscriptions on this topic.	`ListTopicSubscriptions('pubsub-270909', 'topic-1')`

Most common methods

Find below the most common methods used with the Google Cloud Pub/Sub API.

How to create a new Topic

Create a new topic for project with id: pubsub-270909 and topic name topic-1.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.GoogleCloudOptions.OAuth2.ClientId = "... your google client id...";
oPubSub.GoogleCloudOptions.OAuth2.ClientSecret = "... your google client secret...";
oPubSub.CreateTopic("pubsub-270909", "topic-1");
```

Response from Server

```
{
  "name": "projects/pubsub-270909/topics/topic-1"
}
```

Publish a message

Publish a new message in the newly created topic.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.GoogleCloudOptions.OAuth2.ClientId = "... your google client id...";
oPubSub.GoogleCloudOptions.OAuth2.ClientSecret = "... your google client secret...";
oPubSub.Publish("pubsub-270909", "topic-1", "My First Message from sgcWebSockets."));
```

Response from Server

```
{
  "messageIds": [
    "1050732082561505"
```

```
  ]
}
```

Publish a Message with Attributes

```
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.", "origin=gcloud-sample,usernam
```

How to Create a new Subscription

Create a new subscription for project with id: pubsub-270909, with subscription name subscription-1 and topic-1

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.GoogleCloudOptions.OAuth2.ClientId = "... your google client id...";
oPubSub.GoogleCloudOptions.OAuth2.ClientSecret = "... your google client secret...";
oPubSub.CreateSubscription("pubsub-270909", "subscription-1", "topic-1");
```

Response from Server

```
{
  "name": "projects/pubsub-270909/subscriptions/subscription-1",
  "topic": "projects/pubsub-270909/topics/topic-1",
  "pushConfig": {},
  "ackDeadlineSeconds": 10,
  "messageRetentionDuration": "604800s",
  "expirationPolicy": {
    "ttl": "2678400s"
  }
}
```

How to Read messages from a Subscription

Read messages from previous subscription created.

```
oPubSub = new TsgcHTTPGoogleCloud_PubSub_Client();
oPubSub.GoogleCloudOptions.OAuth2.ClientId = "... your google client id...";
oPubSub.GoogleCloudOptions.OAuth2.ClientSecret = "... your google client secret...";
oPubSub.pubsub.Pull("pubsub-270909", "subscription-1");
```

Response from Server

```
{
  "receivedMessages": [
    {
      "ackId": "PjA-RVNEUAYWLF1GSFE3GQhoUQ5PXiM_NSAoRREFC08CKF15MEorQVh0Dj4N",
      "message": {
        "data": "TXkgRmlyc3QgTWVzc2FnZSBmcm9tIHNnY1dlYlNvY2tldHMu",
        "messageId": "1050732082561505",
        "publishTime": "2020-03-14T15:25:31.505Z"
      }
    }
  ]
}
```

Message is received encoded in Base64, so you must decode first to read contents.

```
sgcBase_Helpers.DecodeBase64("TXkgRmlyc3QgTWVzc2FnZSBmcm9tIHNnY1dlYlNvY2tldHMu=");
```

Google Cloud | Calendar

The Google Calendar API lets you integrate your app with Google Calendar, creating new ways for you to engage your users. The Calendar API lets you display, create and modify calendar events as well as work with many other calendar-related objects, such as calendars or access controls.

API Resources

Google Calendar uses the following resources:

- **Event:** An event on a calendar containing information such as the title, start and end times, and attendees. Events can be either single events or recurring events. An event is represented by an Event resource. The Events collection for a given calendar contains all event resources for that calendar.
- **Calendar:** A calendar is a collection of events. Each calendar has associated metadata, such as calendar description or default calendar time zone. The metadata for a single calendar is represented by a Calendar resource. The Calendars collection contains Calendar resources for all existing calendars.
- **CalendarList:** A list of all calendars on a user's calendar list in the Calendar UI. The metadata for a single calendar that appears on the calendar list is represented by a CalendarListEntry resource. This metadata includes user-specific properties of the calendar, such as its color or notifications for new events. The CalendarList collection contains all CalendarListEntry resources for a given user. For a further explanation of the difference between the Calendars and CalendarList collections, see Calendar and Calendar List
- **Setting:** A user preference from the Calendar UI, such as the user's time zone. A single user preference is represented by a Setting Resource. The Settings collection contains all Setting resources for a given user.
- **ACL:** An access control rule granting a user (or a group of users) a specified level of access to a calendar. A single access control rule is represented by an ACL resource. The ACL collection for a given calendar contains all ACL resources that grant access to that calendar.
- **Color:** A color presented in the Calendar UI. The Colors resource represents the set of all colors available in the Calendar UI, in two groups: colors available for events and colors available for calendars.
- **Free/busy:** A time when a calendar has events scheduled is considered "busy", a time when a calendar has no events is considered "free". The Freebusy resource allows querying for the set of busy times for a given calendar or set of calendars.

Main Features

- Fully Featured Google Calendar Client API V3.
- All Methods supported by API can be called using client API.
- Client requests using HTTP/2 protocol (*only Enterprise Edition).
- Automatic Handling of partial responses using PageNextToken.
- Easy access to Calendar and Event data properties.
- Authentication methods:
    - OAuth2: requires user interaction.
    - Service Accounts (requires Domain-Wide Delegation): for windows services, daemons...

Configuration

Google Calendar component client has the following properties:

OAuth2

- **GoogleCloudOptions.OAuth2.ClientId:** is the ClientID provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.ClientSecret:** is the Client Secret string provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.Scope:** is the scope of OAuth2, usually there is no need to modify the default value unless you need to get more access than default.
- **GoogleCloudOptions.OAuth2.LocalIP:** the OAuth2 protocol requires a local server listening for the response from the authentication server. This is the IP or DNS name. By default, it is 127.0.0.1.

- **GoogleCloudOptions.OAuth2.LocalPort:** Local server listening port.
- **GoogleCloudOptions.OAuth2.RedirectURL:** if you need to set a redirect URL different from LocalPort + LocalIP, you can set it in this property (example: http://127.0.0.1:8080/oauth2).

You can modify the Scopes of your client API using Scopes property, just select which scopes are supported by your client.

JWT

The login is done signing the requests using a private key provided by google, this method is recommended for automated services or applications without user interaction. Requires configure the Service Account with Domain-Wide Delegation.

- **GoogleCloudOptions.JWT.ClientEmail:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKeyId:** the Private Key ID provided by Google. "private_key_id" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKey:** the Private Key certificate provided by Google. "private_key" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.Subject:** is the workspace email account linked to the service account using Domain-Wide Delegation.

TLS Options

Use the **TLSOptions** property to customize the secure connection established with Google servers.

- **TLSOptions.IOHandler:** selects the TLS stack (OpenSSL, SChannel...).
- **TLSOptions.Version:** forces a specific TLS protocol version (for example tls1_2 or tls1_3).
- **TLSOptions.VerifyCertificate:** enables or disables server certificate validation.
- **TLSOptions.OpenSSL_Options.LibPath:** points to the folder that contains the OpenSSL libraries deployed with the application.
- **TLSOptions.SChannel_Options.UseLegacyCredentials:** enables the legacy credential flow required by some Windows versions when using SChannel.

The following snippets show how to configure the TLS options:

```
oCalendar = new TsgcHTTPGoogleCloud_Calendar_Client();
oCalendar.TLSOptions.IOHandler = TwsTLSIOHandler.iohOpenSSL;
oCalendar.TLSOptions.Version = TwsTLSVersions.tls1_3;
oCalendar.TLSOptions.VerifyCertificate = true;
oCalendar.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder;
```

Most common uses

- **Configuration**
  - Google Calendar Refresh Token
  - Google Calendar Service Account
- **Synchronization**
  - Google Calendar Sync Calendars
  - Google Calendar Sync Events

Synchronize Calendars

TsgcHTTPGoogleCloud_Calendar_Client component allows you to synchronize the calendars using direct Google API calls or using our easy Calendars methods to synchronize the calendars.

Method	Parameters	Description
NewCalendar	**aSummary:** the title of the calendar.	Creates a new Calendar

DeleteCalendar	**aId:** identifier of the calendar.	Deletes an existing Calendar.
UpdateCalendar	**aResource:** object with the calendar data.	Updates an existing Calendar.
LoadCalendars		Loads all calendars and Calendars property is filled with this data.
LoadCalendarsChanged	**aSyncToken:** last token used to update your calendar.	Loads all changes in your calendars from Token set.

Calendar Client has a property called **Calendars**, where you can access the calendar data after calling any of the previous methods. This property is synchronized automatically.

Synchronize Events

TsgcHTTPGoogleCloud_Calendar_Client component allows you to synchronize the events using direct Google API calls or using our easy Event methods to synchronize the Events.

Method	Parameters	Description
NewEvent	**aCalendarId:** id of the calendar. **aResource:** object with the event data.	Creates a new Event.
DeleteEvent	**aCalendarId:** id of the calendar. **aId:** identifier of the event.	Deletes an existing Event.
UpdateEvent	**aCalendarId:** id of the calendar. **aResource:** object with the event data.	Updates an existing Event.
LoadEvents	**aCalendarId:** id of the calendar.	Loads all events of the calendar.
LoadEventsChanged	**aCalendarId:** id of the calendar. **aSyncToken:** last token used to update your calendar.	Loads all events of the calendar from Token set.

You can access event data using the **Calendars** property. Select any of the existing calendars from the list and access its **Events** property.

Google Calendar API Calls

Method	Description
ACL_Delete	Deletes an access control rule.
ACL_Get	Returns an access control rule.
ACL_Insert	Creates an access control rule.
ACL_List	Returns the rules in the access control list for the calendar.

ACL_Patch	Updates an access control rule. This method supports patch semantics.
ACL_Update	Updates an access control rule.
ACL_Watch	Watch for changes to ACL resources.

Method	Description
CalendarList_Delete	Removes a calendar from the user's calendar list.
CalendarList_Get	Returns a calendar from the user's calendar list.
CalendarList_Insert	Inserts an existing calendar into the user's calendar list.
CalendarList_List	Returns the calendars on the user's calendar list.
CalendarList_Patch	Updates an existing calendar on the user's calendar list. This method supports patch semantics.
CalendarList_Update	Updates an existing calendar on the user's calendar list.
CalendarList_Watch	Watch for changes to CalendarList resources.

Method	Description
Calendar_Clear	Clears a primary calendar. This operation deletes all events associated with the primary calendar of an account.
Calendar_Delete	Deletes a secondary calendar. Use calendars.clear for clearing all events on primary calendars.
Calendar_Get	Returns metadata for a calendar.
Calendar_Insert	Creates a secondary calendar.
Calendar_Patch	Updates metadata for a calendar. This method supports patch semantics.
Calendar_Update	Updates metadata for a calendar.

Method	Description
Channel_Stop	Stop watching resources through this channel.

Method	Description
Color_Get	Returns the color definitions for calendars and events.

Method	Description
Event_Delete	Deletes an event.
Event_Get	Returns an event.
Event_Import	Imports an event. This operation is used to add a private copy of an existing event to a calendar.
Event_Insert	Creates an event.
Event_Instances	Returns instances of the specified recurring event.
Event_List	Returns events on the specified calendar.
Event_Move	Moves an event to another calendar, i.e. changes an event's organizer.
Event_Patch	Updates an event. This method supports patch semantics. The field values you specify replace the existing values. Fields that you don't specify in the request remain unchanged. Array fields, if specified, overwrite the existing arrays; this discards any previous array elements.
Event_QuickAdd	Creates an event based on a simple text string.
Event_Update	Updates an event.
Event_Watch	Watch for changes to Events resources.

Method	Description
Freebusy_Query	Returns free/busy information for a set of calendars.

Method	Description
Settings_Get	Returns a single user setting.
Settings_List	Returns all user settings for the authenticated user.
Settings_Watch	Watch for changes to Settings resources.

Switch Account

If you want to switch from one account to another, first call the method **Clear**, to erase the current calendar session and configure the new session.

Google Calendar | Load Calendars

The process to get all calendars from your account is very easy. Just follow the next steps:

1. Call the method **LoadCalendars**.
2. If the method returns True, then you can access the **Calendars** property and iterate over the list to get access to all Calendars.

```
TsgcHTTPGoogleCloud_Calendar_Client oGoogleCalendar = new TsgcHTTPGoogleCloud_Calendar_Client();
// ... 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++)
  {
    string vCalendarTitle = oGoogleCalendar.Calendars.Calendar[i].Summary;
  }
}
else
{
  throw new Exception("Error Calendar Sync");
}
```

Google Calendar | Sync Events

The process to get all events from a calendar is very easy. Just follow the next steps:

1. Call the method **LoadEvents** and pass the **CalendarId** as parameter**.**
2. If the method returns True, then you can access the **Calendars.Events** property and iterate over the list to get access to all Events of the calendar.

```
TsgcHTTPGoogleCloud_Calendar_Client oGoogleCalendar = new TsgcHTTPGoogleCloud_Calendar_Client();
// ... 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
TsgcGoogleCalendarItem oCalendar = (TsgcGoogleCalendarItem)oGoogleCalendar.Calendars.Calendar[0];
if (oGoogleCalendar.LoadEvents(oCalendar.ID))
{
  // ... get events data
  for (int i = 0; i < oCalendar.Events.Count; i++)
  {
    string vEventTitle = oCalendar.Events[i].Summary;
  }
}
else
{
  throw new Exception("Error Event Sync");
}
```

Google Calendar | RefreshToken

The Google Calendar API uses OAuth2 to authenticate against Google servers. sgcWebSockets has a component that handles the entire authentication process, but if your application closes and you attempt to connect again, you have two options:

1. Authenticate again using your Google APIs
2. Use the Refresh Token (if still valid), so you avoid the authentication process.

Using RefreshToken

The first time you authenticate, use the OnAuthToken event to save the **RefreshToken** if it exists. You can save it in an INI file, for example:

```
void OnGoogleCalendarAuthToken(object sender, string TokenType, string Token, string Data)
{
  TsgcJSON oJSON = new TsgcJSON();
  oJSON.Read(Data);
  if (oJSON.Node["refresh_token"] != null)
  {
    // Save refresh token to your preferred storage
    string refreshToken = oJSON.Node["refresh_token"].Value;
    System.IO.File.WriteAllText("refresh_token.txt", refreshToken);
  }
}
```

Then, when you start your application again, if there is a RefreshToken, call the RefreshToken method and pass the token as an argument (you must first set the Google Calendar API keys). If successful, you will log in to Google servers without having to re-authenticate.

```
GoogleCalendar.RefreshToken("your refresh token here");
```

Google Cloud FCM

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages at no cost.

Using FCM, you can notify a client app that new email or other data is available to sync. You can send notification messages to drive user re-engagement and retention. For use cases such as instant messaging, a message can transfer a payload of up to 4096 bytes to a client app.

The component supports the HTTP v1 API.

Authorization

Google FCM component client can login to Google Servers using the following methods:

- **gcaOAuth2:** OAuth2 protocol
- **gcaJWT:** JWT tokens.

OAuth2

The login is done using a web browser where the user logs in with their own account and authorizes the FCM requests.

- **GoogleCloudOptions.OAuth2.ClientId:** is the ClientID provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.ClientSecret:** is the Client Secret string provided by Google to Authenticate through OAuth2 protocol.
- **GoogleCloudOptions.OAuth2.Scope:** is the scope of OAuth2, usually there is no need to modify the default value unless you need to get more access than default.
- **GoogleCloudOptions.OAuth2.LocalIP:** the OAuth2 protocol requires a local server listening for the response from the authentication server. This is the IP or DNS name. By default, it is 127.0.0.1.
- **GoogleCloudOptions.OAuth2.LocalPort:** Local server listening port.
- **GoogleCloudOptions.OAuth2.RedirectURL:** if you need to set a redirect URL different from LocalPort + LocalIP, you can set it in this property (example: http://127.0.0.1:8080/oauth2).

Service Accounts

The login is done by signing the requests using a private key provided by Google. This method is recommended for automated services or applications without user interaction.

- **GoogleCloudOptions.JWT.ClientEmail:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.Subject:** the client email name provided when creating the new service account. "client_email" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKeyId:** the Private Key ID provided by Google. "private_key_id" node in the JSON configuration file.
- **GoogleCloudOptions.JWT.PrivateKey:** the Private Key certificate provided by Google. "private_key" node in the JSON configuration file.

TLS Options

Firebase Cloud Messaging connections can be tuned through the **TLSOptions** property.

- **TLSOptions.IOHandler:** choose the TLS stack that will be used (OpenSSL, SChannel...).
- **TLSOptions.Version:** request a specific TLS protocol version required by your project.
- **TLSOptions.VerifyCertificate:** enforce server certificate validation when connecting to Google.
- **TLSOptions.OpenSSL_Options.LibPath:** define where the OpenSSL runtime libraries are deployed.
- **TLSOptions.SChannel_Options.UseLegacyCredentials:** enable it when Windows SChannel requires legacy credential negotiation.

Example configuration:

```
oFCM = new TsgcHTTPGoogleCloud_FCM_Client();
oFCM.TLSOptions.IOHandler = TwsTLSIOHandler.iohOpenSSL;
oFCM.TLSOptions.Version = TwsTLSVersions.tls1_3;
oFCM.TLSOptions.VerifyCertificate = true;
oFCM.TLSOptions.OpenSSL_Options.LibPath = oslpDefaultFolder;
```

When a new service account is created, you can download a JSON file with all configurations. This file can be processed by the FCM component, just call the method **LoadSettingsFromFile** and pass the JSON filename as argument.

Most common uses

- **Configuration**
  - **Google OAuth2 Keys**
  - **Service Accounts**

Google FCM Client

OAuth2

In order to work with the Google FCM API, the sgcWebSockets FCM component uses OAuth2 as the default authentication method, so first you must set your **ClientId** and **ClientSecret** from your Google account.

```
oFCM = new TsgcHTTPGoogleCloud_FCM_Client();
oFCM.GoogleCloudOptions.Authorization = gcaOAuth2;
oFCM.GoogleCloudOptions.OAuth2.ClientId = "... your google client id...";
oFCM.GoogleCloudOptions.OAuth2.ClientSecret = "... your google client secret...";
```

Service Accounts

Service Accounts require building a JWT and passing it as an authorization token.

```
TsgcHTTPGoogleCloud_FCM_Client oFCM = new TsgcHTTPGoogleCloud_FCM_Client();
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 = "...private key certificate...";
```

This is required in order to get an Authorization Token Key from Google which will be used for all Rest API calls.

All methods return a response, which may be successful or return an error.

Send Message

Use the method SendMessage to send a notification message. The function has 2 arguments:

1. Project: is the project id of your google cloud account.
2. Payload: is the json text message. Example:

```
{
  "message": {
    "topic": "news",
    "notification": {
      "title": "Breaking News",
```

```
      "body": "New news story available."
    },
    "data": {
      "story_id": "story_12345"
    }
  }
}
```

TsgcWebView2

TsgcWebView2 is a visual VCL component that wraps Microsoft Edge WebView2 (Chromium). Drop it on a form to embed a modern, full-featured web browser in your Delphi application. It supports Delphi 7 through Delphi 13.

Requirements

- Microsoft Edge WebView2 Runtime (included with Windows 10/11 or downloadable from Microsoft)
- WebView2Loader.dll (included, place next to your executable)
- Windows only

Quick Start

Drop TsgcWebView2 on a form and navigate to a URL:

```
sgcWebView21 := TsgcWebView2.Create(Self);
sgcWebView21.Parent := Self;
sgcWebView21.Align := alClient;
sgcWebView21.DefaultURL := 'https://www.example.com';
```

Features

- **Navigation**
  - **Navigate, NavigateToString, GoBack, GoForward, Reload, Stop, POST navigation**
- **JavaScript**
  - **ExecuteScript (async), ExecuteScriptSync, AddInitScript**
- **Cookie Management**
  - **GetCookies, AddOrUpdateCookie, DeleteCookie, DeleteAllCookies**
- **Download Control**
  - **OnDownloadStarting, OnDownloadProgress, OnDownloadCompleted**
- **Settings & Configuration**
  - **ScriptEnabled, DevToolsEnabled, ContextMenuEnabled, and more**
- **Advanced Features**
  - **Print, Audio/Mute, Certificates, Favicon, Virtual Host, Screenshot**

Properties

- **DefaultURL**: string — URL to navigate to after initialization.
- **AutoInitialize**: Boolean — when True, the WebView2 is created automatically when the window handle is allocated. Default: True.
- **ZoomFactor**: Double — zoom level (1.0 = 100%).
- **UserDataFolder**: string — path for cookies, cache, and permissions storage.
- **BrowserExecutableFolder**: string — path to a fixed-version WebView2 Runtime distribution.
- **Settings**: TsgcWebView2Settings — design-time browser settings (ScriptEnabled, DevToolsEnabled, etc.).
- **URL**: string (read-only) — current page URL.
- **DocumentTitle**: string (read-only) — current page title.
- **CanGoBack**: Boolean (read-only) — whether back navigation is possible.
- **CanGoForward**: Boolean (read-only) — whether forward navigation is possible.
- **Initialized**: Boolean (read-only) — whether WebView2 is ready.
- **IsMuted**: Boolean — mute/unmute audio.
- **IsDocumentPlayingAudio**: Boolean (read-only) — whether the page is playing audio.
- **FaviconURI**: string (read-only) — URI of the current page favicon.
- **StatusBarText**: string (read-only) — current status bar text.
- **CookieManager**: TsgcWebView2CookieManager (read-only) — cookie management object.
- **WebView**: ICoreWebView2 (read-only) — direct COM interface access.
- **Controller**: ICoreWebView2Controller (read-only) — direct COM interface access.
- **Environment**: ICoreWebView2Environment (read-only) — direct COM interface access.

Methods

- **Navigate**(aURL: string) — navigate to a URL.
- **NavigateToString**(aHTML: string) — load HTML content directly.
- **GoBack** / **GoForward** — navigate history.
- **Reload** / **Stop** — reload or stop loading.
- **ExecuteScript**(aScript: string) — execute JavaScript asynchronously (result in OnScriptExecuted).
- **ExecuteScriptSync**(aScript: string): string — execute JavaScript synchronously, returns JSON result.
- **AddInitScript**(aScript: string) — add JavaScript that runs on every page load.
- **RemoveInitScript**(aScriptId: string) — remove a previously added init script.
- **NavigateWithPostData**(aURI, aMethod, aPostData, aHeaders: string) — navigate with custom HTTP method, body and headers.
- **PrintToPdf**(aFilePath: string) — save the page as PDF.
- **ShowPrintUI** — show the native print dialog.
- **CapturePreviewToFile**(aFilePath: string; aFormat: Integer) — screenshot (0=PNG, 1=JPEG).
- **OpenDevToolsWindow** — open Edge DevTools.
- **OpenTaskManagerWindow** — open Edge task manager.
- **SetVirtualHostNameToFolderMapping**(aHostName, aFolderPath: string; aAccessKind: Integer) — map hostname to local folder.
- **ClearVirtualHostNameToFolderMapping**(aHostName: string) — remove mapping.
- **PostWebMessageAsString**(aMessage: string) — send message to web content.
- **PostWebMessageAsJson**(aJson: string) — send JSON to web content.
- **GetProfileName**: string — get the browser profile name.
- **ClearBrowsingData**(aKinds: Cardinal) — clear specific browsing data types.
- **ClearAllBrowsingData** — clear all browsing data.
- **PostSharedBufferToScript**(aSharedBuffer: IUnknown; aAccess: Integer; aAdditionalDataAsJson: string) — share memory buffer with web content.
- **InitializeWebView** / **FinalizeWebView** — manual initialization control.

Events

- **OnInitialized** — fired when WebView2 is ready.
- **OnNavigationStarting**(aURI; aIsUserInitiated, aIsRedirected: Boolean; var aCancel: Boolean) — before navigation.
- **OnNavigationCompleted**(aIsSuccess: Boolean; aWebErrorStatus: Integer) — after navigation.
- **OnSourceChanged**(aIsNewDocument: Boolean) — URL changed.
- **OnDocumentTitleChanged**(aTitle: string) — page title changed.
- **OnWebMessageReceived**(aSource, aWebMessageAsJson, aWebMessageAsString: string) — message from web content.
- **OnNewWindowRequested**(aURI: string; aIsUserInitiated: Boolean; var aHandled: Boolean) — popup/new tab requested.
- **OnScriptExecuted**(aErrorCode: HRESULT; aResultAsJson: string) — async JS result.
- **OnContentLoading**(aIsErrorPage: Boolean) — content started loading.
- **OnHistoryChanged** — navigation history changed.
- **OnProcessFailed**(aKind: Integer) — browser process crashed.
- **OnPermissionRequested**(aURI: string; aKind: Integer; aIsUserInitiated: Boolean; var aState: Integer) — permission request.
- **OnWindowCloseRequested** — window.close() called.
- **OnDownloadStarting**(aURI, aResultFilePath: string; var aCancel, aHandled: Boolean; var aFilePath: string) — download started.
- **OnDownloadProgress**(aBytesReceived, aTotalBytes: Int64) — download progress.
- **OnDownloadCompleted**(aFilePath: string; aState: Integer) — download finished.
- **OnContextMenuRequested**(aMenuItems: string; aContextKind: Integer; aLocation: TPoint; var aHandled: Boolean) — right-click menu.
- **OnFaviconChanged**(aFaviconURI: string) — page icon changed.
- **OnClientCertificateRequested**(aHost: string; aPort: Integer; var aHandled: Boolean) — client cert needed.
- **OnServerCertificateError**(aRequestURI: string; aErrorStatus: Integer; var aAction: Integer) — cert error.
- **OnStatusBarTextChanged**(aText: string) — status bar text changed.
- **OnDOMContentLoaded** — DOM ready.
- **OnBasicAuthRequested**(aURI: string; var aUserName, aPassword: string; var aHandled: Boolean) — HTTP basic auth.
- **OnCapturePreviewCompleted**(aErrorCode: HRESULT) — screenshot finished.

TsgcWebView2 | Navigation

TsgcWebView2 provides several methods for navigating to URLs, loading HTML content, and controlling browser history.

Basic Navigation

Use the **Navigate** method to load a URL. The component fires OnNavigationStarting before the request and OnNavigationCompleted when it finishes.

```
// Navigate to a URL
sgcWebView21.Navigate('https://www.example.com');
```

You can also set the **DefaultURL** property at design time or before initialization. The component navigates to this URL automatically after WebView2 is ready.

```
// Set the default URL before initialization
sgcWebView21.DefaultURL := 'https://www.example.com';
```

HTML Content

Use **NavigateToString** to load HTML content directly without a server or file.

```
// Load HTML content directly
sgcWebView21.NavigateToString(
  '<html><body><h1>Hello from Delphi</h1>' +
  '<p>This content was loaded with NavigateToString.</p>' +
  '</body></html>');
```

History Navigation

Use **GoBack** and **GoForward** to navigate the browser history. Check **CanGoBack** and **CanGoForward** before calling these methods.

```
// Navigate back
if sgcWebView21.CanGoBack then
  sgcWebView21.GoBack;

// Navigate forward
if sgcWebView21.CanGoForward then
  sgcWebView21.GoForward;
```

Use **Reload** to refresh the current page and **Stop** to cancel a pending navigation.

```
// Reload the current page
sgcWebView21.Reload;

// Stop loading
sgcWebView21.Stop;
```

POST Navigation

Use **NavigateWithPostData** to send an HTTP request with a custom method, body, and headers. This is useful for submitting form data or calling REST APIs directly in the browser.

```
// POST form data with custom headers
sgcWebView21.NavigateWithPostData(
  'https://api.example.com/login',
```

```
  'POST',
  'username=admin&password=secret',
  'Content-Type: application/x-www-form-urlencoded');
```

```
// POST JSON data
sgcWebView21.NavigateWithPostData(
  'https://api.example.com/data',
  'POST',
  '{"name":"John","age":30}',
  'Content-Type: application/json');
```

Navigation Events

Use **OnNavigationStarting** to inspect or cancel a navigation before it begins. Set **aCancel** to True to block the request.

```
procedure TFormMain.sgcWebView21NavigationStarting(Sender: TObject;
  const aURI: string; aIsUserInitiated, aIsRedirected: Boolean;
  var aCancel: Boolean);
begin
  // Block navigation to unwanted domains
  if Pos('ads.example.com', aURI) > 0 then
    aCancel := True;
end;
```

Use **OnNavigationCompleted** to check whether the navigation succeeded or failed.

```
procedure TFormMain.sgcWebView21NavigationCompleted(Sender: TObject;
  aIsSuccess: Boolean; aWebErrorStatus: Integer);
begin
  if aIsSuccess then
    StatusBar1.SimpleText := 'Page loaded: ' + sgcWebView21.URL
  else
    StatusBar1.SimpleText := 'Navigation failed, error: ' +
      IntToStr(aWebErrorStatus);
end;
```

New Window Handling

When the page opens a popup or a link with target="_blank", the **OnNewWindowRequested** event fires. Set **aHandled** to True to prevent the default popup and navigate in the same window instead.

```
procedure TFormMain.sgcWebView21NewWindowRequested(Sender: TObject;
  const aURI: string; aIsUserInitiated: Boolean;
  var aHandled: Boolean);
begin
  // Navigate in the same window instead of opening a popup
  aHandled := True;
  sgcWebView21.Navigate(aURI);
end;
```

TsgcWebView2 | JavaScript

TsgcWebView2 provides methods to execute JavaScript in the browser context, retrieve results, and establish two-way communication between your Delphi application and web content.

Async Execution

Use **ExecuteScript** to run JavaScript asynchronously. The result is returned in the **OnScriptExecuted** event as a JSON string.

```
// Execute JavaScript asynchronously
sgcWebView21.ExecuteScript('document.title');
```

```
procedure TFormMain.sgcWebView21ScriptExecuted(Sender: TObject;
  aErrorCode: HRESULT; const aResultAsJson: string);
begin
  if aErrorCode = S_OK then
    ShowMessage('Result: ' + aResultAsJson)
  else
    ShowMessage('Script error: ' + IntToStr(aErrorCode));
end;
```

You can execute any valid JavaScript expression. The return value is always serialized as JSON.

```
// Get the number of links on the page
sgcWebView21.ExecuteScript('document.querySelectorAll("a").length');

// Modify page content
sgcWebView21.ExecuteScript(
  'document.body.style.backgroundColor = "lightyellow"');
```

Sync Execution

Use **ExecuteScriptSync** for blocking execution that returns the result directly. This is simpler when you need the value immediately, but it blocks the calling thread until the script completes.

```
var
  vResult: string;
begin
  // Get the page title synchronously
  vResult := sgcWebView21.ExecuteScriptSync('document.title');
  ShowMessage('Title: ' + vResult);

  // Get form field value
  vResult := sgcWebView21.ExecuteScriptSync(
    'document.getElementById("email").value');
  ShowMessage('Email: ' + vResult);
end;
```

Init Scripts

Use **AddInitScript** to register JavaScript that runs automatically on every page load, before any other scripts on the page. This is useful for injecting polyfills, overriding browser APIs, or setting up message handlers.

```
// Add a script that runs on every page load
sgcWebView21.AddInitScript(
  'window.addEventListener("DOMContentLoaded", function() {' +
  '  console.log("Page loaded at: " + new Date().toISOString());' +
  '});');
```

```
// Override window.alert to send messages to Delphi
sgcWebView21.AddInitScript(
  'window.alert = function(msg) {' +
  '  window.chrome.webview.postMessage(msg);' +
  '};');
```

Use **RemoveInitScript** to unregister a previously added init script by its ID.

Web Messaging

Web messaging provides two-way communication between your Delphi application and JavaScript running in the browser. Use **PostWebMessageAsString** or **PostWebMessageAsJson** to send data from Delphi to JavaScript, and handle **OnWebMessageReceived** to receive data from JavaScript.
Sending messages from Delphi to JavaScript:

```
// Send a plain string message
sgcWebView21.PostWebMessageAsString('Hello from Delphi!');

// Send a JSON message
sgcWebView21.PostWebMessageAsJson('{"action":"refresh","id":42}');
```

Receiving messages in JavaScript:

```
// Add an init script to listen for messages from Delphi
sgcWebView21.AddInitScript(
  'window.chrome.webview.addEventListener("message", function(e) {' +
  '  console.log("Received from Delphi: " + e.data);' +
  '});');
```

Receiving messages from JavaScript in Delphi:

```
procedure TFormMain.sgcWebView21WebMessageReceived(Sender: TObject;
  const aSource, aWebMessageAsJson, aWebMessageAsString: string);
begin
  // aWebMessageAsString contains the plain text message
  // aWebMessageAsJson contains the JSON-serialized message
  Memo1.Lines.Add('Message from web: ' + aWebMessageAsString);
end;
```

From JavaScript, send messages to Delphi using:

```
// JavaScript code inside the web page:
// window.chrome.webview.postMessage('Hello from JavaScript!');
// window.chrome.webview.postMessage({action: 'save', data: [1,2,3]});
```

TsgcWebView2 | Cookie Management

TsgcWebView2 exposes a **CookieManager** property for reading, creating, updating, and deleting cookies in the browser context.

Getting Cookies

Use **CookieManager.GetCookies** to retrieve cookies for a given URI. The method returns a list of cookie objects that you can iterate.

```
procedure TFormMain.ButtonGetCookiesClick(Sender: TObject);
var
  vCookies: TsgcWebView2CookieList;
  i: Integer;
begin
  vCookies := sgcWebView21.CookieManager.GetCookies(
    'https://www.example.com');
  try
    for i := 0 to vCookies.Count - 1 do
      Memo1.Lines.Add(
        vCookies[i].Name + ' = ' + vCookies[i].Value);
  finally
    vCookies.Free;
  end;
end;
```

Adding Cookies

Use **CookieManager.AddOrUpdateCookie** to create a new cookie or update an existing one. Specify the name, value, domain, and path at minimum.

```
// Add a session cookie
sgcWebView21.CookieManager.AddOrUpdateCookie(
  'session_id',             // name
  'abc123def456',           // value
  '.example.com',           // domain
  '/');                     // path
```

```
// Add a cookie with an expiration date
sgcWebView21.CookieManager.AddOrUpdateCookie(
  'preferences',            // name
  'theme=dark',             // value
  '.example.com',           // domain
  '/',                      // path
  Now + 30);                // expires in 30 days
```

Deleting Cookies

Use **DeleteCookie** to remove a specific cookie by name and URI. Use **DeleteAllCookies** to clear all cookies from the browser.

```
// Delete a specific cookie
sgcWebView21.CookieManager.DeleteCookie(
  'session_id',
  'https://www.example.com');
```

```
// Delete cookies matching a domain and path
sgcWebView21.CookieManager.DeleteCookiesWithDomainAndPath(
  'session_id',
  '.example.com',
  '/');
```

```
// Delete all cookies
sgcWebView21.CookieManager.DeleteAllCookies;
```

TsgcWebView2 | Download Control

TsgcWebView2 provides events to intercept, monitor, and control file downloads initiated by the browser.

Download Events

The **OnDownloadStarting** event fires when a download begins. You can cancel the download, change the destination file path, or let it proceed with the default behavior.

```
procedure TFormMain.sgcWebView21DownloadStarting(Sender: TObject;
  const aURI, aResultFilePath: string;
  var aCancel, aHandled: Boolean; var aFilePath: string);
begin
  // Log the download
  Memo1.Lines.Add('Download starting: ' + aURI);
  Memo1.Lines.Add('Default path: ' + aResultFilePath);

  // Cancel downloads of .exe files
  if Pos('.exe', LowerCase(aURI)) > 0 then
  begin
    aCancel := True;
    ShowMessage('Executable downloads are blocked.');
  end;
end;
```

Download Progress

The **OnDownloadProgress** event fires periodically during the download, reporting bytes received and total bytes expected.

```
procedure TFormMain.sgcWebView21DownloadProgress(Sender: TObject;
  aBytesReceived, aTotalBytes: Int64);
begin
  if aTotalBytes > 0 then
    ProgressBar1.Position :=
      Round((aBytesReceived / aTotalBytes) * 100)
  else
    ProgressBar1.Position := 0;

  LabelStatus.Caption := Format('Downloaded %d of %d bytes',
    [aBytesReceived, aTotalBytes]);
end;
```

Download Complete

The **OnDownloadCompleted** event fires when the download finishes. Check **aState** to determine whether the download succeeded, was cancelled, or failed.

```
procedure TFormMain.sgcWebView21DownloadCompleted(Sender: TObject;
  const aFilePath: string; aState: Integer);
begin
  case aState of
    0: // Completed
      ShowMessage('Download complete: ' + aFilePath);
    1: // Cancelled
      ShowMessage('Download was cancelled.');
    2: // Failed
      ShowMessage('Download failed.');
  end;
end;
```

Custom Download Path

Set the **aFilePath** parameter in OnDownloadStarting to redirect the download to a custom location. Set **aHandled** to True to suppress the default save dialog.

```
procedure TFormMain.sgcWebView21DownloadStarting(Sender: TObject;
  const aURI, aResultFilePath: string;
  var aCancel, aHandled: Boolean; var aFilePath: string);
begin
  // Redirect all downloads to a custom folder
  aFilePath := 'C:\Downloads\' + ExtractFileName(aResultFilePath);
  aHandled := True; // suppress the default save dialog
end;
```

TsgcWebView2 | Settings

TsgcWebView2 exposes a **Settings** property of type TsgcWebView2Settings that controls browser behavior. These settings can be configured at design time or changed at runtime after initialization.

TsgcWebView2Settings Properties

- **ScriptEnabled**: Boolean — enable or disable JavaScript execution. Default: True.
- **WebMessageEnabled**: Boolean — enable or disable web messaging (postMessage). Default: True.
- **DefaultScriptDialogsEnabled**: Boolean — enable or disable default JavaScript dialogs (alert, confirm, prompt). Default: True.
- **StatusBarEnabled**: Boolean — show or hide the browser status bar. Default: True.
- **DevToolsEnabled**: Boolean — allow or block access to Edge DevTools. Default: True.
- **ContextMenuEnabled**: Boolean — enable or disable the browser right-click context menu. Default: True.
- **ZoomControlEnabled**: Boolean — allow or block user-initiated zoom (Ctrl+scroll, Ctrl+plus/minus). Default: True.
- **BuiltInErrorPageEnabled**: Boolean — show or hide the built-in error page for navigation failures. Default: True.

Design-Time Configuration

Set the Settings properties in the Object Inspector. They are applied automatically after the WebView2 environment is initialized.

```
// These values can also be set in the Object Inspector
sgcWebView21.Settings.ScriptEnabled := True;
sgcWebView21.Settings.DevToolsEnabled := False;
sgcWebView21.Settings.ContextMenuEnabled := False;
sgcWebView21.Settings.ZoomControlEnabled := False;
```

Runtime Configuration

You can change settings at runtime after the component is initialized. Changes take effect immediately.

```
procedure TFormMain.sgcWebView21Initialized(Sender: TObject);
begin
  // Disable right-click menu and DevTools at runtime
  sgcWebView21.Settings.ContextMenuEnabled := False;
  sgcWebView21.Settings.DevToolsEnabled := False;

  // Disable JavaScript dialogs
  sgcWebView21.Settings.DefaultScriptDialogsEnabled := False;
end;
```

UserDataFolder

The **UserDataFolder** property specifies the directory where the browser stores cookies, cache, permissions, and other profile data. Each unique folder creates an isolated browser profile.

```
// Use a custom data folder for isolated profiles
sgcWebView21.UserDataFolder := 'C:\AppData\MyApp\Profile1';
```

Set this property before initialization (before the component creates the WebView2 environment). If not set, a default folder is used in the application's temporary directory.

BrowserExecutableFolder

The **BrowserExecutableFolder** property points to a fixed-version WebView2 Runtime distribution. Use this to ship a specific browser version with your application instead of relying on the system-installed runtime.

```
// Use a fixed-version runtime bundled with the application
sgcWebView21.BrowserExecutableFolder :=
  ExtractFilePath(ParamStr(0)) + 'WebView2Runtime';
```

Set this property before initialization. If left empty, the component uses the system-installed Evergreen WebView2 Runtime.

TsgcWebView2 | Advanced Features

TsgcWebView2 provides access to advanced browser capabilities including printing, screenshots, audio control, certificate handling, and more.

Print Support

Use **PrintToPdf** to save the current page as a PDF file, or **ShowPrintUI** to display the native print dialog.

```
// Save the current page as PDF
sgcWebView21.PrintToPdf('C:\Output\page.pdf');
```

```
// Show the native print dialog
sgcWebView21.ShowPrintUI;
```

Screenshot Capture

Use **CapturePreviewToFile** to take a screenshot of the current page. The format parameter specifies the image type: 0 for PNG, 1 for JPEG. The **OnCapturePreviewCompleted** event fires when the capture finishes.

```
// Capture as PNG
sgcWebView21.CapturePreviewToFile('C:\Output\screenshot.png', 0);

// Capture as JPEG
sgcWebView21.CapturePreviewToFile('C:\Output\screenshot.jpg', 1);
```

```
procedure TFormMain.sgcWebView21CapturePreviewCompleted(
  Sender: TObject; aErrorCode: HRESULT);
begin
  if aErrorCode = S_OK then
    ShowMessage('Screenshot saved.')
  else
    ShowMessage('Screenshot failed: ' + IntToStr(aErrorCode));
end;
```

Audio / Mute Control

Use the **IsMuted** property to mute or unmute audio playback. Check **IsDocumentPlayingAudio** to detect whether the page is currently playing audio.

```
// Toggle mute
sgcWebView21.IsMuted := not sgcWebView21.IsMuted;

// Check if audio is playing
if sgcWebView21.IsDocumentPlayingAudio then
  LabelStatus.Caption := 'Audio is playing';
```

Certificate Handling

Use **OnClientCertificateRequested** to respond when a server requires a client certificate. Use **OnServerCertificateError** to handle TLS certificate errors (for example, self-signed certificates in development).

```
procedure TFormMain.sgcWebView21ClientCertificateRequested(
  Sender: TObject; const aHost: string; aPort: Integer;
  var aHandled: Boolean);
begin
  // Handle client certificate selection
  aHandled := True;
```

```
end;
```

```
procedure TFormMain.sgcWebView21ServerCertificateError(
  Sender: TObject; const aRequestURI: string;
  aErrorStatus: Integer; var aAction: Integer);
begin
  // Accept self-signed certificates in development
  // aAction: 0 = deny, 1 = allow
  aAction := 1;
end;
```

Virtual Host Mapping

Use **SetVirtualHostNameToFolderMapping** to map a hostname to a local folder. This lets web content reference local files using a virtual URL instead of file:// paths.

```
// Map "app.local" to a local folder
sgcWebView21.SetVirtualHostNameToFolderMapping(
  'app.local',
  'C:\MyApp\WebContent',
  0); // 0 = deny remote access

// Now navigate to local content using the virtual host
sgcWebView21.Navigate('https://app.local/index.html');
```

```
// Remove the mapping
sgcWebView21.ClearVirtualHostNameToFolderMapping('app.local');
```

Profile Management

Use **GetProfileName** to retrieve the current browser profile name. Use **ClearBrowsingData** or **ClearAllBrowsing-Data** to remove cached data, cookies, and other browsing artifacts.

```
// Get the profile name
ShowMessage('Profile: ' + sgcWebView21.GetProfileName);

// Clear all browsing data
sgcWebView21.ClearAllBrowsingData;

// Clear specific browsing data types
sgcWebView21.ClearBrowsingData($0001); // cache only
```

Basic Authentication

The **OnBasicAuthRequested** event fires when a server requests HTTP Basic authentication. Provide the creden-tials and set **aHandled** to True.

```
procedure TFormMain.sgcWebView21BasicAuthRequested(Sender: TObject;
  const aURI: string; var aUserName, aPassword: string;
  var aHandled: Boolean);
begin
  aUserName := 'admin';
  aPassword := 'secret';
  aHandled := True;
end;
```

Context Menu

The **OnContextMenuRequested** event fires when the user right-clicks in the browser. Set **aHandled** to True to suppress the default menu and show your own.

```
procedure TFormMain.sgcWebView21ContextMenuRequested(Sender: TObject;
  const aMenuItems: string; aContextKind: Integer;
  aLocation: TPoint; var aHandled: Boolean);
```

```
begin
  // Suppress the default context menu
  aHandled := True;

  // Show a custom popup menu at the click location
  PopupMenu1.Popup(aLocation.X, aLocation.Y);
end;
```

Favicon

The **FaviconURI** property returns the URI of the current page's favicon. The **OnFaviconChanged** event fires when the favicon changes.

```
procedure TFormMain.sgcWebView21FaviconChanged(Sender: TObject;
  const aFaviconURI: string);
begin
  LabelFavicon.Caption := 'Favicon: ' + aFaviconURI;
end;
```

Status Bar

The **StatusBarText** property returns the current status bar text (typically the URL of a hovered link). The **OnStatusBarTextChanged** event fires when the text changes.

```
procedure TFormMain.sgcWebView21StatusBarTextChanged(Sender: TObject;
  const aText: string);
begin
  StatusBar1.SimpleText := aText;
end;
```

Task Manager

Use **OpenTaskManagerWindow** to open the Edge browser task manager, which shows memory and CPU usage for each browser process.

```
sgcWebView21.OpenTaskManagerWindow;
```

Shared Buffer

Use **PostSharedBufferToScript** to share a memory buffer between your Delphi application and web content for high-performance data transfer.

```
// Share a buffer with web content
sgcWebView21.PostSharedBufferToScript(
  vSharedBuffer,  // IUnknown shared buffer object
  0,              // 0 = read-only, 1 = read-write
  '{"type":"image","width":640,"height":480}');
```

Direct COM Access

For advanced scenarios not covered by the component API, use the **WebView**, **Controller**, and **Environment** properties to access the underlying WebView2 COM interfaces directly.

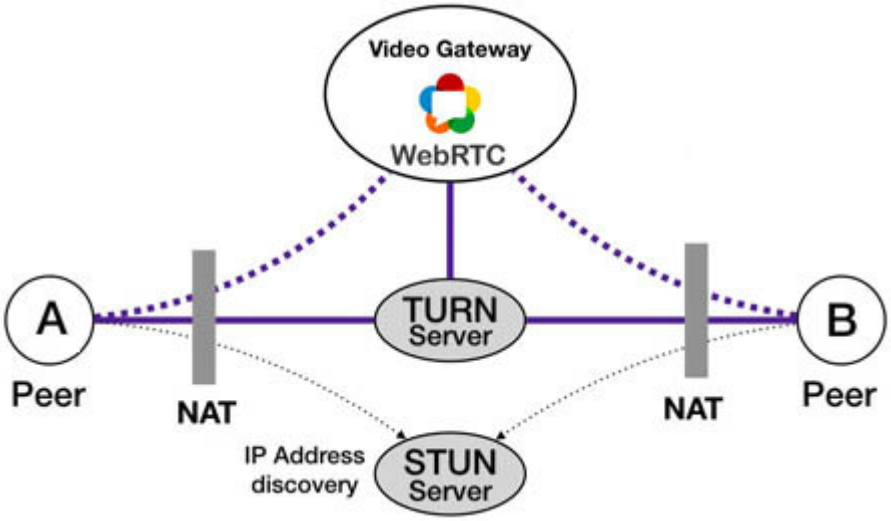
```
var
  vWebView: ICoreWebView2;
begin
  vWebView := sgcWebView21.WebView;
  if Assigned(vWebView) then
  begin
    // Call any ICoreWebView2 method directly
  end;
end;
```

STUN

STUN (Session Traversal Utilities for NAT) is an IETF protocol used for real-time audio and video in IP networks. STUN is a server-client protocol, a STUN server usually operates on both UDP and TCP and listens on port 3478.

The main purpose of the STUN protocol is to enable a device running behind a NAT to discover its public IP and what type of NAT it is behind.

STUN provides a mechanism to communicate between peers behind a NAT. The peers send a request to a STUN server to know which is the public IP address and Port. The binding requests sent from client to server are used to determine the IP and ports bindings allocated by NAT's. The STUN client sends a Binding request to the STUN server, the server examines the source IP and Port used by client, and returns this information to the client.



The STUN server basically sends two types of responses: successful or error. Every response has a list of attributes that contain information about the binding IP address, error code, reason for the error, etc.

Components

- **TsgcSTUNClient:** the client component that implements the STUN protocol and allows you to send binding requests to STUN servers.
- **TsgcSTUNServer:** the server component that implements the STUN protocol.

STUN | TsgcSTUNClient

TsgcSTUNClient is the client that implements the STUN protocol and allows you to send binding requests to STUN servers.

The component allows you to use **UDP** and **TCP** as transport. When using UDP as transport, it implements a **Re-transmission mechanism** to re-send requests if the response has not arrived after a short time.

Basic usage

Usually STUN servers run on UDP port 3478 and don't require authentication, so in order to send a STUN request binding, fill the server properties to allow the client to know where to connect and Handle the events where the component will receive the response from server.

Configure the server

- Host: the IP or DNS name of the server, example: stun.sgcwebsockets.com
- Port: the listening Server port, example: 3478

Call the method **SendRequest**, to send a request binding to STUN server.

Handle the events

- If the server returns a successful response, the event **OnSTUNResponseSuccess** will be called and you can access the binding information by reading the **aBinding** object.
- If the server returns an error, the event **OnSTUNResponseError** will be called and you can access the Error Code and Reason reading the **aError** object.

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient();
oSTUN.Host = "stun.sgcwebsockets.com";
oSTUN.Port = 3478;
oSTUN.SendRequest();

private void OnSTUNResponseSuccess(Component Sender, TsgcSocketConnection aSocket,
  TsgcSTUN_Message aMessage, TsgcSTUN_ResponseBinding aBinding)
{
  DoLog("Remote IP: " + aBinding.RemoteIP + ". Remote Port: " + IntToStr(aBinding.RemotePort));
}

private void OnSTUNResponseError(Component Sender, TsgcSocketConnection aSocket,
  TsgcSTUN_Message aMessage, TsgcSTUN_ResponseError aError)
{
  DoLog("Error: " + Int32.Parse(aError.Code) + " " + aError.Reason);
}
```

Most common uses

- **Bindings**
  - UDP Retransmissions
  - Long Term Credentials
-
  - Attributes

Methods

There is a single method called **SendRequest**, which sends a request to STUN Server, requesting binding information.

Properties

Host: it's the IP Address or DNS name of STUN server where the client will send a binding request.

Port: it's the listening port of STUN server, by default 3478.

IPVersion: it's the Family Address, by default IPv4.

Transport: it's the transport used to connect to STUN server, by default UDP.

STUNOptions: here are defined the specific STUN options of client component

Fingerprint: if enabled, the message includes a fingerprint that aids to identify STUN messages from packets of other protocols when the two are multiplexed on the same transport address.

Software: if enabled, sends an attribute with the name of the software being used by the client.

Authentication: some STUN servers require that requests are authenticated.

- **Credentials:** there are 2 types of Authentication: **LongTermCredentials** and **ShortTermCredentials**. By default the requests are not authenticated
- **Username:** the string that identifies the user.
- **Password:** the secret string.

RetransmissionOptions: when messages are sent using UDP as transport, UDP doesn't includes a mechanism to know if a message has arrived or not to other peer. This property allows you to configure a mechanism to resend UDP messages if not arrived after a small time.

Enabled: if enabled, the message will be re-sent until a confirmation is received or the maximum number of retries has been reached.

RTO: retransmission time in milliseconds, by default 500ms. For example, assuming an RTO of 500 ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500 ms.

MaxRetries: Max number of retries, by default 7.

LogFile: if enabled save stun messages to a specified log file, useful for debugging. The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by client it will be saved on a file.

FileName: full path to the filename.

NotifyEvents: defines which mode to notify WebSocket events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access controls that are not thread-safe, you need to implement your own synchronization methods.

Events

OnSTUNBeforeSend

This event is called before the stun client sends a message to the server. You can access the message properties through the aMessage parameter and modify them if required.

OnSTUNResponseSuccess

When the server processes successfully a request binding, it sends a message with the binding properties (IP Address, Port and family) and other attributes, this event is called when the client receives this successful response.

OnSTUNResponseError

When there is any error in the response sent by server, this event is called with the error details.

OnSTUNException

This event is called when there is any exception processing the STUN protocol messages.

STUN Client | UDP Retransmissions

When running **STUN** over **UDP**, it's possible that the **STUN message** might be **dropped** by the network. Reliability of STUN request/response transactions is accomplished through retransmissions of the request message by the client application itself.

A client should retransmit a STUN request message starting with an interval of RTO ("Retransmission TimeOut"), doubling after each retransmission. The RTO is an estimate of the round-trip time.

By default, the sgcWebSockets STUN Client is already configured with a RTO of 500 ms and a Max Retries value of 7.

For example, assuming an RTO of 500 ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500 ms. If the client has not received a response after 39500 ms, the client will consider the transaction to have timed out.

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient();
oSTUN.Host = "stun.sgcwebsockets.com";
oSTUN.Port = 3478;
oSTUN.RetransmissionOptions.Enabled = true;
oSTUN.RetransmissionOptions.RTO = 500;
oSTUN.RetransmissionOptions.MaxRetries = 7;
oSTUN.SendRequest();
```

STUN Client | Long Term Credentials

The long-term credential mechanism relies on a long-term credential, in the form of a username and password that are shared between client and server. The credential is considered long-term since it is assumed that it is provisioned for a user and remains in effect until the user is no longer a subscriber of the system or until it is changed.

You can configure the Long-term credentials in the sgcWebSockets STUN client using the following code.

```
TsgcSTUNClient oSTUN = new TsgcSTUNClient();
oSTUN.Host = "stun.sgcwebsockets.com";
oSTUN.Port = 3478;
oSTUN.STUNOptions.Authentication.Credentials = TsgcStunCredentials.stauLongTermCredential;
oSTUN.STUNOptions.Authentication.Username = "user_name";
oSTUN.STUNOptions.Authentication.Password = "secret";
oSTUN.SendRequest();
```

If the server requires long-term credentials and the credentials sent by the client are wrong, the client will receive a 401 Unauthorized error as a response in the **OnSTUNResponseError** event.

STUN Client | Attributes

Every time a server sends a message to client, as a response message to a request binding, the STUN message contains a list of attributes with information about the response.

You can access these attributes using the TsgcSTUN_Message class and its Attributes property, which contains a list of TsgcSTUN_Attribute objects with useful information.

STUN | TsgcSTUNServer

TsgcSTUNServer is the server that implements the STUN protocol and allows you to process binding requests from STUN clients.

The STUN server can be configured with or without Authentication, can verify Fingerprint Attribute, send an alternate server and more.

Basic usage

Usually STUN servers run on UDP port 3478 and don't require authentication, so in order to configure a STUN server, set the listening port (by default 3478) and start the server.

Configure the server

- Port: the listening Server port, example: 3478

Set the property **Active = True** to start the STUN server.

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN.Port = 3478;
oSTUN.Active = true;
```

Most common uses

- **Configurations**
  - Long-Term Credentials
  - Alternate Server

Properties

Active: set the property to True to **Start** the STUN server and set to False to **Stop** the Server.

Host: it's the IP Address or DNS name of STUN server.

Port: it's the listening port of STUN server, by default 3478.

IPVersion: it's the Family Address, by default IPv4.

STUNOptions: here are defined the specific STUN options of server component

Fingerprint: if enabled, the message includes a fingerprint that helps identify STUN messages from packets of other protocols when the two are multiplexed on the same transport address.

Software: if enabled, sends an attribute with the name of the software being used by the server.

Authentication: here you can configure if the server requires Authentication requests to send binding responses.

- **Enabled:** set to True if the server requires Authentication requests, by default false.
- **LongTermCredentials:** Enable if the server supports Long-Term credentials. The long-term credential mechanism relies on a long-term credential, in the form of a username and password that are shared between client and server.

- **Enabled:** set to True if the server requires Long-Term credentials.
- **Realm:** the string of the realm sent to client.
- **StaleNonce:** time in seconds after the nonce is no longer valid.

BindingAttributes: when the server sends a successful response after a binding request, here you can customize which attributes will be sent to the client.

- **OtherAddress:** if enabled and the server binds to more than one address, this attribute will be sent with all other addresses except the default one.
- **ResponseOrigin:** is the Local IP of the server to send the response.
- **SourceAddress:** is the Local IP of the server to send the response.

LogFile: if enabled save stun messages to a specified log file, useful for debugging.

Enabled: if enabled every time a message is received and sent by server it will be saved on a file.

FileName: full path to the filename.

NotifyEvents: defines which mode to notify the events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access controls that are not thread-safe, you need to implement your own synchronization methods.

Events

OnSTUNRequestAuthorization

This event is called when a binding request is received and requires authentication.

OnSTUNRequestSuccess

When the server processes successfully a request binding, it sends a message with the binding properties (IP Address, Port and family) and other attributes, this event is called before the message is sent to client.

OnSTUNRequestError

When there is any error in the response sent by the server, this event is called before the message is sent to client.

OnSTUNException

This event is called when there is any exception processing the STUN protocol messages.

STUN Server | Long-Term Credentials

Usually STUN Servers are configured without Authentication, so any STUN client can send a binding request and expect a response from server without Authentication.

sgcWebSockets STUN Server supports Long-Term Credentials, so you can configure TsgcSTUNServer to only allow binding requests with Long-Term credentials info.

To configure it, access to STUNOptions.Authorization property and enable it.
Then access to LongTermCredentials property and enabled it. By default, this type of authorization is already configured with a Realm string and with a default StaleNonce value of 10 minutes (= 600 seconds).

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN.Port = 3478;
oSTUN.STUNOptions.Authentication.Enabled = true;
oSTUN.STUNOptions.Authentication.LongTermCredentials.Enabled = true;
oSTUN.STUNOptions.Authentication.LongTermCredentials.Realm = "sgcWebSockets";
oSTUN.STUNOptions.Authentication.LongTermCredentials.StaleNonce = 600;
oSTUN.Active = true;

private void OnSTUNRequestAuthorization(Component Sender, TsgcSTUN_Message aRequest,
  string aUsername, string aRealm, ref string Password)
{
  if ((aUsername == "my-user") && (aRealm == "sgcWebSockets"))
  {
    Password = "my-password";
  }
}
```

STUN Server | Alternate Server

The alternate server represents an alternate transport address identifying a different STUN server that the STUN client should try.

The STUN Server can be configured to send an alternate server as a response to a binding request, to configure this behaviour, just access to STUNOptions.BindingAttributes.AlternateServer property and configure here the values required.

```
TsgcSTUNServer oSTUN = new TsgcSTUNServer();
oSTUN.Port = 3478;
oSTUN.STUNOptions.BindingAttributes.AlternateServer.Enabled = true;
oSTUN.STUNOptions.BindingAttributes.AlternateServer.IPAddress = "80.54.54.1";
oSTUN.STUNOptions.BindingAttributes.AlternateServer.Port = 3478;
oSTUN.Active = true;
```

When the client receives the Alternate Server response attribute, it will try to send a request binding to the new server.

TURN

Traversal Using Relays around NAT (TURN) protocol enables a server to relay data packets between devices.

If the public IP address of both the caller and callee is not discovered, TURN provides a fallback technique to relay the call between endpoints.

Connecting a WebRTC session is an orchestrated effort done with the assistance of multiple WebRTC servers. The NAT traversal servers in WebRTC are in charge of making sure the media gets properly connected. These servers are STUN and TURN.

How WebRTC sessions connect

Directly

If both devices are on the local network, then no special effort is needed to get them connected to each other. If one device has the local IP address of the other device, then they can communicate with each other directly.

Directly with public IP Address

Connecting WebRTC directly using public IP address obtained via STUN protocol.

Route through a TURN Server

When peers are behind a NAT and there are Firewalls, direct connection is not possible, so a TURN server is required to route the data between the peers.

Components

- **TsgcTURNClient:** the client component that implements the TURN protocol and allows you to allocate, create permissions, send indications, etc. to a TURN server.
- **TsgcTURNServer:** the server component that implements the TURN protocol.

TURN | TsgcTURNClient

TsgcTURNClient is the client that implements the TURN protocol and allows you to send allocation requests to TURN servers. The client inherits from STUN Client, so all methods supported by STUN client are already supported by TURN Client.

Basic usage

Usually TURN servers run on UDP port 3478 and don't require authentication, so in order to send a TURN request, fill the server properties to allow the client know where connect and Handle the events where the component will receive the response from server.

Configure the server

- Host: the IP or DNS name of the server, example: turn.sgcwebsockets.com
- Port: the listening Server port, example: 3478

Call the method **Allocate**, to send a request to allocate an IP Address and a Port to the TURN server.

Handle the events

- If the server returns a successful response, the event **OnTURNAllocateSuccess** will be called and you can access to the Allocation information reading the **aAllocation** object.
- If the server returns an error, the event **OnSTUNResponseError** will be called and you can access the Error Code and Reason reading the **aError** object.

```
TsgcTURNClient oTURN = new TsgcTURNClient();
oTURN.Host = "turn.sgcwebsockets.com";
oTURN.Port = 3478;
oTURN.Allocate();

private void OnTURNAllocate(Component Sender, const TsgcSocketConnection aSocket,
  const TsgcSTUN_Message aMessage, const TsgcTURN_ResponseAllocation aAllocation)
{
  DoLog("Relayed IP: " + aAllocation.RelayedIP + ". Relayed Port: " + aAllocation.RelayedPort.ToString());
}

private void OnSTUNResponseError(Component Sender, const TsgcSTUN_Message aMessage,
const TsgcSTUN_ResponseError aError)
{
  DoLog("Error: " + aError.Code.ToString() + " " + aError.Reason);
}
```

Most common uses

- **Allocation**
  - Allocate IP Address
  - Create Permissions
- **Indications**
  - Send Indication
- **Channels**
  - TURN Client Channels

TURN Relay Data

There are basically 2 ways to send data between peers:

1. **Send Indications**, which encapsulates the data in a STUN packet. Use the method SendIndication to send an indication to other peer.

2. **Use Channel Data**, it's a more efficient way to send data between peers because the packet size is smaller than indications. Use **SendChannelData** method to send a channel data to other peer.

When a TURN server receives a packet in a Relayed IP Address from an IP Address with an active permission, if there is channel data bound to the peer IP Address, the TURN client will receive the data in the event **OnTURN-ChannelData**. But if there is no channel, the TURN client will receive the data in the event **OnTURNData**.

Methods

Allocate

This method sends a request to the server to allocate an IP Address and a Port which will be used to relay data between the peers.

If the server can allocate successfully an IP Address and a Port, the event **OnTURNAllocate** event will be called. If not, the **OnSTUNRequestError** event will be called.

The client saves in the **Allocation** property of the client, the data returned by server about the allocated IP Address.

Refresh

If there is an active allocation, the client can refresh it sending a Refresh request.

This method has a parameter called Lifetime, if the value is zero, the allocation will expire immediately. If the value is greater than zero, it means the number of seconds to expiry.

If the result is successful, the event **OnTURNRefresh** will be called.

CreatePermission

This method creates a new permission for the IP Address set as an argument of the CreatePermission method. If the permission already exists for this IP, it will be refreshed by the server.

If the result is successful, the event **OnCreatePermission** will be called.

SendIndication

This method sends a data to the peer identified as PeerIP and PeerPort. This method requires there is an active permission for this IP in the TURN server.

ChannelBind

This method sends a request to the server to create a new channel to communicate with the peer identified as PeerIP and PeerPort.

if the result is successful, the event **OnChannelBind** will be called. You can access to the channel-id assigned, reading the parameter **aChannelBind** of the event.

SendChannelData

This method sends data to a peer using a ChannelId. This method requires the channel exists and is active.

Properties

Host: it's the IP Address or DNS name of TURN server where the client will send a binding request.

Port: it's the listening port of TURN server, by default 3478.

IPVersion: it's the Family Address, by default IPv4.

Transport: it's the transport used to connect to TURN server, by default UDP.

STUNOptions: here are defined the specific STUN options of client component

Fingerprint: if enabled, the message includes a fingerprint that aids to identify STUN messages from packets of other protocols when the two are multiplexed on the same transport address.

Software: if enabled, sends an attribute with the name of the software being used by the client.

Authentication: some STUN servers requires that requests are authenticated.

- **Credentials:** there are 2 types of Authentication: **LongTermCredentials** and **ShortTermCredentials**. By default the requests are not authenticated
- **Username:** the string that identifies the user.
- **Password:** the secret string.

TURNOptions: here are defined the specific TURN options of client component

Allocation: here are defined the Allocation properties

- **Lifetime:** default lifetime in seconds, by default 600 seconds.

Authentication: usually TURN servers are user protected.

- **Credentials:** by default Long-Term credentials is enabled
- **Username:** the string that identifies the user.
- **Password:** the secret string.

AutoRefresh: when a new allocation is created, requires to be refreshed in order to be used by the peers. Here you can define which methods are automatically refreshed by the TURN Client Component.

- Allocations
- Channels
- Permissions

Fingerprint: if enabled, the message includes a fingerprint that aids to identify STUN messages from packets of other protocols when the two are multiplexed on the same transport address.

Software: if enabled, sends an attribute with the name of the software being used by the client.

RetransmissionOptions: when messages are sent using UDP as transport, UDP doesn't include a mechanism to know if a message has arrived or not to other peer. This property allows you to configure a mechanism to re-send UDP messages if not arrived after a small time.

Enabled: if enabled, the message will be re-send until receives a confirmation or the maximum number of retries has been reached.

RTO: retransmission time in milliseconds, by default 500ms. For example, assuming an RTO of 500 ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500 ms.

MaxRetries: Max number of retries, by default 7.

LogFile: if enabled save stun messages to a specified log file, useful for debugging. The access to log file is not thread safe if it's accessed from several threads.

Enabled: if enabled every time a message is received and sent by client it will be saved on a file.

FileName: full path to the filename.

NotifyEvents: defines which mode to notify WebSocket events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access to controls that are not thread-safe, you need to implement your own synchronization methods.

Events

The TURN client inherits from STUN Client the events: OnSTUNResponseSuccess, OnSTUNResponseError, OnSTUNException and OnSTUNBeforeSend.

Additionally, includes the following events to handle all TURN messages.

OnTURNAllocate

This event is called after a successful IP Address allocation in the TURN server.

OnTURNCreatePermission

This event is called after creating a new permission in the TURN server.

OnTURNRefresh

This event is called after receiving a successful refresh response from the TURN Server.

OnTURNDataIndication

The event is called when the client receives a DATA Indication from other peer.

OnTURNChannelBind

This event is called when the server creates a new channel. Returns the new channel-id created.

OnTURNChannelData

The event is called when the client receives new Data from a Channel previously created.

TURN Client | Allocate IP Address

TURN Protocol allows you to use a Relayed IP Address to exchange data between peers that are behind NATs.

To create a new Relayed IP Address on a TURN server, the client must first call the method **Allocate**, this method sends a Request to the TURN server to create a new Relayed IP Address, if the TURN server can create a new Relayed IP Address, the client will receive a successful response. The client will be able to communicate with other peers during the time defined in the Allocation's lifetime.

```
TsgcTURNClient oTURN = new TsgcTURNClient();
oTURN.Host = "turn.sgcwebsockets.com";
oTURN.Port = 3478;
oTURN.Allocate();


private void OnTURNAllocate(Component Sender, TsgcSocketConnection aSocket, TsgcSTUN_Message aMessage,
  const TsgcTURN_ResponseAllocation aAllocation)
{
  DoLog("Relayed IP: " + aAllocation.RelayedIP + ". Relayed Port: " +
    aAllocation.RelayedPort.ToString());
}

private void OnSTUNResponseError(Component Sender, TsgcSTUN_Message aMessage,
  TsgcSTUN_ResponseError aError)
{
  DoLog("Error: " + aError.Code.ToString() + " " + aError.Reason);
}
```

The lifetime can be updated to avoid expiration using the method **Refresh**. The Lifetime is the number of seconds to expire. If the value is zero the Allocation will be deleted.

```
oTURN.Refresh(600);
```

TURN Client | Create Permissions

When a new Allocation is created in a TURN server, this allocation cannot process any incoming packet from other peers if has no permissions. So, in order to allow other peers to communicate using a Relayed IP Address, first the TURN Client must create permissions for the IP Addresses that are allowed to exchange Data.

To Create a new Permission, just call the method **CreatePermission** and pass as a parameter the IP Address of the peer. If the Peer IP already exists on the TURN server, it will be refreshed, if not, it will be created. Permissions expire after 5 minutes unless are refreshed.

The TURN client, only allows you to call the method CreatePermission if exists an active allocation.

If the permission is created successfully, the event **OnTURNCreatePermission** is called.

```
oTURN.CreatePermission("80.147.23.157");


void OnTURNCreatePermission(Component Sender, TsgcSocketConnection aSocket,
  TsgcSTUN_Message aMessage, TsgcTURN_ResponseCreatePermission aCreatePermission)
{
  DoLog("#Create Permission: " + aCreatePermission.IPAddresses);
}
```

TURN Client | Send Indication

TURN Protocol supports 2 mechanisms for sending and receiving data from peers, one of them is Send and Data mechanisms.

The TURN client can use the **SendIndication** method to send data to the server for relaying to a peer. The TURN client must ensure that there is a permission for the Peer IP Address where the Send Indication will be sent.

The responses to a SendIndication method, are received **OnTURNDataIndication** event.

```
oTURN.SendIndication("80.147.23.157", 5000, "random data");


void OnTURNDataIndication(Component Sender, TsgcSocketConnection aSocket,
  TsgcSTUN_Message aMessage, TsgcTURN_ResponseDataIndication aDataIndication)
{
  DoLog("#Data Indication: [" + aDataIndication.PeerIP + ":" + IntToStr(aDataIndication.PeerPort) + "] " +
    aDataIndication.Data.ToString());
}
```

TURN Client | Channels

Channels provide a way for the TURN Client and Server to send application data using ChannelData messages, which have less overhead than Send and Data Indications.

Before using ChannelData messages to exchange data between peers, the TURN client must create a new channel, to do this, just call the method **ChannelBind** passing the Peer IP Address and Port as parameters.

If the TURN server can bind a new channel, the TURN client will receive a successful response **OnTURNChannelBind** event.

```
oTURN.ChannelBind("80.147.23.157", 5000);


void OnTURNChannelBind(Component Sender, TsgcSocketConnection aSocket,
  TsgcSTUN_Message aMessage, TsgcTURN_ResponseChannelBind aChannelBind)
{
  DoLog("#Channel Bind: " + aChannelBind.Channel.ToString());
}
```

A channel binding lasts for 10 minutes unless refreshed. To refresh a channel just call **ChannelBind** method again.

When the TURN client receives a new ChannelMessage, the event **OnTURNChannelData** is called.

```
void OnTURNChannelData(Component Sender, TsgcSocketConnection aSocket,
  TsgcTURNChannelData aChannelData)
{
  DoLog("#Channel Data: [" + aChannelData.ChannelID.ToString() + "] " +
    aChannelData.Data.ToString());
}
```

TURN | TsgcTURNServer

TsgcTURNServer is the server that implements the TURN protocol and allows you to process requests from TURN clients. The component inherits from TsgcSTUNServer, so all methods and properties are available on TsgcTURNServer.

TURN Server supports Long-Term Authentication, Allocation, Permissions, Channel Data and more.

Basic usage

Usually TURN servers run on UDP port 3478 and require Long-Term credentials, so in order to configure a TURN server, set the listening port (by default 3478) and start the server.

Configure the server

- Port: the listening Server port, example: 3478
- Define the Long-Term Credentials properties in TURNOptions.Authentication.LongTermCredentials
- Handle the OnSTUNRequestAuthorization to set the password when a TURN client sends a request to TURN Server.

Set the property **Active = True** to start the TURN server.

```
TsgcTURNServer oTURN = new TsgcTURNServer();
oTURN.Port = 3478;
oTURN.TURNOptions.Authentication.Enabled = true;
oTURN.TURNOptions.Authentication.LongTermCredentials.Enabled = true;
oTURN.TURNOptions.Authentication.LongTermCredentials.Realm = "esegece.com";
oTURN.Active = true;
void OnSTUNRequestAuthorization(TObject Sender, const TsgcSTUN_Message aRequest,
  const string aUsername, const string aRealm, ref string Password)
{
  if ((aUsername == "user") && (aRealm == "esegece.com"))
  {
    Password = "password";
  }
}
```

Most common uses

- **Configurations**
  - Long-Term Credentials
- **Allocations**
  - Allocations

Properties

Active: set the property to True to **Start** the TURN server and set to False to **Stop** the Server.

Host: it's the IP Address or DNS name of TURN server.

Port: it's the listening port of TURN server, by default 3478.

IPVersion: it's the Family Address, by default IPv4.

STUNOptions: here are defined the specific options for STUN Requests

Fingerprint: if enabled, the message includes a fingerprint that aids to identify STUN messages from packets of other protocols when the two are multiplexed on the same transport address.

Software: if enabled, sends an attribute with the name of the software being used by the server.

Authentication: here you can configure if the server requires Authentication requests to send binding responses.

- **Enabled:** set to True if the server requires Authentication requests, by default false.
- **LongTermCredentials:** Enable if the server supports Long-Term credentials. The long-term credential mechanism relies on a long-term credential, in the form of a username and password that are shared between client and server.
  - **Enabled:** set to True if the server requires Long-Term credentials.
  - **Realm:** the string of the realm sent to client.
  - **StaleNonce:** time in seconds after the nonce is no longer valid.

BindingAttributes: when the server sends a successful response after a binding request, here you can customize which attributes will be sent to the client.

- **OtherAddress:** if enabled and the server binds to more than one address, this attribute will be sent with all other addresses except the default one.
- **ResponseOrigin:** is the Local IP of the server to send the response.
- **SourceAddress:** is the Local IP of the server to send the response.

TURNOptions: here are defined the specific options for TURN Requests

Fingerprint: if enabled, the message includes a fingerprint that aids to identify TURN messages from packets of other protocols when the two are multiplexed on the same transport address.

Software: if enabled, sends an attribute with the name of the software being used by the server.

Allocation: when a new allocation is created, the server takes from this property the default values.

- **DefaultLifeTime:** value in seconds of default LifeTime.
- **MaxLifeTime:** max value of LifeTime, if a TURN client requests a value greater than this value, the value returned will be the MaxLifeTime.
- **MaxUserAllocations:** max number of allocations.
- **MinPort:** Minimum range port of allocations.
- **MaxPort:** Maximum range port of allocations.
- **RelayIP:** if defined, this will be the Relayed IP Address.

Authentication: usually TURN servers require Long-Term Credentials authentication.

- **Enabled:** set to True if the server requires Authentication requests, by default false.
- **LongTermCredentials:** Enable if the server supports Long-Term credentials. The long-term credential mechanism relies on a long-term credential, in the form of a username and password that are shared between client and server.
  - **Enabled:** set to True if the server requires Long-Term credentials.
  - **Realm:** the string of the realm sent to client.
  - **StaleNonce:** time in seconds after the nonce is no longer valid.

LogFile: if enabled save stun messages to a specified log file, useful for debugging.

Enabled: if enabled every time a message is received and sent by server it will be saved on a file.

FileName: full path to the filename.

NotifyEvents: defines which mode to notify the events.

neAsynchronous: this is the default mode, notify threaded events on asynchronous mode, adds events to a queue that are synchronized with the main thread asynchronously.

neSynchronous: if this mode is selected, notify threaded events on synchronous mode, needs to synchronize with the main thread to notify these events.

neNoSync: there is no synchronization with the main thread, if you need to access to controls that are not thread-safe, you need to implement your own synchronization methods.

Events

The TURN server inherits from STUN Server the events: OnSTUNRequestAuthorization, OnSTUNRequestSuccess, OnSTUNRequestError and OnSTUNException.

Additionally, includes the following events to handle all TURN messages.

OnTURNBeforeAllocate

The event is called before creating a new Allocation. It provides the IP Address and Port used to Relay Data, you can reject if don't want to accept the Allocation.

OnTURNCreateAllocation

The event is called after creating successfully an Allocation.

OnTURNDeleteAllocation

The event is called after removing an already created Allocation.

OnTURNMessageDiscarded

The event is called when a message received by server is discarded.

OnTURNChannelDataDiscarded

The event is called when a Channel Data message is discarded.

TURN Server | Long Term Credentials

Usually TURN Servers are configured WITH Authentication for TURN requests and without Authentication for STUN requests.

sgcWebSockets TURN Server supports Long-Term Credentials, so you can configure TsgcTURNServer to only allow requests with Long-Term credentials info.

To configure it, access the TURNOptions.Authorization property and enable it.
Then access to LongTermCredentials property and enabled it. By default, this type of authorization is already configured with a Realm string and with a default StaleNonce value of 10 minutes (= 600 seconds).

```
TsgcTURNServer oTURN = new TsgcTURNServer();
oTURN.STUNOptions.Authentication.Enabled = false;
oTURN.TURNOptions.Authentication.Enabled = true;
oTURN.TURNOptions.Authentication.LongTermCredentials.Enabled = true;
oTURN.TURNOptions.Authentication.LongTermCredentials.Realm = "sgcWebSockets";
oTURN.TURNOptions.Authentication.LongTermCredentials.StaleNonce = 600;
oTURN.Port = 3478;
oTURN.Active = true;

private void OnSTUNRequestAuthorization(TObject Sender, const TsgcSTUN_Message aRequest,
  const string aUsername, const string aRealm, ref string Password)
{
  if ((aUsername == "my-user") && (aRealm == "sgcWebSockets"))
  {
    Password = "my-password";
  }
}
```

TURN Server | Allocations

All TURN operations revolve around allocations and all TURN messages are associated with an Allocation. An allocation consists of:

- The relayed transport address
- The 5-Tuple: client's IP Address, client's IP port, server IP address, server port and transport protocol.
- The authentication information.
- The time-to-expiry for each relayed transport address.
- A list of permissions for each relayed transport address.
- A list of channels bindings for each relayed transport address.

When a TURN client sends an Allocate request, this TURN message is processed by server and tries to create a new Relayed Transport Address. By default, if there is any available UDP port, it will create a new Relayed Address, but you can use **OnTURNBeforeAllocate** event to reject a new Allocation request.

```
void OnTURNBeforeAllocate(TObject Sender, const TsgcSocketConnection aSocket,
  const string aIP, Word aPort, ref bool Reject)
{
  if (your own rules) == false
  {
        Reject = false;
  }
}
```

If the process continues, the server creates a new allocation and the event **OnTURNCreateAllocation** is called. This event provides information about the Allocation through the class TsgcTURNAllocationItem.

```
void OnTURNCreateAllocation(TObject Sender, const TsgcSocketConnection aSocket,
  const TsgcTURNAllocationItem Allocation)
{
  DoLog("New Allocation: " + Allocation.RelayIP + ":" + IntToStr(Allocation.RelayPort));
}
```

When the Allocation expires or is deleted receiving a Refresh Request from client with a lifetime of zero, the event **OnTURNDeleteAllocation** event is fired.

```
void OnTURNDeleteAllocation(TObject Sender, const TsgcSocketConnection aSocket,
  const TsgcTURNAllocationItem Allocation)
{
  DoLog("Allocation Deleted: " + Allocation.RelayIP + ":" + IntToStr(Allocation.RelayPort));
}
```

Demos | Server Chat

This demo shows how to build a Server Chat using TsgcWebSocketHTTPServer and WebSockets as communication protocol.
Every time a new peer sends a message, the server reads the message and broadcast the message to all connected clients.

Start Server

Before start the server, you must configure it to set the listening port, if use a secure connection or not...

- First I create a new instance of TsgcWebSocketHTTPServer.
- If Server will use secure connections, it needs a PEM certificate, just set where is located this certificate and the listening port for SSL You can configure the TLS version and the OpenSSL API (if needed)

```
// ... ssl
switch (cboOpenSSLAPI.SelectedIndex)
{
        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.SelectedIndex)
{
        case 0:
                server.SSLOptions.Version = TwsTLSVersions.tlsUndefined;
                break;
        case 1:
                server.SSLOptions.Version = TwsTLSVersions.tls1_0;
                break;
        case 2:
                server.SSLOptions.Version = TwsTLSVersions.tls1_1;
                break;
        case 3:
                server.SSLOptions.Version = TwsTLSVersions.tls1_2;
                break;
        case 4:
                server.SSLOptions.Version = TwsTLSVersions.tls1_3;
                break;
        default:
                break;
}
```

- By default, if you start the server, it will listening on ALL IPs of listening port, so it's recommended use the binding property to only listen on 1 specific IP.

```
server.Bindings = txtHost.Text + ":" + txtDefaultPort.Text;
```

- Once configured all options, call Server.Active = true to start the server.

Events Configuration

- Use **OnConnect** and **OnDisconnect** events to know when a client connects to server.
- **Messages** sent from **client to server** are received **OnMessage** event, so use this event handler to broadcast the message received to all clients

```
private void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
  server.Broadcast(Text);
}
```

Dispatch HTTP Requests

WebSocket HTTP Server allows you to handle WebSocket and HTTP Protocols on the same listening port, so a web-browser can request a web page to access your server. **OnCommandGet** is the event used to read the HTTP Request and send the HTTP Responses.

Use ARequestInfo parameter to read the HTTP Request and AResponseInfo to write the HTTP Response.

Basically, use the ARequestInfo.Document to read which document is requesting the client and send a response using the following properties: ResponseNo, ContentType and ContentText.

Example: a client request document '/jquery.js'

```
private void OnCommandGetEvent(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo RequestInfo, ref TsgcWSHTTPResp
{
        if (RequestInfo.Document == "/jquery.js")
        {
                ResponseInfo.ContentType = "text/javascript";
                ResponseInfo.ContentText = File.ReadAllText(Application.StartupPath + "\\html\\jquery.js");
                ResponseInfo.ResponseNo = 200;
        }
}
```

Client Chat

This demo shows how to build a client chat, using TsgcWebSocketClient, which connects to a WebSocket Server, sends a message and this message is received by all connected clients.

Connect to Server

- First create a new instance of TsgcWebSocketClient.
- Then configure the server **Host** and **Port**.
- If client uses a secure connection, configure the **TLSOptions** property of the component.

```
if (chkTLS.Checked)
{
        client.Port = Int32.Parse(txtSSLPort.Text);
}
else
{
        client.Port = Int32.Parse(txtDefaultPort.Text);
}
client.Host = txtHost.Text;
switch (cboTLSIOHandler.SelectedIndex)
{
        case 0:
                client.TLSOptions.IOHandler = TwsTLSIOHandler.iohOpenSSL;
                break;
        case 1:
                client.TLSOptions.IOHandler = TwsTLSIOHandler.iohSChannel;
                break;
        default:
                break;
}
switch (cboOpenSSLAPI.SelectedIndex)
{
        case 0:
                client.TLSOptions.OpenSSL_Options.APIVersion = TwsOpenSSLAPI.oslAPI_1_0;
                break;
        case 1:
                client.TLSOptions.OpenSSL_Options.APIVersion = TwsOpenSSLAPI.oslAPI_1_1;
                break;
}
switch (cboTLSVersion.SelectedIndex)
{
        case 0:
                client.TLSOptions.Version = TwsTLSVersions.tlsUndefined;
                break;
        case 1:
                client.TLSOptions.Version = TwsTLSVersions.tls1_0;
                break;
        case 2:
                client.TLSOptions.Version = TwsTLSVersions.tls1_1;
                break;
        case 3:
                client.TLSOptions.Version = TwsTLSVersions.tls1_2;
                break;
        case 4:
                client.TLSOptions.Version = TwsTLSVersions.tls1_3;
                break;
        default:
                break;
}
client.TLS = chkTLS.Checked;
```

- Once all options can be configured, set Client.Active = true to connect to server.

Send Message To Server

- To send a message to server, use **WriteData** method, send any Text message and server will send as a response the same message.

```
client.WriteData(txtName.Text + ": " + txtMessage.Text);
```

Receive Messages from Server

- Every time a new Text message is received by client, **OnMessage** event is fired.

```
private void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
        DoLog(Text);
}
```

Demos | Client

This demo shows how to build a websocket client, using TsgcWebSocketClient.

Connect to Server

- First create a new instance of TsgcWebSocketClient.
- Then configure the server **Host** and **Port**.
- By default the client will connect using **WebSocket protocol**. But you can configure the client to connect using **plain TCP protocol**. Just set **Specifications.RFC6455 = false**, and the client will use plain TCP protocol instead of WebSocket protocol. You can read more about TCP Connections.

```
client.Host = txtHost.Text;
client.Port = Int32.Parse(txtPort.Text);
client.Options.Parameters = txtParameters.Text;
client.TLS = chkTLS.Checked;
client.TLSOptions.Version = TwsTLSVersions.tls1_2;
client.TLSOptions.IOHandler = TwsTLSIOHandler.iohSChannel;
client.Specifications.RFC6455 = chkOverWebSocket.Checked;
client.Proxy.Enabled = chkProxy.Checked;
client.Proxy.Username = txtProxyUsername.Text;
client.Proxy.Password = txtProxyPassword.Text;
client.Proxy.Host = txtProxyHost.Text;
if (txtProxyPort.Text != "")
{
	client.Proxy.Port = Int32.Parse(txtProxyPort.Text);
}
client.Extensions.PerMessage_Deflate.Enabled = chkCompressed.Checked;
client.Active = true;
```

Client Events

Use the following events to control the client flow: when connects, disconnects, receives a message, an error is detected...

```
private void OnExceptionEvent(TsgcWSConnection Connection, Exception E)
{
	DoLog("#exception: " + E.Message);
}
private void OnConnectEvent(TsgcWSConnection Connection)
{
	DoLog("#connected: " + Connection.IP);
}
private void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
	DoLog(Text);
}
private void OnDisconnectEvent(TsgcWSConnection Connection, int CloseCode)
{
	DoLog("Disconnected (" + CloseCode.ToString() + "): " + Connection.IP);
}
private void OnErrorEvent(TsgcWSConnection Connection, string Error)
{
	DoLog("#error: " + Connection.IP + " - " + Error);
}
```

Demos | Client MQTT

This demo shows how connect to a MQTT broker server. Requires a TsgcWebSocketClient to handle WebSocket / TCP protocols.

Configuration

- First create a new TsgcWebSocketClient instance, check the Client Demo.
- Then, create a new instance of TsgcWSPClient_MQTT.
- After that, you must assign the MQTT Protocol to WebSocket client and configure the connection options in WebSocket client.

```
if (mqtt == null)
{
        mqtt = new TsgcWSPClient_MQTT();
        mqtt.OnMQTTBeforeConnect += OnMQTTBeforeConnectEvent;
        mqtt.OnMQTTConnect += OnMQTTConnectEvent;
        mqtt.OnMQTTDisconnect += OnMQTTDisconnectEvent;
        mqtt.OnMQTTSubscribe += OnMQTTSubscribeEvent;
        mqtt.OnMQTTUnSubscribe += OnMQTTUnSubscribeEvent;
        mqtt.OnMQTTPing += OnMQTTPingEvent;
        mqtt.OnMQTTPubAck += OnMQTTPubAckEvent;
        mqtt.OnMQTTPubComp += OnMQTTPubCompEvent;
        mqtt.OnMQTTPublish += OnMQTTPublishEvent;
        mqtt.OnMQTTPubRec += OnMQTTPubRecEvent;
        mqtt.Client = client;
}
mqtt.Client = client;
txtParameters.Text = "/";
chkTLS.Checked = false;
mqtt.Authentication.Enabled = false;
mqtt.Authentication.UserName = "";
mqtt.Authentication.Password = "";
mqtt.MQTTVersion = TwsMQTTVersion.mqtt311;
mqtt.HeartBeat.Interval = 5;
mqtt.HeartBeat.Enabled = true;
switch (Index)
{
        case 0: // esegece.com
                txtHost.Text = "www.esegece.com";
                txtPort.Text = "15675";
                txtParameters.Text = "/ws";
                mqtt.Authentication.Enabled = true;
                mqtt.Authentication.UserName = "sgc";
                mqtt.Authentication.Password = "sgc";
                chkOverWebSocket.Checked = true;
                break;
        case 1: // test.mosquitto.org
                txtHost.Text = "test.mosquitto.org";
                txtPort.Text = "1883";
                chkTLS.Checked = false;
                chkOverWebSocket.Checked = false;
                break;
        case 2: // mqtt.fluux.io
                txtHost.Text = "mqtt.fluux.io";
                txtPort.Text = "1883";
                chkTLS.Checked = false;
                chkOverWebSocket.Checked = false;
                mqtt.MQTTVersion = TwsMQTTVersion.mqtt5;
                break;
        case 3: // broker.hivemq.com
                txtHost.Text = "broker.mqttdashboard.com";
                txtPort.Text = "8000";
                txtParameters.Text = "/mqtt";
                chkTLS.Checked = false;
                chkOverWebSocket.Checked = true;
                mqtt.MQTTVersion = TwsMQTTVersion.mqtt5;
                break;
}
```

MQTT Events

The connection flow is controlled by MQTT Client component, so you must handle the MQTT events to know when it's connected to broker, when a new message is published, when is disconnected...

```
private void OnMQTTConnectEvent(TsgcWSConnection Connection, bool Session, int ReasonCode, string ReasonName, Tsg
{
        DoLog("#MQTT Connect");
        chkTLS.Enabled = false;
        chkCompressed.Enabled = false;
        if (FMQTTSubscribeTopic != "")
        {
                mqtt.Subscribe(FMQTTSubscribeTopic);
                FMQTTSubscribeTopic = "";
        }
}



private void OnMQTTPublishEvent(TsgcWSConnection Connection, string Topic, string Text, TsgcWSMQTTPublishProperti
{
        DoLog(Topic + ": " + Text);
}



private void OnMQTTSubscribeEvent(TsgcWSConnection Connection, int PacketIdentifier, TsgcWSSUBACKS Codes, TsgcWSM
{
        DoLog("#Subscribe: " + PacketIdentifier.ToString());
}



private void OnMQTTDisconnectEvent(TsgcWSConnection Connection, int ReasonCode, string ReasonName, TsgcWSMQTTDISC
{
        DoLog("#disconnected");
        chkTLS.Enabled = true;
        chkCompressed.Enabled = true;
}
```

Demos | Client SocketIO

This demo shows how connect to a Socket.IO Server. Requires a TsgcWebSocketClient to handle WebSocket / TCP protocols.

Configuration

- First create a new TsgcWebSocketClient instance, check the Client Demo.
- Then, create a new instance of TsgcWSAPI_SocketIO.
- After that, you must assign the Socket.IO API to WebSocket client and configure the connection options in WebSocket client.

```
if (socketio == null)
{
        socketio = new TsgcWSAPI_SocketIO();
        socketio.Client = client;
}
socketio.Client = client;
txtParameters.Text = "/";
chkTLS.Checked = true;
chkOverWebSocket.Checked = true;
txtHost.Text = "socketio-chat-h9jt.herokuapp.com";
txtPort.Text = "443";
txtParameters.Text = "/";
```

Send Messages

Socket.IO uses **TsgcWebSocketClient** to send messages to server, so just call **WriteData** and pass as a parameter the JSON message to socket.io server

```
client.WriteData("42[\"new message\", \"" +  txtSocketIOMessage.Text + "\"]");
```

Receive Messages

The messages received as the flow of connection is handled by TsgcWebSocketClient, so use this component to read the messages sent from server and to know if connection is active or not.

```
private void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
  DoLog(Text);
}
```

Demos | Server Monitor

This demo shows how to update 3 HTML Monitors using WebSocket as protocol. Server has an internal timer that updates randomly the values of the gauges and updates the value using a websocket message. This message is read by javascript client and updates the value of the Gauge.

Configuration

- First create a new TsgcWebSocketServer instance, check the Server Chat Demo.
- Then Create a new Timer and every 500 milliseconds update the values of: memory, network or cpu. Send the update to all clients connected.
- In Javascript client, read the message sent by server and update the value of the gauge.

```
<pre><code>
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Server Monitor Demo</title>
        <script src="http://127.0.0.1:5413/sgcWebSockets.js"></script>
        <script src="http://127.0.0.1:5413/esegece.com.js"></script>
        <link rel="stylesheet" href="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.css" />
        <script src="http://code.jquery.com/jquery-1.6.4.min.js"></script>
        <script src="http://code.jquery.com/mobile/1.1.0/jquery.mobile-1.1.0.min.js"></script>
        <script type='text/javascript' src='https://www.google.com/jsapi'></script>
        <style>
        #status {
                padding: 5px;
                color: #fff;
                background: #ccc;
        }
        #status.fail {
          background: #c00;
        }
        #status.offline {
          background: #c00;
        }
        #status.online {
          background: #0c0;
        }
        </style>
        <script type='text/javascript'>
          var vMemory;
          var vCpu;
          var vNetwork;
          var chart;
          var data;
          var options;
          var ws;

          vMemory=30;
          vCpu=55;
          vNetwork=68;
      google.load('visualization', '1', {packages:['gauge']});
          google.setOnLoadCallback(drawChart);
      function drawChart() {
        data = google.visualization.arrayToDataTable([
          ['Label', 'Value'],
          ['Memory', vMemory],
          ['CPU', vCpu],
          ['Network', vNetwork]
        ]);
        options = {
          width: 400, height: 120,
          redFrom: 90, redTo: 100,
          yellowFrom:75, yellowTo: 90,
          minorTicks: 5
        };
        chart = new google.visualization.Gauge(document.getElementById('chart_div'));
        chart.draw(data, options);
        }

        function updateChart() {
```

```
                data = google.visualization.arrayToDataTable([
                        ['Label', 'Value'],
                        ['Memory', vMemory],
                        ['CPU', vCpu],
                        ['Network', vNetwork]
                ]);
                chart.draw(data, options);
        }

        function subscribe(Channel)
        {
          if (document.getElementById(Channel).checked) {
                ws.subscribe(Channel);
          } else {
                ws.unsubscribe(Channel);
          }
        }

        function wsmonitor()
        {
          if ("WebSocket" in window)
          {
                  ws = new sgcws("ws://127.0.0.1:5413");
                  ws.on('open', function(evt){
                                document.getElementById('status').innerHTML = "Socket Open";
                                document.getElementById('status').className = "online";
                                ws.subscribe("memory");
                                ws.subscribe("cpu");
                                ws.subscribe("network");
                        }
                  );
                  ws.on('close', function(evt){
                                document.getElementById('status').innerHTML = "Socket Closed";
                                document.getElementById('status').className = "offline";
                        }
                  );
                  ws.on('sgcevent', function(evt){
                        if (evt.channel == "memory") {
                          vMemory = parseInt(evt.message);
                        } else if (evt.channel == "cpu") {
                          vCpu = parseInt(evt.message);
                        } else if (evt.channel == "network") {
                          vNetwork = parseInt(evt.message);
                        }
                        updateChart();
                        }
                  );
                  ws.on('error', function(evt){
                                document.getElementById('status').innerHTML = "Socket Error";
                                document.getElementById('status').className = "fail";
                        }
                  );
          }
        }
</script>
</head>
<body>
<div data-role="page" id="wsdemo_monitor">
        <div data-role="header" data-theme="b">
                <h1>Server Monitor</h1>
                <a href="#home" data-icon="home" data-iconpos="notext" data-direction="reverse" class="ui-btn-lef
        </div><!-- /header -->
        <div data-role="content">
                <h2>Press Start to Get Monitor Data</h2>
                <p id="status" classname="success"></p>
                <h4>Select which data you want to receive: Memory - CPU - Network</h4>
                <a href="javascript:wsmonitor()" data-role="button" data-inline="true">Start</a>
                <div id='chart_div'></div>
                <div data-role="fieldcontain">
                    <fieldset data-role="controlgroup" data-type="horizontal">
                            <input type="checkbox" name="memory" id="memory" class="custom" checked="True" onclic
                                <label for="memory">Memory</label>
                                <input type="checkbox" name="cpu" id="cpu" class="custom" checked="True" onclick=
                                <label for="cpu">CPU</label>
                                <input type="checkbox" name="network" id="network" class="custom" checked="True"
                                <label for="network">Network</label>
                    </fieldset>
                </div>
        </div><!-- /content -->
        <div data-role="footer" class="footer-docs" data-theme="c">
                        <p>&copy; 2020 eSeGeCe.com</p>
        </div>
</div><!-- /page -->
</body>
</html>
</code></pre>
```

Demos | Server Snapshots

This demo shows how to send images from server to client and how all clients receive the same image using broadcast method of server component.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Enable compression to send less bytes when message is transmitted to clients

  Server.Extensions.PerMessage_Deflate.Enabled = true

- Then every 5 seconds the server broadcast an image stream to all connected clients

```
Random r = new Random();
int rInt = r.Next(1, 12);
Bitmap bmp = new Bitmap(Application.StartupPath + "\\img\\" + rInt.ToString() + ".bmp");
MemoryStream memoryStream = new MemoryStream();
bmp.Save(memoryStream, System.Drawing.Imaging.ImageFormat.Bmp);
server.Broadcast(memoryStream.ToArray());
```

Demos | Client Snapshots

This demo shows how read binary websocket messages, using TsgcWebSocketClient, which connects to a Web-Socket Server, and receives a stream which is an image that is shown to user.

Connect to Server

- First create a new instance of TsgcWebSocketClient.
- Then configure the server **Host** and **Port**.
- Enable compression to receive less bytes when message is transmitted from server.

```
Client.Extensions.PerMessage_Deflate.Enabled = true
```

- The image sent by server arrives as a stream, so use **OnBinary** event to read images.

```
private void OnBinaryEvent(TsgcWSConnection Connection, byte[] Bytes)
{
        MemoryStream stream = new MemoryStream(Bytes);
        pictureBox1.Image = new Bitmap(stream);
}
```

Demos | Upload File

This demo shows how to upload a file from web browser to a server using websocket protocol.

Configuration

- First create a new TsgcWebSocketServer instance, check the Server Chat Demo.
- The file will arrive to server as a binary stream, so you must handle **OnBinary** event to read the file.

```
private void OnBinaryEvent(TsgcWSConnection Connection, byte[] Bytes)
{
  File.WriteAllBytes(filename, Bytes);
  DoLog("File Received: " + filename);
}
```

- If you want to know the name of the file, you can send a text message before the file is sent with the name of the file

```
private void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
        if (Text.StartsWith("uploadfile:") == true)
        {
                string[] filePath = Text.Split(':');
                filename = filePath[1];
        }
        else
        {
                DoLog("Message Received (" + Connection.Guid + "): " + Text);
        }
}
```

The **javascript** code to send a file using **websockets** is shown below:

```
<script type='text/javascript'>
var ws;
function DoOpen()
{
  if ("WebSocket" in window)
  {
          ws = new sgcWebSocket("ws://127.0.0.1:5418");
          ws.on('open', function(evt){
                        ws.binaryType = "arraybuffer";
                        document.getElementById('status').innerHTML = "Socket Open";
                        document.getElementById('status').className = "online";
                }
          );
          ws.on('close', function(evt){
                        document.getElementById('status').innerHTML = "Socket Closed";
                        document.getElementById('status').className = "offline";
                }
          );
          ws.on('error', function(evt){
                        document.getElementById('status').innerHTML = "Socket Error";
                        document.getElementById('status').className = "fail";
                }
          );
  }
}
function DoClose()
{
        ws.close();
}
function DoUploadFile() {
        var file = document.getElementById('filename').files[0];
        var reader = new FileReader();
        var rawData = new ArrayBuffer();

        reader.loadend = function() {
```

```
        }
        reader.onload = function(e) {
                ws.send("uploadfile:" + file.name);
                rawData = e.target.result;
                ws.send(rawData);
                        document.getElementById('status').innerHTML = "File Uploaded";
                        document.getElementById('status').className = "online";
                }
                reader.readAsArrayBuffer(file);
        }
</script>
```

Demos | Server Authentication

This demo shows how to use Server Authentication, if you want to know more about the different types of authentication, read the following article about Authentication.

Authentication

- First create a new instance of TsgcWebSocketServer. Enable Authentication property, server.Authentication.Enabled = true;
- Then, check in **OnAuthentication** event handler if the username and password are correct. If they are correct, set the Authenticated property to true, otherwise set to false.

```
private void OnAuthenticationEvent(TsgcWSConnection Connection, string User, string Password, ref bool Authentica
{
        if ((User == "user") && (Password == "1234"))
        {
                Authenticated = true;
        }
}
```

Demos | KendoUI_Grid

This demo shows how the KendoUI Grid works using WebSockets as protocol and a Web Browser as a client. Basically is a javascript grid that is updated when any of the clients makes any change, these changes are updated using websocket protocol to all connected clients, so all clients can see in real-time the same data, including all changes made by clients.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Then you must handle OnCommandGet to send the required files requested by web browser clients.

```
if (RequestInfo.Document == "/")
{
        string readText = File.ReadAllText(Application.StartupPath + "\\files\\index.html");
        readText = readText.Replace("<#port>", txtDefaultPort.Text);
        readText = readText.Replace("<#host>", txtHost.Text);
        ResponseInfo.ContentText = readText;
        ResponseInfo.ResponseNo = 200;
}
else
{
        ResponseInfo.ResponseNo = 404;
}
```

WebSockets Updates

When a client updates a grid record, this change is transmitted to all connected clients using websocket protocol. Use OnMessage event to get notified about grid changes. The messages are in JSON format so you only must read the JSON text, decode it and send a response to the other peer.

```
private void OnMessageEvent(TsgcWSConnection Connection, string Text)
{
        DoLog(Text);
        var oJSON = JObject.Parse(Text);
        if (oJSON["type"].ToString() == "read")
        {
                JArray oArray = new JArray();
                for (int i = 0; i < 20; i++)
                {
                        JObject oObject = new JObject();
                        oObject.Add("ContactID", i.ToString());
                        oObject.Add("ContactName", ContactName[i]);
                        oObject.Add("ContactTitle", ContactTitle[i]);
                        oObject.Add("CompanyName", CompanyName[i]);
                        oObject.Add("Country", Country[i]);
                        oArray.Add(oObject);
                }
                oJSON.Add("data", oArray);
                Connection.WriteData(oJSON.ToString());
        }
        else if (oJSON["type"].ToString() == "update")
        {
                Text = Text.Replace("\"type\":\"update\"", "\"type\":\"push-update\"");
                server.Broadcast(Text, "", "", Connection.Guid);
        }
        else if (oJSON["type"].ToString() == "destroy")
        {
                Text = Text.Replace("\"type\":\"destroy\"", "\"type\":\"push-destroy\"");
                server.Broadcast(Text, "", "", Connection.Guid);
        }
        else if (oJSON["type"].ToString() == "create")
        {
                Text = Text.Replace("null", DateTime.Now.ToString("yyyyMMddHHmmss"));
                string vText = Text.Replace("\"type\":\"create\"", "\"type\":\"push-create\"");
                server.Broadcast(vText, "", "", Connection.Guid);
                Connection.WriteData(Text);
```

```
        }
}
```

Demos | ServerSentEvents

This demo shows how Server-Sent Events work in WebSocket Server. sgcWebSockets allows that the server can handle more than one protocol on the same listening port.

You can read more about Server Sent Events.

This demo shows how the Server will send every second the time to all connected clients using Server Sent Events.

Once the server is started, **broadcasts** to all connected clients a message with the **Server Time**, so every time the client receives this message, it shows to user.

```
private void timer1_Tick(object sender, EventArgs e)
{
  server.Broadcast("data: " + "Server Time: " + DateTime.Now.ToString("HH:mm:ss"));
}
```

The **javascript code** to handle the websocket connection is shown below:

```
socket = new sgcWebSocket('sse', '', 'sse');
socket.on('open', function(evt){
  document.getElementById('status').innerHTML = "Socket Open";
  document.getElementById('status').className = "online";
  }
);
socket.on('close', function(evt){
  document.getElementById('status').innerHTML = "Socket Closed";
  document.getElementById('status').className = "offline";
  }
);
socket.on('message', function(evt){
  document.getElementById('log').innerHTML = evt.message;
  }
);
socket.on('error', function(evt){
  document.getElementById('status').innerHTML = "Socket Error";
  document.getElementById('status').className = "fail";
  }
);
```

Demos | Server WebRTC

This demo shows how to build a Video Conference Server using TsgcWebSocketHTTPServer and WebRTC as javascript library.

The demo uses WebSocket protocol to signal WebRTC and uses public STUN/TURN servers, for production sites, you need to use your own STUN/TURN servers. Registered users can download Coturn for windows, which is already compiled and with all the required libraries to run in your servers.

The client must be a web browser with support for WebRTC connections.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Then, create a new instance of TsgcWSPServer_WebRTC.
- After that, you must assign the WebRTC Protocol to WebSocket Server and configure the server host and port.

```
server.DocumentRoot = Application.StartupPath + "\\html";
server.Port = Int32.Parse(txtDefaultPort.Text);
server.Active = true;
```

The demo requires an index HTML page which is used to dispatch the WebRTC front page, this page is provided with the demo.

Run in WebBrowser

Once configured the server, start it and select one of the web-browsers available. It will open a new Web-Browser session asking to start a new session. If successful you will see your video and if you open the same url in another web-browser, you will see both peers connected.

The demo runs by default without SSL, this is only valid for localhost connections. For production sites, use SSL connections. Check Server Chat Demo to configure SSL in server side.

Demos | Server AppRTC

This demo shows how to build a Video Conference Server using TsgcWebSocketHTTPServer and AppRTC as javascript library.

The demo uses WebSocket protocol to signal WebRTC and uses public STUN/TURN servers, for production sites, you need to use your own STUN/TURN servers. Registered users can download Coturn for windows, which is already compiled and with all the required libraries to run in your servers.

The client must be a web browser with support for WebRTC connections.

Configuration

- First create a new TsgcWebSocketHTTPServer instance, check the Server Chat Demo.
- Then, create a new instance of TsgcWSPServer_AppRTC.
- After that, you must assign the AppRTC Protocol to WebSocket Server and configure the server host and port. WebRTC requires secure connections, so you will need to use a PEM certificate and configure the SSLOptions property of the component.

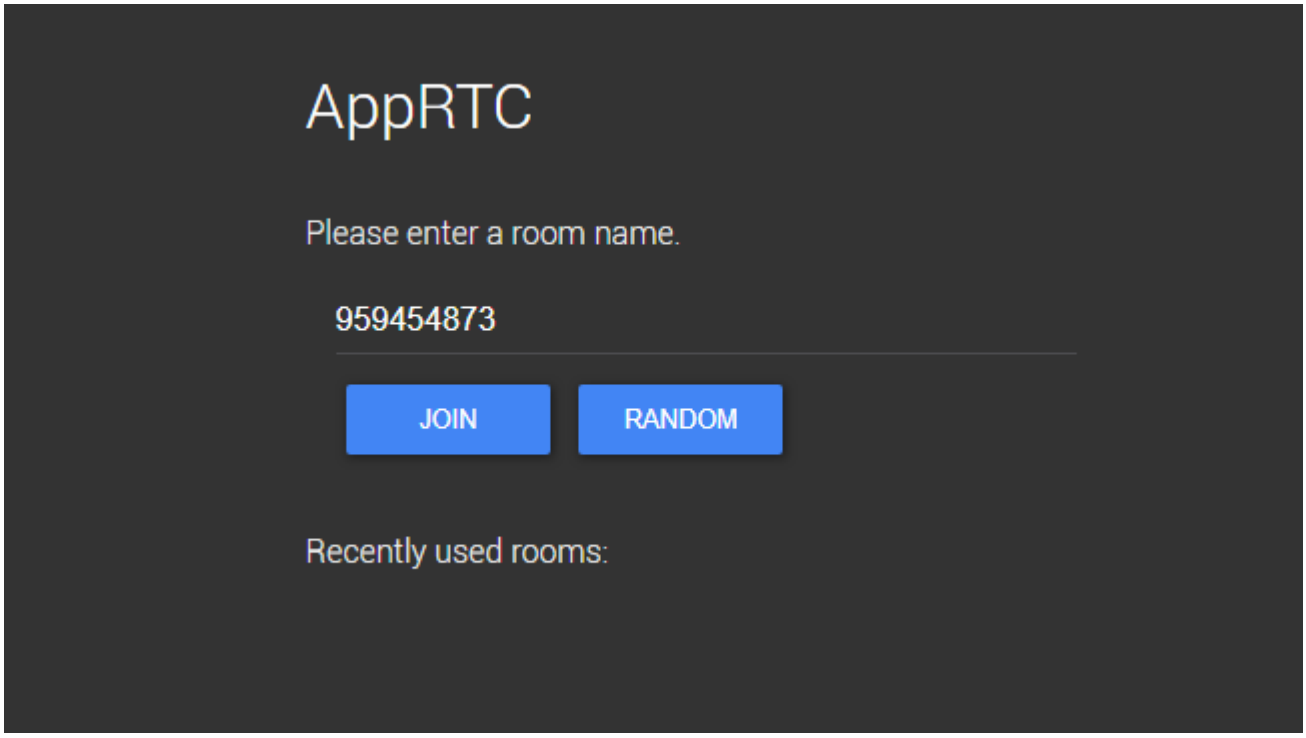
```
server.DocumentRoot = Application.StartupPath + "\\html";
server.Port = Int32.Parse(txtDefaultPort.Text);
AppRTC.AppRTC.RoomLink = "https://" + txtHost.Text + ":" + txtDefaultPort.Text + "/r/";
AppRTC.AppRTC.WebSocketURL = "wss://" + txtHost.Text + ":" + txtDefaultPort.Text;
server.Active = true;
```

- AppRTC.RoomLink is the url where the web-browser will be redirected to login to a room
- AppRTC.WebSocketURL is the url of the websocket connection
- The IceServers can be configured in the AppRTC Server protocol.

The demo requires an index HTML page which is used to dispatch the AppRTC front page, this page is provided with the demo.

Run in WebBrowser

Once configured the server, start it and select one of the web-browsers available. It will open a new Web-Browser session asking to join a new room. Join this room and if successful you will see a link which must be used from another web-browser to start a new video-conference.


AppRTC
Please enter a room name.
959454873
JOIN
RANDOM
Recently used rooms:

Demos | Telegram Client

This demo shows how to connect to Telegram, receive all contacts, send Text messages, send Images... and much more

Configuration

- First create a new instance of TsgcTDLib_Telegram.
- Then, before you attempt to connect to telegram, you must pass some parameters to client component like API Hash, API Id... Once you must set all required parameters, set property Active = true to start a connection.

```
telegram.Telegram.API.ApiHash = txtApiHash.Text;
telegram.Telegram.API.ApiId = txtApiId.Text;
telegram.Telegram.PhoneNumber = "";
telegram.Telegram.BotToken = "";
if (chkLoginBot.Checked)
{
  telegram.Telegram.BotToken = txtBotToken.Text;
}
else
{
  telegram.Telegram.PhoneNumber = txtPhoneNumber.Text;
}


telegram.Active = true;
```

- When client tries to connect to Telegram, usually a code is required, so you must handle **OnTelegramAuthenticationCode** and return the Code parameter with the value provided by your Telegram account.

```
private void OnAuthenticationCodeEvent(TsgcTDLib_Telegram Sender, ref string Code)
{
  Code = InputBox("Telegram", "Introduce Telegram Code");
}
```

Send Telegram Messages

To send a telegram message (text, files, images...) always requires first set the ChaId where you want to send the message and then the parameter that can be a text message, a filename...

```
// send text message
sgcTelegram.SendTextMessage("456413", "Hello From sgcWebSockets!!!");


// send file message
sgcTelegram.SendDocumentMessage("383784", "c:\yourfile.txt");
```

Receive Telegram Messages

Messages received by Telegram client, are handled on specific event Handlers. There is an event when a next Text Message is received, when a new Document is received, photo...

```
private void OnMessageTextEvent(TsgcTDLib_Telegram Sender, TsgcTDLib_Telegram_Client.TsgcTelegramMessageText Mess
{
  DoLogMessage(MessageText.ChatId, MessageText.SenderUserId.ToString(), MessageText.Text);
```

```
}


private void OnMessageDocumentEvent(TsgcTDLib_Telegram Sender, TsgcTDLib_Telegram_Client.TsgcTelegramMessageDocum
{
  DoLogMessage(MessageDocument.ChatId, MessageDocument.SenderUserId.ToString(), MessageDocument.FileName);
}
```

Coturn

Coturn

From sgcWebSockets 4.5.2 ENTERPRISE Edition, you can build your own STUN/TURN server using Delphi/CBuilder.

It's a free open source implementation of TURN and STUN Servers.
The TURN Server is a VoIP media traffic NAT traversal server and gateway. It can be used as a general-purpose network traffic TURN server and gateway, too.

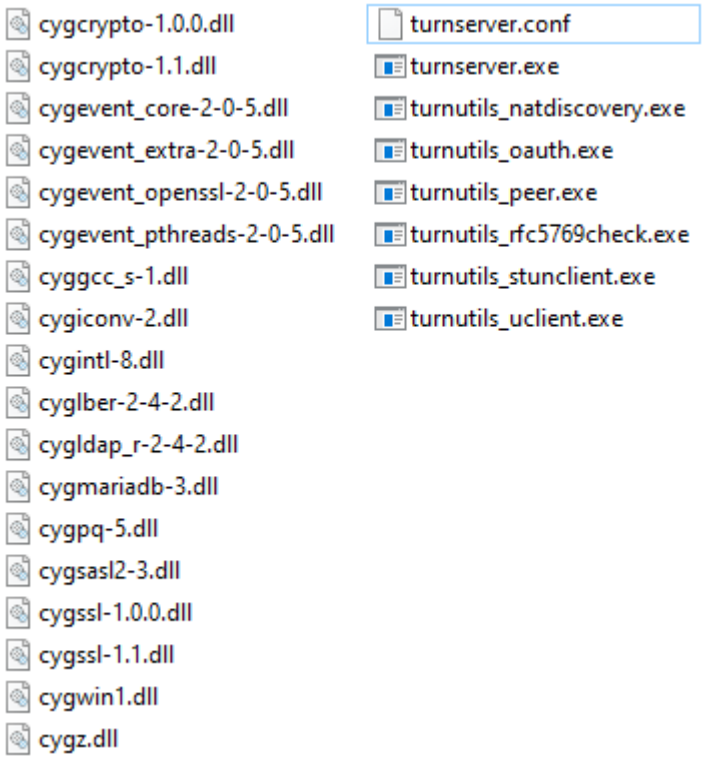
The supported project target platforms are:

- **Linux**
- **Mac OS X**
- **Windows** (Cygwin): compiled binaries are available for registered users.

Windows Configuration

First you must download compiled binaries from your account, there are 2 available versions: win32 and win64. Select the desired platform and uncompress binaries in a folder. The following files will be created:

1. Some cygwin libraries required to run application, you must deploy these libraries with coturn server.
2. Some console applications:
    2.1 turnserver.exe: is the main console application to run a TURN/STUN server
    2.2 Other applications: are used to configure or testing purposes.
3. Turnserver.conf: is the configuration file for coturn server.



turnserver.conf

This is the configuration file for coturn server, if you open you will see a default configuration.

Simple Configuration

Your server has the following public IP 80.15.44.123 and listens on port 80. The credentials for connecting are: username = demo, password = secret
Set the following configuration:

```
listening-ip=80.15.44.123
listening-port=80
realm=yourrealm.com
user=demo:secret
```

Configuration with TLS enabled

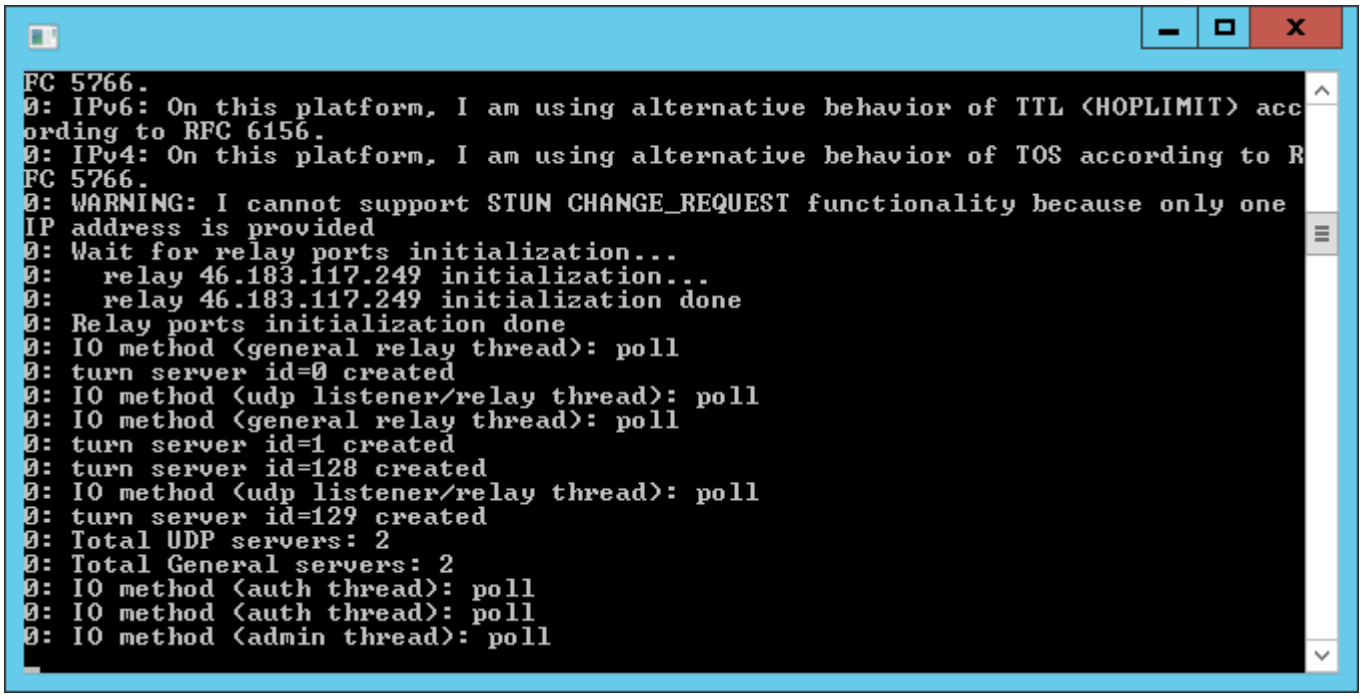
Server has the following public IP 80.15.44.123 and listens on port 80 and 443 (TLS connections). The credentials for connecting are: username = demo, password = secret. Your certificate name (must be in PEM format) is certificate.crt and private key is private.key.
Set the following configuration:

```
listening-ip=80.15.44.123
listening-port=80
realm=yourrealm.com
tls-listening-port=443
cert=certificate.crt
pkey=private.key
user=demo:secret
```

There are more configurations available, just open turnserver.conf and read the documented sections.

Run coturn

Once configured, you can run server just executing turnserver.exe, a new console application will be opened and a log file will be created. You can increase the verbose of console application (get more detailed messages) if you enable "verbose" in turnserver.conf file.



WebSockets

WebSocket is a web technology providing for bi-directional, full-duplex communications channels, over a single Transmission Control Protocol (TCP) socket.

The WebSocket API is being standardized by the W3C, and the WebSocket protocol has been standardized by the IETF as RFC 6455.
WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. The WebSocket protocol makes possible more interaction between a browser and a web site, facilitating live content and the creation of real-time games. This is made possible by providing a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way a two-way (bi-direction) ongoing conversation can take place between a browser and the server. A similar effect has been done in non-standardized ways using stop-gap technologies such as comet.

In addition, the communications are done over the regular TCP port number 80, which is of benefit for those environments which block non-standard Internet connections using a firewall. WebSocket protocol is currently supported in several browsers including Firefox, Google Chrome, Internet Explorer and Safari. WebSocket also requires web applications on the server to be able to support it.

More Information
Browser Support

HTTP/2

HTTP/2 will make our applications faster, simpler, and more robust — a rare combination — by allowing us to undo many of the HTTP/1.1 workarounds previously done within our applications and address these concerns within the transport layer itself. Even better, it also opens up a number of entirely new opportunities to optimize our applications and improve performance!

The primary goals for HTTP/2 are to reduce latency by enabling full request and response multiplexing, minimize protocol overhead via efficient compression of HTTP header fields, and add support for request prioritization and server push. To implement these requirements, there is a large supporting cast of other protocol enhancements, such as new flow control, error handling, and upgrade mechanisms, but these are the most important features that every web developer should understand and leverage in their applications.

HTTP/2 does not modify the application semantics of HTTP in any way. All the core concepts, such as HTTP methods, status codes, URIs, and header fields, remain in place. Instead, HTTP/2 modifies how the data is formatted (framed) and transported between the client and server, both of which manage the entire process, and hides all the complexity from our applications within the new framing layer. As a result, all existing applications can be delivered without modification.

More information

JSON

JSON or **JavaScript Object Notation**, is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

More Information

JSON-RPC 2.0

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

Example: client call method subtract with 2 params (42 and 23). Server sends a result of 19.

Client To Server --> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}

Server To Client<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

Parsers

sgcWebSockets provides a built-in JSON component, but you can use your own JSON parser. Just implement following interfaces located at sgcJSON.pas:

IsgcJSON
IsgcObjectJSON

There are 3 implementations of these interfaces

- **sgcJSON.pas:** default JSON parser provided.
- **sgcJSON_System.pas:** uses JSON parser provided with latest versions of delphi.
- **sgcJSON_XSuperObject.pas:** uses JSON library written by Onur YILDIZ, you can download sources from: https://github.com/onryldz/x-superobject

To use your own JSON parser or use some of the JSON parsers provided, just call **SetJSONClass** in your initialization method. For example: if you want to use XSuperObject JSON parser, just call:

```
SetJSONClass(TsgcXSOJSON)
```

If you don't call this method, sgcJSON will be used by default.

More information

WAMP

The WebSocket Application Messaging Protocol (WAMP) is an open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

The WebSocket Protocol is already built into modern browsers and provides bidirectional, low-latency message-based communication. However, as such, WebSocket is quite low-level and only provides raw messaging.

Modern Web applications often have a need for higher level messaging patterns such as Publish & Subscribe and Remote Procedure Calls.

This is where The WebSocket Application Messaging Protocol (WAMP) enters. WAMP adds the higher level messaging patterns of RPC and PubSub to WebSocket - within one protocol.

Technically, WAMP is an officially registered WebSocket subprotocol (runs on top of WebSocket) that uses JSON as message serialization format.

More Information

WebRTC

WebRTC is a free, open project that enables web browsers with Real-Time Communications (RTC) capabilities via simple Javascript APIs. The WebRTC components have been optimized to best serve this purpose. The WebRTC initiative is a project supported by Google, Mozilla and Opera.

WebRTC offers web application developers the ability to write rich, real-time multimedia applications (think video chat) on the web, without requiring plugins, downloads or installs. Its purpose is to help build a strong RTC platform that works across multiple web browsers, across multiple platforms.

More Information

MQTT

MQTT (MQ Telemetry Transport or Message Queue Telemetry Transport) is an ISO standard (ISO/IEC PRF 20922) publish-subscribe-based "lightweight" messaging protocol for use on top of the TCP/IP protocol. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. The publish-subscribe messaging pattern requires a message broker. The broker is responsible for distributing messages to interested clients based on the topic of a message. Andy Stanford-Clark and Arlen Nipper of Cirrus Link Solutions authored the first version of the protocol in 1999.

The specification does not specify the meaning of "small code footprint" or the meaning of "limited network bandwidth". Thus, the protocol's availability for use depends on the context. In 2013, IBM submitted MQTT v3.1 to the OASIS specification body with a charter that ensured only minor changes to the specification could be accepted. MQTT-SN is a variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as ZigBee.
Historically, the "MQ" in "MQTT" came from IBM's MQ Series message queuing product line. However, queuing itself is not required to be supported as a standard feature in all situations.

Specification
More Info

Server-Sent Events

Server-sent events (SSE) is a technology where a browser gets automatic updates from a server via HTTP connection. The Server-Sent Events EventSource API is standardized as part of HTML5 by the W3C.

A server-sent event is when a web page automatically gets updates from a server. This was also possible before, but the web page would have to ask if any updates were available. With server-sent events, the updates come automatically.

Examples: Facebook/Twitter updates, stock price updates, news feeds, sport results, etc.

More information
Browser Support

OAuth2

OAuth 2 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, and GitHub. It works by delegating user authentication to the service that hosts the user account, and authorizing third-party applications to access the user account. OAuth 2 provides authorization flows for web and desktop applications, and mobile devices.

Read more
Specification

JWT

JSON Web Token is an Internet proposed standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. The tokens are signed either using a private secret or a public/private key. For example, a server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could then use that token to prove that it is logged in as admin.

The tokens can be signed by one party's private key (usually the server's) so that party can subsequently verify the token is legitimate. If the other party, by some suitable and trustworthy means, is in possession of the corresponding public key, they too are able to verify the token's legitimacy. The tokens are designed to be compact, URL-safe, and usable especially in a web-browser single-sign-on (SSO) context. JWT claims can typically be used to pass identity of authenticated users between an identity provider and a service provider, or any other type of claims as required by business processes.

Read more
Specification

STUN

Session Traversal Utilities for NAT (STUN) is a standardized set of methods, including a network protocol, for traversal of network address translator (NAT) gateways in applications of real-time voice, video, messaging, and other interactive communications.

STUN is a tool used by other protocols, such as Interactive Connectivity Establishment (ICE), the Session Initiation Protocol (SIP), and WebRTC. It provides a tool for hosts to discover the presence of a network address translator, and to discover the mapped, usually public, Internet Protocol (IP) address and port number that the NAT has allocated for the application's User Datagram Protocol (UDP) flows to remote hosts. The protocol requires assistance from a third-party network server (STUN server) located on the opposing (public) side of the NAT, usually the public Internet.

Read more
Specification

AMQP

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are interoperable, in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems. Previous standardizations of middleware have happened at the API level (e.g. JMS) and were focused on standardizing programmer interaction with different middleware implementations, rather than on providing interoperability between multiple implementations. Unlike JMS, which defines an API and a set of behaviors that a messaging implementation must provide, AMQP is a wire-level protocol. A wire-level protocol is a description of the format of the data that is sent across the network as a stream of bytes. Consequently, any tool that can create and interpret messages that conform to this data format can interoperate with any other compliant tool irrespective of implementation language.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardised but extensible 'messaging capabilities.'

Specification
More Info

TURN

Traversal Using Relays around NAT (TURN) is a protocol that assists in traversal of network address translators (NAT) or firewalls for multimedia applications. It may be used with the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). It is most useful for clients on networks masqueraded by symmetric NAT devices. TURN does not aid in running servers on well known ports in the private network through a NAT; it supports the connection of a user behind a NAT to only a single peer, as in telephony, for example.

Read more
Specification

License

eSeGeCe Components End-User License Agreement
eSeGeCe Components ("eSeGeCe") End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and the Author of eSeGeCe for all the eSeGeCe components which may include associated software components, media, printed materials, and "online" or electronic documentation ("eSeGeCe components"). By installing, copying, or otherwise using the eSeGeCe components, you agree to be bound by the terms of this EULA. This license agreement represents the entire agreement concerning the program between you and the Author of eSeGeCe, (referred to as "LICENSER"), and it supersedes any prior proposal, representation, or understanding between the parties. If you do not agree to the terms of this EULA, do not install or use the eSeGeCe components.
The eSeGeCe components are protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The eSeGeCe components are licensed, not sold.
If you want SOURCE CODE you need to pay the registration fee. You must NOT give the license keys and/or the full editions of eSeGeCe (including the DCU editions and Source editions) to any third individuals and/or entities. And you also must NOT use the license keys and/or the full editions of eSeGeCe from any third individuals' and/or entities'.
1. GRANT OF LICENSE
The eSeGeCe components are licensed as follows:
(a) Installation and Use.
LICENSER grants you the right to install and use copies of the eSeGeCe components on your computer running a validly licensed copy of the operating system for which the eSeGeCe components were designed [e.g., Windows 2000, Windows 2003, Windows XP, Windows ME, Windows Vista, Windows 7, Windows 8, Windows 10].
(b) Royalty Free.
You may create commercial applications based on the eSeGeCe components and distribute them with your executables, no royalties required.
(c) Modifications (Source editions only).
You may make modifications, enhancements, derivative works and/or extensions to the licensed SOURCE CODE provided to you under the terms set forth in this license agreement.
(d) Backup Copies.
You may also make copies of the eSeGeCe components as may be necessary for backup and archival purposes.
2. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS
(a) Maintenance of Copyright Notices.
You must not remove or alter any copyright notices on any and all copies of the eSeGeCe components.
(b) Distribution.
You may not distribute registered copies of the eSeGeCe components to third parties. Evaluation editions available for download from the eSeGeCe official websites may be freely distributed.
You may create components/ActiveX controls/libraries which include the eSeGeCe components for your applications but you must NOT distribute or publish them to third parties.
(c) Prohibition on Distribution of SOURCE CODE (Source editions only).
You must NOT distribute or publish the SOURCE CODE, or any modification, enhancement, derivative works and/or extensions, in SOURCE CODE form to third parties.
You must NOT make any part of the SOURCE CODE be distributed, published, disclosed or otherwise made available to third parties.
(d) Prohibition on Reverse Engineering, Decompilation, and Disassembly.
You may not reverse engineer, decompile, or disassemble the eSeGeCe components, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.
(e) Rental.
You may not rent, lease, or lend the eSeGeCe components.
(f) Support Services.
LICENSER may provide you with support services related to the eSeGeCe components ("Support Services"). Any supplemental software code provided to you as part of the Support Services shall be considered part of the eSeGeCe components and subject to the terms and conditions of this EULA.
eSeGeCe is licensed to be used by only one developer at a time. And the technical support will be provided to only one certain developer.
(g) Compliance with Applicable Laws.
You must comply with all applicable laws regarding use of the eSeGeCe components.
3. TERMINATION
Without prejudice to any other rights, LICENSER may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the eSeGeCe components in your possession.

4. COPYRIGHT

All title, including but not limited to copyrights, in and to the eSeGeCe components and any copies thereof are owned by LICENSER or its suppliers. All title and intellectual property rights in and to the content which may be accessed through use of the eSeGeCe components are the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This EULA grants you no rights to use such content. All rights not expressly granted are reserved by LICENSER.

5. NO WARRANTIES

LICENSER expressly disclaims any warranty for the eSeGeCe components. The eSeGeCe components are provided "As Is" without any express or implied warranty of any kind, including but not limited to any warranties of merchantability, non-infringement, or fitness of a particular purpose. LICENSER does not warrant or assume responsibility for the accuracy or completeness of any information, text, graphics, links or other items contained within the eSeGeCe components. LICENSER makes no warranties respecting any harm that may be caused by the transmission of a computer virus, worm, time bomb, logic bomb, or other such computer program. LICENSER further expressly disclaims any warranty or representation to Authorized Users or to any third party.

6. LIMITATION OF LIABILITY

In no event shall LICENSER be liable for any damages (including, without limitation, lost profits, business interruption, or lost information) rising out of "Authorized Users" use of or inability to use the eSeGeCe components, even if LICENSER has been advised of the possibility of such damages. In no event will LICENSER be liable for loss of data or for indirect, special, incidental, consequential (including lost profit), or other damages based in contract, tort or otherwise. LICENSER shall have no liability with respect to the content of the eSeGeCe components or any part thereof, including but not limited to errors or omissions contained therein, libel, infringements of rights of publicity, privacy, trademark rights, business interruption, personal injury, and loss of privacy, moral rights or the disclosure of confidential information.

Index

info@esegece.com
www.esegece.com

API FXCM

FXCM

FXCM, also known as Forex Capital Markets, is a retail broker for trading on the foreign exchange market. FXCM allows people to speculate on the foreign exchange market and provides trading in contract for difference (CFDs) on major indices and commodities such as gold and crude oil. It is based in London.

FXCM offers a web-based REST API which can be used to establish secure connectivity with FXCM's trading systems for the purpose of receiving market data and trading.

The FXCM sgcWebSockets component uses WebSocket (socket.io) and HTTP as transports to connect to FXCM servers.

Connection

To use the REST API, you will need:

- **Access Token** generated with Trading Station Web https://tradingstation.fxcm.com/.
- **Demo:** if enabled, it will connect to FXCM Demo server (by default false).

```
// ... create components
TsgcWebSocketClient oClient = new TsgcWebSocketClient();
TsgcWSAPI_FXCM oFXCM = new TsgcWSAPI_FXCM();
oFXCM.Client = oClient;

// ... set properties FXCM
oFXCM.FXCM.AccessToken = "here your access token";
oFXCM.FXCM.Demo = false;

// ... connect to server
oClient.Active = true;
```

Messages

Once authenticated against server, FXCM uses websocket to receive unsolicited messages like price updates and you can request data from server using HTTP methods.

```
oClient.OnMessage += delegate(TsgcWSConnection Connection, string Text)
{
  // ... here we receive all messages from server
};
```

Methods

- **GetSymbols:** Request a list of all available symbols.
- **SubscribeMarketData:** After subscribing, market price updates will be pushed to the client via the socket.
- **SubscribeTradingTables:** Subscribes to the updates of the data models. Updates will be pushed to client via the socket. Type of update can be determined by "Model" Parameter.
- **SnapshotTradingTables:** In case continuous updates of the trading tables is not needed, it is possible to request a one-time snapshot. Gets current content snapshot of the specified data models.
    - Model choices:
        - Offer
        - OpenPosition
        - ClosedPosition
        - Order
        - Summary

    - LeverageProfile
    - Account
    - Properties
- **UpdateSubscriptions:** Offers table will show only symbols that we have subscribed to using update_subscriptions.
- **TradingOrder:** allows you to send open orders, modify, close...
- **GetHistoricalData:** Allows the user to retrieve candles for a given instrument at a given time frame. If time range is specified, number of candles parameter is ignored, but still required. There is a limit on the number of candles that can be returned in one request.

API Huobi

Huobi (HTX)
Huobi (now rebranded as HTX) is an international multi-language cryptocurrency exchange.

Configuration

If you want to subscribe to the private account updates, you need to create an API Key in your Huobi Account. Once created, set the API Key and Secret in the Huobi API Client

- ApiKey: the api key created in your huobi account.
- ApiSecret: the api secret.

If the ApiKey is not empty, the client will attempt to connect to the private websocket server, so only the private methods will be available. If the ApiKey is empty, the client will connect to the public websocket server and only the public methods will be available. If you need to subscribe to public and private methods, you need 2 connections.

Public Methods

You can subscribe to the following public channels (api key is not required)

Method	Description
SubscribeKLine	This topic sends a new candlestick whenever it is available. Supported periods: 1min, 5min, 15min, 30min, 60min, 4hour, 1day, 1mon, 1week, 1year.
SubscribeMarketDepth	This topic sends the latest market by price order book in snapshot mode at 1-second interval. Supported depth aggregation levels: step0 through step15.
SubscribeTradeDetail	This topic sends the latest completed trades. It updates in tick by tick mode.
SubscribeMarketDetail	This topic sends the latest market stats with 24h summary. It updates in snapshot mode, at a frequency of no more than 10 times per second
SubscribeBBO	User can receive BBO (Best Bid/Offer) update in tick by tick mode.
SubscribeMarketTicker	Retrieve the market ticker. Data is pushed every 100ms.
SubscribeMarket-ByPrice	User could subscribe to this channel to receive refresh update of Market By Price order book. The update interval is around 100ms. Supported levels: 5, 10, 20, 150, 400.

Futures Public Methods

The following methods are available for the Futures API client (TsgcWS_API_Huobi_Fut) in addition to all the public methods above.

Method	Description
SubscribeIncremental-MarketDepth	Subscribe to incremental market depth updates with configurable size and data type (snapshot or incremental).
SubscribePremiumIn-dexKLine	Subscribe to premium index kline/candlestick data for futures contracts.
SubscribeEstimate-dRateKLine	Subscribe to estimated funding rate kline/candlestick data for futures contracts.
SubscribeBasisData	Subscribe to basis data (spot-futures price spread). Supports different basis price types: open, close, high, low.

SubscribeMarkPriceK-Line	Subscribe to mark price kline/candlestick data for futures contracts.
SubscribeLiquida-tionOrders	Subscribe to public liquidation order feed for a given contract. No authentication required.
SubscribeFundingRate	Subscribe to public funding rate updates for a given contract. No authentication required.
SubscribeContractInfo	Subscribe to contract parameter changes (e.g. contract listings, delistings, parameter adjustments).

Private Methods

You can subscribe to the following private channels (an api key is required). If the credentials are not correct, the connection will be closed automatically.

Method	Description
SubscribeOrderUp-dates	An order update can be triggered by any of following: - Conditional order triggering failure (eventType=trigger) - Conditional order cancellation before trigger (eventType=deletion) - Order creation (eventType=creation) - Order matching (eventType=trade) - Order cancellation (eventType=cancellation)
SubscribeTradeClear-ing	Only update when order is in transaction or cancellation. Order transaction update is in tick by tick mode, which means, if a taker's order matches with multiple maker's orders, the simultaneous multiple trades will be disseminated one by one. But the update sequence of the multiple trades, may not be exactly the same as the sequence of the transactions made. Also, if an order is auto cancelled immediately just after its partial fills, for example a typical IOC order, this channel would possibly disseminate the cancellation update first prior to the trade.
SubscribeAc-countChange	Upon subscription field value specified, the update can be triggered by either of following events. The aMode parameter controls the update behavior: Mode 0: Only update when account balance is changed. Mode 1: Update when either account balance or available balance is changed (separate updates). Mode 2: Update when account balance or available balance is changed (combined update).

Events

OnHuobiSubscribed: event called after a successful subscription.
OnHuobiUnSubscribed: event called after a successful unsubscription.
OnHuobiUpdate: every time there is an update in data (kline, market depth...) this event is called.
OnHuobiError: if there is an error in Huobi API, this event will provide information about error.

API Bitstamp

Bitstamp

Bitstamp is a bitcoin exchange based in Luxembourg. It allows trading between USD currency and bitcoin cryptocurrency. It allows USD, EUR, bitcoin, litecoin, ethereum, ripple or bitcoin cash deposits and withdrawals.

- Supports the latest **WebSocket API V2**.
- Supports the following **REST API** Endpoints: **Account Balance**, **User Transactions**, **Fees**, **Orders**, **Withdrawal**, **Deposit Addresses**, **Sub-Account Transfers**, **Earn/Staking**, **Travel Rule** and **Markets**.

Configuration

Private API requires you to create an API key from your Bitstamp account. Once you've your API keys, configure these keys in the following properties:

- Bitstamp.ApiKey
- Bitstamp.ApiSecret

WebSocket Public Methods

SubscribeLiveTicker: get live trades from currency selected. JSON data:

Property	Description
id	Trade unique ID.
amount	Trade amount.
amount_str	Trade amount represented in string format.
price	Trade price.
price_str	Trade price represented in string format.
type	Trade type (0 - buy; 1 - sell).
timestamp	Trade timestamp.
microtimestamp	Trade microtimestamp.
buy_order_id	Trade buy order ID.
sell_order_id	Trade sell order ID.

SubscribeLiveOrders: get live orders from currency selected. JSON data:

Property	Description
id	Order ID.
amount	Order amount.
amount_str	Order amount represented in string format.
price	Order price.
price_str	Order price represented in string format.
order_type	Order type (0 - buy; 1 - sell).
datetime	Order datetime.
microtimestamp	Order action timestamp represented in microseconds.

SubscribeLiveOrderBook: get live order book from currency selected. JSON data:

Property	Description
bids	List of top 100 bids.
asks	List of top 100 asks.
timestamp	Order book timestamp.
microtimestamp	Order book microtimestamp.

SubscribeLiveDetailOrderBook: get live detail order book from currency selected. JSON data:

Property	Description
bids	List of top 100 bids [price, amount, order id].
asks	List of top 100 asks [price, amount, order id].
timestamp	Order book timestamp.
microtimestamp	Order book microtimestamp.

SubscribeLiveFullOrderBook: get live full order book from currency selected. JSON data:

Property	Description
bids	List of changed bids since last broadcast.
asks	List of changed asks since last broadcast.
timestamp	Order book timestamp.
microtimestamp	Order book microtimestamp.

WebSocket Private Methods

SubscribeMyOrders: get updates about the orders sent. JSON data:

Property	Description
id	Order ID.
id_str	Order ID represented in string format.
client_order_id	Client order ID (if used when placing order).
amount	Order amount.
amount_str	Order amount represented in string format.
price	Order price.
price_str	Order price represented in string format.
order_type	Order type (0 - buy, 1 - sell).
datetime	Order datetime.
microtimestamp	Order action timestamp represented in microseconds.

SubscribeMyTrades: get updates about the trades. JSON data:

Property	Description
id	Trade ID.
order_id	Order ID associated with the trade.
client_order_id	Client order ID associated with the trade.
amount	Trade amount.
price	Trade price.
fee	Trade fee.
side	Trade side (buy or sell).
microtimestamp	Trade timestamp represented in microseconds.

REST API Public Methods

Tickers

- **GetCurrencies:** View that returns list of all currencies with basic data.
- **GetAllCurrencyPairsTickers:** Return ticker data for all currency pairs. Passing any GET parameters, will result in your request being rejected.
- **GetCurrencyPairTicker:** Return ticker data for the requested currency pair. Passing any GET parameters, will result in your request being rejected.

- **GetHourlyTicker:** Return hourly ticker data for the requested currency pair. Passing any GET parameters, will result in your request being rejected.

Order Book

- **GetOrderBook:** Returns order book data.

Transactions

- **GetTransactions:** Return transaction data from a given time frame.

Market Info

- **GetEURUSDConversionRate:** Return EUR/USD conversion rate.
- **GetOLHCData:** View that returns OHLC (Open High Low Close) data on api request.
- **GetTradingPairsInfo:** Return trading pairs info.
- **GetMarkets:** Return list of all available markets with basic data.

Travel Rule

- **GetTravelRuleVASPs:** Return list of Virtual Asset Service Providers (VASPs) for Travel Rule compliance.

REST API Private Methods

Account Balance

- **GetAccountBalances:** Return account balances.
- **GetAccountBalanceForCurrency:** Return account balances for currency.

Orders

- **BuyInstantOrder:** Open a buy instant order. By placing an instant order you acknowledge that the execution of your order depends on the market conditions and that these conditions may be subject to sudden changes that cannot be foreseen. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **BuyMarketOrder:** Open a buy market order. By placing a market order you acknowledge that the execution of your order depends on the market conditions and that these conditions may be subject to sudden changes that cannot be foreseen. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **BuyLimitOrder:** Open a buy limit order. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **CancelAllOrders:** Cancel all open orders. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **CancelAllOrdersForCurrencyPair:** Cancel all open orders for a currency pair. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **CancelOrder:** Cancel an order. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **GetTradingPairs:** Returns all trading pairs that can be traded on selected account.
- **GetAllOpenOrders:** Return user's open orders. This API call is cached for 10 seconds. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **GetOpenOrders:** Return user's open orders for currency pair. This API call is cached for 10 seconds. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **GetOrderStatus:** Returns order status. This call will be executed on the account (Sub or Main), to which the used API key is bound to. Order can be fetched by using either id or client_order_id parameter. For closed orders, this call only returns information for the last 30 days. 'Order not found' error will be returned for orders outside this time range.

- **SellInstantOrder:** Open an instant sell order. By placing an instant order you acknowledge that the execution of your order depends on the market conditions and that these conditions may be subject to sudden changes that cannot be foreseen. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **SellMarketOrder:** Open a sell market order. By placing a market order you acknowledge that the execution of your order depends on the market conditions and that these conditions may be subject to sudden changes that cannot be foreseen. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **SellLimitOrder:** Open a sell limit order. This call will be executed on the account (Sub or Main), to which the used API key is bound to.

Withdrawals

- **RippleIOUWithdrawal:** This call will be executed on the account (Sub or Main), to which the used API key is bound to. This endpoint supports withdrawals of USD, BTC or EUR IOU on the XRP Ledger.
- **WithdrawalRequests:** Return user's withdrawal requests. This call will be executed on the account (Sub or Main), to which the used API key is bound to.
- **CancelBankOrCryptoWithdrawal:** Cancels a bank or crypto withdrawal request. This call can only be performed by your Main Account.
- **OpenBankWithdrawal:** Opens a bank withdrawal request (SEPA or international). Withdrawal requests opened via API are automatically confirmed (no confirmation e-mail will be sent), but are processed just like withdrawals opened through the platform's interface. This call can only be performed by your Main Account.
- **FiatWithdrawalStatus:** Checks the status of a fiat withdrawal request. This call can only be performed by your Main Account.
- **CryptoWithdrawal:** Request a crypto withdrawal.

User Transactions

- **GetUserTransactions:** Return all user transactions. Supports limit, offset and sort parameters.
- **GetUserTransactionsForCurrencyPair:** Return user transactions for a specific currency pair. Supports limit, offset and sort parameters.

Fees

- **GetTradingFees:** Return all trading fees.
- **GetTradingFeesForCurrencyPair:** Return trading fees for a specific currency pair.
- **GetWithdrawalFees:** Return withdrawal fees for all currencies.

Deposit Addresses

- **GetCryptoDepositAddress:** Return the deposit address for the specified cryptocurrency.

Sub-Account Transfers

- **TransferToMain:** Transfer funds from a sub-account to the main account.
- **TransferFromMain:** Transfer funds from the main account to a sub-account.

Earn / Staking

- **EarnSubscribe:** Subscribe to the Earn program for the specified currency and amount.
- **EarnUnsubscribe:** Unsubscribe from the Earn program for the specified currency and amount.
- **GetEarnSubscriptions:** Return current Earn subscriptions.
- **GetEarnTransactions:** Return Earn transaction history (rewards, subscriptions, unsubscriptions).