



# sgcWebSockets .NET 2026.2

February 2026

Documentation for .NET

Copyright © 2012-2026 eSeGeCe Software  
[info@esegece.com](mailto:info@esegece.com)  
[www.esegece.com](http://www.esegece.com)

# Contents

---

<b>Introduction .....</b>	<b>10</b>
<b>Overview .....</b>	<b>12</b>
Editions.....	12
Installation .....	13
<b>QuickStart.....</b>	<b>14</b>
Overview .....	14
QuickStart WebSockets .....	16
Threading Flow .....	18
Build .....	19
OpenSSL .....	20
OpenSSL Windows .....	23
OpenSSL OSX .....	25
OpenSSL Own CA Certificates.....	27
OpenSSL P12 Certificates .....	29
OpenSSL Verify Certificate .....	30
<b>Topics .....</b>	<b>31</b>
WebSocket Events.....	31
WebSocket Parameters Connection .....	32
Using inside a DLL.....	33
Web Browser Test .....	34
Authentication .....	35
Secure Connections .....	37
HeartBeat .....	38
WatchDog.....	39
Logs.....	40

HTTP.....	41
Broadcast and Channels .....	42
Bindings.....	43
Post Big Files .....	44
Compression.....	46
Flash.....	47
IOCP .....	48
ALPN .....	49
Forward HTTP Requests .....	50
TCP Connections .....	51
SubProtocol .....	52
Throttle.....	53
Server-Sent Events .....	54
Fragmented Messages .....	56
<b>Components .....</b>	<b>57</b>
TsgcWebSocketClient.....	57
Connect WebSocket Server .....	63
Client Open Connection .....	64
Client Close Connection .....	66
Client Keep Connection Open .....	67
Dropped Disconnections.....	68
Connect TCP Server .....	69
Connections TIME_WAIT.....	70
WebSocket Redirections.....	71
Connect Secure Server .....	72
Certificates OpenSSL.....	73
Certificates SChannel.....	74
Client Send Text Message .....	76
Client Send Binary Message.....	77
Client Send Text and Binary Message .....	78

Receive Text Messages .....	79
Receive Binary Messages .....	80
Client Authentication .....	81
Client Exceptions .....	83
Client WebSocket HandShake .....	84
Client Register Protocol .....	85
Client Proxies .....	86
TsgcWebSocketServer .....	87
Server Start .....	94
Server Bindings .....	95
Server Startup Shutdown .....	96
Server Keep Active .....	97
Server SSL .....	98
Server Verify Certificate .....	100
Server Keep Connections Alive .....	101
Server Plain TCP .....	102
Server Close Connection .....	103
Server Authentication .....	104
Server Send Text Message .....	106
Server Send Binary Message .....	107
Server Receive Text Message .....	108
Server Receive Binary Message .....	109
Server Read Headers from Client .....	110
TsgcWebSocketHTTPServer .....	111
HTTP Server Requests .....	115
HTTP Dispatch Files .....	116
HTTP/2 Server .....	117
HTTP/2 Server Push .....	118
HTTP/2 Alternate Service .....	120
HTTP/2 Server Threads .....	121
TsgcWSConnection .....	123

Protocols .....	125
Protocols Javascript.....	126
Protocol MQTT.....	129
TsgcWSPClient_MQTT .....	131
Client MQTT Connect.....	137
Connect Mosquitto MQTT Servers .....	138
Client MQTT Sessions .....	139
Client MQTT Version .....	140
MQTT Publish Subscribe .....	141
MQTT Topics .....	142
MQTT Subscribe .....	143
MQTT Publish Message .....	144
MQTT Receive Messages .....	145
MQTT Publish and Wait Response .....	146
MQTT Clear Retained Messages.....	147
Protocol AppRTC .....	148
TsgcWSPServer_AppRTC.....	149
Protocol WebRTC .....	150
TsgcWSPServer_WebRTC.....	151
Protocol WebRTC Javascript.....	152
Protocol Files .....	153
TsgcWSPServer_Files.....	154
TsgcWSPClient_Files.....	156
TsgcWSMessageFile .....	158
How Send Files To Server .....	159
How Send Files To Clients .....	160
How Send Big Files .....	161
API Binance .....	162
Binance Connect WebSocket API .....	169
Binance Subscribe WebSocket Channel.....	170
Binance Get Market Data .....	171

Binance Private REST API.....	172
Binance Trade Spot.....	173
Binance Private Requests Time .....	175
API Binance Futures .....	176
API Binance Futures Trade .....	181
API SocketIO.....	182
API Whatsapp .....	184
WhatsApp Create App .....	188
WhatsApp Phone Number Id.....	190
WhatsApp Token .....	191
WhatsApp Webhook .....	192
WhatsApp Security .....	193
WhatsApp Send Messages .....	194
WhatsApp Send Interactive Messages.....	197
WhatsApp Send Template Messages.....	201
WhatsApp Receive Messages and Status Notifications.....	202
WhatsApp Send Files .....	204
WhatsApp Download Media .....	206
API Telegram.....	207
Send Telegram Message With Inline Buttons .....	215
Send Telegram Message With Buttons.....	216
Send Telegram Message Bold .....	217
Telegram Chat not found as Bot .....	218
Telegram Sponsored Messages .....	219
Send Telegram Invoice Message .....	220
Telegram Get SuperGroup Members .....	221
Add Telegram Proxy.....	222
Register Telegram User .....	223
RTCMultiConnection .....	224
WebPush .....	226
TsgcWSAPIServer_WebPush .....	227

TsgcWebPush_Client.....	229
Extensions .....	230
PerMessage-Deflate.....	231
Deflate-Frame .....	232
OAuth2 .....	233
TsgcHTTP_OAuth2_Client .....	234
OAuth2 Client for Web Applications .....	240
OAuth2 Client for Desktop Applications.....	241
TsgcHTTP_OAuth2_Server .....	242
OAuth2 Server Example .....	245
OAuth2 Customize Sign-In HTML .....	249
OAuth2 Server Endpoints.....	250
OAuth2 Register Apps .....	251
OAuth2 Recover Access Tokens .....	252
OAuth2 Server Authentication.....	253
OAuth2 None Authenticate URLs.....	254
JWT .....	255
TsgcHTTP_JWT_Client .....	257
TsgcHTTP_JWT_Server.....	260
Webauthn Javascript Client.....	262
STUN .....	266
TsgcSTUNClient .....	267
STUN Client UDP Retransmissions.....	270
STUN Client Long Term Credentials.....	271
TsgcSTUNServer .....	272
STUN Server Long Term Credentials .....	274
STUN Server Alternate Server.....	275
TURN .....	276
TsgcTURNClient .....	277
TURN Client Allocate IP Address.....	281
TURN Client Create Permissions .....	282

TURN Client Send Indication.....	283
TURN Client Channels.....	284
TsgcTURNServer .....	285
TURN Server Long Term Credentials .....	288
TURN Server Allocations.....	289
<b>Demos .....</b>	<b>290</b>
Server Chat.....	290
Client Chat.....	292
Client.....	294
Client MQTT .....	295
Client SocketIO .....	297
Server Monitor.....	298
Server Snapshots .....	301
Client Snapshots.....	302
Upload File .....	303
Server Authentication.....	305
KendoUI_Grid.....	306
ServerSentEvents .....	308
Server WebRTC.....	309
Server AppRTC.....	310
Telegram Client .....	312
<b>Reference .....</b>	<b>314</b>
WebSockets.....	314
HTTP/2 .....	315
JSON .....	316
WAMP .....	317
WebRTC .....	318
MQTT .....	319
Server-Sent Events .....	320
OAuth2 .....	321

JWT .....	322
STUN .....	323
TURN .....	324
<b>License .....</b>	<b>325</b>
License.....	325
<b>Index.....</b>	<b>327</b>
<b>PDF-Back-Cover .....</b>	<b>330</b>

# Introduction

---

WebSockets represent a long-awaited evolution in client/server web technology. They allow a single long-lived TCP socket connection to be established between the client and server, enabling bi-directional, full-duplex messages to be distributed instantly with little overhead, resulting in a very low latency connection.

Both the WebSocket API and native WebSocket support in browsers such as Google Chrome, Firefox, Opera, and a prototype Silverlight-to-JavaScript bridge implementation for Internet Explorer mean there are now WebSocket library implementations in Objective-C, .NET, Ruby, Java, Node.js, ActionScript, and many other languages.

The Internet wasn't designed to be so dynamic. It was designed to be a collection of HyperText Markup Language (HTML) pages, linked together to form a conceptual web of information. Over time, static resources increased in number and richer elements such as images became part of the web fabric. Server technologies evolved to allow dynamic server pages - pages whose content is generated in response to a request.

Soon the need for more dynamic web pages led to the availability of Dynamic HyperText Markup Language (DHTML), all thanks to JavaScript (let's pretend VBScript never existed). In the years that followed, we saw cross-frame communication in an attempt to avoid page reloads, followed by in-frame HTTP polling. Things started to get interesting with the introduction of LiveConnect, then the forever frame technique, and finally, thanks to Microsoft, we ended up with the XMLHttpRequest object and thus Asynchronous JavaScript and XML (AJAX). AJAX in turn enabled XHR Long-Polling and XHR Streaming. But none of these provided a truly standardised, cross-browser solution for real-time, bi-directional communication between a server and a client.

Finally, WebSockets are a standard for bi-directional, real-time communication between servers and clients. Initially in web browsers, but ultimately between any server and any client. The standards-first approach means that we as developers can finally create functionality that works consistently across multiple platforms. Connection limitations are no longer an issue as WebSockets represent a single TCP socket connection. Cross-domain communication has been considered from day one and is handled within the connection handshake. This means that services like Pusher can easily use them to provide a massively scalable real-time platform that can be used by any website, web, desktop or mobile application.

WebSockets don't make AJAX obsolete, but they do replace Comet (HTTP Long-polling/HTTP Streaming) as the solution of choice for true real-time functionality. AJAX should still be used for short-lived web service calls, and when we eventually see a good uptake in CORS supporting web services, it will become even more useful. WebSockets should now be the standard for real-time functionality, as they provide low-latency, bi-directional communication over a single connection. Even if a web browser doesn't natively support the WebSocket object, there are polyfill fallback options that almost guarantee that any web browser can actually make a WebSocket connection.

sgcWebSockets is a complete package providing access to the WebSockets protocol, allowing you to create Web-Socket servers and clients for .NET applications.

- Fully functional **multithreaded WebSocket** server according to **RFC 6455**.
- Supports **Windows 32 / Windows 64**.
- Supports **MacOS 64**.
- Supports **Linux64**.
- Assemblies for **.NET FRAMEWORK (2.0+)**, **.NET STANDARD (1.6+)**, **.NET CORE (1.0+)**.
- Supports **Chrome, Firefox, Safari, Opera** and **Internet Explorer** (including **iPhone, iPad and iPod**)
- **Multiple Threads** Support. Indy Servers support **IOCP** or default Indy one thread perconnection model.
- Supports **Message Compression** using PerMessage\_Deflate extension RFC 7692.
- Supports **Text and Binary Messages**.
- Supports **Server and Client Authentication**.
- Server component providing **WebSocket** and **HTTP connections** through the **same port**.
- **FallBack** support through Adobe **Flash** for old Web Browsers like Internet Explorer from 6+.
- Supports **Server-Sent Events** (Push Notifications) over HTTP Protocol.
- **WatchDog** and **HeartBeat** built-in support.
- Supports **client Socket.IO** connections.
- Supports **Telegram Client**.
- **Binance** Stock and Futures are supported (WebSocket, User Stream and REST APIs).
- **STUN** and **TURN** protocols are fully supported (client and Server components).
- Client WebSocket supports connections through **HTTP Proxy** Servers and **SOCKS Proxy** Servers.

# INTRODUCTION

- Events Available: **OnConnect**, **OnDisconnect**, **OnMessage**, **OnError**, **OnHandshake**
- Built-in **Javascript libraries** to support browser clients.
- **Easy** to setup
- **Javascript Events** for full control
- **SSL/TLS Support** for Server / Client Components (OpenSSL libraries required). **OpenSSL 1.1.1** and **3.0.0** libraries are supported. Client supports **SChannel** for Windows.

Find below a list of the components included in 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 install nuget package and use sgcWebSockets components

[Visual Studio Windows](#)

[Visual Studio Mac OS](#)

## Assembly Reference

You can work with sgcWebSockets .NET package without using nuget package, just open your project and **Add Reference** from contextual menu over project and select in Assemblies Folder the assembly you want to add as reference.

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 components which implements the HTTP/2 protocol.

[QuickStart HTTP](#)

## Threading Flow

sgcWebSockets components are threaded, which means that **connections runs 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 configure threading flow:

[How Configure NotifyEvents](#)

## How Build Applications

Build Applications with sgcWebSockets library is very easy, just follow the next tips which will helps to **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 | 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 his own thread to run asynchronously the responses from 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 runs well for clients that doesn't receive a lot of messages and for easy of use, because doesn't require to synchronize with the main thread when you want for example update a control of your form.

But when the server / client must process several messages in short period of time, it's better change this threading flow to another where the events are dispatched in the context of 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 the 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 don't expect to handle several messages per seconds and you need 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 which needs a high performance because you expect will require 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`

# QuickStart | Build

---

Build an application with sgcWebSockets library is very easy, only keep in mind if 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**, is enough to deploy the openSSL libraries in the same folder where application is located.

# 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 **configure openSSL** libraries for every Personality:

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

## openSSL Configurations

sgcWebSockets Indy based components allows 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 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-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 in a single one. Just create a new file and copy the content of both files on the new file. So you will have an 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 a different version of SSL/TLS protocol, to fix it, try to set the correct version in Server and/or client component

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

### SSL3\_GET\_RECORD: decryption failed or bad record mac

Usually these error is raised when:

1. Check that you are using the latest OpenSSL version, if is too old, update to latest supported.
2. If this error appears randomly, usually is because more than one thread is accessing to 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 Sys-Wow64 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 Sys-Wow64 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 Sys-Wow64 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 Sys-Wow64 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 doesn'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 is your application.

You can download latest libraries from your account.

## API 3.0

Requires the following libraries:

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

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

If you include the openSSL libraries in a 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 Build a MacOSX64 Application with openSSL libraries

<https://www.esgece.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 **oslsSymLinksDontLoad**. 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 proeprty 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 LibPath-Custom.
  - **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.

# 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 on 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 no, then you can set default property to **neAsynchronous**.

# WebSocket Parameters Connection

---

## Supported by

[TsgcWebSocketClient](#)  
Java script

Sometimes is useful to pass parameters from client to server when a new WebSocket the 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 to this Test Page, you need to type this 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)

WebSockets Specification doesn't have any authentication method and Web browser implementations don't allow sending custom headers on new WebSocket connections.

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

## Authentication/ Enabled

sgcWebSockets implements 3 different 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. 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:** client open WebSocket connection passing username and password as a parameter.

`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 Authenticated)
{
    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 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 would you use:

- **osIAPI\_1\_0:** it's default indy API, you can use standard Indy package with openssl 1.0.2 libraries.
- **osIAPI\_1\_1:** only select if you are compiling sgcWebSockets with our custom Indy library (Enterprise Edition). Will use openssl 1.1.1 libraries.
- **osIAPI\_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 will wait a response from server. Example: if value is 30, means will wait 30 seconds to receive a response before close 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 has 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 to the following property:

### LogFile/ Enabled

Once enabled, every time a new connection is established it will be logged in a text file. On Server component, if the file it's not created it will be created but with you can't access until the server is closed, if you want to open log file while the server is active, log file needs to be created before start 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 can not 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 doesn't know if 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 send 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 OnConnect event of server if a client is mobile, you can do something like following.

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

First cast Connection to TsgcWSConnectionServer to access subscription methods and if fits your filter, will be subscribed to desktop channel. Subscription to a channel can be done in any event, example, you can ask to client to tell you if it's 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, message will be only sent to other 70.

# Bindings

---

## Supported by

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

Usually, Servers have more than one IP, if you enable a WebSocket Server and set listening port to 80, when the server starts, tries to listen port 80 of ALL IP, so if you have 3 IP, it will block port 80 of each IP's.

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 a 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, 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 **RemoveBoundaries** 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 Server has activated this feature, it sends a response to the client with this protocol activated and all messages will be compressed, if Server doesn't 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, is better don't enable this property because it consumes cpu cycle to compress/decompress messages, but if you are using a big amount of data, you will notify and increase on messages 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.

# IOCP

---

## Supported by

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

IOCP for Windows is an API which allows handles thousands of connections using a limited pool of threads instead of using a thread for connection like Indy by default does.

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** only must be enabled if you want that connections are processed always 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 in the performance of the server and it's only recommended to set a value greater of zero only if you require this feature.

Enabling IOCP for windows servers is recommended when you need handle thousands of connections, if your server is only handling 100 concurrent connections at maximum you can stay with default Indy Thread model.

## OnDisconnect event not fired

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

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 trying to write in socket, if there is any error means that connection is closed. There are 2 options to detect disconnections:

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

# 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 compression 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)
begin
    if (AResponseInfo.Document == "/internal")
    {
        aForward.Enabled = true;
        aForward.URL = "http://localhost:8080";
    }
}

```

## Other Options

When you want forward an HTTP request, you have the 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 will use the same of 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');`

# TCP Connections

---

## Supported by

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

By default, sgcWebSocket use WebSocket as 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 don't know if the message is text or binary, so all messages received are handle `OnBinary` event.

### End of Message

If messages are big, sometimes can be received fragmented. There is a method to try to find end of message setting which bytes find. Example: STOMP protocol, all messages ends with byte 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, basically adds a header with protocols name supported by request, these protocols are received and if the receiver supports one of them, sends a response with subprotocol supported.

**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 = TsgcWebSocketServer.Create();
oServerBroker = TsgcWSPServer_Broker.Create();
oServerBroker.Server = oServer;
oServerSGC = TsgcWSPServer_sgc.Create();
oServerSGC.Broker = oServerBroker;
oServerFiles = TsgcWSPServer_files.create();
oServerFiles.Broker = oServerBroker;
// ... client
oClient = TsgcWebSocketClient.Create();
oClientBroker = TsgcWSPClient_Broker.Create();
oClientBroker.Client = oClient;
oClientSGC = TsgcWSPClient_sgc.Create();
oClientSGC.Broker = oClientBroker;
oClientFiles = TsgcWSPClient_files.create();
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. As much slower is client reading data, much slower is server writing 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("id: 1 \r data: Notification from server");
```

javascript code to listen "notifications" channel:

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

# 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 behaviour can be customized by **Options.FragmentedMessages** property, which accepts following values:

1. `frgOnlyBuffer`: this is the default value, means that packet messages will be buffered and only when all stream is received, `OnBinary` message will 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 big streams and receiver must show progress of the transfer.

**Example:** the client must send a stream of size 1.000.000 bytes to server and server wants 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 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 tries 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.

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

## Most common uses

- **Connection**
  - [How Connect 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.

# COMPONENTS

**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.

# COMPONENTS

**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 LibPath-Custom.

**LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.

**UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default, 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.

**osIsSymLinksDontLoad:** 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

**osIx509chHostName:** verifies the hostname certificate.

**osIx509chIPAddress:** verifies the ip address of the certificate.

**HostName:** set the hostname if it's different from the request.

**IPAddress:** set the ip address if it's different from the request.

**SChannel\_Options:** allows you to use a certificate from Windows Certificate Store.

**CertHash:** is the certificate Hash. You can find the certificate Hash running a `dir` command in powershell.

**CipherList:** here you can set which Ciphers will be used (separated by `:`). Example: `CALG_AES_256:CALG_AES_128`

**CertStoreName:** the store name where 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`

# TsgcWebSocketClient | Connect WebSocket Server

---

## URL Property

The most easy way to connect to a WebSocket server is use **URL** property and call **Active = true**.

**Example:** connect to [www.esgece.com](https://www.esgece.com) using secure connection.

```
oClient = new TsgcWebSocketClient();
oClient.URL = "wss://www.esgece.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.esgece.com](https://www.esgece.com) using secure connections

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

# TsgcWebSocketClient | Client Open Connection

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

## Active Property

The most easy way to open a new connection is Set Active property to true. This will attempts to connect to server using component configuration.

If you set Active property to false, will close connection if active.

This method is executed in the same thread that caller. So if you call in the Main Thread, method will be executed in Main Thread of application.

### Open Connection

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

When you call Active = true, **you can't still send any data to server** because client maybe is still connecting, you must first wait to OnConnect event is fired and then you can start to send messages to 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 to OnDisconnect event is fired.

## Start/Stop methods

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

### Open Connection

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

When you call Start(), **you can't still send any data to server** because client maybe is still connecting, you must first wait to OnConnect event is fired and then you can start to send messages to 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 to OnDisconnect event is fired.

## Connect/Disconnect methods

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

**Example:** connect to server and wait till 5 seconds

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

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

**Example:** connect to server and wait till 10 seconds

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

If after calling Disconnect() event the result is successful, this means that 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 connection is closed, you can notify other peer that connection is closed sending a message about close connection, to enable this feature, Set Options.CleanDisconnect property to true.

If this property is enabled, before connection is closed, a Close message will be sent to server to notify that client is closing connection.

## Disconnect

[TsgcWSConnection](#) has a method called Disconnect(), that allows you to disconnect connection at socket level. If you call this method, socket will be disconnected directly without waiting any response from 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 server requesting to close connection, if server receives this message, must close the connection and client will receive a notification that connection is closed. You can send a Close Code with this method.

# TsgcWebSocketClient |Client Keep Connection Open

Once your client has connected to server, sometimes connection can be closed due to poor signal, connection errors... there are 2 properties which helps 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, but you can set a TimeOut interval if you want to close connection if a response from 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 behaviour. By default, if HeartBeat is enabled, client will send a websocket ping every X seconds set by 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 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 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 attempts to connect. **Example:** after 3 retries, if the client cannot connect to a server, will attempts 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 that the connection is active.

## Disconnection reasons

- **Application closes:** when a process is finished, usually sends a FIN packet which acknowledges the other peer that connection has been closed. But if a process crashes there is no guarantee that this packet will be sent to other peer.
- **Device Closes:** if devices closes, most probably there won't be any notification about this.
- **Network cable unplugged:** if network cable is unplugged it's the same that a router closes, there is no data being transferred so connection is not closed.
- **Loss signal from router:** if application loses signal from router, 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 attempts to connect but this has some disadvantages, like you are consuming more resources, create new threads... and if other peer has rebooted, second connection will work but first won't.

### 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 that TCP protocol check if 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();
TCPKeepAlive.Enabled = true;
TCPKeepAlive.Time = 5000;
TCPKeepAlive.Interval = 1000;
```

# TsgcWebSocketClient | Connect TCP Server

---

TsgcWebSocketClient can connect to WebSocket servers but can connect to plain TCP Servers too.

## URL Property

The most easy way to connect to a WebSocket server is use **URL** property and call **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 server, there is an exchange between client and server to inform about the state of disconnection. When the process is finished, the client socket connection states as TIME\_WAIT during a variable time. This is a normal behavior, in windows operating systems, this time defaults to about 4 minutes.

You can reduce or eliminate this behaviour, do with careful, 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, means that TIME\_WAIT will waits as max as 5 seconds.
4. Save and restart the system.

### LINGER

Another option to avoid TIME\_WAIT state, is 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 **LingerState** property, by default has a value of -1. If you set a value of zero, the connection will be reset when disconnecting from socket without Timeout.

This options is probably the less recommended and only use as a last option.

# TsgcWebSocketClient | WebSocket Redirections

---

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

The WebSocket client, handle redirections automatically, so if detects the Server Response contains a redirection, it will disconnect the actual connection and attempts to connect with then 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's a redirection and reads the Location
  1. First Disconnects the actual connection.
  2. Update the URL property with the value of 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 none-secure connections.

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

## TLSTOptions

In **TLSTOptions** 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 **TLSTOptions.Version to tlsUndefined** is enough for the wide majority of WebSocket Servers.

TLSTOptions.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** this server **requires a TLS version newer** than you set.

If **TLSTOptions.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 **TLSTOptions.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 client connects using a 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, first must 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 certificate is protected by a password, set here the secret.

# TsgcWebSocketClient | Certificates SChannel

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

Connection through SChannel requires that **TLSSettings.IOHandler** = **iohSChannel**.

SChannel support 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 use it, you must convert 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 has PFX format, you only need to deploy the certificate and set in the **TLSSettings.Certificate** property the path to it.

```
TLSSettings.IOHandler = iohSChannel
TLSSettings.CertFile = <certificate path>
TLSSettings.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 in the **TLSSettings.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 in the **TLSSettings.SChannel\_Options** property the hash and where is located the certificate.

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

## COMPONENTS

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

# TsgcWebSocketClient | Client Send Text Message

---

Once client has connected to server, it can send Text Messages to server. To send a Text Message, just call WriteData() method and send your text message.

## Send a Text Message

Call To **WriteData()** method and send a Text message. This method is executed on the **same thread** that is called.

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

If **QueueOptions.Text** has a **different value from qmNone**, instead of be 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 the Response

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

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

# TsgcWebSocketClient | Client Send Binary Message

---

Once client has connected to server, it can send Binary Messages to server. To send a Text Message, just call WriteData() method and send your binary message.

## Send a Binary Message

Call To **WriteData()** method and send a Binary message. This method is executed on the **same thread** that is called.

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

If **QueueOptions.Binary** has a **different value from qmNone**, instead of be 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 the Response

Call To **WriteAndWaitData()** method to send a Binary message and wait 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 you to types of messages: Text or Binary. But you can't send a binary with text in the same message.

One way to solve this, is add a header to binary message before is sent and decode this binary message when is received.

There are 2 functions in sgcWebSocket\_Helpers which can be used to set a short description of binary packet, basically adds a header to stream which is used to identify binary packet.

**Before send a binary message, call method to encode stream.**

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

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

```
sgcWSBytesRead(oBytes, VID);
```

The only limitation is that text used to identify 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 client receives a Text Message, **OnMessage** event is fired, just read Text parameter to know the string of 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 **executed 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 client receives a Binary Message, **OnBinary** event is fired, just read Data parameter to know 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 **executed 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 HTTP Header during WebSocket HandShake, just set in Authentication.Token.AuthToken the required token by server.
- **Session:** first client request an HTTP session to server and if server returns a session this is passed in GET HTTP Header of WebSocket HandShake. (\* own authorization method for sgcWebSockets library).
- **URL:** client request authorization using GET HTTP Header of WebSocket HandShake. (\* own authorization method for sgcWebSockets library).

## Authorization Basic

Is a simple authorization method where user and password are encoded and passes as an HTTP Header. Just set User and Password and enable only 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 to get Authorization using JWT, requires you obtain a Token using any external tool (example: using an HTTP connection, OAuth2...).

If you Attach an OAuth2 component, you can obtain this token automatically. Read more about [OAuth2](#).  
Basically 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 client connects to server using an HTTP connection requesting a new Session, if successful, server returns a SessionId and client sends this SessionId in GET HTTP Header of WebSockets HandShake.

Requires to set UserName and Password and set Session Authentication to True.

```
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, just passes username and password in GET HTTP Header of WebSockets 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 some errors in communications, server can disconnect a connection because it's not authorized or a message hasn't 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: " + Error);
}
```

## OnException

This event is fired every time there is an exception like write a socket is not active, access to an object that not exists

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

By default, when **connection is closed by server**, an **exception will be fired**, if you don't want that these exceptions are fired, just disable in **Options.RaiseDisconnectExceptions**.

# TsgcWebSocketClient | WebSocket Hand-Shake

---

WebSocket protocol uses an HTTP HandShake to upgrade from HTTP Protocol to WebSocket protocol. This hand-shake is handled internally by TsgcWebSocket Client component, but you can add your custom HTTP headers if server requires some custom HTTP Headers info.

**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 check HandShake string before is sent to server using OnHandShake event too.

# TsgcWebSocketClient | Client Register Protocol

---

By default, TsgcWebSocketClient doesn't make use of any SubProtocol, basically websocket sub-protocol are built on top of websocket protocol and defines a custom message protocol, example of websocket sub-protocols can be MQTT, STOMP...

WebSocket SubProtocol name is sent as an HTTP Header in WebSocket HandShake, this header is processed by server and if server supports this subprotocol will accept connection, if is not supported, 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 support connections through proxies, to configure a proxy connection, just fill the **Proxy** properties of 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).

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](#)

## 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 is 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 have 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 support 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 its 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).

# COMPONENTS

**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:**

**FailIfNoCertificate:** 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.

**oslpAPI\_1\_0:** uses API 1.0 OpenSSL, it's latest supported by Indy

**oslpAPI\_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).

**oslpAPI\_3\_0:** uses API 3.0 OpenSSL, requires our custom Indy library and allows using OpenSSL 3.0.0 libraries (with TLS 1.3 support).

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

**oslpNone:** this is the default, the openSSL libraries should be in the same folder where is the binary or in a known path.

**oslpDefaultFolder:** sets automatically the openSSL path where the libraries should be located for all IDE personalities.

**oslpCustomFolder:** if this is the option selected, define the full path in the property LibPath-Custom.

**LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.

**UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):

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

**oslsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.

**oslsSymLinksLoad:** Load SymLinks after trying to load the version libraries.

**oslsSymLinksDontLoad:** don't load the SymLinks.

**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

**osIx509chHostName:** verifies the hostname certificate.  
**osIx509chIPAddress:** 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.

# TsgcWebSocketServer | Start Server

---

The first you must set when you want start a Server is set a Listening Port, by default, this is set to port 80 but you can change for any port.

Once the port is set, there are 2 methods to start a server.

## Active Property

If you set Active property to true, server will start to listening all incoming connection on port set.

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

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

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

## Start / Stop methods

While if you call Active property the process of start / stop server is done in the same thread, calling Start and Stop methods will be executed in a secondary thread.

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

If you call Stop() method, 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**, server **binds listening port of ALL IPs**, so if for example, you have 3 IP: 127.0.0.1, 80.5411.22 and 12.55.41.17. Your server will bind this port on 3 IPs.

Usually is recommended only binding to needed IPs, here is where you can user Bindings property.

Instead of use Port property, just use Binding property and fill with IP and Port required.

**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 server has started and when has stopped.

## OnStartup

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

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

## OnShutdown

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

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

# TsgcWebSocketServer | Server Keep Active

---

Once server is started and OnShutdown event is fired, sometimes server can stopped for any reason. If you want to restart server after an unexpected close, you can use WatchDog property

## WatchDog

If WatchDog is enabled, when server detects a Shutdown, WatchDog try to restart again every X seconds until 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

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... For **testing purposes** you can use a **self-signed certificate** (check out in Demos/Chat 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 certificate contains entire certificate (public key, private key...) 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 None 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 (none 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;
```

# TsgcWebSocketServer | Server Verify Certificate

---

By default, the server doesn't verify the peer certificates. To configure the server to verify the client certificate implement the next 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 provides a certificate, the event is not fired.

You can configure the **server that only allow SSL connections using a certificate**, to do this, set the 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 your client has connected to server, sometimes connection can be closed due to poor signal, connection errors... use to keep connection alive.

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. solves this problem, sending a ping every a specific interval. Usually this is enough to maintain a connection active, but you can set a TimeOut interval if you want to close connection if a response from client is not received after X seconds.

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

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

# TsgcWebSocketServer | Server Plain TCP

---

WebSocket server accepts WebSocket, HTTP, SSE... protocols, but can work too with plain tcp connections. Read more about [TCP Connections](#).

There are 2 events, which can be used to handle TCP connections better.

## OnTCPConnect

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

This event allows you to know that a new client is trying to connect to server and server can accept or not the connection. By default, server always accept connection.

## OnUnknownProtocol

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

## OnConnect

This event is fired after a successful and complete connection, if connection is plain TCP, is fired after protocol is accepted in 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 connection at socket level. If you call this method, socket will be disconnected directly without waiting any response from 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 server requesting to close connection, if client receives this message, must close the connection and server will receive a notification that connection is closed. You can send a Close Code with this method.

## DisconnectAll

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

# TsgcWebSocketServer | Server Authentication

TsgcWebSocket server supports 3 types of Authentifications:

- **Basic:** read an HTTP Header during WebSocket HandShake with User and Password encoded as Basic Authorization.
- **Session:** first client request an HTTP session to server and if server returns a session this is passed in GET HTTP Header of WebSocket HandShake. (\* own authorization method for sgcWebSockets library).
- **URL:** read request authorization using GET HTTP Header of 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 server receives an Authentication Request from a client, this event is called to return if 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 Authentication is not supported by default, like JWT, still you can use this event to accept or not the connection. Just read the parameters and accept or not 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 client has connected to server, server can send text messages. To send a Text Message, just call 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 To **WriteData()** method and send a Text message.

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

If **QueueOptions.Text** has a **different value from qmNone**, instead of be 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 call to 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 To **Broadcast()** method to send a Text message to all connected clients.

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

# TsgcWebSocketServer | Server Send Binary Message

---

Once client has connected to server, server can send binary messages. To send a Binary Message, just call 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 To **WriteData()** method and send a Binary message.

```
TsgcWebSocketClient1.WriteData("guid", new MemoryStream());
```

If **QueueOptions.Binary** has a **different value from qmNone**, instead of be 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 call to 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 To **Broadcast()** method to send a Binary message to all connected clients.

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

# TsgcWebSocketServer | Server Receive Text Message

---

When server receives a Text Message, **OnMessage** event is fired, just read Text parameter to know the string of 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 **executed 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 server receives a Binary Message, **OnBinary** event is fired, just read Data parameter to know 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 **executed 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 **client connects** to WebSocket server, sends a list of **headers** with information about client connection. In order to read these client headers, you can **OnHandshake** event of Server component, which is called when server receives the headers from client and before sends a response to client.

Client headers are stores in **HeadersRequest** property of **TsgcWSConnectionServer**.

```
void OnServerHandshake(TsgcWSConnection Connection; var TStringList Headers);
begin
  MessageBox.Show(Headers.HeadersRequest.Text());
end;
```

# TsgcWebSocketHTTPServer

TsgcWebSocketHTTPServer implements Server WebSocket Component and can handle multiple threaded client connections as [TsgcWebSocketServer](#), and allows you to server 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.

**OnCommandOther:** 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 is 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.

\* 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](#)

## 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.

# 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, example: if you want that some endpoints are using an authorization scheme while others can be accessed without authorization, use the options parameter to allow or disable it. Find below an example when Authorization Basic is enabled but when a user requests the endpoint /public the authorization is not required.

```
public void OnBeforeCommand(TsgcWSConnection aConnection, TIdHTTPRequestInfo ARequestInfo, TIdHTTPResponseInfo AF
{
  if (AResponseInfo.Document == "/public")
    aOptions = TsgcHTTPCommandOptions.hcoAuthorizedBasic;
}
```

# TsgcWebSocketHTTPServer | HTTP Dispatch Files

---

When a client request a file, **OnCommandGet** event is fired, but you can use **DocumentRoot** property to dispatch automatically files.

**Example:** if you set **DocumentRoot** to **c:/www/files**. Every time a new file is requested, will search in this folder if file exists and if exists, 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 user exactly equal than with HTTP/1.1 protocol, [read more](#).

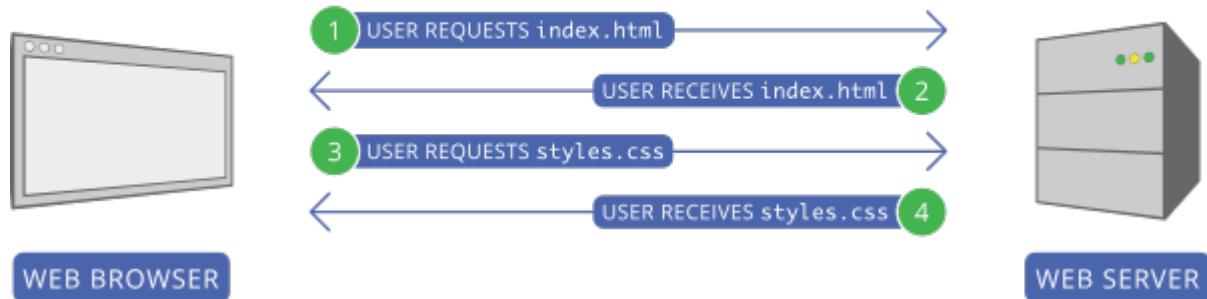
When HTTP/2 protocol is enabled, server will still support HTTP/1.1 requests.

By default, `OnConnect` and `OnDisconnect` events won't be called when there is a new HTTP/2 connection, but this can be modified accessing to properties `HTTP2Options.Events`, here you can customize if you want be notified every time there is a new HTTP/2 connection and/or disconnection.

# TsgcWebSocketHTTPServer | HTTP/2 Server Push

HTTP usually works with Request/Response pattern, where client REQUEST a resource to SERVER and SERVER sends a RESPONSE with the resource requested or an error. Usually the client, like a browser, makes a bunch of requests for those assets which are provided by the server.

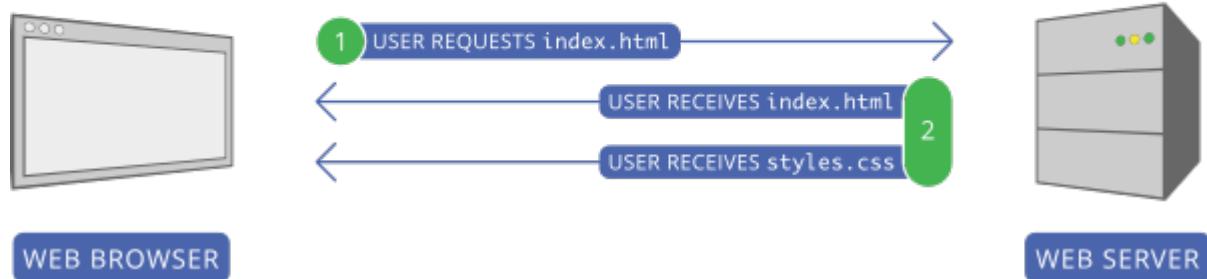
## TYPICAL WEB SERVER COMMUNICATION



The main problem of this approach is that first client must send a request to get the resource, example: index.html, wait till server sends the response, the client reads the content and then make all other requests, example: styles.css

HTTP/2 server push tries to solve this problem, when the client requests a file, if server thinks that this file needs another file/s, those files will be PUSHED to client automatically.

## WEB SERVER COMMUNICATION WITH HTTP/2 SERVER PUSH



In the prior screenshot, first client request index.html, server reads this request and sends as a response 2 files: index.html and styles.css, so it avoids a second request to get styles.css

## Configure Server Push

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

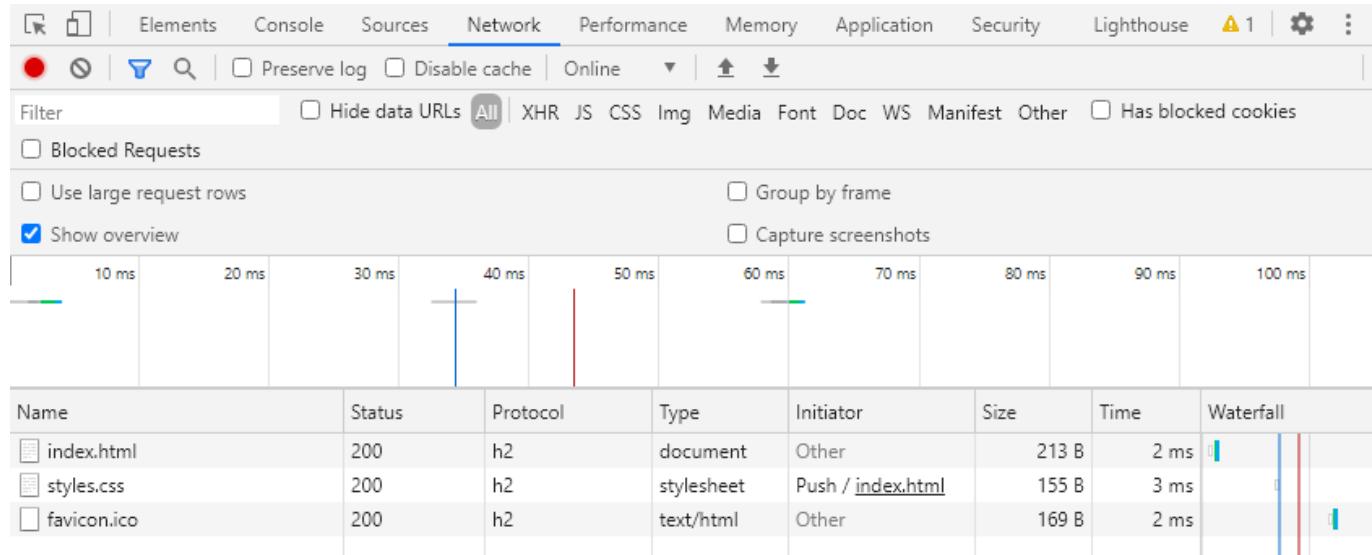
Use the method **PushPromiseAddPreLoadLinks**, to associate every request to a push promise list.

```
TsgcWebSocketHTTPServer server = new TsgcWebSocketHTTPServer(this);
server->PushPromiseAddPreLoadLinks("/index.html", "/styles.css");
void OnCommandGet(TsgcWSConnection Connection, TsgcWSHTTPRequestInfo RequestInfo, ref TsgcWSHTTPResponseInfo Response)
{
    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 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 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, automatically adds this header if the connection is not running on 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, the Web-Browsers implemented a workaround. It opens six-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-eight parallel connections by the 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 avoid waits when a single connection sends a lot of concurrent requests which will require processing sequentially (in the context of the connection thread) in the absence of this pool of threads.

The behaviour of the PoolOfThreads can be configured in the following properties.

- **HTTP2Options.PoolOfThreads.Enabled:** (by default false) enable to dispatch the 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 the HTTP/2 requests. Set a number according the number of processors of your server.

To **fine-tune the requests**, selecting which must be processed in the Pool Of Threads (because are time consuming) while others can be processed in the connection thread, you can use the event **OnHttp2BeforeAsyncRequest**, this event is raised before queue the request in the pool of threads, use the parameter **Async** to set if the request is threaded or not.

# TsgcWSConnection

---

TsgcWSConnection is a wrapper of client WebSocket connections, you can access to this object on 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 of closing won't 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.

**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 from 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 arrive, 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:** search in all packet if find bytes equal to bytes added with AddTCPEndOfFrame method. If true, OnBinary message is called, otherwise this packet is buffered

# 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 open on the Server side, it will validate if sub-protocol sent by the client is supported by the server, if not, then 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 to create customized applications and be sure 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 send files: packet size, authorization, QoS, message acknowledgement and more.

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 common javascript library used on 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 is the 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 light weight, 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 a MQTT server is simple, you need to drop this component in the form and select a [TsgcWebSocket-Client](#) 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 it supports 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 to 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 message 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 s 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 than 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.

# COMPONENTS

**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.

**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.

# COMPONENTS

- **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 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 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 s 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.

**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 subscribe method. There are the following parameters:

**PacketIdentifier:** is packet identifier sent initially.

**Codes:** codes with the result of a subscription.

**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.

**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 server must retain message after publish 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 s 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.

# 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

Somes servers require an user and password to **authorize 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 client **want 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-server 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 server, are received **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)
{
  WriteLine("Topic: " + aTopic + ". Message: " + aText);
}
```

# 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 message 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 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 that normal Publish method, now you have a new parameter called Timeout, where method will return with value false if after 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 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 Server Protocol AppRTC Component, you need to drop this component in the form and select a [TsgcWeb-  
SocketServer](#) 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 the browser to browser applications for voice calling, video chat and P2P file sharing without plug-ins. 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.esgece.com.html>

# TsgcWSPServer\_WebRTC

---

This is Server Protocol WebRTC Component, you need to drop this component in the form and select a [TsgcWeb-  
SocketServer](#) 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 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.esgece.com.js"></script>
```

## Open Connection

When a WebSocket connection is opened, browser request access to local camera and microphone, you need to allow access.

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esgece.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 attempts to connect to another client using `webrtc_connect` procedure

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esgece.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.esgece.com.js"></script>
<script>
  socket.webrtc_disconnect('custom channel');
</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 [TsgcWeb-SocketServer](#) 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 reconnects, try to resume file download for qosLevel2, by default is enabled.

**ClearSentStreamsOnDisconnect:** if disabled, when reconnects, try to resume file upload for qosLevel2, by default is enabled.

## Events

**OnFileBeforeSent:** fired before a file is sent. You can use this event to check file data before 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 Server Files Protocol Component, you need to drop this component in the form and select a [TsgcWeb-SocketClient](#) 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 reconnects, try to resume file download for qosLevel2, by default is enabled.

**ClearSentStreamsOnDisconnect:** if disabled, when reconnects, try to resume file upload for qosLevel2, by default is enabled.

## Events

**OnFileBeforeSent:** fired before a file is sent. You can use this event to check file data before 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 only will be sent to clients subscribed to specific channel.
- Method: internal method.
- FileId: identifier of a file, 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, 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 Server, just call the method **SendFile** of Files Protocol and pass the full **FileName** as argument. The file received by server, will be saved by default in the same directory where is the server executable 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 client, will be saved by default in the same directory where is the client executable 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, "");
```

# 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](#)
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, example: get ticker prices. Only are only private and related to user data, those methods requires 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 deletes 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)

## Most common uses

- **WebSockets API**
  - [How Connect WebSocket API](#)
  - [How Subscribe WebSocket Channel](#)
- **REST API**
  - [How Get Market Data](#)
  - [How Use Private REST API](#)
  - [How Trade Spot](#)
  - [Private Requests Time](#)

## 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: <ul style="list-style-type: none"><li>• Deposits or withdrawals from the account</li><li>• Transfer of funds between accounts (e.g. Spot to Margin)</li></ul>
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
GetHistorical-Trades	Symbol	Get older trades.
GetAggregate-Trades	Symbol	Get compressed, aggregate trades. Trades that fill at the time, from the same order, with the same price will have the quantity aggregated.
GetKLines	Symbol, Interval	Kline/candlestick bars for a symbol. Klines are uniquely identified by their open time.
GetAveragePrice	Symbol	Current average price for a symbol.

Get24hrTicker	Symbol	24 hour rolling window price change statistics. Careful when accessing this with no symbol.
GetPriceTicker	Symbol	Latest price for a symbol.
GetPriceTickers	Symbols	Latest price for an array of symbols. Example: ["BTCUSDT", "BNBUSDT"]
GetBookTicker	Symbol	Best price/qty on the order book for a symbol or symbols.

## Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

### Account Data EndPoints

Method	Parameters	Description
NewOrder	Symbol, Side, Type	Send in a new order.
PlaceMarketOrder	Side, Symbol, Quantity	Places a New Market Order
PlaceMarketQuoteOrder	Side, Symbol, QuoteOrderQty	Places a New Market Quote Order
PlaceLimitOrder	Side, Symbol, Quantity, LimitPrice	Places a New Limit Order
PlaceStopOrder	Side, Symbol, Quantity, StopPrice, LimitPrice	Places a New Stop Order
PlaceStopTrailingOrder	Side, Symbol, Quantity, TrailingDelta, LimitPrice	Places a New Stop Trailing Order
PlaceTakeProfitOrder	Side, Symbol, Quantity, StopPrice, LimitPrice	Places a New Take Profit Order
PlaceTakeProfitTrailingOrder	Side, Symbol, Quantity, TrailingDelta, LimitPrice	Places a New Take ProfitTrailing Order
PlaceLimitMakerOrder	Side, Symbol, Quantity	Places a New Limit Market Order
TestNewOrder	Symbol, Side, Type	Test new order creation and signature/recvWindow long. Creates and validates a new order but does not send it into the matching engine.
QueryOrder	Symbol	Check an order's status.
CancelOrder	Symbol	Cancel an active order. Cancel an active order. Either OrderId or OrigClientOrderId must be sent.
CancelAllOpenOrders	Symbol (optional)	
GetOpenOrders		Get all open orders on a symbol. Careful when accessing this with no symbol.
GetAllOrders	Symbol	Get all account orders; active, canceled, or filled.
NewOCO	Symbol, Side, Quantity, Price, StopPrice	Send in a new OCO
CancelOCO	Symbol	Cancel an entire Order List
QueryOCO	Symbol	Retrieves a specific OCO based on provided optional parameters
GetAllOCO		Retrieves all OCO based on provided optional parameters
GetOpenOCO		Get All Open OCO.
GetAccountInformation		Get current account information.

# COMPONENTS

GetAccountTradeList	Symbol	Get trades for a specific account and symbol.
---------------------	--------	-----------------------------------------------

## 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	QuotId	Accept the offered quote by quote ID.
GetConvertOrderStatus	OrderId or QuotId	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" <ul style="list-style-type: none"> <li>The query time period must be less than 30 days</li> <li>Support query within the last one month only</li> <li>If startTime and endTime not sent, return records of the last 7 days by default</li> </ul>
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.

## COMPONENTS

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: <ul style="list-style-type: none"><li>MAIN_CMFUTURE Spot account transfer to COIN-M Futures account</li><li>MAIN_MARGIN Spot account transfer to Margin (cross) account</li><li>UMFUTURE_MAIN USD\$-M Futures account transfer to Spot account</li><li>UMFUTURE_MARGIN USD\$-M Futures account transfer to Margin (cross) - account</li><li>CMFUTURE_MAIN COIN-M Futures account transfer to Spot account</li><li>CMFUTURE_MARGIN COIN-M Futures account transfer to Margin(cross) account</li><li>MARGIN_MAIN Margin (cross) account transfer to Spot account</li><li>MARGIN_UMFUTURE Margin (cross) account transfer to USD\$-M Futures</li><li>MARGIN_CMFUTURE Margin (cross) account transfer to COIN-M Futures</li><li>ISOLATEDMARGIN_MARGIN Isolated margin account transfer to Margin(cross) account</li><li>MARGIN_ISOLATEDMARGIN Margin(cross) account transfer to Isolated margin account</li><li>ISOLATEDMARGIN_ISOLATEDMARGIN Isolated margin account transfer to Isolated margin account</li><li>MAIN_FUNDING Spot account transfer to Funding account</li><li>FUNDING_MAIN Funding account transfer to Spot account</li><li>FUNDING_UMFUTURE Funding account transfer to UMFUTURE account</li><li>UMFUTURE_FUNDING UMFUTURE account transfer to Funding account</li><li>MARGIN_FUNDING MARGIN account transfer to Funding account</li><li>FUNDING_MARGIN Funding account transfer to Margin account</li><li>FUNDING_CMFUTURE Funding account transfer to CMFUTURE account</li><li>CMFUTURE_FUNDING CMFUTURE account transfer to Funding account</li></ul>
GetWalletQueryUserUniversalTransferHistory	<ul style="list-style-type: none"><li>fromSymbol must be sent when type are ISOLATEDMARGIN_MARGIN and ISOLATEDMARGIN_ISOLATEDMARGIN</li><li>toSymbol must be sent when type are MARGIN_ISOLATEDMARGIN and ISOLATEDMARGIN_ISOLATEDMARGIN</li><li>Support query within the last 6 months only</li><li>If startTime and endTime not sent, return records of the last 7 days by default</li></ul>
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.
GetWalletApiKeyPermission	

## Events

Binance Messages are received in TsgcWebSocketClient component, you can use the following events:

**OnConnect**

After a successful connection to Binance server.

**OnDisconnect**

After a disconnection from Binance server

**OnMessage**

Messages sent by server to client are handled in this event.

**OnError**

If there is any error in protocol, this event will be called.

**OnException**

If there is an unhandled exception, this event will be called.

Additionally, there is a specific event in Binance API Component, called **OnBinanceHTTPException**, which is raised every time there is an error calling an HTTP Request (REST API or WebSocket User Stream).

**(\*) Due to changes in Binance Servers, Indy versions before Rad Studio 10.1, won't be able to connect to Test Servers. This issue doesn't affect to Enterprise Edition or if the Indy version has been upgraded to latest.**

# Binance | Connect WebSocket API

---

In order to connect to Binance WebSocket API, just create a new Binance API client and attach to TsgcWebSocketClient.

See below an example:

```
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 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 an snapshot of the market data requested.

The Market Data Endpoints doesn't require authentication, so are freely available to all users.

**Example:** to get an snapshot of the ticker BNBBTC, do the following call

```
TsgcWSAPI_Binance oBinance = new TsgcWSAPI_Binance();
MessageBox.Show(oBinance.REST_API.GetPriceTicker("BNBBTC"));
```

# Binance | Private REST API

---

The Binance REST API offer public and private endpoints. The Private endpoints requires that messages 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 with spot using his 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 to method **REST\_API.NewOrder** of Binance Client Component.

Depending of the type of the order (market, limit...) the API requires more or less 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 so 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 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**.

# 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. Only are only private and related to user data, those methods requires 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 deletes 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.

## COMPONENTS

MarkPrice	Symbol, UpdateSpeed	Mark price and funding rate for a single symbol pushed every 3 seconds or every second.
AllMarkPrice	Update-Speed	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 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
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 Update occurs during the following:
Balance and Position Update	<ul style="list-style-type: none"> <li>When balance or position get updated, this event will be pushed.</li> <li>When "FUNDING FEE" changes to the user's balance.</li> <li>•</li> </ul>
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
GetHistorical-Trades	Symbol	Get older trades.
GetAggregate-Trades	Symbol	Get compressed, aggregate trades. Trades that fill at the time, from the same order, with the same price will have the quantity aggregated.
GetKLines	Symbol, Interval	Kline/candlestick bars for a symbol. Klines are uniquely identified by their open time.
Get24hrTicker	Symbol	24 hour rolling window price change statistics. Careful when accessing this with no symbol.
GetPriceTicker	Symbol	Latest price for a symbol or symbols.
GetBookTicker	Symbol	Best price/qty on the order book for a symbol or symbols.
GetMarkPrice	Symbol	Mark Price and Funding Rate
GetFundingRateHistory	Symbol	
GetOpenInterest	Symbol	Get present open interest of a specific symbol.
GetOpenInterestStatistics	Symbol, Period	
GetTopTrader-AccountRatio	Symbol, Period	
GetTopTrader-PositionRatio	Symbol, Period	
GetGlobalAccountRatio	Symbol, Period	

GetTakerVolume    Symbol, Period

## Private API EndPoints

Requires an APIKey and APISecret to get authorized by server.

### Account and Trades EndPoints

Method	Parameters	Description
ChangePosition-Mode	DualPosition	Change user's position mode (Hedge Mode or One-way Mode ) on EVERY symbol
GetCurrentPositionMode		Get user's position mode (Hedge Mode or One-way Mode ) on EVERY symbol
NewOrder	Symbol, Side, PositionSide, Type	Send in a new order.
PlaceMarketOrder	Side, Symbol, Quantity	
PlaceLimitOrder	Side, Symbol, Quantity, Limit-Price	
PlaceStopOrder	Side, Symbol, Quantity, Stop-Price, LimitPrice	
PlaceTrailingStopOrder	Side, Symbol, Quantity, aActivationPrice, aCallbackRate	
QueryOrder	Symbol	Check an order's status.
CancelOrder	Symbol	Cancel an active order. Either OrderId or OrigClientOrderId must be sent.
CancelAllOpenOrders	Symbol	
AutoCancelAllOpenOrders	Symbol, CountDownTimer	Cancel all open orders of the specified symbol at the end of the specified countdown.
QueryCurrentOpenOrder	Symbol	
GetOpenOrders	Symbol	Get all open orders on a symbol. Careful when accessing this with no symbol.
GetAllOrders	Symbol	Get all account orders; active, canceled, or filled.
GetAccountBalance		
GetAccountInformation		Get current account information.
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	
GetAccount-TradeList	Symbol	

GetIncomeHistory	Symbol
GetNotional-LeverageBracket	Symbol

## 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 with futures using his 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 of 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":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 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 to send 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 WriteData method to send messages to socket.io server (following Message Types sections)

1. call method add user and one parameter with John as user name

```
writeData("42[\"add user\", \"John\"]");
```

## Events

### OnHTTPRequest

Before a new websocket connection is established, socket.io server requires client open a new HTTP connection to get a new session id. In some cases, 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 socket.io connection is successful and client can send messages to server. Here you can subscribe to namespaces for example.

# 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](https://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 on-boarding 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 property 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 = "EAA040pgZAs98BAGj3nCFGr...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 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.esgece.com>, the endpoint will be "<https://www.esgece.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

The event is called before the message is sent to the WhatsApp servers, you can access to the internal message accessing to the RawMessage parameter.

### OnBeforeSubscribe

The event is called before the server subscribes to a topic, use the parameter Accept to subscribe or not, by default, the server will subscribe to all events requested.

### OnRawMessage

This event is called when the server receives a new message and still is not parsed, so you get access to the raw message.

### OnMessageReceived

This event is called after the server receives a new message and is parsed. If you set the parameter MarkAsRead to True, the sender will receive a double check.

### OnMessageSent

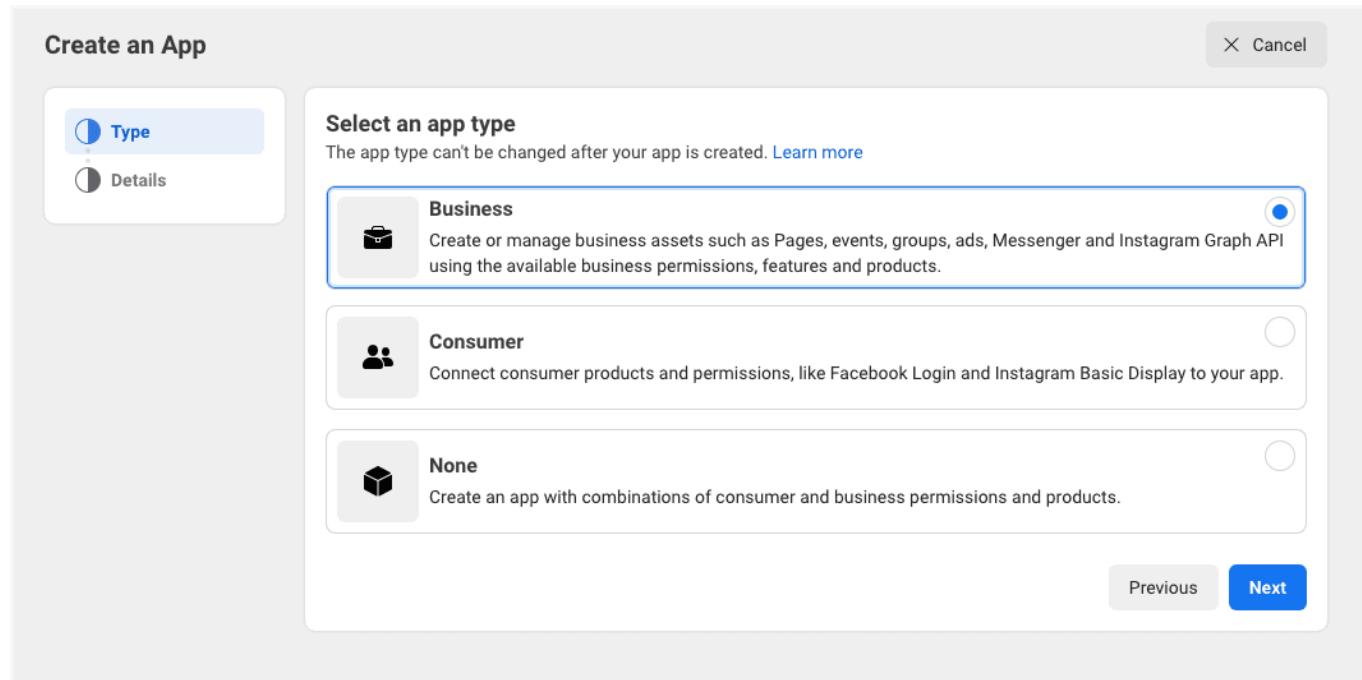
This event is called every time the server receives a new status message about the message previously sent. Read the Status property to know if the message has been sent, delivered or read.



# WhatsApp Create App

Go to [developers.facebook.com](https://developers.facebook.com) and **Create App**.

Select **Business Type** as the app type and proceed.



**Create an App**

**Select an app type**  
The app type can't be changed after your app is created. [Learn more](#)

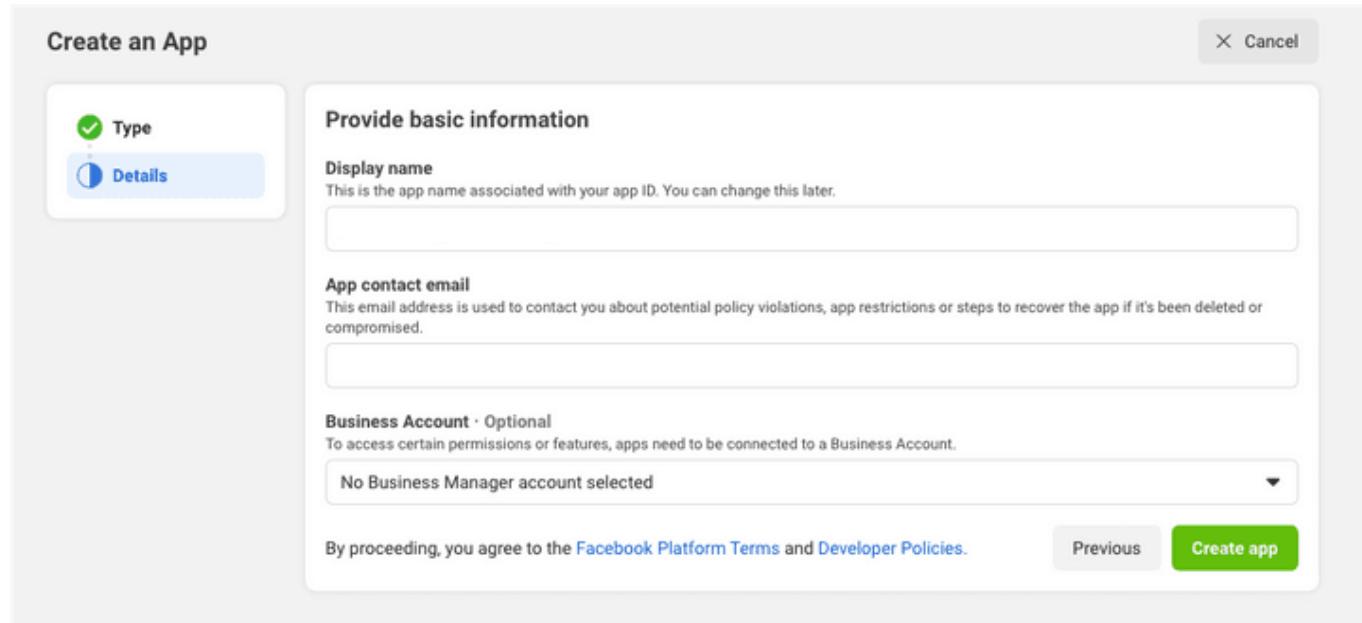
**Business**  Create or manage business assets such as Pages, events, groups, ads, Messenger and Instagram Graph API using the available business permissions, features and products.

**Consumer**  Connect consumer products and permissions, like Facebook Login and Instagram Basic Display to your app.

**None**  Create an app with combinations of consumer and business permissions and products.

[Previous](#) [Next](#)

Provide a name for your app (avoid using trademarked names such as “WhatsApp” or “Facebook”).



**Create an App**

**Type**  **Details**

**Provide basic information**

**Display name**  
This is the app name associated with your app ID. You can change this later.

**App contact email**  
This email address is used to contact you about potential policy violations, app restrictions or steps to recover the app if it's been deleted or compromised.

**Business Account · Optional**  
To access certain permissions or features, apps need to be connected to a Business Account.  
 No Business Manager account selected

By proceeding, you agree to the [Facebook Platform Terms](#) and [Developer Policies](#).

[Previous](#) [Create app](#)

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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFGr...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

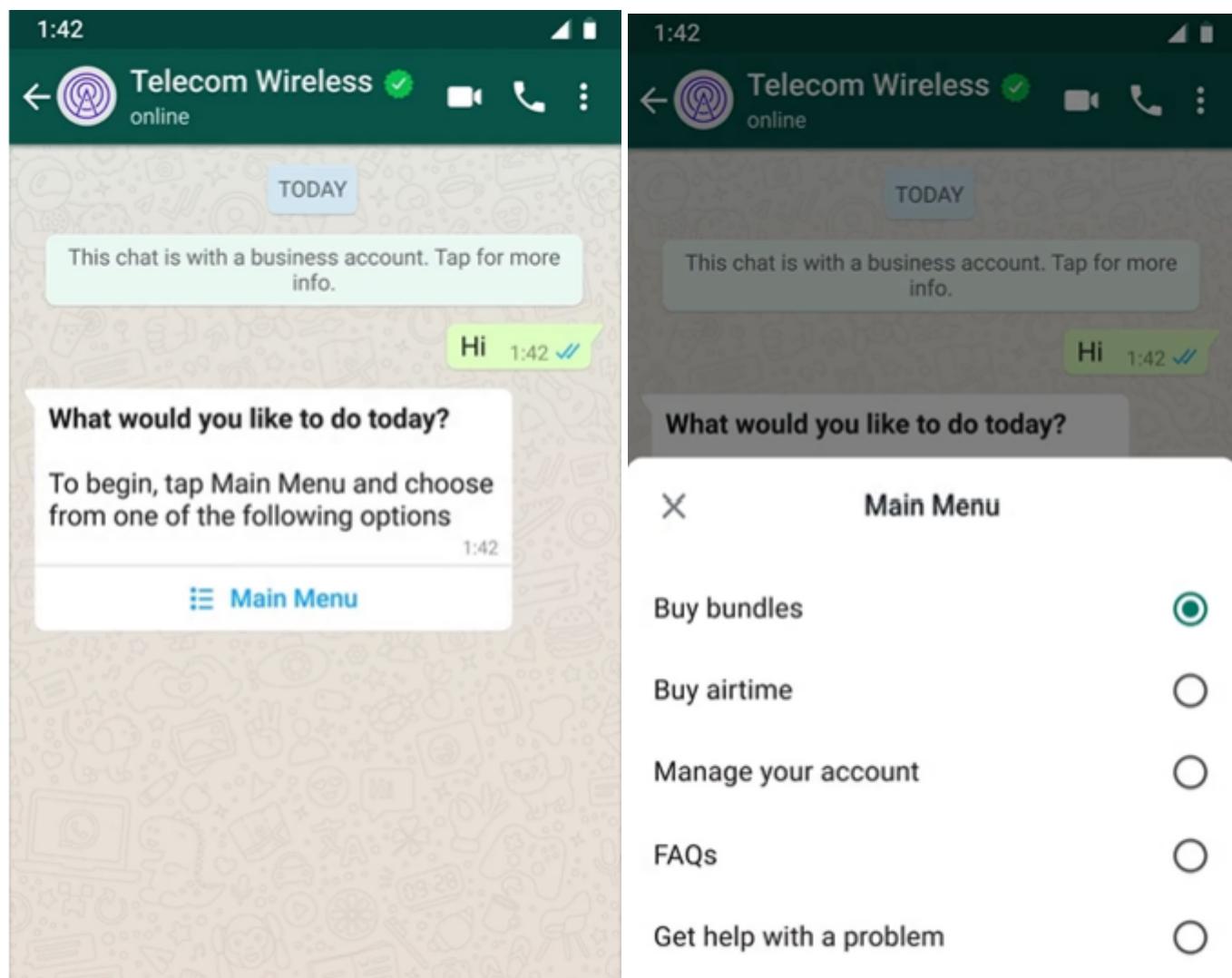
Reply buttons are particularly valuable for ‘personalized’ use cases where a generic response is not adequate.

## Interactive List

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAA040pgZAs98BAGj3nCFG...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageInteractiveList("
```

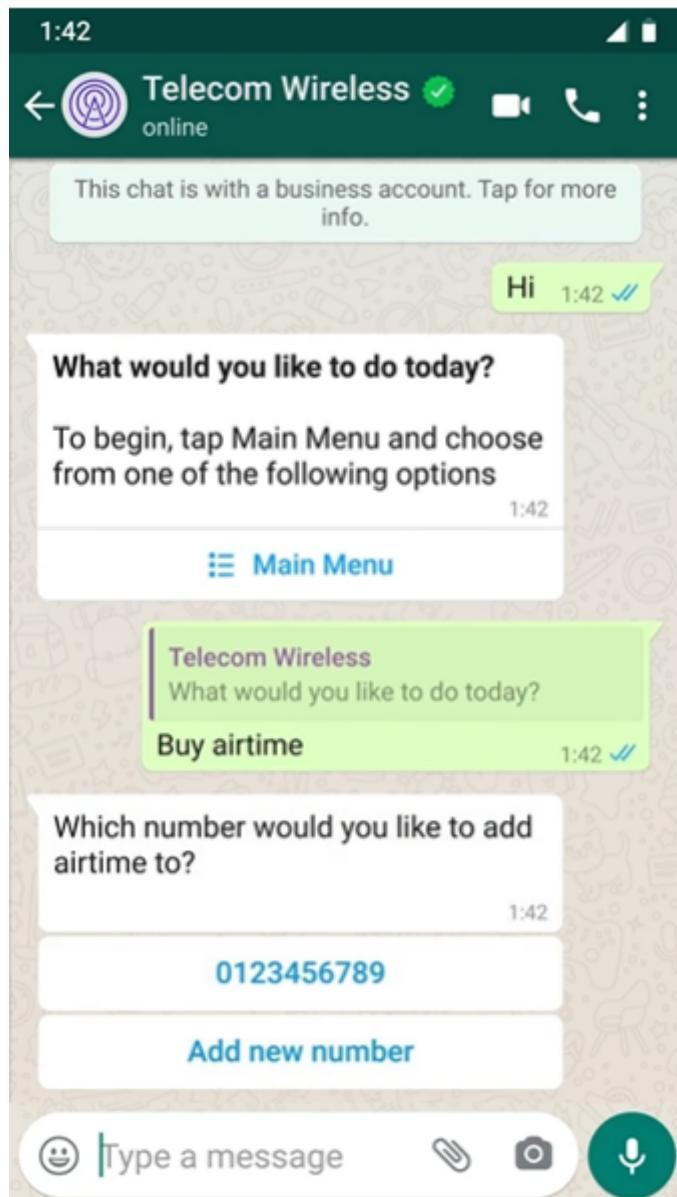
34605889421

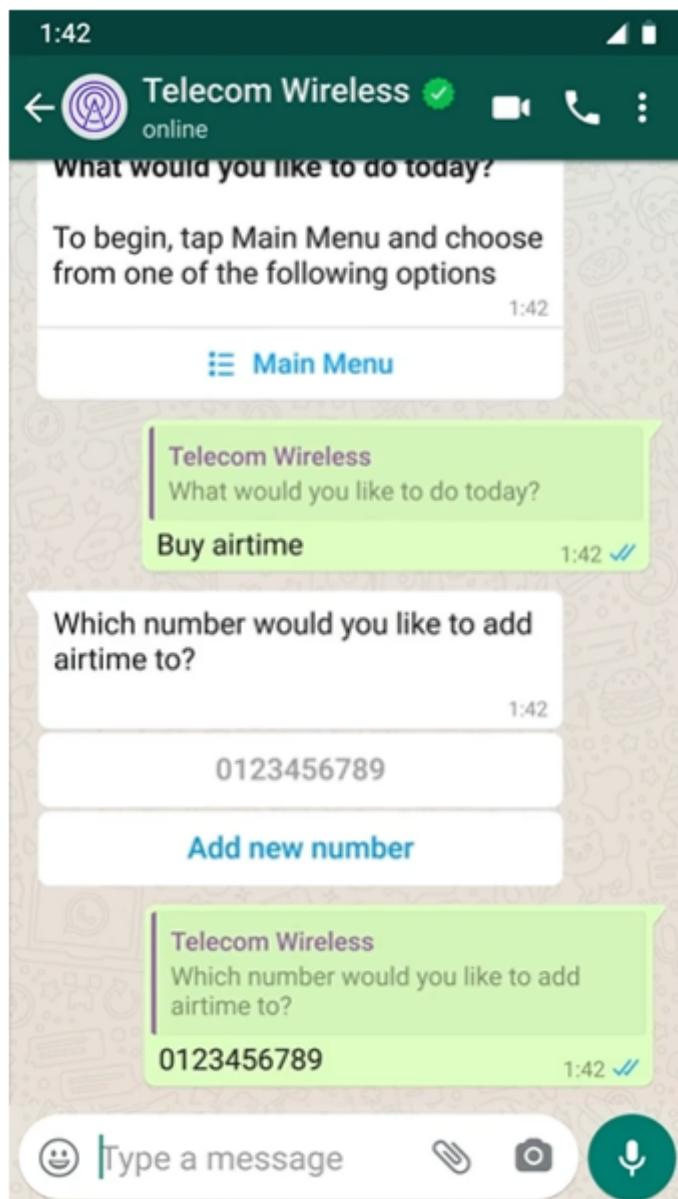
", "What Would you like to do today?", "To begin, Tap Main Menu and choose from of the following options", "", "



## Reply Buttons

```
TsgcWhatsapp_Client oClient = new TsgcWhatsapp_Client();
oClient.WhatsappOptions.PhoneNumberId = "107809351952205";
oClient.WhatsappOptions.Token = "EAA040pgZAs98BAGj3nCFG...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageInteractiveButtons("34605889421", "Select an option", "Which number would you like to add airt
```





# 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 = "EAA040pgZAs98BAGj3nCFG...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.SendMessageTemplate("34605889421", "hello_world", "en_US");
```

# 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 MarkAsF
{
    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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...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 = "EAA040pgZAs98BAGj3nCFG...ZB2t8mmLB2LRXJkte2Y5PMNh2";
oClient.sendFileSticker("34605889421", "c:\Stickers\MySticker.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 = "EAA040pgZAs98BAGj3nCFG...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** allows 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 requires VCRuntime which can be download 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 in the same directory where is your application:

Windows 32	Windows 64
tdjson.dll	tdjson.dll
libcrypto-1_1-x64.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 tdlb 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.

Optionaly, 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:

- authorizationStateWaitTdlbParameters
- 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
<b>Send-TextMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aText:</b> Text of Message. <b>InlineKeyboard:</b> Optional Buttons (only bots).	Sends a Text Message to a Chat
<b>SendRich-TextMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aText:</b> Text of Message. <b>InlineKeyboard:</b> Optional Buttons (only bots).	Sends a Rich Text Message to a Chat. Markdown syntax: <ul style="list-style-type: none"> <li>• Bold: <b>**bold**</b></li> <li>• Italic: <b>_italic_</b></li> <li>• Strike: <b>--strike--</b></li> <li>• Underline: <b>~~underline~~</b></li> <li>• Code: <b>##code##</b></li> </ul>
<b>SendDocumentMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aFilePath:</b> full file path of document <b>InlineKeyboard:</b> Optional Buttons (only bots).	Sends a Document to a Chat.
<b>SendPhotoMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aFilePath:</b> full file path of photo <b>Width:</b> width of photo. <b>Height:</b> width of photo. <b>InlineKeyboard:</b> Optional Buttons (only bots).	Sends a Photo to a Chat.
<b>SendVideoMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aFilePath:</b> full file path of video <b>aWidth:</b> width of video. <b>Height:</b> width of video. <b>aDuration:</b> duration of video in seconds. <b>InlineKeyboard:</b> Optional Buttons (only bots).	Sends a Video to a Chat.
<b>SendInvoiceMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aInvoice:</b> Text of Message. <b>InlineKeyboard:</b> Optional Buttons (only bots).	Sends an Invoice (only available when is a Bot and in Private Channels).
<b>Edit-TextMessage</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aMessageId:</b> Id of Message to modify <b>Text:</b> Text of Message.	Edits the text of a message (or a text of a game message)

	<b>InlineKeyboard:</b> Optional Buttons (only bots). <b>ShowKeyboard:</b> Optional Buttons (only bots).	
<b>AddChat- Member</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aUserId:</b> Identifier of the user. <b>aForwardLimit:</b> 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.
<b>AddChat- Members</b>	<b>aChatId:</b> Id of Chat which message will be sent <b>aUserIds:</b> 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.
<b>GetChat- Member</b>	<b>aChatId:</b> Chat Identifier. <b>aUserId:</b> User Identifier.	Returns information about a single member of a chat.
<b>GetBasic- Group- FullInfo</b>	<b>aGroupId:</b> Basic Group Identifier	Returns full information about a basic group by its identifier.
<b>GetSuper- groupMem- bers</b>	<b>aSuperGroupId:</b> Identifier of the supergroup or channel. <b>aSupergroupMembersFilter:</b> The type of users to return. By default null <b>aOffset:</b> Number of users to skip. <b>aLimit:</b> The maximum number of users be returned; up to 200.	Returns information about members or banned users in a supergroup or channel.
<b>JoinChat- ByIn- viteLink</b>	<b>aLink:</b> 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.
<b>Create- NewSe- cretChat</b>	<b>aUserId:</b> Identifier of the user.	Creates a new secret chat.
<b>CreateNew- Basic- GroupChat</b>	<b>aUserIds:</b> Identifiers of the users to be added to the chat. <b>aTitle:</b> Title of the new basic group	Creates a new basic group
<b>CreateNew- Super- groupChat</b>	<b>aTitle:</b> Title of the new SuperGroup <b>alsChannel:</b> True, if a channel chat should be created. <b>aDescription:</b> Chat Description.	Creates a new supergroup or channel.
<b>CreatePri- vateChat</b>	<b>aUserId:</b> Identifier of the user. <b>aForce:</b> 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
<b>GetChats</b>	<b>aOffsetOrder:</b> Chat order to return chats from <b>aOffsetChatId:</b> Chat identifier to return chats from <b>aLimit:</b> The	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)

	maximum number of chats to be returned.	
<b>GetChat</b>	<b>aChatId:</b> Chat identifier	Returns information about a chat by its identifier
<b>GetChatHistory</b>	<b>aChatId:</b> Chat identifier <b>aFromMessageId:</b> Identifier of the message starting from which history must be fetched; use 0 to get results from the last message. <b>aOffset:</b> Specify 0 to get results from exactly the from_message_id or a negative offset up to 99 to get additionally some newer messages. <b>aLimit:</b> The maximum number of messages to be returned	Returns messages in a chat. The messages are returned in a reverse chronological order
<b> GetUser</b>	<b>aUserId:</b> User Identifier	Returns information about a user by their identifier.
<b>AddProxy-HTTP</b>	<b>aServer:</b> Server name of proxy. <b>aPort:</b> Number of proxy port. <b>aUserName:</b> Username for logging in; may be empty. <b>aPassword:</b> Password for logging in; may be empty. <b>aHTTPOnly:</b> 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.
<b>AddProxyMTProto</b>	<b>aServer:</b> Server name of proxy. <b>aPort:</b> Number of proxy port. <b>aSecret:</b> The proxy's secret in hexadecimal encoding.	Adds a MTProto proxy server for network requests. Can be called before authorization.
<b>AddProxySocks5</b>	<b>aServer:</b> Server name of proxy. <b>aPort:</b> Number of proxy port. <b>aUserName:</b> Username for logging in; may be empty. <b>aPassword:</b> Password for logging in; may be empty.	Adds a Socks5 proxy server for network requests. Can be called before authorization.
<b>EnableProxy</b>	<b>aid:</b> ID of proxy	Enables a proxy. Only one proxy can be enabled at a time. Can be called before authorization.
<b>DisableProxy</b>		Disables the currently enabled proxy. Can be called before authorization.
<b>Remove-Proxy</b>	<b>aid:</b> ID of proxy	Removes a proxy server. Can be called before authorization.
<b>GetProxies</b>		Returns list of proxies that are currently set up. Can be called before authorization.
<b>getChat-SponsoredMessage</b>	<b>aChatId:</b> 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.
<b>ViewMessage</b>	<b>aSponsorChatId:</b> ID of the sponsor Chat <b>aMessageId:</b> 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
<b>Logout</b>		Logs out 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.
- **SenderId:** 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.
- **SenderId:** Sender Identifier (read SenderChat and SenderUser from tlib 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.

# COMPONENTS

- **SenderId**: Sender Identifier (read SenderChat and SenderUser from tlib 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.
- **SenderId**: Sender Identifier (read SenderChat and SenderUser from tlib 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.
- **SenderId**: 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 before send a telegram message, call **GetChat** method 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** must **call GetChat the FIRST TIME** before send a message if you never receive any bot message from this chat. If you close the application and start again, there is no need to call first GetChat because the Chat is already saved on 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 **GetSuperGroupMembers** 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 of 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 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";
}
```

# 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 communicate 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, as higher is the resolution, more bandwidth is required by the connection.

### HTMLDocuments

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

Example: if the server is running on website [www.webrtc.com](http://www.webrtc.com) on port 8443 and the HTMLDocuments.VideoConferencing = /RTCMultiConnection-VideoConferencing.html, the url to access the Video-Conferencing 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
Video-Conferencing	Multi-user (many-to-many) video chat using mesh networking model.
Screen-Sharing	Multi-user (one-to-many) screen sharing using star topology.

### Video- Broad- casting

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

# WebPush

---

[RFC 8030](#)  
[RFC 8291](#)

The WebPush protocol is defined by the **RFC 8030** (Delivery using HTTP Push) and **RFC 8291** (Message Encryption).

Web Push is a **standardized protocol for delivering push notifications to web browsers**. It uses the Push API, which is a standard web API that enables websites to register and receive push messages. The Push API allows a website to send push messages to a user's browser, even when the user is not actively browsing the website.

To use Web Push, a website first needs to **obtain a push subscription from the user's browser**. The subscription consists of a unique endpoint URL and an encryption key. The endpoint URL is a URL that the website can use to send push messages to the user's browser, and the encryption key is used to encrypt and decrypt the push messages.

Once the website has **obtained a push subscription**, it can **send push messages** to the user's browser by making an HTTP request to the endpoint URL. The push message is sent in a special format called the Web Push Protocol Message, which consists of a set of headers and a payload. The headers contain information such as the encryption key and the TTL (time-to-live) of the message, while the payload contains the actual content of the message.

When the **user's browser receives a push message**, it **first decrypts the message** using the encryption key. It then **displays the notification to the user**, along with any additional actions that the user can take, such as dismissing the notification or opening the website.

To ensure the security and privacy of push messages, Web Push uses end-to-end encryption and requires that push subscriptions be obtained over a secure connection (e.g., HTTPS). Additionally, the protocol provides mechanisms for authenticating the sender of a push message and preventing abuse (e.g., by limiting the number of push messages that a website can send to a user).

## Components

There are 2 components which support WebPush:

- **TsgcWSAPIServer\_WebPush:** implements the WebPush protocol on the server side, allowing you to ask users for permission, register subscriptions, send notifications, and more. This component already encapsulates a WebPush client to send notifications.
- **TsgcWebPush\_Client:** implements the WebPush protocol on the client side, allowing you to send notifications to users via desktop and mobile web. This is useful if you already have the keys and endpoint and only want to publish WebPush messages to the subscribed clients.

# TsgcWSAPIServer\_WebPush

**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 to 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.
  - **VAPIDPublicKey:** 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 eSeGeCe\"}");
}
```

# 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 Server and Client size of the sliding window used by LZ77 algorithm (between 8 - 15). As greater is this value, more probably will find and eliminate duplicates but consumes more memory and CPU cycles. 15 is the default value.

## No Context Take Over

By default, previous messages are used to compression and decompression, if messages are similar, this improves the compression ratio. If Enabled, then each message is compressed using only its message data. By default 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 is set to 1.

# Extensions | Deflate-Frame

---

Is a WebSocket protocol extension which allows the compression of frames sent using WebSocket protocol, supported by WebKit browsers like chrome or safari. This extension is supported on Server and Client Components.

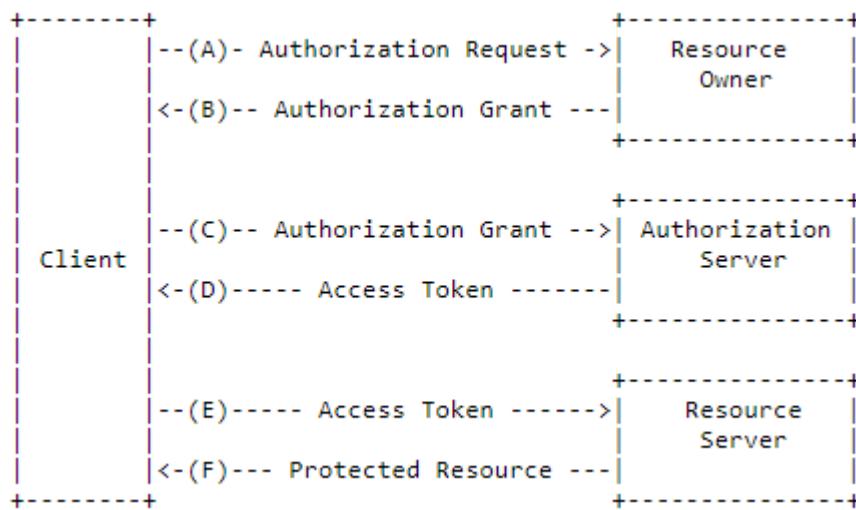
This extension has been deprecated.

# HTTP | OAuth2

OAuth2 allows third-party applications to receive a limited access to an HTTP service which is either on behalf of a resource owner or by allowing a third-party application 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 supports **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).

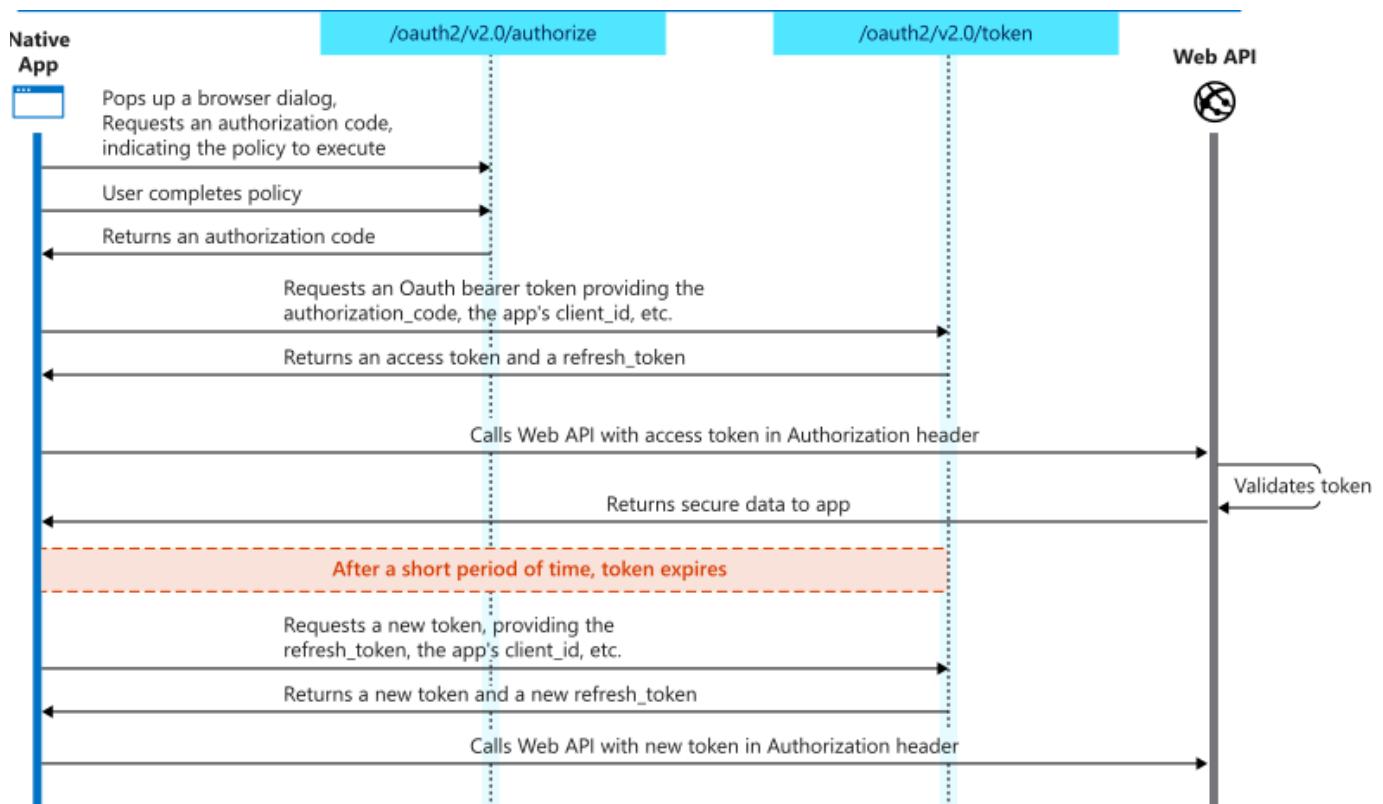
# OAuth2 | TsgcHTTP\_OAuth2\_Client

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

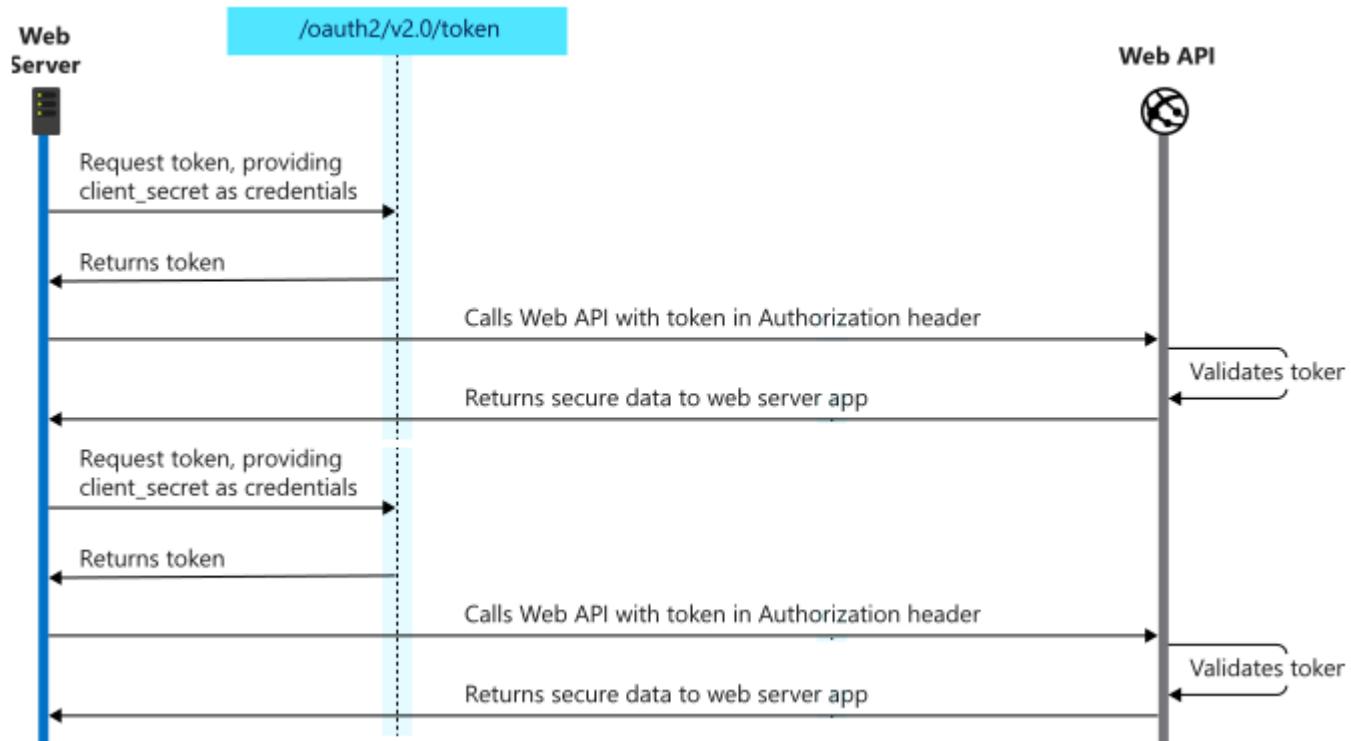
Client supports 2 types of Authorization:

**auth2Code:** It's used to perform authentication and authorization in the majority of application types, including single page applications, web applications, and natively installed applications. The flow enables apps to securely acquire access\_tokens that can be used to access resources secured, as well as refresh tokens to get additional access\_tokens, and ID tokens for the signed in user.



**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.



## 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 customizingd 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).

## 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/>

## 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
- **ClientId:** 180803918307-eqitm20gqfhcs6gjklbrreng022mqqc.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 connection using TLS.

**iohOpenSSL:** uses OpenSSL library and is the default for Indy components. Requires to deploy openssl libraries for win32/win64.

**iohSChannel:** uses Secure Channel which is a security protocol implemented by Microsoft for Windows, doesn't require to deploy openssl libraries. Only works in Windows 32/64 bits.

**OpenSSL\_Options:** 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 LibPath-Custom.

**LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.

**UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):

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

**oslsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.

**oslsSymLinksLoad:** Load SymLinks after trying to load the version libraries.

**oslsSymLinksDontLoad:** don't load the SymLinks.

**SChannel\_Options:** allows you to use a certificate from Windows Certificate Store.

**CertHash:** is the certificate Hash. You can find the certificate Hash running a dir command in powershell.

**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

## 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 used 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("BeforeAccessToken: " + 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 Handled)
{
    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 Handled)
{
    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);
}
```

## 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 = "10803918357-eqjtn20gqfhcs6gjkebbrenh022mqc.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 Handled)
{
    // write your code here
}
```

```
oAuth2->OnAfterAccessToken = OnOAuth2AfterAccessToken;  
oAuth2->Start();
```

# 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.GrantType** to **auth2CodePKCE** to add an extra security.

# 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 OAuth2 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.

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

So if server is listening on port 443 and domain is www.esgece.com, the EndPoints will be:

**Authorization:** <https://www.esgece.com/sgc/oauth2/auth>

**Token:** <https://www.esgece.com/sgc/oauth2/token>

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.

# COMPONENTS

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.

## 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.

# 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.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.

OAuth2

**Authorization Server Options**

Configuration	<input type="button" value="..."/>
Auth. URL	<input type="text" value="https://127.0.0.1/sgc/oauth2/auth"/>
Token URL	<input type="text" value="https://127.0.0.1/sgc/oauth2/token"/>
Scope	<input type="text" value="scope"/>

**Local Server Options**

IP	<input type="text" value="127.0.0.1"/>	Redirect URL	<input type="text"/>
Port	<input type="text" value="8080"/>		

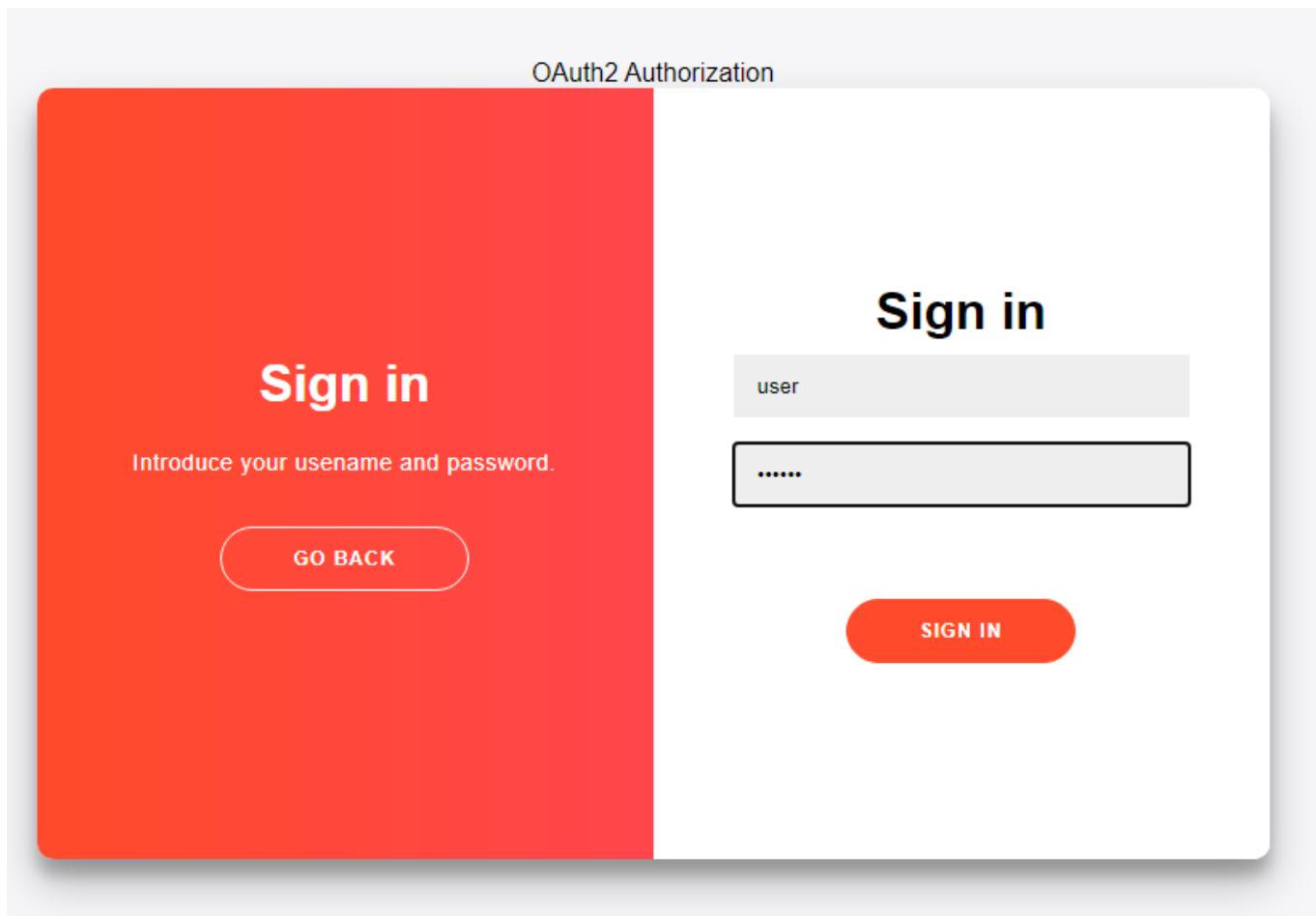
**OAuth2 Options**

ClientId	<input type="text" value="client-id"/>
Secret	<input type="text" value="client-secret"/>
Username	<input type="text"/>
Password	<input type="text"/>

**New Access Token**

Access Token	<input type="text"/>
Token Type	<input type="text"/>
Expires In	<input type="text"/>
Refresh Token	<input type="text"/> Refresh Token
Scope	<input type="text"/>

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**

**Authorization Server Options**

Configuration	Gmail
Auth. URL	https://127.0.0.1/sgc/oauth2/auth
Token URL	https://127.0.0.1/sgc/oauth2/token
Scope	scope

**Local Server Options**

IP	127.0.0.1
Port	8080

**OAuth2 Options**

ClientId	client-id
Secret	client-secret
Username	
Password	

**New Access Token**

Access Token	be4d115a9e6a44f0a859cb303d7f03890d3bf290fc4342c1a08befa550b738bd
Token Type	Bearer
Expires In	3600
Refresh Token	b5ec418745ef4c9e94d3b1a90b34324826152b7edbf040a6a55271813aac5849
Scope	scope

After Authorize Code: code=4b387ffb4255412083057f29ca35933a  
state=EB2FD5EB11B346BF9CE14F7974DBEE7

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

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>

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

# 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.

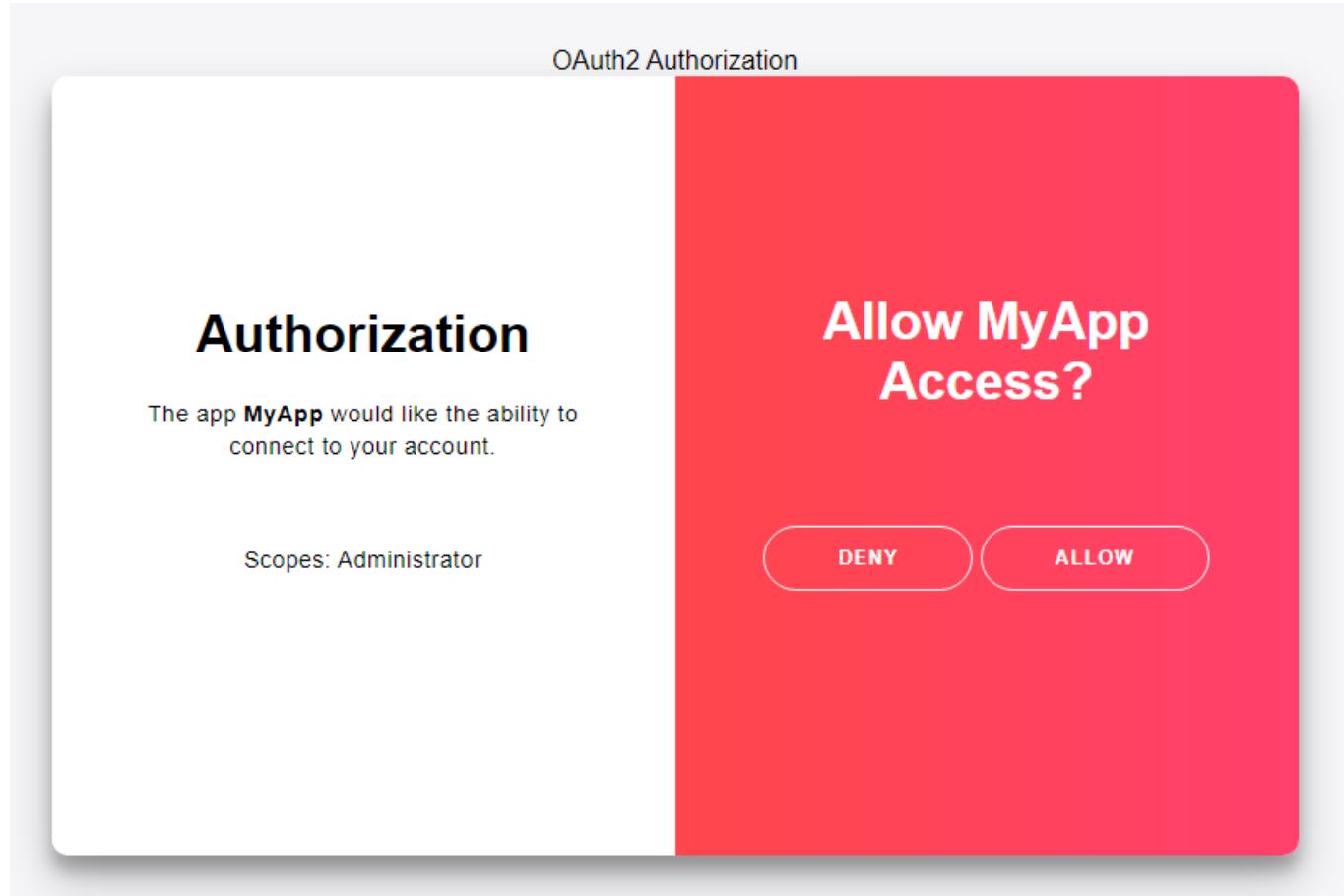
You can save the issued tokens handling the **OAuth2AfterAccessToken** event.

```
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", "12146ce12b0e4813987f2794f768905cef39da6fbd54f6d9b38387489280608", 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;
  }
}
```

# HTTP | JWT

JWT (JSON Web Token) typically consists of a header + payload + signature.

## Header

Contains the metadata information about JWT

- **alg**: is the algorithm used to sign the token
- **typ**: is the type of the token, always JWT

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

You can find more headers but the previous will be always there.

## Payload

The payload contains the claims of the JWT. The standard headers are the following:

- **iss**: is the issuer, the entity who generates and issue the JWT.
- **sub**: is the subject, the entity identified by this token.
- **aud**: is the audience, the target audience for this JWT.
- **exp**: is the expiry, is the timestamp in UNIX format after the token should not be accepted.
- **iat**: is issued at, specifies the date when the token has been 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.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvAG4gRG91IiwiawF0IjoxNTE2MjM5M
```

```
DIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_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 **Pay-load**, 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 LibPath-Custom.

**LibPathCustom:** when LibPath = oslpCustomFolder define here the full path where are located the openSSL libraries.

**UnixSymLinks:** enable or disable the loading of SymLinks under Unix systems (by default is enabled, except under OSX64):

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

**oslsSymLinksLoadFirst:** Load SymLinks and do before trying to load the version libraries.

**oslsSymLinksLoad:** Load SymLinks after trying to load the version libraries.

**oslsSymLinksDontLoad:** don't load the SymLinks.

## 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.esgece.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 = oJWT;
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 | 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(assemResp), // 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();
```

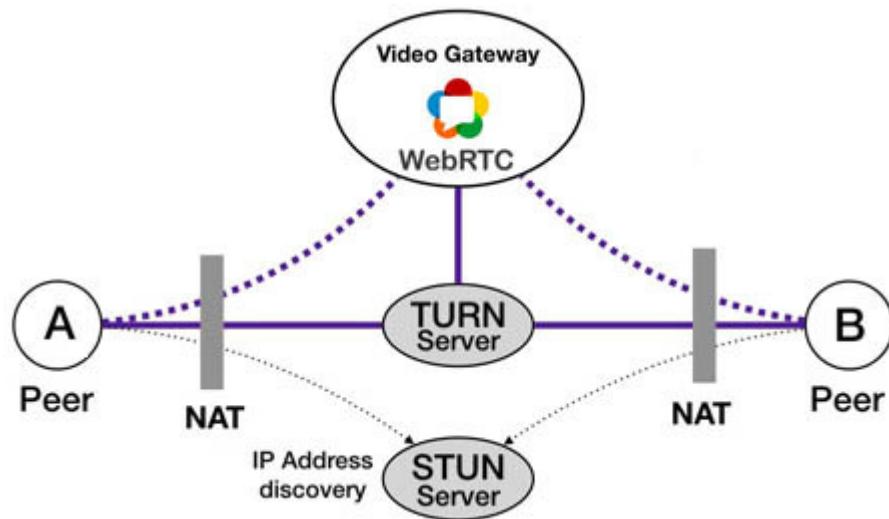
```
if (result.verified) {  
  document.getElementById('Success').textContent = 'User authenticated!';  
} else {  
  document.getElementById('Error').textContent = 'Authentication failed!';  
}
```

# STUN

STUN (Session Traversal Utilities for NAT) it's an IETF protocol used for real-time audio 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 discover its public IP and what type of NAT is.

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 2 types of responses: successful or error, every response has a list of attributes which contains information about binding IP address, error code, reason of error...

## Components

- **TsgcSTUNClient:** it's the client component that implements the STUN protocol and allows you to send binding requests to STUN servers.
- **TsgcSTUNServer:** it's 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 components allows you to use **UDP** and **TCP** as transport, and when used UDP as transport implements a **Re-transmission mechanism** to re-send requests if the response not arrived after a small time.

## Basic usage

Usually stun servers runs 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 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: `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 to the Binding information 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](#)

## 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 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

### OnSTUNBeforeSend

This event is called before the stun client sends a message to the server. You can access to the message properties through the aMessage parameter and modify 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.sgcwebsocckets.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.sgcwebsokcets.com";
oSTUN.Port = 3478;
oSTUN.STUNOptions.Authentication.Credentials = TsgcStunCredentials.stauLongTermCredential;
oSTUN.STUNOptions.Authentication.Username = "user_name";
oSTUN.STUNOptions.Authentication.Password = "secret";
oSTUN.SendRequest();
```

If server requires long-term credentials and the credentials sent by the client are wrong, the will receive a 401 Unauthorized error as a response in the **OnSTUNResponseError** event.

# 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 runs 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 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.

**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

### 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 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 there's no special effort needed to be done 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](#): it's the client component that implements the TURN protocol and allows you to Allocate, create permissions, Send Indications... to TURN Server.
- [TsgcTURNServer](#): it's 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 runs 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 `OnTURNChannelData`. 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 of 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 fo 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 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.

**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-sent 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.

### OnTURNAccAllocate

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 use 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.SendIndication("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 runs 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 STUN 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 of 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 create 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 remove 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 to 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 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 TsgcWSHTTPResponseInfo ResponseInfo)
{
    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 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 = TwstTLSVersions.tlsUndefined;
        break;
    case 1:
        client.TLSOptions.Version = TwstTLSVersions.tls1_0;
        break;
    case 2:
        client.TLSOptions.Version = TwstTLSVersions.tls1_1;
        break;
    case 3:
        client.TLSOptions.Version = TwstTLSVersions.tls1_2;
        break;
    case 4:
        client.TLSOptions.Version = TwstTLSVersions.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 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: // esegce.com
        txtHost.Text = "www.esegce.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.fluxx.io
        txtHost.Text = "mqtt.fluxx.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, TsgcWSMQTTConnectProperties Properties)
{
    DoLog("#MQTT Connect");
    chkTLS.Enabled = false;
    chkCompressed.Enabled = false;
    if (FMQTTSubscribeTopic != "")
    {
        mqtt.Subscribe(FMQTTSubscribeTopic);
        FMQTTSubscribeTopic = "";
    }
}

private void OnMQTTPublishEvent(TsgcWSConnection Connection, string Topic, string Text, TsgcWSMQTTPublishProperties Properties)
{
    DoLog(Topic + ": " + Text);
}

private void OnMQTTSubscribeEvent(TsgcWSConnection Connection, int PacketIdentifier, TsgcWSMQTTSubACKS Codes, TsgcWSMQTTSubscribeProperties Properties)
{
    DoLog("#Subscribe: " + PacketIdentifier.ToString());
}

private void OnMQTTDisconnectEvent(TsgcWSConnection Connection, int ReasonCode, string ReasonName, TsgcWSMQTTDisconnectProperties Properties)
{
    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 show how 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/esegcece.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() {
</code></pre>

```

```

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-left">

</div><!-- /header -->

<div data-role="content">

<h2>Press Start to Get Monitor Data</h2>

<p id="status" class="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" onclick="ws.subscribe('memory');">

<label for="memory">Memory</label>

<input type="checkbox" name="cpu" id="cpu" class="custom" checked="True" onclick="ws.subscribe('cpu');">

<label for="cpu">CPU</label>

<input type="checkbox" name="network" id="network" class="custom" checked="True" onclick="ws.subscribe('network');">

<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 show how 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 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.onloadend = function() {
```

```
        }
        reader.onload = function(e) {
            ws.send("uploadfile:" + file.name);
            rawData = e.target.result;
            ws.send(rawData);
                document.getElementById('status').innerHTML = "File Uploaded";
                document.getElementById('status').className = "online";
            }
        reader.readAsArrayBuffer(file);
    }
</script>
```

# Demos | Server Authentication

---

This demo show how 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 Authenticated)
{
    if ((User == "user") && (Password == "1234"))
    {
        Authenticated = true;
    }
}
```

# Demos | KendoUI\_Grid

This demo show how 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 show how Server Sent Events works 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 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 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.RommLink 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 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 attempts 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 ChatId 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 Message)
{
    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);
}
```

# 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](#)

# 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 for 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](#)

# 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

---

- Add Telegram Proxy [222](#)
- ALPN [49](#)
- API Binance [162, 176, 181](#)
- API Binance Futures [176, 181](#)
- API Binance Futures Trade [181](#)
- API SocketIO [182](#)
- API Telegram [207](#)
- APIs [162, 169, 172, 176, 181, 182, 207](#)
- Authentication [35, 81, 104](#)
- Binance Connect [169](#)
- Binance Get Market Data [171](#)
- Binance Private Requests Time [175](#)
- Binance Private REST API [172](#)
- Binance Subscribe [170](#)
- Binance Trade Spot [173](#)
- Binary Message [78](#)
- Bindings [43, 95](#)
- Bot [218](#)
- Broadcast [42](#)
- Build [19](#)
- Certificates OpenSSL [73](#)
- Certificates SChannel [74](#)
- Channels [42, 170, 284](#)
- Client [64, 66, 67, 76, 77, 78, 81, 83, 84, 85, 86, 110, 137, 139, 140, 160, 270, 271, 281, 282, 283, 284, 292, 294, 295, 297, 302, 312](#)
- Client Authentication [81](#)
- Client Chat [292](#)
- Client Close Connection [66](#)
- Client Exceptions [83](#)
- Client Keep Connection Open [67](#)
- Client MQTT Connect [137](#)
- Client MQTT Sessions [139](#)
- Client MQTT Version [140](#)
- Client Open Connection [64](#)
- Client Proxies [86](#)
- Client Register Protocol [85](#)
- Client Send Binary Message [77](#)
- Client Send Text [76, 78](#)
- Client Send Text Message [76](#)
- Client Snapshots [302](#)
- Client SocketIO [297](#)
- Client WebSocket HandShake [84](#)
- Clients [139, 160, 292, 295](#)
- Send Files [160](#)
- Compression [46](#)
- Connect Mosquitto [138](#)
- Connect Secure Server [72](#)
- Connect TCP Server [69](#)
- Connect WebSocket Server [63](#)
- Connections TIME\_WAIT [70](#)
- Deflate-Frame [232](#)
- Dropped Disconnections [68](#)
- Editions [12](#)
- Extensions [230](#)
- Files [44, 116, 159, 160, 161, 303](#)
- Flash [47](#)
- Flow [18](#)
  - Threading [18](#)
- Forward HTTP Requests [50](#)
- Found [218](#)
- Fragmented Messages [56](#)
- HeartBeat [38](#)
- HTTP [41, 50, 115, 116](#)
- HTTP Dispatch Files [116](#)
- HTTP/2 [117, 118, 120, 315](#)
- HTTP/2 Alternate Service [120](#)
- HTTP/2 Server Push [118](#)
- In HTML [249](#)
- Installation [13](#)
- Introduction [10](#)
- IOCP [48](#)
- JSON [316](#)
- JWT [255, 322](#)
- KendoUI\_Grid [306](#)
- License [325](#)
- Logs [40](#)
- MQTT [137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 295, 319](#)
- MQTT Clear Retained Messages [147](#)
- MQTT Publish [141, 144, 146](#)
- MQTT Publish Message [144](#)

MQTT Publish Subscribe 141  
MQTT Receive Messages 145  
MQTT Subscribe 143  
MQTT Topics 142  
OAuth2 233, 245, 249, 250, 251, 252, 253, 254, 321  
OAuth2 Customize Sign 249  
OAuth2 None Authenticate URLs 254  
OAuth2 Recover Access Tokens 252  
OAuth2 Register Apps 251  
OpenSSL 20, 23, 25, 73  
OpenSSL OSX 25  
OpenSSL Windows 23  
Overview 14  
PerMessage-Deflate 231  
Post Big Files 44  
Protocol AppRTC 148  
Protocol Files 153  
Protocol MQTT 129  
Protocol WebRTC 150, 152  
Protocol WebRTC Javascript 152  
Protocols 85, 125, 126, 129, 148, 150, 152, 153  
Protocols Javascript 126  
Proxy 86, 222  
QuickStart WebSockets 16  
Receive Binary Messages 80, 109  
Receive Text Messages 79, 108  
Register 223  
Register Telegram User 223  
RTCMultiConnection 224  
Secure Connections 37  
Send Big Files 161  
Send Files 159, 160  
    Clients 160  
    Server 159  
Send Files To Clients 160  
Send Files To Server 159  
Send Telegram Message Bold 217  
Send Telegram Message With Buttons 215, 216  
Send Telegram Message With Inline Buttons 215  
Server 94, 95, 96, 97, 98, 101, 102, 103, 104, 106, 107, 108, 109, 110, 115, 117, 138, 159, 245, 250, 253, 274, 275, 288, 289, 290, 298, 301, 305, 309, 310  
    Send Files 159  
Server AppRTC 310  
Server Authentication 104, 253, 305  
Server Bindings 95  
Server Chat 290  
Server Close Connection 103  
Server Endpoints 250  
Server Example 245  
Server Keep Active 97  
Server Keep Connections Alive 101  
Server Monitor 298  
Server Plain TCP 102  
Server Read Headers 110  
Server Requests 115  
Server Send Binary Message 107  
Server Send Text Message 106  
Server Snapshots 301  
Server SSL 98  
Server Start 94  
Server Startup Shutdown 96  
Server-Sent Events 54, 320  
ServerSentEvents 308  
Service 120  
STUN 266, 270, 271, 274, 275, 323  
STUN Client Long Term Credentials 271  
STUN Client UDP Retransmissions 270  
STUN Server Alternate Server 275  
STUN Server Long Term Credentials 274  
SubProtocol 52  
TCP Connections 51  
Telegram Chat 218  
Telegram Client 312  
Telegram Get SuperGroup Members 221  
Threading 18  
    Flow 18  
Throttle 53  
TsgcHTTP\_JWT\_Client 257  
TsgcHTTP\_JWT\_Server 260  
TsgcHTTP\_OAuth2\_Client 234  
TsgcHTTP\_OAuth2\_Server 242  
TsgcSTUNClient 267  
TsgcSTUNServer 272  
TsgcTURNClient 277  
TsgcTURNServer 285

- TsgcWebSocketClient [57](#)
- TsgcWebSocketHTTPServer [111](#)
- TsgcWebSocketServer [87](#)
- TsgcWSConnection [123](#)
- TsgcWSMessageFile [158](#)
- TsgcWSPClient\_Files [156](#)
- TsgcWSPClient\_MQTT [131](#)
- TsgcWSPServer\_AppRTC [149](#)
- TsgcWSPServer\_Files [154](#)
- TsgcWSPServer\_WebRTC [151](#)
- TURN [276, 281, 282, 283, 284, 288, 289, 324](#)
- TURN Client Allocate IP Address [281](#)
- TURN Client Create Permissions [282](#)
- TURN Client Send Indication [283](#)
- TURN Server Allocations [289](#)
- TURN Server Long Term Credentials [288](#)
- Upload File [303](#)
- Using DLL [33](#)
- Wait Response [146](#)
- WAMP [317](#)
- WatchDog [39](#)
- Web Browser Test [34](#)
- WebRTC [309, 318](#)
- WebSocket Events [31](#)
- WebSocket Parameters Connection [32](#)
- WebSocket Redirections [71](#)
- WebSockets [16, 31, 32, 63, 71, 84, 169, 170, 314](#)

**Copyright © 2012-2026 eSeGeCe Software**

info@esegece.com

**www.esegece.com**