sgcWebSockets 4.2.3

Table of Contents

IDE Editions 5
Delphi supported IDE 5
CBuilder supported IDE 5
AppMethod supported IDE 5
FreePascal supported IDE 5
HTML supported IDE 6
Trial Version 6
Library 6
Installation 7
Delphi / CBuilder / Lazarus 7
HTML5 Builder 9
Configure Install 11
Indy 11
Intraweb 12
Install Package 13
Install Errors 19
Intraweb package not found 19
Indy Package not found 20
QuickStart 21
VCL Server 21
VCL Client 21
Web Browser Client 21
How To Use 22
Linux Compiler 22
Linux (Lazarus) 23
WebBrowser Test 24
Fast Performance Server 25
Custom Sub-Protocols 26
Client Example Code 26
Server Example Code 27
Implementation 28
Features 31
Authentication 31
Custom Objects 33
Secure Connections 34
Compression 36
Flash 37
Bindings 38
Quality Of Service 39
Queues 41
Fragmented Messages 43
Transactions 44
TCP Connections 46
SubProtocol 47
HTTP 48
Throttle 49
Server-sent Events (Push Notifications) 50
LoadBalancing 53
HeartBeat 54
WatchDog 55
Files 56
Proxy 57
Logs 58
General 59
WebSocket Events 59
WebSocket Parameters Connection 60
Using inside a DLL 61

TsgcWebSocketServer 63
Methods 63
Properties 64
TsgcWebSocketHTTPServer 67
Methods 68
Properties 68
TsgcWebSocketServer_HTTPAPI 70
Properties 71
Methods 71
TsgcWebSocketClient 72
Methods 73
Properties 73
TsgcWebSocketClient_SocketIO 76
Methods 76
Events 77
Properties 77
TsgcWSHTTPWebBrokerBridgeServer 79
TsgcWebSocketClient_WinHTTP 80
Methods 80
Properties 80
TsgcWebSocketLoadBalancerServer 82
TsgcWebSocketProxyServer 83
TsgcIWWebSocketClient 84
Methods 84
Properties 84
Connections 86
TsgcWSConnection 86
Protocols 88
Protocols 88
Protocols Javascript 90
Subprotocols 93
Default 94
Dataset 104
Files 113
Presence 118
MQTT 129
STOMP 132
AppRTC 141
WebRTC 143
WAMP 146
WAMP 2.0 153
APIs 159
WebSocket APIs 159
API 160
API SocketIO 160
API Blockchain 162
API Bitfinex 164
API Pusher 169
API SignalR 173
API Bittrex 175
API Binance 179
API Bitstamp 182
API Huobi 184
API Cex 187
API Bitmex 196
API SignalRCore 199
Extensions 204
Extensions 204
Extensions | PerMessage-Deflate 205
Extensions | Deflate-Frame 206

IoT 207
IoT 207
Amazon 208
IoT Amazon MQTT Client 208
Library 213
Library | Build 214
QuickStart 215
QuickStart | Library | Delphi 215
QuickStart | Library | C# 216
Client 217
Library | Client 217
Library Client | SocketIO 219
Library | Client Protocol SGC 223
Server 226
Library | Server 226
Library | Server Protocol SGC 228
WebSockets 231
JSON 232
JSON-RPC 2.0 233
Parsers 233
WAMP 234
WebRTC 235
MQTT 236
Server-Sent Events 237
License 239

Introduction

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

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

The Internet wasn't built to be all that dynamic. It was conceived to be a collection of HyperText Markup Language (HTML) pages linking to one another to form a conceptual web of information. Over time the static resources increased in number and richer items, such as images, began to be part of the web fabric. Server technologies advanced allowing for dynamic server pages - pages whose content was generated based on a query.

Soon the requirement to have more dynamic web pages lead to the availability of Dynamic HyperText Markup Language (DHTML) all thanks to JavaScript (let's pretend VBScript never existed). Over the following years, we saw cross frame communication in an attempt to avoid page reloads followed by HTTP Polling within frames. 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 therefore Asynchronous JavaScript and XML (AJAX). In turn, AJAX made XHR Long-Polling and XHR Streaming possible. But none of these solutions offered a truly standardised cross-browser solution to real-time bi-directional communication between a server and a client.

Finally, WebSockets represent a standard for bi-directional real-time communication between servers and clients. Firstly in web browsers, but ultimately between any server and any client. The standards first approach means that as developers we can finally create functionality that works consistently across multiple platforms. Connection limitations are no longer a problem since WebSockets represent a single TCP socket connection. Cross-domain communication has been considered from day one and is dealt with within the connection handshake. This means that services such as Pusher can easily use them when offering 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 supersede Comet (HTTP Long-polling/HTTP Streaming) as the solution of choice for true real-time functionality. AJAX should still be used for making short-lived web service calls, and if we eventually see a good uptake in CORS supporting web services, it will get even more useful. WebSockets should now be the go-to standard for real-time functionality since they offer 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 which all but guarantee any web browser can actually establish a WebSocket connection.

sgcWebSockets is a complete package providing access to WebSockets protocol, allowing to create WebSockets Servers, Intraweb Clients or WebSocket Clients in VCL, Firemonkey, Linux and FreePascal applications.

- Fully functional **multithreaded WebSocket server** according to **RFC 6455**.
- Supports **Firemonkey** (**Windows** and **MacOS**).
- Supports **NEXTGEN Compiler** (IOS and Android Support).
- Supports **LINUX Compiler**.
- Supports **Lazarus / FreePascal.**
- Supports **CBuilder.**

- Supports **Chrome**, **Firefox, Safari, Opera and Internet Explorer** (including **iPhone**, **iPad** and **iPod**)
- Supports **C#.NET** using a compiled library (for Windows 32 and 64 bits).
- Supports **Microsoft HTTP Server API** and **IOCP** for high-performance Windows Servers.
- **Multiple Threads** Support
- 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**.
- **Proxy Server** component allowing to Web Browsers to connect to any TCP server.
- **WebBroker Server** which supports **DataSnap, HTTP** and **WebSocket** connections using the same port.
- **Load Balancing** Server.
- Client WebSocket based on WinHTTP API.
- Client WebSocket supports connections through **Socket.IO Servers**.
- **FallBack** support through Adobe **Flash** for old Web Browsers like Internet Explorer from 6 to 9.
- Supports **Server-Sent Events** (Push Notifications) over HTTP Protocol.
- **WatchDog** and **HeartBeat** built-in support.
- **Client WebSocket** supports connections through **HTTP Proxy Servers** and **SOCKS Proxy Servers.**
- **Events** Available: OnConnect, OnDisconnect, OnMessage, OnError, OnHandshake
- **Built-in sub-protocols**: JSON-RPC 2.0, Dataset, Presence, WebRTC, MQTT and WAMP (1.0 and 2.0)
- Client **Built-in API**: **Blockchain**, **Bitfinex**, **Pusher, SignalR, Huobi, CEX, Bitmex and Bittrex**.
- Support for JSON parsers: **Delphi JSON** and **XSuperObject**.
- **Built-in Javascript libraries** to support browser clients.
- **Easy** to setup
- **Javascript Events** for full control
- **Async Events** using Ajax
- **SSL/TLS Support** for Server / Client Components (OpenSSL libraries required).

Main components available are:

- **TsgcWebSocketServer:** Non-visual component, it's used to manage client threaded connections. Supports RFC 6455. Based on Indy library.
- **TsgcWebSocketHTTPServer:** Non-visual component, it's used to manage client threaded connections. Supports RFC 6455. Supports HTTP Requests using a unique port for WebSocket and HTTP Connections. Based on Indy library.
- **TsgcWSHTTPWebBrokerBridgeServer:** Non-visual component, the server which allows combine HTTP, DataSnap and WebSocket requests using the same server and port.
- **TsgcWebSocketLoadBalancerServer:** Non-visual component, it's used to distribute messages across several back-up servers. Based on Indy library.
- **TsgcWebSocketClient:** Non-visual component, used to establish a WebSocket connection with a WebSocket server, based on Indy LIbrary.
- **TsgcWebSocketClient_WinHTTP:** Non-visual component, used to establish a WebSocket connection with a WebSocket server based on WinHTTP API.
- **TsgcWebsocketProxyServer:** Non-visual component, used to translate WebSocket connections to normal TCP connections. Based on Indy library.

- **TsgcIWWebSocketClient:** Non-visual component, used on Intraweb forms to establish a WebSocket connection with a WebSocket Server.

You can use WebSockets too, using sgcWebSockets.dll, modules available:

- Delphi
- C# .NET

Overview

IDE Editions

Delphi supported IDE

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

CBuilder supported IDE

- CBuilder 2010
- CBuilder XE
- CBuilder XE2
- CBuilder XE3
- CBuilder XE4
- CBuilder XE5
- CBuilder XE6
- CBuilder XE7
- CBuilder XE8
- CBuilder 10 Seattle
- CBuilder 10.1 Berlin
- CBuilder 10.2 Tokyo
- CBuilder 10.3 Rio

AppMethod supported IDE

- AppMethod 2014

FreePascal supported IDE

- Lazarus

HTML supported IDE

- HTML5 Builder

Trial Version

Compiled *.dcu files provided with free version are using default Indy and Intraweb version. If you have upgraded any of these packets, probably it won't work or you need to buy full source code version.

Library

Just copy sgcWebSockets.dll to your Application folder or to a common path. Supported languages:

- Delphi
- C# .NET

Installation

Delphi / CBuilder / Lazarus

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

2. From Delphi\CBuilder:

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

All Delphi\CBuilder Versions

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

For specific Delphi version

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

For specific CBuilder version

C++ Builder 2010 : Add the directory {$DIR}\libD2010 to the library path
C++ Builder XE : Add the directory {$DIR}\libDXE to the library path
C++ Builder XE2 : Add the directory {$DIR}\libDXE2\$(Platform) to the library path
C++ Builder XE3 : Add the directory {$DIR}\libDXE3\$(Platform) to the library path
C++ Builder XE4 : Add the directory {$DIR}\libDXE4\$(Platform) to the library path
C++ Builder XE5 : Add the directory {$DIR}\libDXE5\$(Platform) to the library path
C++ Builder XE6 : Add the directory {$DIR}\libDXE6\$(Platform) to the library path
C++ Builder XE7 : Add the directory {$DIR}\libDXE7\$(Platform) to the library path
C++ Builder XE8 : Add the directory {$DIR}\libDXE8\$(Platform) to the library path
C++ Builder 10 : Add the directory {$DIR}\libD10\$(Platform) to the library path

C++ Builder 10.1 : Add the directory {$DIR}\libD10_1\$(Platform) to the library path
C++ Builder 10.2 : Add the directory {$DIR}\libD10_2\$(Platform) to the library path
C++ Builder 10.3 : Add the directory {$DIR}\libD10_3\$(Platform) to the library path

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

For AppMethod

AppMethod : Add the directory {$DIR}\libAppMethod\$(Platform) to the library path

3. From Delphi

Choose

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

packages files for Delphi

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

4. From CBuilder

Choose

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

packages files for CBuilder

sgcWebSocketsC2010.groupproj : C++ Builder 2010
sgcWebSocketsCXE.groupproj : C++ Builder XE
sgcWebSocketsCXE2.groupproj : C++ Builder XE2
sgcWebSocketsCXE3.groupproj : C++ Builder XE3
sgcWebSocketsCXE4.groupproj : C++ Builder XE4
sgcWebSocketsCXE5.groupproj : C++ Builder XE5
sgcWebSocketsCXE6.groupproj : C++ Builder XE6
sgcWebSocketsCXE7.groupproj : C++ Builder XE7

sgcWebSocketsCXE8.groupproj : C++ Builder XE8
sgcWebSocketsC10.groupproj : C++ Builder 10
sgcWebSocketsC10_1.groupproj : C++ Builder 10.1
sgcWebSocketsC10_2.groupproj : C++ Builder 10.2
sgcWebSocketsC10_3.groupproj : C++ Builder 10.3

5. From AppMethod

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

packages files for AppMethod

sgcWebSocketsAppMethod.groupproj : AppMethod

6. From Lazarus

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

Compiled files are located on Lazarus Directory, inside this, there is a Indy directory with latest Indy source version.

Tested with Lazarus 1.6.4 and Indy 10.5.9.4930

7. Demos

All demos are available in subdirectory Demos. Just open the project and run it. Intraweb demos may need to modify some units due to different Intraweb Versions.

HTML5 Builder

1. Copy "esegece" directory into HTML5 Builder's RPCL directory.

C:\Program Files\Embarcadero\HTML5 Builder\5.0\rpcl

2. From HTML5 Builder

go to Home > Packages and press button "Add Package"

Locate your package file (sgcWebSockets.package.php), select it and click Open.

3. If you start a new project you will see a new palette called "SGC - WebSockets", drop a sgcWebSocket component on a Form and configure required properties.

4. Demos

All demos are available in subdirectory Demos. Just open the project and run it.

Install

Configure Install

In the source folder, there is a file called sgcVer.inc which includes all compiler defines for all Delphi, CBuilder and Lazarus IDEs. Here you can customize your configuration for Intraweb, Indy...

For every Delphi version, there is a section where you can configure all compiler defines, an example for Delphi 10.3

```
{$IFDEF VER330} { Delphi  10.3 }
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE}
  {$DEFINE DXE2}
  {$DEFINE DXE3}
  {$DEFINE DXE4}
  {$DEFINE DXE5}
  {$DEFINE DXE6}
  {$DEFINE DXE7}
  {$DEFINE DXE8}
  {$DEFINE D10}
  {$DEFINE D10_1}
  {$DEFINE D10_2}
  {$DEFINE D10_3}
  {$DEFINE INDY10_1}
  {$DEFINE INDY10_2}
  {$DEFINE INDY10_5_5}
  {$DEFINE INDY10_5_7}
  {$DEFINE INDY10_6}
  {$DEFINE INDY10_6_2_5366}

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

Indy

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

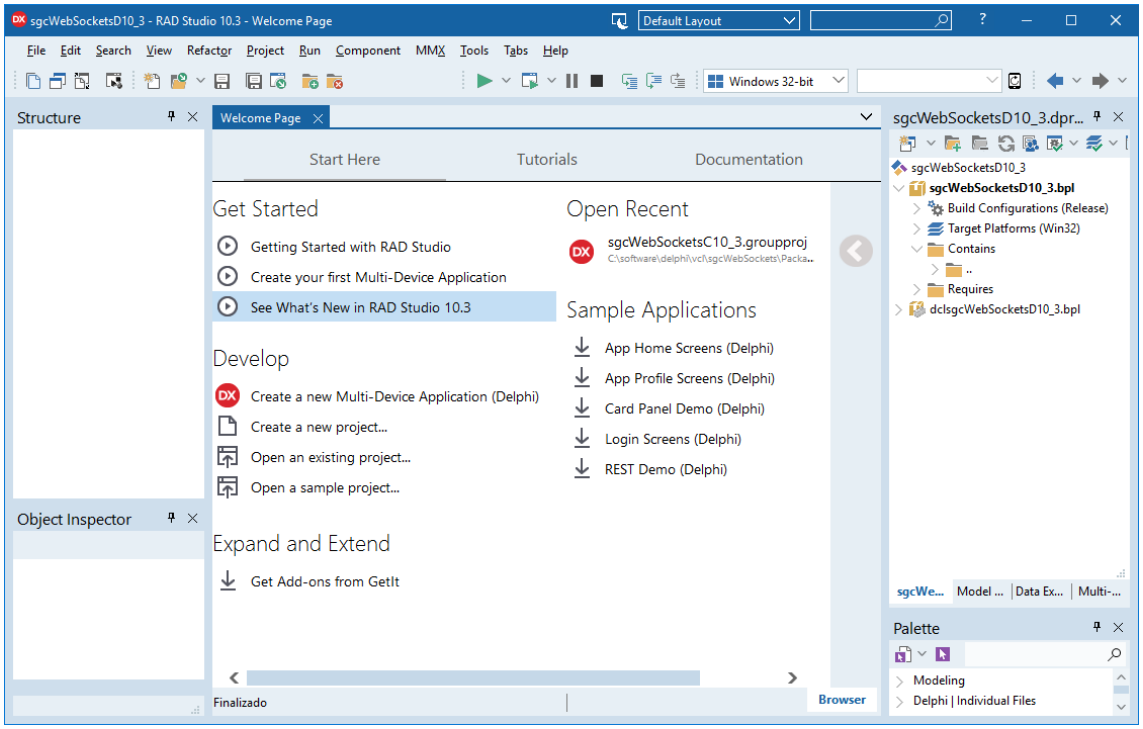
Intraweb

If Intraweb is not installed, just comment compiler defines for Intraweb (those who starts with IW...)

Install Package

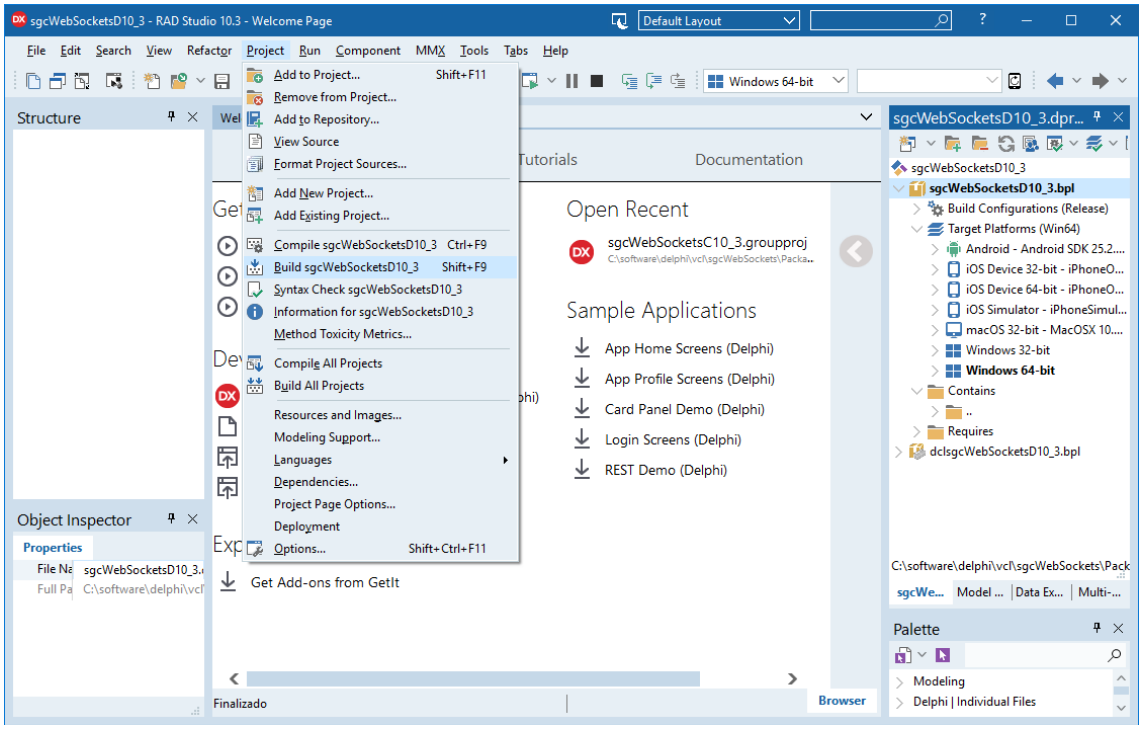
Follow next steps to install sgcWebSockets package, screenshots use Delphi 10.3 version.

1. Open sgcWebSocketsD10_3 group project.

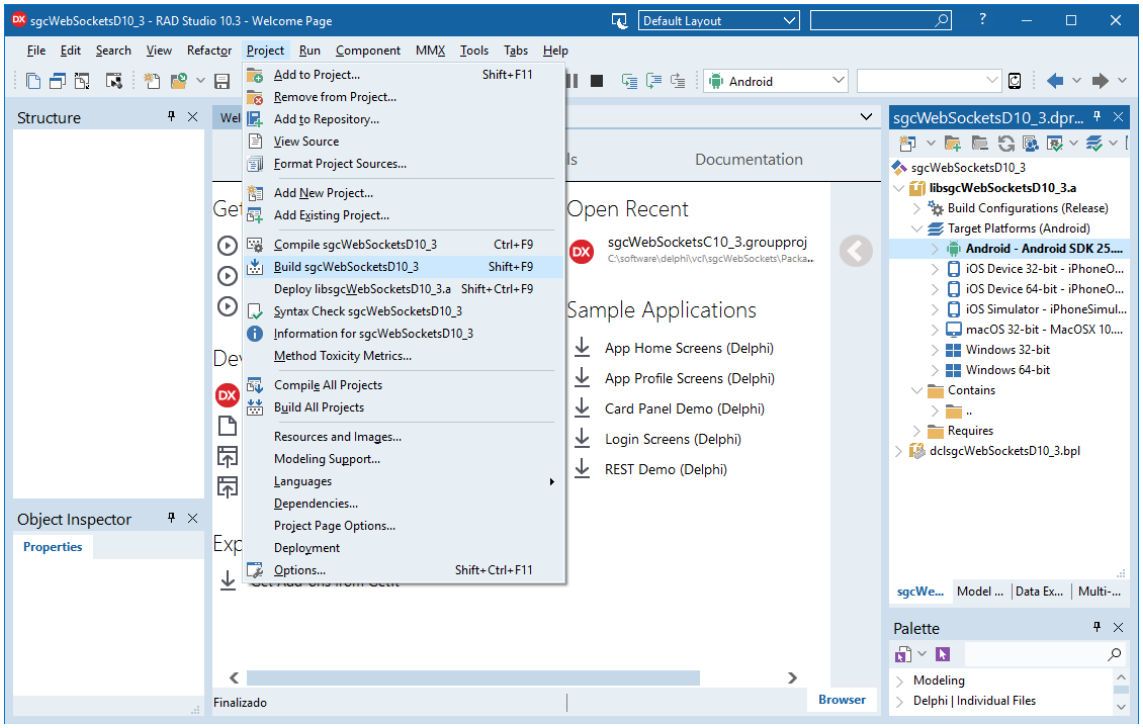


2. Now we must **compile first runtime packages** (name starts with sgcWebSockets). There is one package for every target platform and this depends of Delphi version, so **select target platform one by one and build** every package.

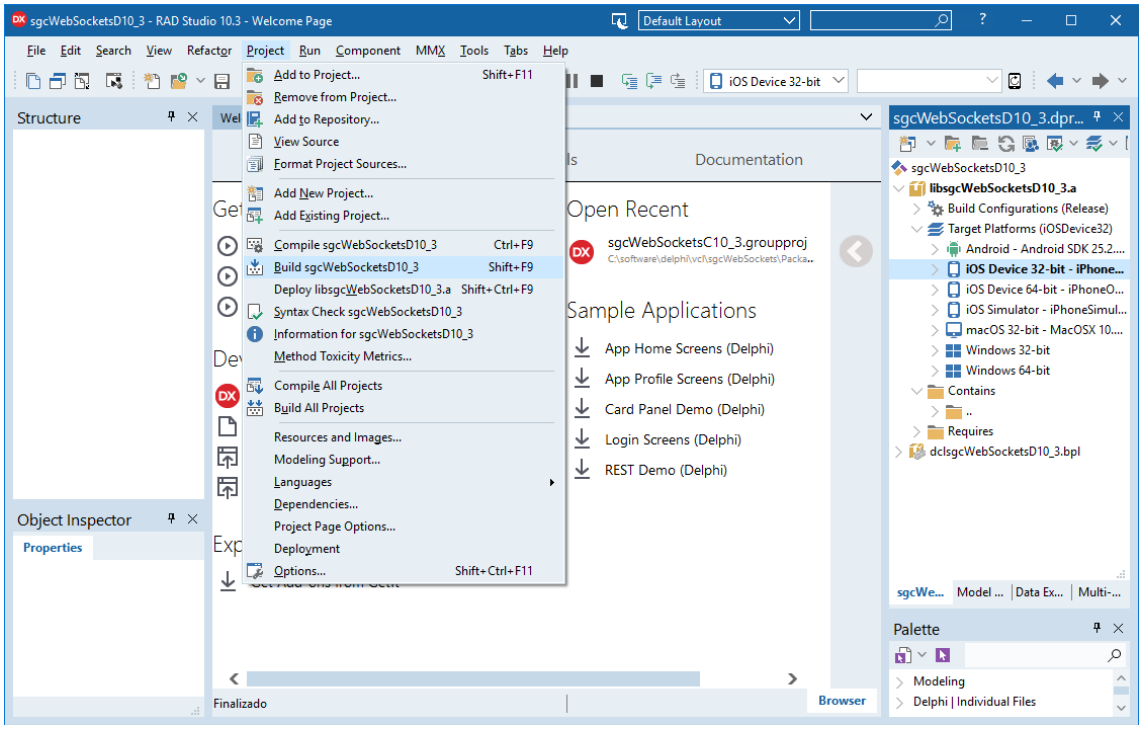
3. Select win64 as Target platform and build package.



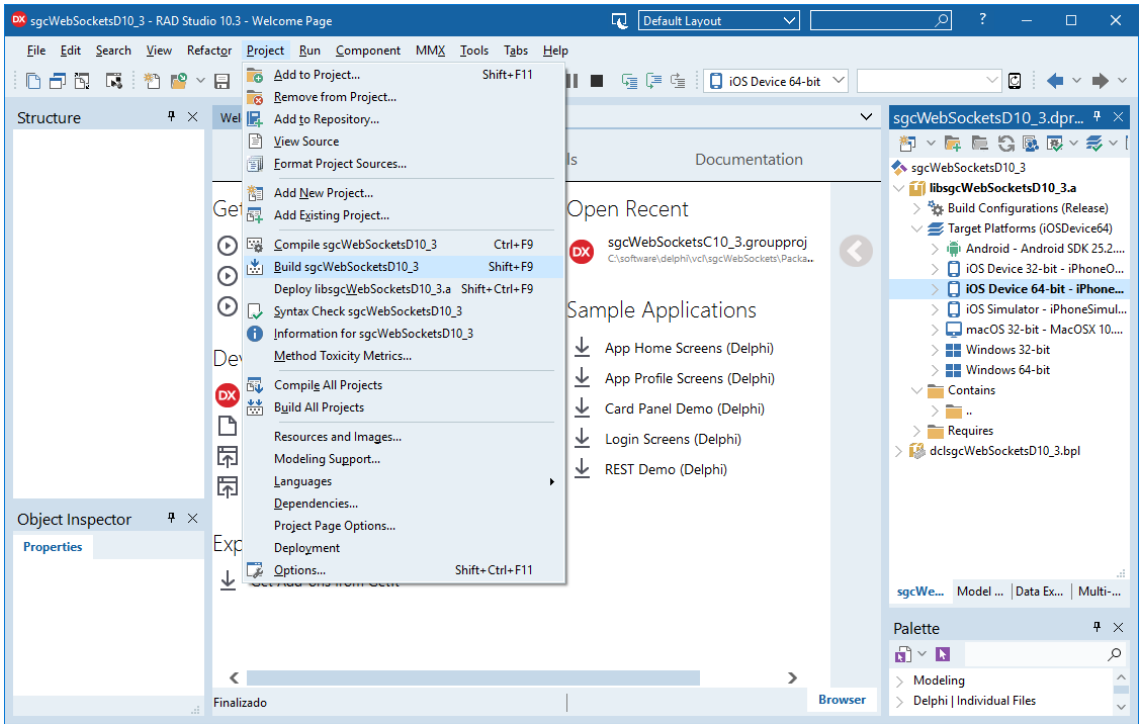
4. Select Android as Target Platform and build package.



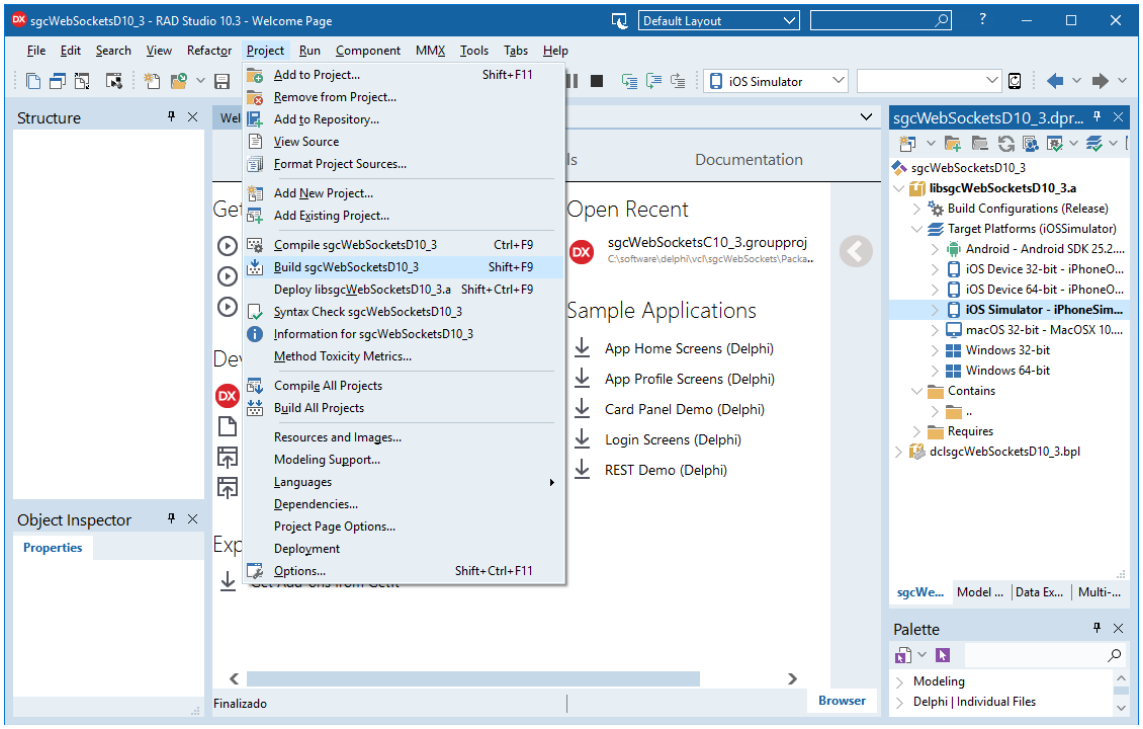
5. Select **iOS Device 32** as Target Platform and build package.



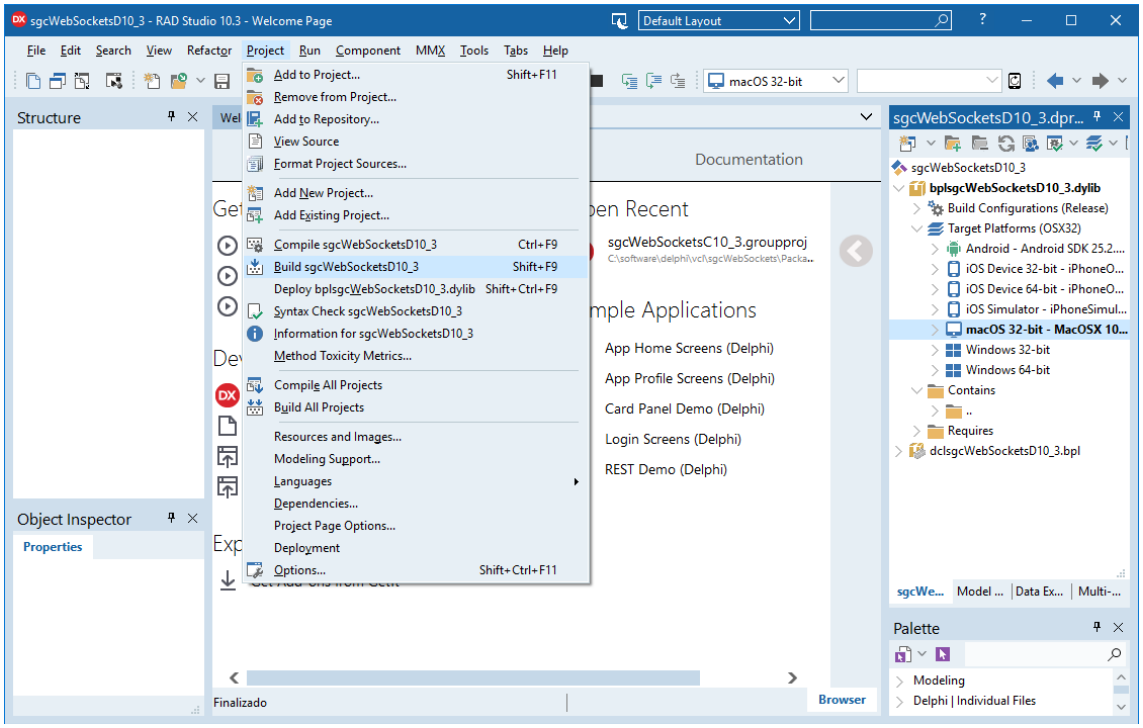
6. Select **iOS Device 64** as Target Platform and build package.



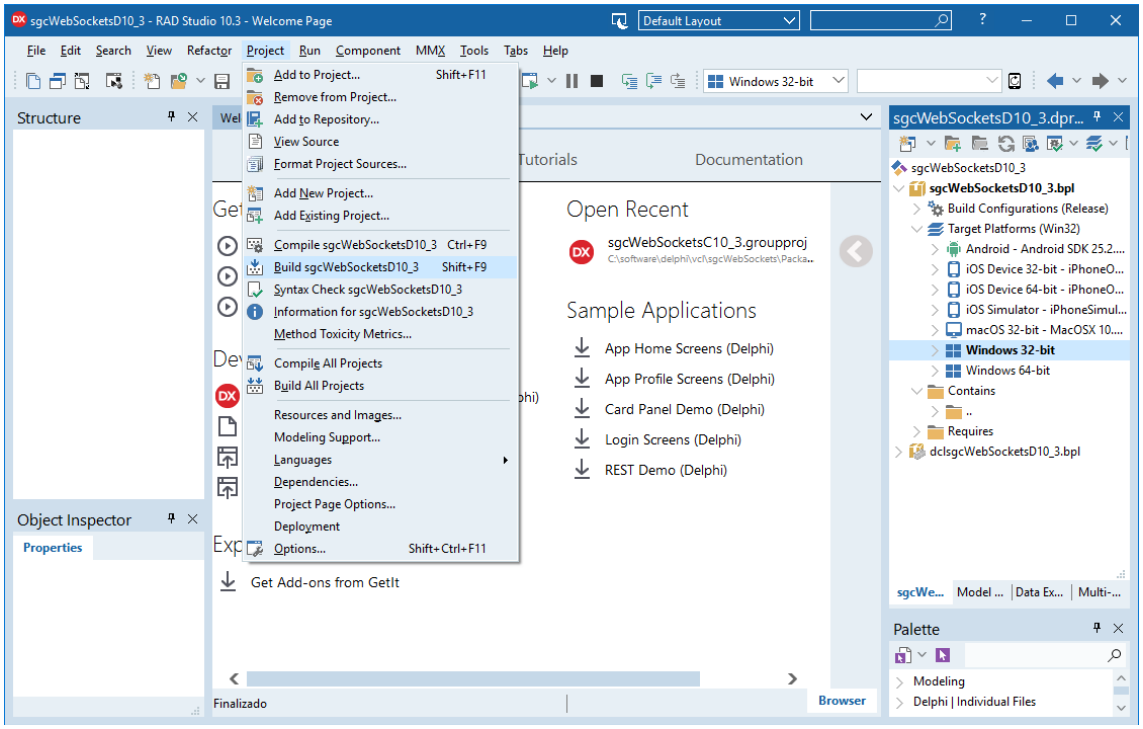
7. Select **iOS Device** Simulator as Target Platform and build package.



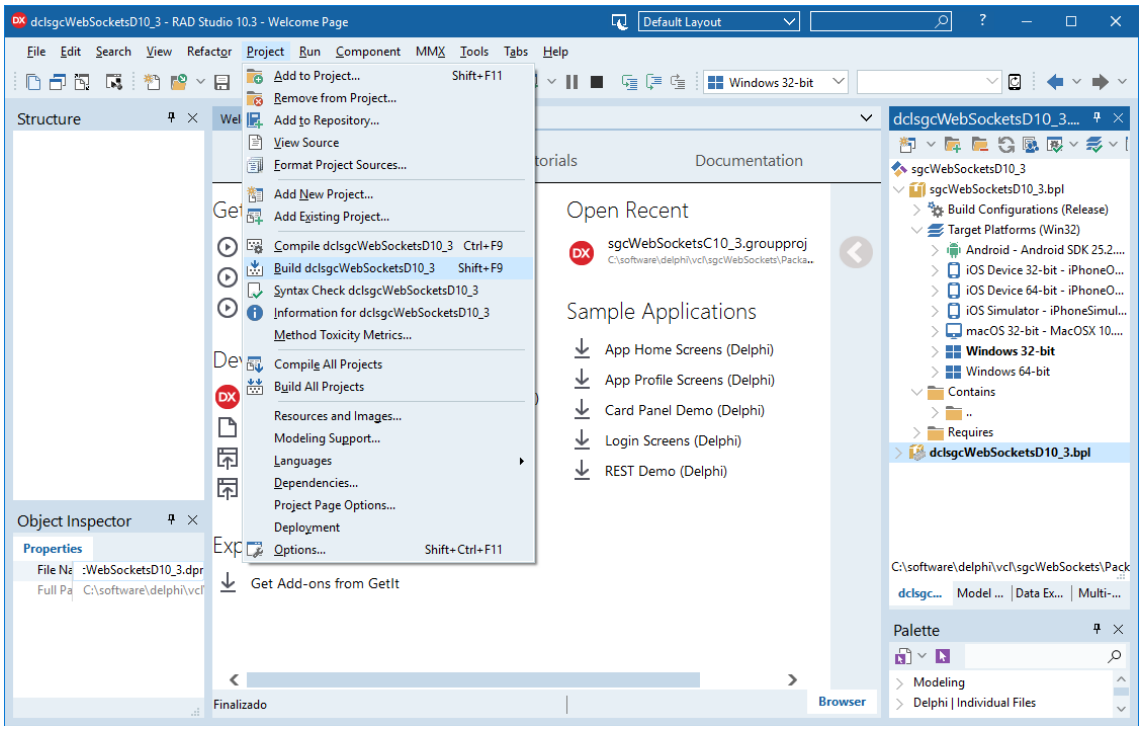
8. Select **MacOS 32** as Target Platform and build package.

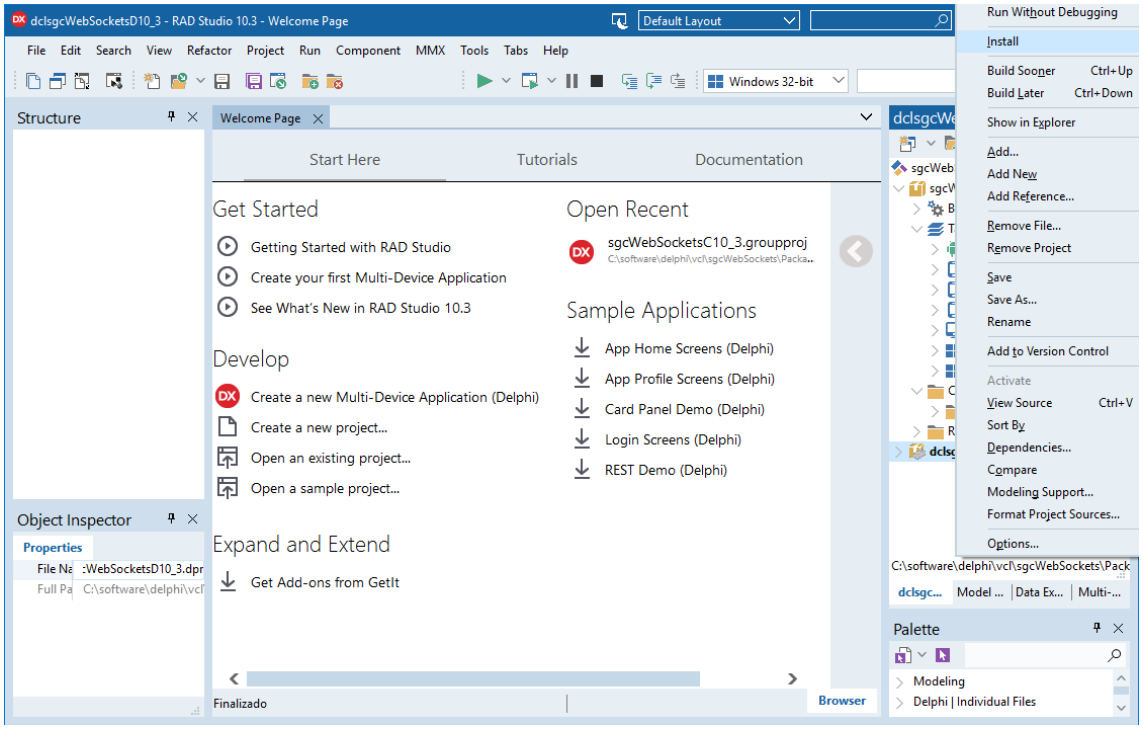


9. Select **Win32** as Target Platform and build package.

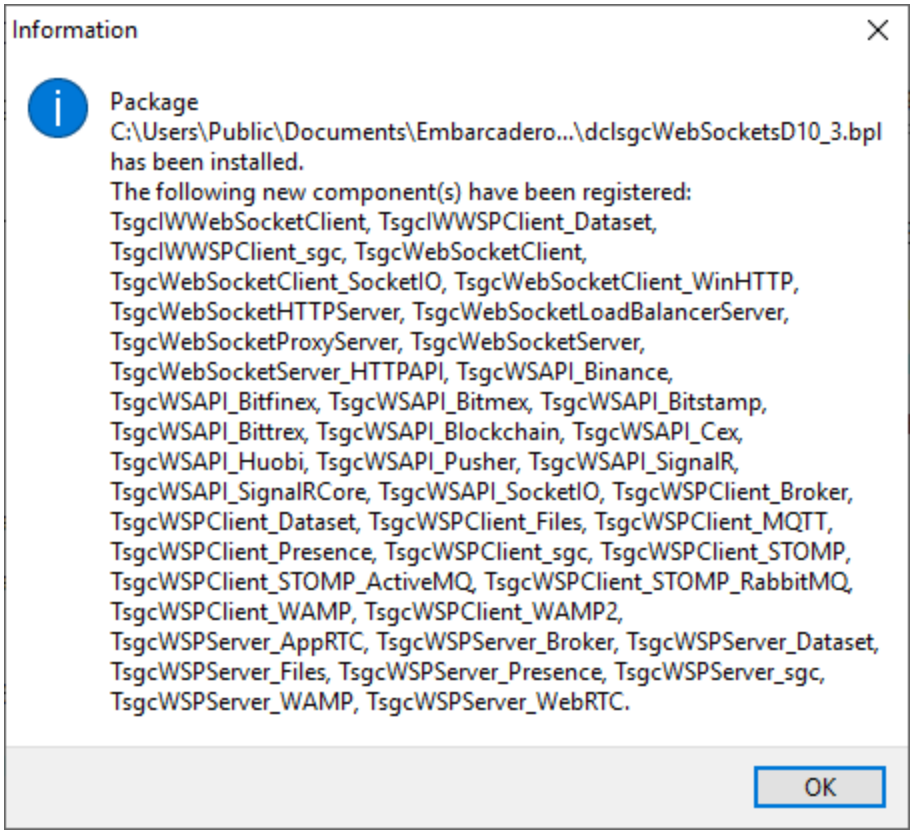


10. Once all runtime packages are compiled, select **design time package** (name starts with dcl) and first **build** and then **install** (design time packages only have Win32 as target platform).





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



Install Errors

Sometimes you may get some errors installing components.

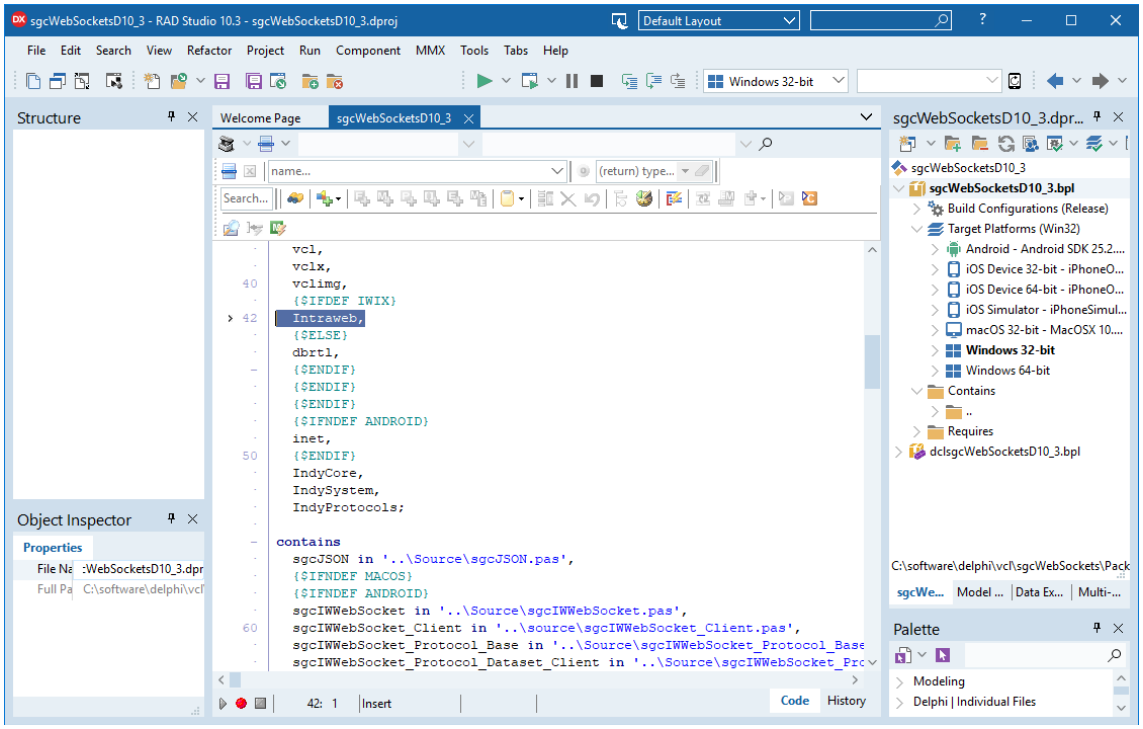
Intraweb package not found

sgcWebsockets is compiled using the **default Intraweb version** provided with Delphi. If you don't have Intraweb installed, you can **modify sgcVer.inc** file (located in Source folder). Search your Delphi version and **comment all compiler defines for Intraweb** (starts with IW).
Example: for Delphi 10.3 comment all compiler defines for Intraweb

```
{$IFDEF VER330} { Delphi  10.3 }
  {$DEFINE D2006}
  {$DEFINE D2007}
  {$DEFINE D2009}
  {$DEFINE D2010}
  {$DEFINE DXE}
  {$DEFINE DXE2}
  {$DEFINE DXE3}
  {$DEFINE DXE4}
  {$DEFINE DXE5}
  {$DEFINE DXE6}
  {$DEFINE DXE7}
  {$DEFINE DXE8}
  {$DEFINE D10}
  {$DEFINE D10_1}
  {$DEFINE D10_2}
  {$DEFINE D10_3}
  {$DEFINE INDY10_1}
  {$DEFINE INDY10_2}
  {$DEFINE INDY10_5_5}
  {$DEFINE INDY10_5_7}
  {$DEFINE INDY10_6}
  {$DEFINE INDY10_6_2_5366}

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

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



Indy Package not found

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

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

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

If you have full source code, then you only must check:

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

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

```
{$DEFINE INDY10_1}
{$DEFINE INDY10_2}
{$DEFINE INDY10_5_5}
{$DEFINE INDY10_5_7}
{$DEFINE INDY10_6}
{$DEFINE INDY10_6_2_5366}
```

QuickStart

QuickStart

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

VCL Server

1. Create a new VCL Forms Application on Delphi

2. Drop a TsgcWebSocketServer in a Form.

3. On Events Tab, Double click OnMessage Event, and type following code:

```
ShowMessage('Message Received From Client: ' + Text);
```

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

```
sgcWebSocketServer1.Active := True;
```

5. Build Project and that's all, you have configured a basic WebSocket Server.

VCL Client

1. Create a new VCL Forms Application on Delphi

2. Drop a TsgcWebSocketClient in 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:

```
sgcWebSocketClient1.Active := True;
```

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

```
sgcWebSocketClient1.WriteData('Hello Server From VCL Client');
```

5. Build Project and that's all, you have configured a basic WebSocket 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 VCL Server Application and press button to start WebSocket Server to listen new connections.

2. Start VCL 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 VCL Client.

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

Linux Compiler

Simple Server example (listening on port 5000).

```
program sgcWebSockets_linux;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils, sgcWebSocket;

var
  oServer: TsgcWebSocketServer;

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

    while oServer.Active do
      Sleep(10);
```

```
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Linux (Lazarus)

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

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

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

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

Fast Performance Server

TsgcWebSocketServer and TsgcWebSocketHTTPServer are based on Indy library, so every connection is handled by a thread, so if you have 1000 concurrent connections, you will have, at least, 1000 threads to handle these connections. When performance is important, you must do some "tweaks" to increase performance and improve server work.

Use the following tips to increase server performance.

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

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

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

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

4. Set in Server component property **Optimizations.ConnectionsFree.Enabled := True**. If this property is enabled, every time there is a disconnection, instead of destroying TsgcWSConnection, the object is stored in a List and every X seconds, all objects stored in this list are destroyed.

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

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

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

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

    {$DEFINE UseReleaseStack}

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

Custom Sub-Protocols

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

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

Client Example Code

```
unit sgcWebSocket_Protocol_Example_Client;

interface

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

uses
  sgcWebSocket_Protocol_Base_Client, Classes,
sgcWebSocket_Classes;

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

implementation

{$IFDEF SGC_PROTOCOLS}

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

procedure
TsgcWSProtocol_Example_Client.DoEventConnect(aConnection:
    TsgcWSConnection);
```

```
begin
  inherited;
  // ... add your own code when client connects to server
end;

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

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

end.
```

Server Example Code

```
unit sgcWebSocket_Protocol_Example_Server;

interface

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

uses
  sgcWebSocket_Protocol_Base_Server, Classes,
sgcWebSocket_Classes;

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

implementation

{$IFDEF SGC_PROTOCOLS}
```

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

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

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

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

end.
```

Implementation

Once your custom subprotocol is implemented, then you only need to assign to your Client or Server websocket component. Example:

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


procedure InitalizeServer;
var
  oServer: TsgcWebSocketServer;
  oProtocol: TsgcWSProtocol_Example_Server;
begin
```

```
      oClient := TsgcWebSocketServer.Create(nil);
      oProtocol := TsgcWSProtocol_Example_Server.Create(nil);
      oProtocol.Server := oServer;
    end;
```

Topics

Features

Authentication

Supported by

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

WebSockets Specification doesn't have any authentication method and Web Browsers implementation don't allow to send 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:

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

Custom Objects

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

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

You can create a new class called MyClass and create some properties, example:

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

Then, when a new client connects, OnConnect Event, create a new TMyClass and Assign to Data:

```
procedure WSServerConnect(Connection: TsgcWSConnection);
begin
  Connection.Data := TMyClass.Create;
end;
```

Every time a new message is received by the server, you can access your custom object using Connection.Data property.

```
procedure WSServerMessage(Connection: TsgcWSConnection; const
    Text: string);
begin
  if TMyClass(Connection.Data).Registered then
    DoSomeStuff;
end;
```

When a connection is closed, you may free your object:

```
procedure TfrmServerChat.WSServerDisconnect(Connection:
TsgcWSConnection;
  Code: Integer);
var
  oMyClass: TMyClass;
begin
  oMyClass := TMyClass(Connection.Data);
  if Assigned(oMyClass) then
  begin
    oMyClass.Free
    Connection.Data := nil;
  end;
end;
```

Secure Connections

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO
Web Browsers

SSL support is based on Indy implementation, so you need to deploy openssl libraries in order to use this feature. In Compiled Demos file, there is a directory called **Third-Parties/ openssl**, where you can find the libraries need for every Delphi Version.

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

Events

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

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

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

2. OnSSLAfterCreateHandler: if no custom SSL object has been created, it creates by default using OpenSSL handler. You can access to SSL Handler properties and modify if needed:

```
procedure OnSSLAfterCreateHandler(Sender: TObject;
    aType: TwsSSLHandler; aSSLHandler:
TIdServerIOHandlerSSLBase);
begin
     TIdServerIOHandlerSSLOpenSSL(aSSLHandler).SSLOptions.Method
     := sslvSSLv3;
end;
```

Compression

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO
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.

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:

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

Quality Of Service

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
TsgcWSPClient_MQTT
Java script

SGC Default Protocol and MQTT implements a QoS (Quality of Service) for message delivery, there are 3 different types:

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

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

Level 2: "Exactly once", where 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.

Level 0

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

The table below shows the QoS level 0 protocol flow.

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

Level 1

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

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

The table below shows the QoS level 1 protocol flow.

Client	Message and direction	Server
QoS = 1	PUBLISH	**Actions:**

Message ID = x **Action:** Store message	---------->	• Store message • Publish a message to subscribers • Delete message
Action: Discard message	ACKNOWLEDGEMENT <----------	

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

Level 2

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

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

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

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

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

Queues

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

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

Level 0: Messages are not queued on Server

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

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

Level 0

The message is not queued by Server

The table below shows the Queue level 0 protocol flow.

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

Level 1

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

The table below shows the Queue level 1 protocol flow.

Client	Message	Server

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

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

Level 2

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

The table below shows the Queue level 2 protocol flow.

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

Fragmented Messages

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

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

```
  procedure OnFragmented(Connection: TsgcWSConnection;  const Data:
TMemoryStream; const OpCode: TOpcode; const Continuation: Boolean);
  begin
    ShowProgress(Data.Size);
    if Continuation = False then
      SaveStream(Data);
  end;
```

Transactions

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

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

StartTransaction

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

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

Commit

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

Client	Message and	Server

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

RollBack

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

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

TCP Connections

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketServer_HTTPAPI
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.

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

Then when a client connects to the server, this connection will be defined as TCP and will use plain TCP protocol instead of WebSockets.

Plain TCP connections don't know if the message is text or binary, so all messages received are handle **OnBinary** event.

SubProtocol

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient

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.

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

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

```
Client.Host := server host;
Client.Port := server.port;
Client.RegisterProtocol('Test 1.0');
Client.Active := True;
```

HTTP

Supported by

TsgcWebSocketHTTPServer

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

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

http://localhost/test.html

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

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

Throttle

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

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.

```
   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' + #10 + 'data: Notification from
server');
```

If you need to specify an event name:

```
   Connection.WriteData('event: notifications' + #10 + 'data:
Notification from server');
```

javascript code to listen "notifications" channel:

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

LoadBalancing

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketLoadBalancerServer

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

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

To enable this feature:

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

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

HeartBeat

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

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.

WatchDog

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

On Server components, automatically restart server after unexpected disconnection.

On Client components, automatically reconnect to server after unexpected disconnection.

Files

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc

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

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

Proxy

Supported by

TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

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

Proxy / Enabled

Once set to True, you can set up:

Host: Proxy server address
Port: Proxy server port
UserName/Password: Authentication to connect to proxy, only if required.

You can configure SOCKS proxies accessing to SOCKS property and set Enable to True.

Logs

Supported by

TsgcWebSocketServer
TsgcWebSocketHTTPServer
TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

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.

General

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
TsgcWebSocketClient_SocketIO
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:

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

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

```
<script src="http://localhost/sgcWebSockets.js"></script>
<script>
  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**.

Components

TsgcWebSocketServer

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

1. Drop a TsgcWebSocketServer 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. If you want, you can handle events:

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

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

OnError: every time there is a WebSocket error (like mal-formed handshake), this event is fired.

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

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

OnHandhake: this event is fired after the handshake is evaluated on the server side.

OnException: every time an exception occurs, this event is fired.

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.

OnStartup: raised after the server has started.

OnShutdown: raised after the server has stopped.

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

```
sgcWebSocketServer1.Active := True
```

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 freezes while trying to connect.

Stop: uses a secondary thread to disconnect from the server, this prevents your application freezes 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).

Bindings: used to manage IP and Ports.

Connections: contains a list of all clients connections.

Count: Connections number count.

Extensions: you can enable compression on messages sent (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 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 try to keeps alive Websocket client connections sending a ping every x seconds.

Interval: number of seconds between each ping.

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

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

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

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

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

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

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

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

Host: Load Balancer Server Host.

Port: Load Balancer Server Port.

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

NotifyEvents: defines which mode to notify WebSocket events.

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

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

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

Options:

FragmentedMessages: allows handling Fragmented Messages

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

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

JavascriptFiles: if enabled, allows to request Javascript Built-in libraries, enabled by default.

RaiseDisconnectExceptions: enabled by default, raises an exception every time there is a disconnection by protocol error.

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

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

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.

SSL: enables secure connections.

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

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.

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, restart the server after unexpected disconnection.

Interval: seconds before reconnects.

Attempts: max number of reconnects, if zero, then unlimited.

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

TsgcWebSocketHTTPServer

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

Follow the next steps 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. If you want, you can handle events:

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

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

OnError: every time there is a WebSocket error (like mal-formed handshake), this event is fired.

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

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

OnHandhake: this event is fired after handshake is evaluated on the server side.

OnCommandGet: this event is fired when HTTP Server receives a GET command requesting a HTML page, an image... Example:

```
AResponseInfo.ContentText :=
'<HTML><HEADER>TEST</HEAD><BODY>Hello!</BODY></HTML>';
```

OnCommandOther: this event is fired when HTTP Server receives a command different of GET.

OnCreateSession: this event is fired when HTTP Server creates a new session.

OnInvalidSession: this event is fired when an HTTP request is using an invalid/expiring session.

OnSessionStart: this event is fired when HTTP Server starts a new session.

OnCommandOther: this event is fired when HTTP Server closes a session.

OnException: this event is fired when HTTP Server throws an exception.

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

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

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

```
sgcWebSocketHTTPServer1.Active := True
```

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 compression on messages sent (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.

TsgcWebSocketServer_HTTPAPI

The HTTP Server API enables applications to communicate over HTTP without using Microsoft Internet Information Server (IIS). Applications can register to receive HTTP requests for particular URLs, receive WebSocket requests, and send WebSocket responses. The HTTP Server API includes SSL support so that applications can exchange data over secure HTTP connections without IIS. It is also designed to work with I/O completion ports.

Follow the next steps to configure this component:

1. Drop a TsgcWebSocketServer_HTTPAPI component in the form

2. Add which URL/s you need register, example: if connections to port 8080 with "/ws/" parameter is WebSocket connection, register the following URL

```
WSServer.Bindings.NewBinding('127.0.0.1', 8080, '/ws/');
```

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

RFC6455: is standard and recommended WebSocket specification.

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

4. If you want, you can handle events:

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

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

OnError: every time there is a WebSocket error (like mal-formed handshake), this event is fired.

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

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

OnHandhake: this event is fired after the handshake is evaluated on the server side.

OnException: this event is fired 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.

OnAsynchronous: every time an asynchronous event has been completed, this event is called.

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

```
sgcWebSocketServer_HTTPAPI1.Active := True
```

Properties

Timeouts: allows overriding default timeouts of HTTP API Server.

EntityBody: the time, in seconds, allowed for the request entity body to arrive.
DrainEntityBody: The time, in seconds, allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection.
RequestQueue: The time, in seconds, allowed for the request to remain in the request queue before the application picks it up.
IdleConnection: The time, in seconds, allowed for an idle connection.
HeaderWait: The time, in seconds, allowed for the HTTP Server API to parse the request header.
MinSendRate: The minimum sends rate, in bytes-per-second, for the response. The default response sends rate is 150 bytes-per-second.

MaxConnections: maximum number of connections (zero means unlimited, value by default).

MaxBandwidth: maximum allowed bandwidth rate in bytes per second (zero means unlimited, value by default).

ThreadPoolSize: by default 32 (maximum allowed is 64), allows setting number of threads of HTTP API Server.

Asynchronous: by default is disabled, if enabled, messages sent don't wait till completed. You can check when asynchronous is completed **OnAsynchronous** event.

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.

TsgcWebSocketClient

TsgcWebSocketClient implements Client VCL WebSocket Component and can connect to a WebSocket Server, it's based on Indy library. Follow the next steps to configure this component:

1. Drop a TsgcWebSocketClient component in the form

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

```
Host := '127.0.0.1';
Port := 80;
TLS := True;
Options.Paramters := '/ws/';
```

3. You can select if you want TLS (secure connection) or not, by default is not Activated.

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

Host: hostname of the proxy server.
Port: port number of the proxy server.
Username: user to authenticate, blank if anonymous.
Password: password to authenticate, blank if anonymous.

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

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

RFC6455: is standard and recommended WebSocket specification.

Hlxie76: always is false

7. If you want, you can handle events

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

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

OnError: every time there is a WebSocket error (like mal-formed handkshake), this event is fired

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

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

OnHandhake: this event is fired when handshake is evaluated on the client side.

OnException: every time an exception occurs, this event is fired.

8. Create a procedure and set property Active := True

```
sgcWebSocketClient1.Active := True
```

Methods

WriteData: sends a message to a WebSocket Server. Could be a String or TStream. 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 freezes while trying to connect.

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

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

Disconnect: try 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 2 types of WebSocket Authentication

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.

Host: IP or DNS name of the server.

HeartBeat: if enabled try to keeps alive WebSocket connection sending a ping every x seconds.

Interval: number of seconds between each ping.

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

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
```

ReadTimeout: max time in milliseconds to read messages.

Port: Port used to connect to the host.

LogFile: if enabled save 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.

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 every time there is a disconnection by protocol error.

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

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

Extensions: you can enable compression on messages sent.

Protocol: if exists, shows the current protocol used

Proxy: here you can define if you want to connect through a HTTP Proxy Server. If you need to connect to SOCKS proxies, just enable SOCKS.Enable property too.

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

Interval: seconds before reconnects.

Attempts: max number of reconnects, if zero, then unlimited.

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

TLS: enables a secure connection.

TsgcWebSocketClient_SocketIO

This component is deprecated, use API SocketIO.

TsgcWebSocketClient_SocketIO inherits all properties and methods from TsgcWebSocketClient and allows connecting to a Socket.IO Server.

Methods

These messages are only supported by ioAPI0:

SendDisconnect: Signals disconnection. If no endpoint is specified, disconnects the entire socket.

Examples:

Disconnect a socket connected to the /test endpoint.

```
SendDisconnect('/test');
```

Disconnect the whole socket

```
SendDisconnect;
```

SendConnect: Only used for multiple sockets. Signals a connection to the endpoint. Once the server receives it, it's echoed back to the client.

Example, if the client is trying to connect to the endpoint /test, a message like this will be delivered:

```
SendConnect('[path] [query]');
```

Example:

```
SendConnect('/test?my=param');
```

To acknowledge the connection, the server echoes back the message. Otherwise, the server might want to respond with an error packet.

SendHeartBeat: Sends a heartbeat. Heartbeats must be sent within the interval negotiated with the server. It's up to the client to decide the padding (for example, if the heartbeat timeout negotiated with the server is 20s, the client might want to send a heartbeat every 15s).

Example:

```
SendHeartBeat;
```

SendTextMessage: A regular message.

Example: send a text message "Hi Folks", with id "fjghs121" to clients connected to EndPoint "/test"

```
SendTextMessage('Hi Folks', 'fjghs121', '/test');
```

SendJSONMessage: A JSON encoded message.

Example: send a JSON encoded message "{"a":"b"}"

```
SendJSONMessage('{"a":"b"}');
```

SendEvent: An event is like a JSON message, but has a mandatory name and args fields. The name is a string and args an array.

The event names: 'message', 'connect', 'disconnect', 'open', 'close', 'error', 'retry', 'reconnect' are reserved, and cannot be used by clients or servers with this message type.

Example: send event "test" with arguments "["1","2","3"]"

```
SendEvent('test', ["1","2","3"]);
```

SendACK: An acknowledgement contains the message id as the message data. If a + sign follows the message id, it's treated as an event message packet.

Example: simple acknowledgement of message id "2"

```
SendACK("2");
```

SendError: For example, if a connection to a sub-socket is unauthorized.

Example: send error "not authorized" with advise "connect with admin user"

```
SendError("not authorized", "connect with admin user");
```

SendNoop: No operation. Used for example to close a poll after the polling duration times out.

Example:

```
SendNoop;
```

Events

These events are only raised if "RawMessages" property is disabled and ioAPI0 is selected.

OnMessageDisconnect

OnMessageConnect

OnMessageHeartBeat

OnMessageText

OnMessageJSON

OnMessageEvent

OnMessageACK

OnMessageError

OnMessageNoop

Properties

RawMessages: if not enabled (which is default) socket.io messages are processed and specific socket.io messages events are raised, if enabled, then socket.io messages are not processed and OnMessage event is raised.

IO_API: specifies SocketIO version:

ioAPI0: supports socket.io 0.* servers (selected by default)

ioAPI1: supports socket.io 1.* servers

IO_CloseTimeout: close timeout received from Socket.io server.

IO_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).

IO_HeartBeatTimeout: HeartBeat timeout received from Socket.io server.

IO_SessionId: SessionId received from Socket.io server.

IO_Base64: if enabled, binary messages are received as base64.

IO_HandShakeCustomURL: allows customizing url to get socket.io session.

IO_Namespace: allows setting a namespace when connects to a server.

TsgcWSHTTPWebBrokerBridgeServer

TsgcWSHTTPWebBrokerBridgeServer use **TIdHttpWebBrokerBridge** as server base and is useful if you want to use a single server for DataSnap, HTTP and WebSocket connections.

TsgcWSHTTPWebBrokerBridgeServer inherits from **TsgcWebSocketHTTPServer**, so you can refer to this server.

Follow next steps to replace TIdHttpWebBrokerBridge for TsgcWSHTTPWebBrokerBridgeServer :

1. Create a new instance of TsgcWSHTTPWebBrokerBridgeServer.

2. Replace all calls to TIdHttpWebBrokerBridge for TsgcWSHTTPWebBrokerBridgeServer.

3. To Handle WebSocket connections just refer to TsgcWebSocketHTTPServer.

TsgcWebSocketClient_WinHTTP

TsgcWebSocketClient implements Client VCL WebSocket Component and can connect to a WebSocket Server, it's based on WinHTTP API and requires Windows 8 or higher. Follow the next steps to configure this component:

1. Drop a TsgcWebSocketClient_WinHTTP component in the form

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

```
Host := '127.0.0.1';
Port := 80;
TLS := True;
Options.Paramters := '/ws/';
```

3. You can select if you want TLS (secure connection) or not, by default is not Activated.

4. If you want, you can handle events

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

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

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

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

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

OnException: every time an exception occurs, this event is fired.

8. Create a procedure and set property Active := True

```
sgcWebSocketClient_WinHTTP1.Active := True
```

Methods

WriteData: sends a message to a WebSocket Server. Could be a String or TStream.

Properties

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

Implements 1 type of WebSocket Authentication

Basic: client open WebSocket connection passing username and password inside the header.

Asynchronous: by default, requests are synchronous, execution of your application stops when you make new requests and resumes when you get a response. If you don't want that requests stop your application, enable this property.

Host: IP or DNS name of the server.

HeartBeat: if enabled try to keeps alive a WebSocket connection sending a ping every x seconds.

Interval: number of seconds between each ping.

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

ReadTimeout: max time in milliseconds to read messages.

Port: Port used to connect to the host.

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.

Parameters: define parameters used on GET.

Origin: customize connection origin.

Protocol: if exists, shows the current protocol used

Proxy: here you can define if you want to connect through an HTTP Proxy Server.

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

Interval: seconds before reconnects.

Attempts: max number of reconnects, if zero, then unlimited.

TLS: enables a secure connection.

TsgcWebSocketLoadBalancerServer

TsgcWebSocketLoadBalancerServer allows distributing messages across a group of servers and distributes clients connections using a random sequence or fewer connections algorithm.

Load Balancer Server, inherits all methods and properties from TsgcWebSocketServer.

Follow the next steps to configure this component:

1. Drop a TsgcWebSocketLoadBalancerServer component in the form

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

3. Configure LoadBalancer properties:

> **3.1** lbRandom: every time a new client request a new connection, it will return a random server.
>
> **3.1** lbConnections: every time a new client request a new connection, it will return server with fewer clients connected.

4. There are serveral events which are specific of LoadBalancerServer:

OnBeforeSendServerBinding: raised before binding is sent to a new client connection.

OnClientConnect: every time a client connection is stablished, this event is fired.

OnClientDisconnect: every time a client connection is dropped, this event is fired.

OnClientMessage: raised when a new text message is received from the server.

OnClientBinary: raised when a new binary message is received from the server.

OnClientFragmented: raised when a new fragmented message is received from the server.

OnServerConnect: raised when a new server connects to LoadBalancerServer.

OnServerDisconnect: raised when a server disconnects from LoadBalancerServer.

OnServerReady: raised when a server is ready to accept messages.

5. Configure "LoadBalancer" property of back-up servers, to connect to LoadBalancer Server.

6. Create a procedure and set property Active := True

```
sgcWebSocketLoadBalancerServer1.Active := True
```

TsgcWebSocketProxyServer

TsgcWebSocketProxyServer implements a WebSocket Server Component which listens to client WebSocket connections and forwards data connections to a normal TCP/IP server. This is especially useful for browser connections because allows a browser to virtually connect to any server.

TsgcIWWebSocketClient

TsgcIWWebSocketClient implements Intraweb WebSocket Component and can connect to a WebSocket Server. Follow the next steps to configure this component:

1. Drop a TsgcIWWebSocketClient component in the form

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

```
Host := '127.0.0.1';
Port := 80;
TLS := True;
Paramters := '/ws/';
```

3. You can select if you want TLS (secure connection) or not, by default is not Activated.

4. Set Transports allowed.

WebSockets: it will use standard WebSocket implementation

Emulation: if browser doesn't support WebSockets, then it will use a loop AJAX callback connection

5. If you want, you can handle events

OnAsyncConnect: when a WebSocket connection is established, this event is fired

OnAsyncDisconnect: when a WebSocket connection is dropped, this event is fired

OnAsyncError: every time there is a WebSocket error (like mal-formed handshake), this event is fired

OnAsyncMessage: every time the server sends a message, this event is fired

OnAsyncEmulation: this event is fired on every loop of emulated connection

6. Create an Async Procedure and set property Active := True

```
sgcIWWebSocketClient1.Open;
```

Methods

Open: Opens a WebSocket Connection.

Close: Closes a WebSocket Connection.

WriteData: sends a message to WebSocket Server.

Properties

Connected: is a read-only variable and returns True if the connection is Active, otherwise returns False.

JSOpen: here you can include JavaScript Code on the client side when a connection is opened.

JSClose: here you can include JavaScript Code on the client side when a connection is closed.

JSMessage: here you can include JavaScript Code on the client side when clients receive a message from the server. You can get Message String, using Javascript variable "text".

JSError: here you can include JavaScript Code on the client side when an error is raised. You can get Message Error, using Javascript variable "text".

Connections

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.

Disconnect: close client connection from the server side. A "CloseCode" can be specified optionally.

Ping: sends a ping to the client.

Subscribed: returns if the connection is subscribed to a custom channel.

Properties

Data: user session data object, here you can pass an object and access this every time you need, for example: you can pass a connection to a database, user session properties...

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.

Protocols

Protocols

With WebSockets, you can implement Sub-protocols allowing to create customized communications, **for example** you can implement a sub-protocol over WebSocket protocol to communicate 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 SGC: implemented using JSON-RPC 2.0 messages, provides the following patterns: RPC, PubSub, Transactional Messages, Messages Acknowledgment and more.

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

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

4. Protocol Presence: allows to know who is subscribed to a channel, example: chat rooms, collaborators on a document, people viewing the same web page, competitors in a game...

5. 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.

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

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

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

9. Protocol 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.

10. Protocol STOMP: STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

10.1 STOMP for RabbitMQ: client for RabbitMQ Broker.

10.2 STOMP for ActiveMQ: client for ActiveMQ Broker.

If you need to use **more than one protocol using a single connection** (example: you may need to use **default protocol** to handle Remote Procedure Calls and **Dataset protocol** to handle database connections) you can assign a "Broker" to each protocol component and all messages will be exchanged using this intermediary protocol (you can check "Tickets Demo" to get a simple example of this).

Protocols can be registered at **runtime**, just call Method **RegisterProtocol** and pass protocol component as a parameter.

Javascript Reference

Here you can get more information about 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>
```

Subprotocols

Default

Protocol Default

This is default sub-protocol implemented using "JSONRPC 2.0" messages, every time you send a message using this protocol, a JSON object is created with the following properties:

jsonrpc: A String specifying the version of the JSON-RPC protocol. MUST be exactly "2.0".

method: A String containing the name of the method to be invoked. Method names that begin with the word rpc followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else.

params: A Structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id: An identifier established by the Client that MUST contain a String, Number, or NULL value if included. If it is not included it is assumed to be a notification. The value SHOULD normally not be Null [1] and Numbers SHOULD NOT contain fractional parts [2]

JSON object example:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

Features

- **Publish/subscribe** message pattern to provide one-to-many message distribution and decoupling of applications. Supports Wildcard characters, so you can subscribe to a hierarchy of channels. Example: if you want to subscribe to all channels which start with 'news', then call Subscribe('news*').
- A messaging transport that is **agnostic** to the content of the payload
- **Acknowledgment** of messages sent.
- Supports **transactional messages** through server local transactions. When the client commits the transaction, the server processes all messages queued. If client rollback the transaction, then all messages are deleted.
- Implements **QoS** (Quality of Service) for message delivery.

Components

TsgcWSPClient_sgc: Server Protocol Default VCL Component.

TsgcWSPClient_sgc: Client Protocol Default VCL Component.

Javascript Component: Client Javascript Reference.

Browser Test

If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/esegece.com.html
```

TsgcWSPServer_sgc

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

Methods

Subscribe / UnSubscribe: subscribe/unsubscribe to a channel. Supports wildcard characters, so you can subscribe to a hierarchy of channels. Example: if you want to subscribe to all channels which start with 'news', then call Subscribe('news*').

Publish: sends a message to all subscribed clients. Supports wildcard characters, so you can publish to a hierarchy of channels. Example: if you want to send a message to all subscribers to channels which start with 'news', then call Publish('news*').

RPCResult: if a call RPC from the client is successful, the server will respond with this method.

RPCError: if a call RPC from the client it has an error, the server will respond with this method.

Broadcast: sends a message to all connected clients, if you need to broadcast a message to selected channels, use Channel argument.

WriteData: sends a message to single or multiple selected clients.

Properties

RPCAuthentication: if enabled, every time a client requests an RPC, method name needs to be authenticated against a username and password.

Methods: is a list of allowed methods. Every time a client sends an RPC first it will search if this method is defined on this list, if it's not in this list, OnRPCAuthentication event will be fired.

Subscriptions: returns a list of active subscriptions.

UseMatchesMasks: if enabled, subscriptions and publish methods accepts wildcards, question marks... check MatchesMask Delphi function to see all supported masks.

Events

OnRPCAuthentication: if RPC Authentication is enabled, this event is fired to define if a client can call this method or not.

OnRPC: fired when the server receives an RPC from a client.

OnNotification: fired every server receive a Notification from a client.

OnBeforeSubscription: fired every time before a client subscribes to a custom channel. Allows denying a subscription.

OnSubscription: fired every time a client subscribes to a custom channel.

OnUnSubscription: fired every time a client unsubscribes from a custom channel.

OnRawMessage: this event is fired before a message is processed by component.

TsgcWSPClient_sgc

This is Client Protocol Default Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

Publish: sends a message to all subscribed clients.

RPC: Remote Procedure Call, client request a method and response will be handled OnRPCResult or OnRPCError events.

Notify: the client sends a notification to a server, this notification doesn't need a response.

Broadcast: sends a message to all connected clients, if you need to broadcast a message to selected channels, use Channel argument.

WriteData: sends a message to a server. If you need to send a message to a custom TsgcWSProtocol_Server_sgc, use "Guid" Argument. If you need to send a message to a single channel, use "Channel" Argument.

Subscribe: subscribe client to a custom channel. If the client is subscribed, OnSubscription event will be fired.

Unsubscribe: unsubscribe client to a custom channel. If the client is unsubscribed, OnUnsubscription event will be fired.

UnsubscribeAll: unsubscribe client from all subscribed channel. If the client is unsubscribed, OnUnsubscription event will be fired for every channel.

GetSession: requests to server session id, data session is received OnSession Event.

StartTransaction: begins a new transaction.

Commit: server processes all messages queued in a transaction.

RollBack: server deletes all messages queued in a transaction.

Events

OnEvent: this event is fired every time a client receives a message from a custom channel.

OnRPCResult: this event is fired when the client receives a successful response from the server after a RPC is sent.

OnRPCError: this event is fired when the client receives a error response from the server after an RPC is sent.

OnAcknowledgment: this event is fired when the client receives error an acknowledgment from the server that message has been received.

OnRawMessage: this event is fired before a message is processed by the component.

OnSession: this event is fired after a successful connection or after a GetSession request.

Properties

Queue: disabled by default, if True all text/binary messages are not processed and queued until queue is disabled.

QoS: Three "Quality of Service" provided:

Level 0: "At most once", the message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the server either once or not at all.

Level 1: "At least once", the receipt of a message by the server is acknowledged by an ACKNOWLEDGMENT message. If there is an identified failure of either the communications link or the sending device, or the acknowledgement message is not received after a specified period of time, the sender resends the message. The message arrives at the server at least once. A message with QoS level 1 has an ID param in the message.

Level 2: "Exactly once", where 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.

Subscriptions: returns a list of active subscriptions.

TsgcIWWSPClient_sgc

This is Intraweb Client Protocol Default Component, you need to drop this component in the form and select a TsgcIWWebSocketClient Component using Client Property.

Methods

WriteData: sends a message to a server. If you need to send a message to a custom TsgcWSProtocol_Server_sgc, use "Guid" Argument. If you need to send a message to a single channel, use "Channel" Argument.

Subscribe: subscribe client to a custom channel. If the client is subscribed, OnSubscription event will be fired.

Unsubscribe: unsubscribe client to a custom channel. If client is unsubscribed, OnUnsubscription event will be fired.

Protocol Default Javascript

Default Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and esegece.com.js** files.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Publish Message to test channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'test');
</script>
```

Show Alert with Event Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcevent', function(event)
  {
    alert('channel:' + event.channel + '. message: ' + event.message);
  }
</script>
```

Call RPC

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
```

```
  var socket = new sgcws('ws://{%host%}:{%port%}');
  var params = {param:10};
  socket.rpc(GUID(), 'test', JSON.stringify(params));
</script>
```

Handle RPC Response

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcrpcresult', function(event)
  {
    alert('result:' + event.result);
  }
  socket.on('sgcrpcerror', function(event)
  {
    alert('error:' + event.code + ' ' + event.message);
  }
</script>
```

Call Notify

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  var params = {param:10};
  socket.notify('test', JSON.stringify(params));
</script>
```

Send Messages in a Transaction

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.starttransaction('sgc:test');
  socket.publish('Message1', 'sgc:test');
  socket.publish('Message2', 'sgc:test');
  socket.publish('Message3', 'sgc:test');
  socket.commit('sgc:test');
</script>
```

Show Alert OnSubscribe or OnUnSubscribe to a channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcsubscribe', function(event)
  {
    alert('subscribed: ' + event.channel);
  }
  socket.on('sgcunsubscribe', function(event)
  {
    alert('unsubscribed: ' + event.channel);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
```

```
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Get Session

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcsession', function(event)
  {
    alert(event.guid);
  };
  socket.getsession();
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.state();
</script>
```

Set QoS

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');

  socket.qoslevel1();
  socket.publish('message', 'channel');
</script>
```

Set Queue Level

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
```

```
  socket.queuelevel2();
  socket.publish('message1', 'channel1');
  socket.publish('message2', 'channel1');
</script>
```

Dataset

Protocol Dataset

This protocol inherits from Protocol Default and it's useful if you want to broadcast dataset changes over clients connected to this protocol.

It uses "JSON-RPC 2.0" Object, and every time there is a dataset change, it sends all field values (* only fields supported) using Dataset Object.

To allow the component to search records on the dataset, you need to specify which fields are the Key, example: if in your dataset, ID field is the key you will need to write a code like this

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

Components
TsgcWSPServer_Dataset: Server Protocol Dataset VCL Component.

TsgcWSPClient_Dataset: Client Protocol Dataset VCL Component.

Javascript Component: Client Javascript Reference.

Browser Test
If you want to test this protocol with your favourite Web Browser, please type this URL (you need to define your custom host and port)

```
http://host:port/dataset.esegece.com.html
```

TsgcWSPServer_Dataset

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

This component inherits from TsgcWSProtocol_Server_sgc all methods and properties.
Properties
ApplyUpdates: if enabled, every time the server receives a dataset update from client, it will be saved on the server side.

NotifyUpdates: if enabled, every time dataset server changes, server broadcasts this change to all connected clients.

NotifyDeletes: if enabled, every time a record is deleted, server broadcasts this to all connected clients.

AutoEscapeText: if enabled (disabled by default), automatically escape/unescape characters inside field values like "{", "["...

AutoSynchronize: if enabled, every time a client connects to the server, the server will sent metadata and all dataset records to client.

UpdateMode: if umWhereAll (by default) all fields are broadcasted to clients, if umWhereChanged only Fields that have changed will be broadcasted to connected clients.

Methods
BroadcastRecord: sends dataset record values to all connected clients.

MetaData: sends metadata info to a client.

Synchronize: sends all dataset records to a client.

Events
These events are specific on the dataset protocol.

OnAfterDeleteRecord: event fired after a record is deleted from Dataset.

OnAfterNewRecord: event fired after a record is created on Dataset.

OnAfterUpdateRecord: event fired after a record is updated on Dataset.

OnBeforeDeleteRecord: event fired before a record is deleted from Dataset. If Argument "Handled" is True, means that the user handles this event and if won't be deleted (by default this argument is False)

OnBeforeNewRecord: event fired before a record is created on Dataset. If Argument "Handled" is True, means that the user handles this event and if won't be inserted (by default this argument is False)

OnBeforeUpdateRecord: event fired before a record is updated on Dataset. If Argument "Handled" is True, means that the user handles this event and if won't be updated (by default this argument is False)

OnBeforeDatasetUpdate: event fired before a dataset record is updated.

TsgcWSPClient_Dataset

This is Client Protocol Dataset Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property and select a Dataset Component using Dataset Property.

This component inherits from TsgcWSProtocol_Client_sgc all methods and properties.

Methods

Subscribe_all: subscribe to all available channels

new: fired on new dataset record.
update: fired on post dataset record.
delete: fired on delete dataset record.

Synchronize: requests all dataset records from the server

GetMetaData: requests all dataset fields from server

Events

These events are specific on the dataset protocol.

OnAfterDeleteRecord: event fired after a record is deleted from Dataset.

OnAfterNewRecord: event fired after a record is created on Dataset.

OnAfterUpdateRecord: event fired after a record is updated on Dataset.

OnAfterSynchronize: event fired after synchronization has ended.

OnBeforeDeleteRecord: event fired before a record is deleted from Dataset. If Argument "Handled" is True, means that the user handles this event and if won't be deleted (by default this argument is False)

OnBeforeNewRecord: event fired before a record is created on Dataset. If Argument "Handled" is True, means that user the handles this event and if won't be inserted (by default this argument is False)

OnBeforeUpdateRecord: event fired before a record is updated on Dataset. If Argument "Handled" is True, means that user the handles this event and if won't be updated (by default this argument is False)

OnBeforeSynchronization: event fired before a synchronization starts.

OnMetaData: event fired after a GetMetaData request. Example:

```
procedure OnMetaData(Connection:
    TsgcWSConnection; const JSON: TsgcObjectJSON);
var
  i: integer;
  vFieldName, vDataType: string;
  vDataSize: Integer;
  vKeyField: Boolean;
begin
  for i:= 0 to JSON.Count -1 do
  begin
    vFieldName := JSON.Item[i].Node['fieldname'].Value;
```

```
        vDataType := JSON.Item[i].Node['datatype'].Value;
        vDataSize := JSON.Item[i].Node['datasize'].Value;
        vKeyField := JSON.Item[i].Node['keyfield'].Value;
      end;
    end;
```

Properties

AutoSubscribe: enabled by default, if True, client subscribes to all available channels after successful connection.

ApplyUpdates: if enabled, every time the client receives a dataset update from server, it will be saved on the client side.

AutoEscapeText: if enabled (disabled by default), automatically escape/unescape characters inside field values like "{", "["...

NotifyUpdates: if enabled, every time dataset client changes, it sends a message to server notifying this change.

UpdateMode: if umWhereAll (by default) all fields are transmitted to the server, if umWhereChanged only Fields that have changed will be transmitted to the server.

TsgcIWWSPClient_Dataset

This is Intraweb Client Protocol Dataset Component, you need to drop this component in the form and select a TsgcIWWebSocketClient Component using Client Property and select a Dataset Component using Dataset Property.

This component inherits from TsgcIWWSPClient_sgc all methods and properties.

Methods
Subscribe_New: fired on new dataset record
Subscribe_Update: fired on post dataset record
Subscribe_Delete: fired on delete dataset record

Protocol Dataset Javascript

Dataset Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and dataset.esegece.com.js** files.

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script
src="http://www.example.com:80/dataset.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
</script>
```

Send Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.send('Hello sgcWebSockets!');
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcdataset', function(event)
  {
    alert(event.dataset);
  }
</script>
```

Show Alert with Dataset Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Show Alert OnSubscribe or OnUnSubscribe to a channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
```

```
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.on('sgcsubscribe', function(event)
  {
    alert('subscribed: ' + event.channel);
  }
  socket.on('sgcunsubscribe', function(event)
  {
    alert('unsubscribed: ' + event.channel);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  var socket = new sgcws_dataset('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Subscribe All Dataset Changes

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.subscribe_all();
</script>
```

UnSubscribe All Dataset Changes

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.unsubscribe_all();
</script>
```

Handle Dataset Changes

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
     var socket = new sgcws_dataset('ws://{%host%}:{%port%}');

     socket.on('sgcdataset', function(evt){

          if ((evt.channel == "sgc@dataset@new") || (evt.channel ==
          "sgc@dataset@update")) {

               ... here you need to implement your own code
               insert/update records ...
```

```
          }
          else if (evt.channel == "sgc@dataset@delete") {

          ... here you need to implement your own code to delete
          records ...

          }
     });
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/dataset.esegece.com.js"></script>
<script>
  socket.state();
</script>
```

Files

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.

TsgcWSPServer_Files

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

Methods

SendFile: sends a file to a client, you can set the following parameters

aSize: size of every packet in bytes.
aData: user custom data, here you can write any text you think is useful for client.
aChannel: if you only want to send data to all clients subscribed to this channel.
aQoS: type of quality of service.

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.

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.

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 TsgcWebSocketClient Component using Client Property.

Methods

SendFile: sends a file to the server, you can set the following parameters

aSize: size of every packet in bytes.
aData: user custom data, here you can write any text you think is useful for the server.
aQoS: type of quality of service.

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.

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.

Presence

Protocol Presence

Presence protocol allows to know who is subscribed to a channel, this makes more easy to create chat applications and know who is online, example: game users, chat rooms, users viewing the same document...
Features

- By default user is **identified by a name**, but this can be **customized** passing more data: email, company, twitter...
- Events to **Authorize** if a **Channel** can be created, if a **member** is allowed...
- Every time a new **member joins** a channel, all members are **notified**.
- **Publish messages** to all channel subscribers.
- **Low memory** usage.

Components
TsgcWSPServer_Presence: Server Protocol Presence VCL Component.

TsgcWSPClient_Presence: Client Protocol Presence VCL Component.

Classes
TsgcWSPresenceMessage: the object which encapsulates presence packet information.

TsgcWSPServer_Presence

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

Methods

All methods are handled internally by the server in response to client requests.

Properties

You must link this component to a **Server** or to a **Broker** if you are using more than one protocol.

Acknowledgment: if enabled, every time a server sends a message to client assign an ID to this message and queues in a list. When the client receives a message, if detect it has an ID, it sends an Acknowledgment to the server, which means the client has processed message and server can delete from the queue.

- **Interval:** interval in seconds where server checks if there are messages not processed by client.
- **Timeout:** maximum wait time before the server sends the message again.

Events

There are several events to handle actions like: a new member request to join a channel, a new channel is created by a member, a member unsubscribes from a channel...

New Member

When a new client connects to a server, first sends member data to the server to request a new member. Following events can be called:

- **OnBeforeNewMember:** server receives a request from the client to join and the server accepts or not this member. Use **Accept** parameter to allow or not this member. By default all members are accepted.

```
procedure OnBeforeNewMember(aConnection:  TsgcWSConnection;
const aMember: TsgcWSPresenceMember; var Accept: Boolean);
begin
  if aMember.Name = 'Spam' then
    Accept := False;
end;
```

- **OnNewMember:** After a new member is accepted, then this event is called and means this member has join member list. You can use aMember.Data property to store objects in memory like database access, session objects...

```
procedure OnNewMember(aConnection: TsgcWSConnection; const
aMember: TsgcWSPresenceMember);
begin

end;
```

Subscriptions

When a client has joined as a member, can subscribe to **new channels**, if a channel not exists, the following events can be called:

- **OnBeforeNewChannel:** server receives a subscription request from the client to join this channel but the channel doesn't exist, the server can accept or not to create this channel. Use **Accept** parameter to allow or not this channel. By default, all channels are accepted.

```
procedure OnBeforeNewChannelBeforeNewChannel(Connection:
TsgcWSConnection; const aChannel: TsgcWSPresenceChannel;
const aMember: TsgcWSPresenceMember; var Accept: Boolean)
begin
  if aChannel.Name = 'Spam' then
    Accept := False;
end;
```

- **OnNewChannel:** After a new channel is accepted, then this event is called and means a new channel has been created. Channel properties can be customized in this event.

```
procedure OnNewChannel(Connection: TsgcWSConnection; var
aChannel: TsgcWSPresenceChannel);
begin

end;
```

If the **channel already exists** or **has been created**, the server can accept or no new subscriptions.

- **OnBeforeNewChannelMembers:** server receives a subscription request from a client to join this channel, the server can accept or not a member join. Use **Accept** parameter to allow or not this member. By default, all members are accepted.

```
procedure OnBeforeNewChannelMember(Connection:
TsgcWSConnection; const aChannel: TsgcWSPresenceChannel;
const aMember: TsgcWSPresenceMember; var Accept: Boolean)
begin
  if aMember.Name = 'John' then
    Accept := True
  else if aMember.Name = 'Spam' then
    Accept := False;
end;
```

- **OnNewChannelMember:** After a new member is accepted, then this event is called and means a new member has joined the channel. All subscribers to this channel, will be notified about new members.

```
procedure OnNewChannelMember(Connection: TsgcWSConnection;
const aChannel: TsgcWSPresenceChannel; const aMember:
TsgcWSPresenceMember);
begin

end;
```

UnSubscriptions

Every time a member unjoin a channel or disconnects, the server is notified by following events:

- **OnRemoveChannelMember:** server receives a subscription request from a client to join this channel but the channel doesn't exist, the server can accept or not to create this channel. Use **Accept** parameter to allow or not this channel. By default all channels are accepted.

```
procedure OnRemoveChannelMember(Connection:
TsgcWSConnection; const aChannel: TsgcWSPresenceChannel;
const aMember: TsgcWSPresenceMember);
begin
  Log('Member: ' + aMember.Name + ' unjoin channel: ' +
aChannel.Name);
end;
```

When a member disconnects, automatically server is notified:

- **OnRemoveMember:** when the client disconnects from protocol, this event is called and server is notified of which never has disconnected.

```
procedure OnRemoveMember(aConnection: TsgcWSConnection;
aMember: TsgcWSPresenceMember);
begin
  Log('Member: ' + aMember.Name);
end;
```

Errors

Every time there is an error, these events are called, example: server has denied a member to subscribe to a channel, a member try to subscribe to an already subscribed channel...

- **OnErrorMemberChannel:** this event is called every time there is an error trying to create a new channel, join a new member, subscribe to a channel...

```
procedure OnErrorMemberChannel(Connection: TsgcWSConnection;
const aError: TsgcWSPresenceError; const aChannel:
TsgcWSPresenceChannel; const aMember: TsgcWSPresenceMember);
begin
  Log('#Error: ' + aError.Text);
end;
```

When a member disconnects, automatically server is notified:

- **OnErrorPublishMsg:** when a client publish a message and this is denied by the server, this event is raised.

```
procedure OnErrorPublishMsg(Connection: TsgcWSConnection;
const aError: TsgcWSPresenceError; const aMsg:
TsgcWSPresenceMsg; const aChannel: TsgcWSPresenceChannel;
const aMember: TsgcWSPresenceMember);
begin
  Log('#Error: ' + aError.Text);
end;
```

TsgcWSPresenceMessage

This object encapsulates all internal messages exchange by server and client presence protocol.

TsgcWSPresenceMember

ID: internal identifier
Name: member name, provided by the client.
Info: member additional info, provided by the client.
Data: TObject which can be used for server purposes.

TsgcWSPresenceMemberList

Member[i]: member of a list by index
Count: number of members of the list

TsgcWSPresenceChannel

Name: channel name, provided by the client.

TsgcWSPresenceMsg

Text: text message, provided by the client when call Publish method

TsgcWSPresenceError

Code: integer value identifying the error
Text: error description.

TsgcWSPClient_Presence

This is Server Presence Protocol Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Properties
Presence: member data

- **Name:** member name.
- **Info:** any additional info related to member (example: email, twitter, company...)

Acknowledgment: if enabled, every time a client sends a message to server assign an ID to this message and queues in a list. When the server receives the message, if detect it has an ID, it sends an Acknowledgment to the client, which means the server has processed message and the client can delete from the queue.

- **Interval:** interval in seconds where the client checks if there are messages not processed by server.
- **Timeout:** maximum wait time before the client sends the message again.

Methods

There are several methods to subscribe to a channel, get a list of members...

Connect

When a client connects, the first event called is **OnSession**, the server sends a session ID to the client, which identifies this client in the server connection list.

After OnSession event is called, automatically client sends a request to the server to join as a member, if successful, **OnNewMember** event is raised

```
procedure OnNewMember(aConnection: TsgcWSConnection; const
aMember: TsgcWSPresenceMember);
begin

end;
```

Subscriptions

When a client wants subscribe to a channel, use method **"Subscribe"** and pass channel name as argument

```
Client.Subscribe('MyChannel');
```

If the client is successfully subscribed, **OnNewChannelMember** event is called. All members of this channel will be notified using the same event.

```
procedure OnNewChannelMember(Connection: TsgcWSConnection;
const aChannel: TsgcWSPresenceChannel; const aMember:
TsgcWSPresenceMember);
begin
  Log('Subscribed: ' + aChannel.Name);
end;
```

if server denies access to a member, **OnErrorMemberChannel** event is raised.

```
procedure OnErrorMemberChannel(Connection: TsgcWSConnection;
const aError: TsgcWSPresenceError; const aChannel:
TsgcWSPresenceChannel; const aMember: TsgcWSPresenceMember)
begin
  Log('Error: ' + aError.Text);
end;
```

UnSubscriptions

When a client unsubscribe from a channel, use method **"Unsubscribe"** and pass channel name as argument

```
Client.Unsubscribe('MyChannel');
```

If a client is successfully unsubscribed, **OnRemoveChannelMember** event is called. All members of this channel will be notified using the same event. All members of this channel will be notified using the same event.

```
procedure OnNewChannelMember(Connection: TsgcWSConnection;
const aChannel: TsgcWSPresenceChannel; const aMember:
TsgcWSPresenceMember);
begin
  Log('Unsubscribed: ' + aChannel.Name);
end;
```

if a client can't unsubscribe from a channel, example: because is not subscribed, **OnErrorMemberChannel** event is raised.

```
procedure OnErrorMemberChannel(Connection: TsgcWSConnection;
const aError: TsgcWSPresenceError; const aChannel:
TsgcWSPresenceChannel; const aMember: TsgcWSPresenceMember)
begin
  Log('Error: ' + aError.Text);
end;
```

When a client disconnects from the server, **OnRemoveEvent** is called.

```
procedure OnRemoveMember(aConnection: TsgcWSConnection;
aMember: TsgcWSPresenceMember);
begin
  Log('#RemoveMember: ' + aMember.Name);
end;
```

Publish

When a client wants to send a message to all members or all subscribers of a channel, use **"Publish"** method

```
Client.Publish('Hello All Members');

Client.Publish('Hello All Members of this channel',
'MyChannel');
```

If a message is successfully published, **OnPublishMsg** event is called. All members of this channel will be notified using the same event.

```
procedure OnPublishMsg(Connection: TsgcWSConnection; const
aMsg: TsgcWSPresenceMsg; const aChannel:
TsgcWSPresenceChannel; const aMember: TsgcWSPresenceMember);
begin
  Log('#PublishMsg: ' + aMsg.Text + ' ' + aMember.Name);
end;
```

if a message can't be published, **OnErrorPublishMsg** event is raised.

```
procedure OnErrorPublishMsg(Connection: TsgcWSConnection;
const aError: TsgcWSPresenceError; const aMsg:
TsgcWSPresenceMsg; const aChannel: TsgcWSPresenceChannel;
const aMember: TsgcWSPresenceMember);
begin
  Log('#Error: ' + aError.Text);
end;
```

GetMembers

A client can request to the server a list of all members or all members subscribed to a channel. Use **"GetMembers"** method

```
Client.GetMembers;

Client.GetMembers('MyChannel');
```

If a message is successfully processed by the server, OnGetMembers event is called

```
procedure OnGetMembers(Connection: TsgcWSConnection; const
aMembers: TsgcWSPresenceMemberList; const aChannel:
TsgcWSPresenceChannel);
var
  i: Integer;
begin
  for i := 0 to aMembers.Count - 1 do
    Log('#GetMembers: ' + aMembers.Member[i].ID + ' ' +
aMembers.Member[i].Name);
end;
```

If there is an error because the member is not allowed or is not subscribed to channel, OnErrorMemberChannel event is raised

```
procedure OnErrorMemberChannel(Connection: TsgcWSConnection;
const aError: TsgcWSPresenceError; const aChannel:
TsgcWSPresenceChannel; const aMember: TsgcWSPresenceMember)
begin
  Log('Error: ' + aError.Text);
end;
```

Invite

A client can invite to other member to subscribe to a channel.

```
Client.Invite('MyChannel', 'E54541D0F0E5R40F1E00FEEA');
```

When the other member receives the invitation, OnChannelInvitation event is called and member can Accept or not the invitation.

```
procedure OnChannelInvitation(Connection: TsgcWSConnection;
const aMember: TsgcWSPresenceMember; const aChannel:
TsgcWSPresenceChannel; var Accept: Boolean)
begin
  if aChannel = 'MyChannel' then
    Accept := True
  else
    Accept := False;
end;
```

Protocol Presence Javascript

Presence Protocol Javascript sgcWebSockets uses **sgcWebSocket.js and presence.esegece.com.js** files.

Here you can find available methods, you must replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script
src="http://www.example.com:80/presence.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
</script>
```

New Member after connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcsession', function(event)
  {
    socket.newmember(event.id, 'John', 'Additional Info');
  });
</script>
```

Subscribe to Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.subscribe('Topic 1');
</script>
```

Unsubscribe from Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.unsubscribe('Topic 1');
</script>
```

Publish Message to Topic 1 channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.publish('Hello sgcWebSockets!', 'Topic 1');
</script>
```

Receive Message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcpublishmsg', function(event)
  {
    console.log('#publishmsg: ' + event.message.text);
  });
</script>
```

Get All Members Connected

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcgetmembers', function(event)
  {
    for (var i  in event.members) {
     console.log(event.members[i].id + ' ' + event.members[i].name);
    }
  });
  socket.getmembers();
</script>
```

Show Alert when Members subscribe/unsubscribe

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('sgcnewmember', function(event)
  {
    alert('#new member: ' + event.member.name);
  }
  );
  socket.on('sgcremovemember', function(event)
  {
    alert('#removed member: ' + event.member.name);
  }
  );
  socket.on('sgcnewchannelmember', function(event)
  {
    alert('#new member: ' + event.member.name + ' in channel: ' +
event.channel.name);
  }
  );
  socket.on('sgcremovechannelmember', function(event)
  {
    alert('#remove member: ' + event.member.name + ' from channel: ' +
event.channel.name);
  }
  );
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
```

```
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  var socket = new sgcws_presence('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('sgcerrormemberchannel', function(event)
  {
    alert('#error member channel: ' + event.error.text);
  }
  );
  socket.on('sgcerrorpublishmsg', function(event)
  {
    alert('#error publish: ' + event.error.text);
  }
  );
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script
src="http://{%host%}:{%port%}/presence.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

MQTT

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 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.

· A small transport overhead and protocol exchanges minimized to reduce network traffic.

· A mechanism to notify interested parties when an abnormal disconnection occurs.

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 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.

TsgcWSPClient_MQTT

This is Client Protocol MQTT Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

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.

Unsubscribe: unsubscribe client to a custom channel. If the client is unsubscribed, OnMQTTUnsubscribe event will be fired.

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.

Disconnect: disconnects from MQTT server.

Events

OnMQTTBeforeConnect: this event is fired 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 fired 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

ReturnCode: returns code with the result of connection.

OnQTTDisconnect: this event is fired when the client is disconnected from MQTT server.

OnMQTTPing: this event is fired when the client receives an acknowledgment from a ping previously sent.

OnMQTTPubAck: this event is fired when receives the response to a Publish Packet with QoS level 1. There is one parameter:

PacketIdentifier: is packet identifier sent initially.

OnMQTTPubComp: this event is fired when receives the response to a PubRel Packet. It is the fourth and final packet of the QoS 2 protocol exchange. There is one parameter:

PacketIdentifier: is packet identifier sent initially.

OnMQTTPublish: this event is fired 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.

OnMQTTPubRec: this event is fired when receives the response to a Publish Packet with QoS 2. It is the second packet of the QoS 2 protocol exchange. There is one parameter:

PacketIdentifier: is packet identifier sent initially.

OnMQTTSubscribe: this event is fired as a response to subscribe method. There are 2 parameters:

PacketIdentifier: is packet identifier sent initially.

Codes: codes with the result of a subscription.

OnMQTTUnSubscribe: this event is fired as a response to subscribe method. There is one parameter:

PacketIdentifier: is packet identifier sent initially.

Properties

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.

LastWillTestament: if there is a disconnection and is enabled, a message is sent to all connected clients to inform that connection has been closed.

STOMP

Protocol STOMP

STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.

Our STOMP client components support following STOMP versions: 1.0, 1.1 and 1.2.

Components

TsgcWSPClient_STOMP: generic STOMP Protocol client, allows to connect to any STOMP Server.

TsgcWSPClient_STOMP_RabbitMQ: STOMP client for RabbitMQ Broker.

TsgcWSPClient_STOMP_ActiveMQ: STOMP client for ActiveMQ Broker.

TsgcWSPClient_STOMP

This is Client Protocol STOMP Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

Send: The SEND frame sends a message to a destination in the messaging system.

Subscribe: The SUBSCRIBE frame is used to register to listen to a given destination.

UnSubscribe: The UNSUBSCRIBE frame is used to remove an existing subscription.

ACK: ACK is used to acknowledge the consumption of a message from a subscription.

NACK: NACK is the opposite of ACK. It is used to tell the server that the client did not consume the message.

BeginTransaction: is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be processed atomically based on the transaction.

CommitTransaction: is used to commit a transaction in progress.

AbortTransaction: is used to roll back a transaction in progress.

Disconnect: use to graceful shutdown connection, where the client is assured that all previous frames have been received by the server.

Events

OnSTOMPConnected: this event is fired after a new connection is established.

version : The version of the STOMP protocol the session will be using. See Protocol Negotiation for more details.

STOMP 1.2 servers MAY set the following headers:

heart-beat : The Heart-beating settings.

session : A session identifier that uniquely identifies the session.

server : A field that contains information about the STOMP server. The field MUST contain a server-name field and MAY be followed by optional comment fields delimited by a space character.

OnSTOMPMessage: this event is fired when the client receives a message.

The MESSAGE frame MUST include a destination header indicating the destination the message was sent to. If the message has been sent using STOMP, this destination header SHOULD be identical to the one used in the corresponding SEND frame.

The MESSAGE frame MUST also contain a message-id header with a unique identifier for that message and a subscription header matching the identifier of the subscription that is receiving the message.

If the message is received from a subscription that requires explicit acknowledgment (either client or client-individual mode) then the MESSAGE frame MUST also contain an ack header with an arbitrary value. This header will be used to relate the message to a subsequent ACK or NACK frame.

MESSAGE frames SHOULD include a content-length header and a content-type header if a body is present.

MESSAGE frames will also include all user-defined headers that were present when the message was sent to the destination in addition to the server-specific headers that MAY get added to the frame.

OnSTOMPReceipt: this event is fired once a server has successfully processed a client frame that requests a receipt.

A RECEIPT frame is an acknowledgment that the corresponding client frame has been processed by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been received by the server. However, these previous frames may not yet be fully processed. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

OnSTOMPError: this event is fired if something goes wrong.
The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).
If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.
ERROR frames SHOULD include a content-length header and a content-type header if a body is present.

Properties

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: Heart-beating can optionally be used to test the healthiness of the underlying TCP connection and to make sure that the remote end is alive and kicking. In order to enable heart-beating, each party has to declare what it can do and what it would like the other party to do. 0 means it cannot send/receive heart-beats, otherwise it is the desired number of milliseconds between heart-beats.

Options: The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.

Versions: Set which STOMP versions are supported.

Brokers

TsgcWSPClient_STOMP_RabbitMQ

This is Client Protocol STOMP Component for RabbitMQ Broker, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Destinations
The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific. The RabbitMQ STOMP adapter supports a number of different destination types:

- **Topic:** SEND and SUBSCRIBE to transient and durable topics.
- **Queue:** SEND and SUBSCRIBE to queues managed by the STOMP gateway.
- **QueueOutside:** SEND and SUBSCRIBE to queues created outside the STOMP gateway.
- **TemporaryQueue:** create temporary queues (in reply-to headers only).
- **Exchange:** SEND to arbitrary routing keys and SUBSCRIBE to arbitrary binding patterns.

Methods
Publish: The SEND frame sends a message to a destination in the messaging system.
PublishTopic
PublishQueue
PublishQueueOutside
PublishTemporaryQueue
PublishExchange

Subscribe: The SUBSCRIBE frame is used to register to listen to a given destination.
Supports following subscriptions
SubscribeTopic
SubscribeQueue
SubscribeQueueOutside
SubscribeTemporaryQueue
SubscribeExchange

UnSubscribe: The UNSUBSCRIBE frame is used to remove an existing subscription.
Supports following UnSubscriptions
UnSubscribeTopic
UnSubscribeQueue
UnSubscribeQueueOutside
UnSubscribeTemporaryQueue
UnSubscribeExchange

ACK: ACK is used to acknowledge the consumption of a message from a subscription.

NACK: NACK is the opposite of ACK. It is used to tell the server that the client did not consume the message.

BeginTransaction: is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be processed atomically based on the transaction.

CommitTransaction: is used to commit a transaction in progress.

AbortTransaction: is used to roll back a transaction in progress.

Disconnect: use to graceful shutdown connection, where the client is assured that all previous frames have been received by the server.

Events

OnRabbitMQConnected: this event is fired after a new connection is established.

version : The version of the STOMP protocol the session will be using. See Protocol Negotiation for more details.

STOMP 1.2 servers MAY set the following headers:

heart-beat : The Heart-beating settings.

session : A session identifier that uniquely identifies the session.

server : A field that contains information about the STOMP server. The field MUST contain a server-name field and MAY be followed by optional comment fields delimited by a space character.

OnRabbitMQMessage: this event is fired when the client receives a message.

The MESSAGE frame MUST include a destination header indicating the destination the message was sent to. If the message has been sent using STOMP, this destination header SHOULD be identical to the one used in the corresponding SEND frame.

The MESSAGE frame MUST also contain a message-id header with a unique identifier for that message and a subscription header matching the identifier of the subscription that is receiving the message.

If the message is received from a subscription that requires explicit acknowledgment (either client or client-individual mode) then the MESSAGE frame MUST also contain an ack header with an arbitrary value. This header will be used to relate the message to a subsequent ACK or NACK frame.

MESSAGE frames SHOULD include a content-length header and a content-type header if a body is present.

MESSAGE frames will also include all user-defined headers that were present when the message was sent to the destination in addition to the server-specific headers that MAY get added to the frame.

OnRabbitMQReceipt: this event is fired once a server has successfully processed a client frame that requests a receipt.

A RECEIPT frame is an acknowledgment that the corresponding client frame has been processed by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been received by the server.

However, these previous frames may not yet be fully processed. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

OnRabbitMQError: this event is fired if something goes wrong.

The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).

If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.

ERROR frames SHOULD include a content-length header and a content-type header if a body is present.

Properties

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: Heart-beating can optionally be used to test the healthiness of the underlying TCP connection and to make sure that the remote end is alive and kicking. In order to enable heart-beating, each party has to declare what it can do and what it would like the other party to do. 0

means it cannot send/receive heart-beats, otherwise it is the desired number of milliseconds between heart-beats.

Options: The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.

Versions: Set which STOMP versions are supported.

TsgcWSPClient_STOMP_ActiveMQ

This is Client Protocol STOMP Component for ActiveMQ Broker, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Destinations
The STOMP specification does not prescribe what kinds of destinations a broker must support, instead the value of the destination header in SEND and MESSAGE frames is broker-specific. The Active STOMP adapter supports a number of different destination types:

- **Topic:** SEND and SUBSCRIBE to transient and durable topics.
- **Queue:** SEND and SUBSCRIBE to queues managed by the STOMP gateway.

Publish Options
Note that STOMP is designed to be as simple as possible - so any scripting language/platform can message any other with minimal effort. STOMP allows pluggable headers on each request such as sending & receiving messages. ActiveMQ has several extensions to the Stomp protocol, so that JMS semantics can be supported by Stomp clients. An OpenWire JMS producer can send messages to a Stomp consumer, and a Stomp producer can send messages to an OpenWire JMS consumer. And Stomp to Stomp configurations, can use the richer JMS message control.

STOMP supports the following standard JMS properties on SENT messages:

- **CorrelationId:** Good consumers will add this header to any responses they send.
- **Expires:** Expiration time of the message.
- **JMSXGroupID:** Specifies the Message Groups.
- **JMSXGroupSeq:** Optional header that specifies the sequence number in the Message Groups.
- **Persistent:** Whether or not the message is persistent.
- **Priority:** Priority on the message.
- **ReplyTo:** Destination you should send replies to.
- **MsgType:** Type of the message.

Methods
Publish: The SEND frame sends a message to a destination in the messaging system.
PublishTopic
PublishQueue

Subscribe: The SUBSCRIBE frame is used to register to listen to a given destination.
Supports following subscriptions
SubscribeTopic
SubscribeQueue

UnSubscribe: The UNSUBSCRIBE frame is used to remove an existing subscription.
Supports following UnSubscriptions
UnSubscribeTopic
UnSubscribeQueue

ACK: ACK is used to acknowledge the consumption of a message from a subscription.

NACK: NACK is the opposite of ACK. It is used to tell the server that the client did not consume the message.

BeginTransaction: is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be processed atomically based on the transaction.

CommitTransaction: is used to commit a transaction in progress.

AbortTransaction: is used to roll back a transaction in progress.

Disconnect: use to graceful shutdown connection, where the client is assured that all previous frames have been received by the server.

Events

OnActiveMQConnected: this event is fired after a new connection is established.

version : The version of the STOMP protocol the session will be using. See Protocol Negotiation for more details.

STOMP 1.2 servers MAY set the following headers:

heart-beat : The Heart-beating settings.

session : A session identifier that uniquely identifies the session.

server : A field that contains information about the STOMP server. The field MUST contain a server-name field and MAY be followed by optional comment fields delimited by a space character.

OnActiveMQMessage: this event is fired when the client receives a message.

The MESSAGE frame MUST include a destination header indicating the destination the message was sent to. If the message has been sent using STOMP, this destination header SHOULD be identical to the one used in the corresponding SEND frame.

The MESSAGE frame MUST also contain a message-id header with a unique identifier for that message and a subscription header matching the identifier of the subscription that is receiving the message.

If the message is received from a subscription that requires explicit acknowledgment (either client or client-individual mode) then the MESSAGE frame MUST also contain an ack header with an arbitrary value. This header will be used to relate the message to a subsequent ACK or NACK frame.

MESSAGE frames SHOULD include a content-length header and a content-type header if a body is present.

MESSAGE frames will also include all user-defined headers that were present when the message was sent to the destination in addition to the server-specific headers that MAY get added to the frame.

OnActiveMQReceipt: this event is fired once a server has successfully processed a client frame that requests a receipt.

A RECEIPT frame is an acknowledgment that the corresponding client frame has been processed by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been received by the server. However, these previous frames may not yet be fully processed. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

OnActiveMQError: this event is fired if something goes wrong.

The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).

If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.

ERROR frames SHOULD include a content-length header and a content-type header if a body is present.

Properties

Authentication: disabled by default, if True a UserName and Password are sent to the server to try user authentication.

HeartBeat: Heart-beating can optionally be used to test the healthiness of the underlying TCP connection and to make sure that the remote end is alive and kicking. In order to enable heart-beating, each party has to declare what it can do and what it would like the other party to do. 0 means it cannot send/receive heart-beats, otherwise it is the desired number of milliseconds between heart-beats.

Options: The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.

Versions: Set which STOMP versions are supported.

AppRTC

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 TsgcWebSocketServer Component using Server Property.

Parameters

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

WebRTC

Protocol 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 plugins. The RTC in WebRTC stands for Real-Time Communications, a technology that enables audio/video streaming and data sharing between browser clients (peers). As a set of standards, WebRTC provides any browser with the ability to share application data and perform teleconferencing peer to peer, without the need to install plug-ins or third-party software.

WebRTC components are accessed with JavaScript APIs. Currently, in development are the Network Stream API, which represents an audio or video data stream, and the PeerConnection API, which allows two or more users to communicate browser-to-browser. Also under development is a DataChannel API that enables communication of other types of data for real-time gaming, text chat, file transfer, and so forth.

Components
TsgcWSPServer_WebRTC: Server Protocol WebRTC VCL Component.

Browser Test
If you want to test this protocol with your favourite Web Browser, please type this url (you need to define your custom host and port)

```
http://host:port/webrtc.esegece.com.html
```

TsgcWSPServer_WebRTC

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

There is no need to configure any parameter.

Protocol WebRTC Javascript

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script
src="http://www.example.com:80/webrtc.esegece.com.js"></script>
```

Open Connection
When a WebSocket connection is opened, 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.esegece.com.js"></script>
<script>
  var socket = new sgcws_webrtc('ws://{%host%}:{%port%}');
</script>
```

Open WebRTC Channel
When a browser has access to local camera and microphone, 'sgcmediastart' event is fired and then you can try to connect to another client using webrtc_connect procedure

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  var socket = new sgcws_webrtc('ws://{%host%}:{%port%}');
  socket.on('sgcmediastart', function(event)
  {
    socket.webrtc_connect('custom channel');
  }
</script>
```

Close WebRTC channel

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/webrtc.esegece.com.js"></script>
<script>
  socket.webrtc_disconnect('custom channel');
</script>
```

WAMP

Protocol WAMP

WAMP is an open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

Technically, WAMP is an officially registered WebSocket subprotocol (runs on top of WebSocket) that uses JSON as message serialization format.

What is RPC?

Remote Procedure Call (RPC) is a messaging pattern involving peers to two roles: client and server.
A server provides methods or procedure to call under well-known endpoints.
A client calls remote methods or procedures by providing the method or procedure endpoint and any arguments for the call.
The server will execute the method or procedure using the supplied arguments to the call and return the result of the call to the client.

What is PubSub?

Publish & Subscribe (PubSub) is a messaging pattern involving peers of three roles: publisher, subscriber and broker.
A publisher sends (publishes) an event by providing a topic (aka channel) as the abstract address, not a specific peer.
A subscriber receives events by first providing topics (aka channels) he is interested. Subsequently, the subscriber will receive any events publishes to that topic.
The broker sits between publishers and subscribers and mediates messages publishes to subscribers. A broker will maintain lists of subscribers per topic so it can dispatch new published events to the appropriate subscribers.
A broker may also dispatch events on its own, for example when the broker also acts as an RPC server and a method executed on the server should trigger a PubSub event.
In summary, PubSub decouples publishers and receivers via an intermediary, the broker.

Components
TsgcWSPServer_WAMP: Server Protocol WAMP VCL Component.

TsgcWSPClient_WAMP: Client Protocol WAMP VCL Component.

Javascript Component: Client Javascript Reference.

Browser Test
If you want to test this protocol with your favourite Web Browser, please type this URL(you need to define your custom host and port)

```
http://host:port/wamp.esegece.com.html
```

TsgcWSPServer_WAMP

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

Methods

CallResult: When the execution of the remote procedure finishes successfully, the server responds by sending a message with the result.

- **CallId:** this is the ID generated by client when request a call to a procedure
- **Result:** is the result, can be a number, a JSON object...

CallError: When the remote procedure call could not be executed, an error or exception occurred during the execution or the execution of the remote procedure finishes unsuccessfully for any other reason, the server responds by sending a message with error details.

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **ErrorURI:** identifies the error.
- **ErrorDesc:** error description.
- **ErrorDetails:** application error details, is optional.

Event: Subscribers receive PubSub events published by subscribers via the EVENT message.

- **TopicURI:** channel name where is subscribed.
- **Event:** message text.

Events

OnCall: event fired when the server receives RPC called by the client

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **ProcUri:** procedure identifier...
- **Arguments:** procedure params, can be a integer, a JSON object, a list...

OnPrefix: Procedures and Errors are identified using URIs or CURIEs, this event is fired when a client sends a new prefix

- **Prefix:** compact URI expression.
- **URI:** full URI.

TsgcWSPClient_WAMP

This is Client Protocol WAMP Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Methods

Prefix: Procedures and Errors are identified using URIs or CURIEs, the client uses this method to send a new prefix.

- **aPrefix:** compact URI expression.
- **aURI:** full URI.

Subscribe: A client requests access to a valid topicURI (or CURIE from Prefix) to receive events published to the given topicURI. The request is asynchronous, the server will not return an acknowledgement of the subscription.

- **aTopicURI:** channel name.

UnSubscribe: Calling unsubscribe on a topicURI informs the server to stop delivering messages to the client previously subscribed to that topicURI.

- **aTopicURI:** channel name.

Call: sent by the client when requests a Remote Procedure Call (RPC)

- **aCallId:** this is the UUID generated by client
- **aProcURI:** procedure identifier.
- **aArguments:** procedure params, can be a integer, a JSON object, a list...

Publish: The client will send an event to all clients connected to the server who have subscribed to the topicURI.

- **TopicURI:** channel name.
- **Event:** message text.

Events

OnWelcome: is the first server-to-client message sent by a WAMP server

- **SessionId:** is a string that is randomly generated by the server and unique to the specific WAMP session. The sessionId can be used for at least two situations: 1) specifying lists of excluded or eligible clients when publishing event and 2) in the context of performing authentication or authorization.
- **ProtocolVersion:** is an integer that gives the WAMP protocol version the server speaks, currently it MUST be 1.
- **ServerIdent:** is a string the server may use to disclose it's version, software, platform or identity.

OnCallError: event fired when the remote procedure call could not be executed, an error or exception occurred during the execution or the execution of the remote procedure finishes unsuccessfully for any other reason, the server responds by sending a message with error details

- **CallId:** this is the ID generated by the client when requesting a call to a procedure
- **ErrorURI:** identifies the error.
- **ErrorDesc:** error description.
- **ErrorDetails:** application error details, is optional.

OnCallResult: event fired when the execution of the remote procedure finishes successfully, the server responds by sending a message with the result.

- **CallId:** this is the ID generated by client when request a call to a procedure
- **Result:** is the result, can be a number, a JSON object...

OnEvent: event fired when the client receives PubSub events published by subscribers via the EVENT message.

- **TopicURI:** channel name where is subscribed.
- **Event:** message text.

Protocol WAMP Javascript

Here you can find available methods, you need to replace {%host%} and {%port%} variables as needed, example: if you have configured your sgcWebSocket server to listen port 80 on www.example.com website you need to configure:

```
<script src="http://www.example.com:80/sgcWebSockets.js"></script>
<script src="http://www.example.com:80/wamp.esegece.com.js"></script>
```

Open Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
</script>
```

Send New Prefix

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.prefix('sgc', 'http://www.esegece.com');
</script>
```

Request RPC (Remote Procedure Call)

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.call('', 'sgc:CallTest', '20')
</script>
```

Subscribe to a TopicURI

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.subscribe('sgc:test)
</script>
```

UnSubscribe to a TopicURI

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.unsubscribe('sgc:test)
</script>
```

Publish message

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.publish('sgc:channel', 'Test Message', [], []);
</script>
```

Show Alert with Message Received

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws('ws://{%host%}:{%port%}');
  socket.on('sgcmessage', function(event)
  {
    alert(event.message);
  }
</script>
```

Show Alert OnCallResult or OnCallError

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('wampcallresult', function(event)
  {
    alert('call result: ' + event.CallId + ' - ' + event.CallResult);
  }
  socket.on('wampcallerror', function(event)
  {
    alert('call error: ' + event.CallId + ' - ' + event.ErrorURI + ' -
' + event.ErrorDesc + ' - ' + event.ErrorDetails);
  }
</script>
```

Show Alert OnEvent

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('wampevent', function(event)
  {
    alert('call result: ' + event.TopicURI + ' - ' + event.Event);
  }
</script>
```

Show Alert OnConnect, OnDisconnect and OnError Events

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  var socket = new sgcws_wamp('ws://{%host%}:{%port%}');
  socket.on('open', function(event)
  {
    alert('sgcWebSocket Open!');
  };
  socket.on('close', function(event)
  {
    alert('sgcWebSocket Closed!');
  };
  socket.on('error', function(event)
  {
    alert('sgcWebSocket Error: ' + event.message);
  };
</script>
```

Close Connection

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.close();
</script>
```

Get Connection Status

```
<script src="http://{%host%}:{%port%}/sgcWebSockets.js"></script>
<script src="http://{%host%}:{%port%}/wamp.esegece.com.js"></script>
<script>
  socket.state();
</script>
```

WAMP 2.0

Protocol WAMP 2

WAMP provides Unified Application Routing in an open WebSocket protocol that works with different languages.

Using WAMP you can build distributed systems out of application components which are loosely coupled and communicate in (soft) real-time.

At its core, WAMP offers two communication patterns for application components to talk to each other:

- Publish & Subscribe (PubSub)

- Remote Procedure Calls (RPC)

WAMP is easy to use, simple to implement and based on modern Web standards: WebSocket, JSON and URIs.

Components
TsgcWSPClient_WAMP2: Client Protocol WAMP2 VCL Component.

TsgcWSPClient_WAMP2

This is Client Protocol WAMP Component, you need to drop this component in the form and select a TsgcWebSocketClient Component using Client Property.

Session Methods

- **ABORT:** Both the Router and the Client may abort the opening of a WAMP session by sending an ABORT message.

  Reason MUST be an URI.
  Details MUST be a dictionary that allows to provide additional, optional closing information (see below).

No response to an ABORT message is expected.

- **GOODBYE:** A WAMP session starts its lifetime with the Router sending a WELCOME message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a GOODBYE message sent by one Peer and a GOODBYE message sent from the other Peer in response.

  Reason MUST be a URI.
  Details MUST be a dictionary that allows providing additional, optional closing information.

Publish/Subscribe Methods

- **PUBLISH:** When a Publisher requests to publish an event to some topic, it sends a PUBLISH message to a Broker:

  Request is a random, ephemeral ID chosen by the Publisher and used to correlate the Broker's response with the request.
  Options is a dictionary that allows to provide additional publication request details in an extensible way. This is described further below.
  Topic is the topic published to.
  Arguments is a list of application-level event payload elements. The list may be of zero length.
  ArgumentsKw is an optional dictionary containing application-level event payload, provided as keyword arguments. The dictionary may be empty.

If the Broker is able to fulfil and allowing the publication, the Broker will send the event to all current Subscribers of the topic of the published event.
By default, publications are unacknowledged, and the Broker will not respond, whether the publication was successful indeed or not.

- **SUBSCRIBE:** A Subscriber communicates its interest in a topic to a Broker by sending a SUBSCRIBE message:

  Request MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
  Options MUST be a dictionary that allows providing additional subscription request details in an extensible way.

Topic is the topic the Subscriber wants to subscribe to and MUST be a URI.

•**UNSUBSCRIBE:** When a Subscriber is no longer interested in receiving events for a subscription it sends an UNSUBSCRIBE message

Request MUST be a random, ephemeral ID chosen by the Subscriber and used to correlate the Broker's response with the request.
SUBSCRIBED.Subscription MUST be the ID for the subscription to unsubscribe from, originally handed out by the Broker to the Subscriber.

RPC Methods

•**CALL:** When a Caller wishes to call a remote procedure, it sends a CALL message to a Dealer:

Request is a random, ephemeral ID chosen by the Caller and used to correlate the Dealer's response with the request.
Options is a dictionary that allows to provide additional call request details in an extensible way. This is described further below.
Procedure is the URI of the procedure to be called.
Arguments is a list of positional call arguments (each of arbitrary type). The list may be of zero length.
ArgumentsKw is a dictionary of keyword call arguments (each of arbitrary type). The dictionary may be empty.

•**REGISTERCALL:** A Callee announces the availability of an endpoint implementing a procedure with a Dealer by sending a REGISTER message:

Request is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
Options is a dictionary that allows providing additional registration request details in a extensible way. This is described further below.
Procedure is the procedure the Callee wants to register

•**UNREGISTERCALL:** When a Callee is no longer willing to provide an implementation of the registered procedure, it sends an UNREGISTER message to the Dealer:

Request is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
REGISTERED.Registration is the ID for the registration to revoke, originally handed out by the Dealer to the Callee.

•**INVOCATION:** If the Dealer is able to fulfil (mediate) the call and it allows the call, it sends a INVOCATION message to the respective Callee implementing the procedure:

Request is a random, ephemeral ID chosen by the Dealer and used to correlate the Callee's response with the request.
REGISTERED.Registration is the registration ID under which the procedure was registered at the Dealer.
Details is a dictionary that allows to provide additional invocation request details in an extensible way. This is described further below.

CALL.Arguments is the original list of positional call arguments as provided by the Caller.
CALL.ArgumentsKw is the original dictionary of keyword call arguments as provided by the Caller.

- **YIELD:** If the Callee is able to successfully process and finish the execution of the call, it answers by sending a YIELD message to the Dealer:

INVOCATION.Request is the ID from the original invocation request.
Options is a dictionary that allows providing additional options.
Arguments is a list of positional result elements (each of arbitrary type). The list may be of zero length.
ArgumentsKw is a dictionary of keyword result elements (each of arbitrary type). The dictionary may be empty.

Events

OnWAMPSession: After the underlying transport has been established, the opening of a WAMP session is initiated by the Client sending a HELLO message to the Router

- **Realm:** is a string identifying the realm this session should attach to
- **Details:** is a dictionary that allows to provide additional opening information

OnWAMPWelcome: A Router completes the opening of a WAMP session by sending a WELCOME reply message to the Client.

- **Session:** MUST be a randomly generated ID specific to the WAMP session. This applies for the lifetime of the session.
- **Details:** is a dictionary that allows to provide additional information regarding the open session.

OnWAMPAbort: Both the Router and the Client may abort the opening of a WAMP session by sending an ABORT message.

- **Reason:** MUST be an URI.
- **Details:** MUST be a dictionary that allows providing additional, optional closing information.

OnWAMPGoodBye: A WAMP session starts its lifetime with the Router sending a WELCOME message to the Client and ends when the underlying transport disappears or when the WAMP session is closed explicitly by a GOODBYE message sent by one Peer and a GOODBYE message sent from the other Peer in response.

- **Reason:** MUST be an URI.
- **Details:** MUST be a dictionary that allows to provide additional, optional closing information.

OnWAMPSubscribed: If the Broker is able to fulfill and allow the subscription, it answers by sending a SUBSCRIBED message to the Subscriber

- **SUBSCRIBE.Request:** MUST be the ID from the original request.
- **Subscription:** MUST be an ID chosen by the Broker for the subscription.

OnWAMPUnSubscribed: Upon successful unsubscription, the Broker sends an UNSUBSCRIBED message to the Subscriber

- **UNSUBSCRIBE.Request:** MUST be the ID from the original request.

OnWAMPPublished: If the Broker is able to fulfill and allowing the publication, and PUBLISH.Options.acknowledge == true, the Broker replies by sending a PUBLISHED message to the Publisher:

- **PUBLISH.Request:** is the ID from the original publication request.
- **Publication:** is a ID chosen by the Broker for the publication.

OnWAMPEvent: When a publication is successful and a Broker dispatches the event, it determines a list of receivers for the event based on Subscribers for the topic published to and, possibly, other information in the event. Note that the Publisher of an event will never receive the published event even if the Publisher is also a Subscriber of the topic published to. The Advanced Profile provides options for more detailed control over publication. When a Subscriber is deemed to be a receiver, the Broker sends the Subscriber an EVENT message.

- **SUBSCRIBED.Subscription:** is the ID for the subscription under which the Subscriber receives the event - the ID for the subscription originally handed out by the Broker to the Subscribe*.
- **PUBLISHED.Publication:** is the ID of the publication of the published event.
- **DETAILS:** is a dictionary that allows the Broker to provide additional event details in a extensible way.
- **PUBLISH.Arguments:** is the application-level event payload that was provided with the original publication request.
- **PUBLISH.ArgumentKw:** is the application-level event payload that was provided with the original publication request.

OnWAMPError: When the request fails, the Broker sends an ERROR

- **METHOD:** is the ID of the Method.
- **REQUEST.ID:** is the ID of the Request.
- **DETAILS:** is a dictionary that allows the Broker to provide additional event details in a extensible way.
- **ERROR:** describes the message error.
- **PUBLISH.Arguments:** is the application-level event payload that was provided with the original publication request.
- **PUBLISH.ArgumentKw:** is the application-level event payload that was provided with the original publication request.

OnWAMPResult: The Dealer will then send a RESULT message to the original Caller:

- **CALL.Request:** is the ID from the original call request.
- **DETAILS:** is a dictionary of additional details.

- **YIELD.Arguments:** is the original list of positional result elements as returned by the Callee.
- **YIELD.ArgumentsKw:** is the original dictionary of keyword result elements as returned by the Callee.

OnWAMPRegistered: If the Dealer is able to fulfill and allowing the registration, it answers by sending a REGISTERED message to the Callee:

- **REGISTER.Request:** is the ID from the original request.
- **Registration:** is an ID chosen by the Dealer for the registration.

OnWAMPUnRegistered: When a Callee is no longer willing to provide an implementation of the registered procedure, it sends an UNREGISTER message to the Dealer:

- **Request:** is a random, ephemeral ID chosen by the Callee and used to correlate the Dealer's response with the request.
- **REGISTERED.Registration:** is the ID for the registration to revoke, originally handed out by the Dealer to the Callee.

APIs

WebSocket APIs

There are several implementations based on WebSockets: finance, message publishing, queues... sgcWebSockets implements the most important APIs based on WebSocket protocol. In order to use an API, just attach API component to the client and all messages will be handled by API component (only one API component can be attached to a client).

List of APIs:

1. SocketIO: is a JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers.

2. Blockchain: Blockchain WebSocket API allows developers to receive Real-Time notifications about new transactions and blocks.

3. Bitfinex: Bitfinex is one of the world's largest and most advanced cryptocurrency trading platform. Users can exchange Bitcoin, Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Classic...

4. Pusher: Pusher is an easy and reliable platform with flexible pub/sub messaging, live user lists, authentication...

5. SignalR: is a library for ASP.NET developers that makes developing real-time web functionality using WebSockets as transport.

6. Bittrex: is a US-based cryptocurrency exchange.

7. Binance: is an international multi-language cryptocurrency exchange.

8. Bitstamp: Bitstamp is one of the world's longest standing crypto exchange, supporting the blockchain ecosystem since 2011.

9. Huobi: is an international multi-language cryptocurrency exchange.

10. Cex: is a cryptocurrency exchange and former Bitcoin cloud mining provider.

11. Bitmex: s a cryptocurrency exchange and derivative trading platform.

12. SignalRCore: ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps.

APIs can be registered at **runtime**, just call Method **RegisterAPI** and pass API component as a parameter.

API

API SocketIO

SocketIO

Socket.IO is a JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical API. Like Node.js, it is event-driven.

Messages Types

0: open (Sent from the server when a new transport is opened (recheck))

1: close (Request the close of this transport but does not shut down the connection itself.)

2: ping (Sent by the client. The server should answer with a pong packet containing the same data)
example
client sends: 2probe
server sends: 3probe

3: pong (Sent by the server to respond to ping packets.)

4: string message (actual message, client and server should call their callbacks with the data.)
example:
42/chat,[“join”,”{room:1{"]
4 is the message packet type in the engine.io protocol
2 is the EVENT type in the socket.io protocol
/chat is the data which is processed by socket.io
socket.io will fire the “join” event
will pass "room: 1" data. It is possible to omit namespace only when it is /.

5: upgrade (Before engine.io switches a transport, it tests, if server and client can communicate over this transport. If this test succeeds, the client sends an upgrade packets which requests the server to flush its cache on the old transport and switch to the new transport.)

6: noop (A noop packet. Used primarily to force a poll cycle when an incoming WebSocket connection is received.)

Properties

API: specifies SocketIO version:

ioAPI0: supports socket.io 0.* servers (selected by default)

ioAPI1: supports socket.io 1.* servers

ioAPI2: supports socket.io 2.* servers

Base64: if enabled, binary messages are received as base64.

HandShakeCustomURL: allows customizing URLl 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).

Namespace: allows setting a namespace when connects to the server.

Events

OnHTTPConnectionSSL

When a WebSocket server requires secure connections, you can get an error message like this when a client tries to connect to server:

Error connecting with SSL. error:XXXXXXXX:SSL routines:ssl3_read_bytes:tlsv1 alert protocol version

This error means that your client is trying to connect using a TLS version which is not supported by the server. To resolve this error you must handle OnSSLAfterCreateHandler of WebSocket client component and set a newer TLS version. For example: here we are setting TLS 1.2 as a protocol version.

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

API Blockchain

Blockchain

Blockchain WebSocket API allows developers to receive Real-Time notifications about new transactions and blocks.

Once WebSocket is open you can subscribe to a channel:

- **SubscribeTransactions:** Subscribe to notifications for all new bitcoin transactions.
- **UnsubscribeTransactions:** UnSubscribe to notifications for all new bitcoin transactions.

- **SubscribeAddress:** Receive new transactions for a specific bitcoin address.
- **UnSubscribeAddress:** Stop receiving new transactions for a specific bitcoin address.

Transactions are received **OnNewTransaction** Event:


```
{
    "op": "utx",
    "x": {
        "lock_time": 0,
        "ver": 1,
        "size": 192,
        "inputs": [
            {
                "sequence": 4294967295,
                "prev_out": {
                    "spent": true,
                    "tx_index": 99005468,
                    "type": 0,
                    "addr": "1BwGf3z7n2fHk6NoVJNkV32qwyAYsMhkWf",
                    "value": 65574000,
                    "n": 0,
                    "script":
"76a91477f4c9ee75e449a74c21a4decfb50519cbc245b388ac"
                },
                "script":
"483045022100e4ff962c292705f051c2c2fc519fa775a4d8955bce1a3e29884b27852
77999ed02200b537ebd22a9f25fbbbcc9113c69c1389400703ef2017d80959ef0f1d68
5756c012102618e08e0c8fd4c5fe539184a30fe35a2f5fccf7ad62054cad29360d871f
8187d"
            }
        ],
        "time": 1440086763,
        "tx_index": 99006637,
        "vin_sz": 1,
        "hash":
"0857b9de1884eec314ecf67c040a2657b8e083e1f95e31d0b5ba3d328841fc7f",
        "vout_sz": 1,
        "relayed_by": "127.0.0.1",
        "out": [
            {
                "spent": false,
                "tx_index": 99006637,
                "type": 0,
                "addr": "1A828tTnkVFJfSvLCqF42ohZ51ksS3jJgX",
```

```
                "value": 65564000,
                "n": 0,
                "script":
"76a914640cfdf7b79d94d1c980133e3587bd6053f091f388ac"
            }
        ]
    }
}
```

- **SubscribeBlocks:** Receive notifications when a new block is found. Note: if the chain splits you will receive more than one notification for a specific block height.
- **UnSubscribeBlocks:** Stop receiving notifications when a new block is found. Note: if the chain splits you will receive more than one notification for a specific block height.

Blocks are received **OnNewBlock** event:

```
{
    "op": "block",
    "x": {
        "txIndexes": [
            3187871,
            3187868
        ],
        "nTx": 0,
        "totalBTCSent": 0,
        "estimatedBTCSent": 0,
        "reward": 0,
        "size": 0,
        "blockIndex": 190460,
        "prevBlockIndex": 190457,
        "height": 170359,
        "hash":
"000000000000006436073c07dfa188a8fa54fefadf571fd774863cda1b884b90f",
        "mrklRoot":
"94e51495e0e8a0c3b78dac1220b2f35ceda8799b0a20cfa68601ed28126cfcc2",
        "version": 1,
        "time": 1331301261,
        "bits": 436942092,
        "nonce": 758889471
    }
}
```

API Bitfinex

Bitfinex

Bitfinex is one of the world's largest and most advanced cryptocurrency trading platform. Users can exchange Bitcoin, Ethereum, Ripple, EOS, Bitcoin Cash, Iota, NEO, Litecoin, Ethereum Classic...

Bitfinex WebSocket API version is 2.0

Each message sent and received via the Bitfinex's WebSocket channel is encoded in JSON format

A symbol can be a trading pair or a margin currency:

- Trading pairs symbols are formed prepending a "t" before the pair (i.e tBTCUSD, tETHUSD).
- Margin currencies symbols are formed prepending an "f" before the currency (i.e fUSD, fBTC, ...)

After a successful connection, **OnBitfinexConnect** event is raised and you get Bitfinex API Version number as a parameter.

You can call **Ping** method to test connection to the server.

If the server sends any information, this can be handle using **OnBitfinexInfoMessage** event, where a Code and a Message are parameters with information about the message sent by the server. Example codes:

20051 : Stop/Restart WebSocket Server (please reconnect)
20060 : Entering in Maintenance mode. Please pause any activity and resume after receiving the info message 20061 (it should take 120 seconds at most).
20061 : Maintenance ended. You can resume normal activity. It is advised to unsubscribe/subscribe again all channels.

In case of error, **OnBitfinexError** will be raised, and information about error provided. Example error codes:

10000 : Unknown event
10001 : Unknown pair

In order to change the configuration, call **Configuration** method and pass as a parameter one of the following flags:

CS_DEC_S = 8; // Enable all decimal as strings.
CS_TIME_S = 32; // Enable all times as date strings.
CS_SEQ_ALL = 65536; // Enable sequencing BETA FEATURE
CHECKSUM = 131072; // Enable checksum for every book iteration. Checks the top 25 entries for each side of the book. The checksum is a signed int.

Subscribe Public Channels

There are channels which are public and there is no need to authenticate against the server. All messages are raised **OnBitfinexUpdate** event.

SubscribeTicker

The ticker is a high level overview of the state of the market. It shows you the current best bid and ask, as well as the last trade price. It also includes information such as daily volume and how much the price has moved over the last day.

```
// Trading pairs
[
  CHANNEL_ID,
  [
    BID,
    BID_SIZE,
    ASK,
    ASK_SIZE,
    DAILY_CHANGE,
    DAILY_CHANGE_PERC,
    LAST_PRICE,
    VOLUME,
    HIGH,
    LOW
  ]
]
// Funding pairs
[
  CHANNEL_ID,
  [
    FRR,
    BID,
    BID_PERIOD,
    BID_SIZE,
    ASK,
    ASK_PERIOD,
    ASK_SIZE,
    DAILY_CHANGE,
    DAILY_CHANGE_PERC,
    LAST_PRICE,
    VOLUME,
    HIGH,
    LOW
  ]
]
```

SubscribeTrades

This channel sends a trade message whenever a trade occurs at Bitfinex. It includes all the pertinent details of the trade, such as price, size and time.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      ID,
      MTS,
      AMOUNT,
      PRICE
    ],
    ...
  ]
]
```

```
// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      ID,
      MTS,
      AMOUNT,
      RATE,
      PERIOD
    ],
    ...
  ]
]
```

SubscribeOrderBook

The Order Books channel allows you to keep track of the state of the Bitfinex order book. It is provided on a price aggregated basis, with customizable precision. After receiving the response, you will receive a snapshot of the book, followed by updates upon any changes to the book.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      PRICE,
      COUNT,
      AMOUNT
    ],
    ...
  ]
]

// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      RATE,
      PERIOD,
      COUNT,
      AMOUNT
    ],
    ...
  ]
]
```

SubscribeRawOrderBook

These are the most granular books.

```
// on trading pairs (ex. tBTCUSD)
[
  CHANNEL_ID,
  [
    [
      ORDER_ID,
```

```
      PRICE,
      AMOUNT
    ],
    ...
  ]
]

// on funding currencies (ex. fUSD)
[
  CHANNEL_ID,
  [
    [
      OFFER_ID,
      PERIOD,
      RATE,
      AMOUNT
    ],
    ...
  ]
]
```

SubscribeCandles

Provides a way to access charting candle info. Time Frames:

```
        1m: one minute
        5m : five minutes
        15m : 15 minutes
        30m : 30 minutes
        1h : one hour
        3h : 3 hours
        6h : 6 hours
        12h : 12 hours
        1D : one day
        7D : one week
        14D : two weeks
        1M : one month

[
  CHANNEL_ID,
  [
    [
      MTS,
      OPEN,
      CLOSE,
      HIGH,
      LOW,
      VOLUME
    ],
    ...
  ]
]
```

Subscribe Authenticated Channels

This channel allows you to keep up to date with the status of your account. You can receive updates on your positions, your balances, your orders and your trades.

Use **Authenticate** method in order to Authenticate against the server and set required parameters.

Once authenticated, you will receive updates of: Orders, positions, trades, funding offers, funding credits, funding loans, wallets, balance info, margin info, funding info, funding trades...

You can request **UnAuthenticate** method if you want to log off from the server.

API Pusher

Pusher

Pusher it's an easy and reliable platform with nice features based on WebSocket protocol: flexible pub/sub messaging, live user lists (presence), authentication...

Pusher WebSocket API is 7.

Data is sent bi-directionally over a WebSocket as text data containing UTF8 encoded JSON (Binary WebSocket frames are not supported).

You can call **Ping** method to test connection to the server. Essentially any messages received from the other party are considered to mean that the connection is alive. In the absence of any messages, either party may check that the other side is responding by sending a ping message, to which the other party should respond with a pong.

Before you connect, you must complete the following fields:

```
Pusher.Cluster := 'eu'; // cluster where is located your pusher account
Pusher.Key := '9c3b7ef25qe97a00116c'; // your pusher api key
Pusher.Name := 'js'; // optional, name of your application
Pusher.Version := '4.1';  // optional, version of your application
Pusher.TLS := True; // if encrypted, set to True
Pusher.Secret := '2dc792e1916ac49e6b3f'; // pusher secret string (needed for private and absence channels)
```

After a successful connection, **OnPusherConnect** event is raised and you get following fields:

- Socket ID: A unique identifier for the connected client.
- Timeout: The number of seconds of server inactivity after which the client should initiate a ping message (this is handled automatically by component).

In case of error, **OnPusherError** will be raised, and information about error provided. An error may be sent from Pusher in response to invalid authentication, an invalid command, etc.

4000-4099

Indicates an error resulting in the connection being closed by Pusher, and that attempting to reconnect using the same parameters will not succeed.

4000: Application only accepts SSL connections, reconnect using wss://
4001: Application does not exist
4003: Application disabled
4004: Application is over connection quota
4005: Path not found
4006: Invalid version string format
4007: Unsupported protocol version
4008: No protocol version supplied

4100-4199

Indicates an error resulting in the connection being closed by Pusher, and that the client may reconnect after 1s or more.

4100: Over capacity

4200-4299

Indicates an error resulting in the connection being closed by Pusher, and that the client may reconnect immediately.

4200: Generic reconnect immediately
4201: Pong reply not received: ping was sent to the client, but no reply was received - see ping and pong messages
4202: Closed after inactivity: The client has been inactive for a long time (currently 24 hours) and client does not support ping. Please upgrade to a newer WebSocket draft or implement version 5 or above of this protocol.

4300-4399

Any other type of error.

4301: Client event rejected due to rate limit

Channels

Channels are a fundamental concept in Pusher. Each application has a number of channels, and each client can choose which channels it subscribes to.

Channels provide:

- A way of filtering data. For example, in a chat application, there may be a channel for people who want to discuss ‘dogs’
- A way of controlling access to different streams of information. For example, a project management application would want to authorise people to get updates about ‘projectX’

It's strongly recommended that channels are used to filter your data and that it is not achieved using events. This is because all events published to a channel are sent to all subscribers, regardless of their event binding.

Channels don’t need to be explicitly created and are instantiated on client demand. This means that creating a channel is easy. Just tell a client to subscribe to it.

There are 3 types of channels:

- **Public channels** can be subscribed to by anyone who knows their name
- **Private channels** introduce a mechanism which lets your server control access to the data you are broadcasting
- **Presence channels** are an extension of private channels. They let you ‘register’ user information on subscription, and let other members of the channel know who’s online

Public Channels
Public channels should be used for publicly accessible data as they do not require any form authorisation in order to be subscribed to.

You can subscribe and unsubscribe from channels at any time. There’s no need to wait for the Pusher to finish connecting first.

Example: subscribe to channel "my-channel".

```
APIPusher.Subscribe('my-channel');
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.

All messages from the subscribed channel will be received **OnPusherEvent** event.

Private Channels
Requires Indy 10.5.7 or later

Private channels should be used when access to the channel needs to be restricted in some way. In order for a user to subscribe to a private channel permission must be authorised.

Example: subscribe to channel "my-private-channel".

```
  APIPusher.Subscribe('my-private-channel', pscPrivateChannel);
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.

All messages from the subscribed channel will be received **OnPusherEvent** event.

Presence Channels
Requires Indy 10.5.7 or later

Presence channels build on the security of Private channels and expose the additional feature of an awareness of who is subscribed to that channel. This makes it extremely easy to build chat room and “who’s online” type functionality to your application. Think chat rooms, collaborators on a document, people viewing the same web page, competitors in a game, that kind of thing.

Presence channels are subscribed to from the client API in the same way as private channels but the channel name must be prefixed with presence-. As with private channels an HTTP Request is made to a configurable authentication URL to determine if the current user has permissions to access the channel.

Information on users subscribing to, and unsubscribing from a channel can then be accessed by binding to events on the presence channel and the current state of users subscribed to the channel is available via the channel.members property.

Example: subscribe to channel "my-presence-channel".

```
  APIPusher.Subscribe('my-presence-channel', pscPresenceChannel,
'{"user_id":"John_Smith","user_info":{"name":"John Smith"}}');
```

If you are subscribed successfully **OnPusherSubscribe** event will be raised, if there is an error you will get a message in **OnPusherError** event.

All messages from the subscribed channel will be received **OnPusherEvent** event.

Publish Messages

Not only you can receive messages from subscribed channels, but you can also send messages to other subscribed users.
Call method **Publish** to send a message to all subscribed users of channel.

Example: send an event to all subscribed users of "my-channel'

```
  APIPusher.Publish('my-event', 'my-channel');
```

Publish no more than 10 messages per second per client (connection). Any events triggered above this rate limit will be rejected by Pusher API. This is not a system issue, it is a client issue. 100 clients in a channel sending messages at this rate would each also have to be processing 1,000 messages per second! Whilst some modern browsers might be able to handle this it’s most probably not a good idea.

API SignalR

SignalR

SignalR component uses WebSocket as transport to connect to a SignalR server, if this transport is not supported, an error will be raised.
SignalR client component has a property called SignalR where you can set following data:

- **Hubs:** contains a list of hubs the client is subscribing to.
- **ProtocolVersion:** the version of the protocol used by the client, supports protocol versions from 1.2 to 1.5
- **UserAgent:** user agent used to connect to SignalR server.

Hubs Messages

Hubs API makes it possible to invoke server methods from the client and client methods from the server. The protocol used for persistent connection is not rich enough to allow expressing RPC (remote procedure call) semantics. It does not mean however that the protocol used for hub connections is completely different from the protocol used for persistent connections. Rather, the protocol used for hub connections is mostly an extension of the protocol for persistent connections.

When a client invokes a server method it no longer sends a free-flow string as it was for persistent connections. Instead, it sends a JSON string containing all necessary information needed to invoke the method. Here is a sample message a client would send to invoke a server method:

```
WriteData('{"H":"chathub","M":"Send","A":["Delphi Client","Test
message"],"I":0}');
```

The payload has the following properties:
I – invocation identifier – allows to match up responses with requests
H – the name of the hub
M – the name of the method
A – arguments (an array, can be empty if the method does not have any parameters)

The component has the following events:

OnSignalRConnect

When the client connects successfully to the server, this event is raised.

OnSignalRDisconnect

When the client is disconnected from the server, this event is raised.

OnSignalRError

When there is an error in WebSocket connection.

OnSignalRMessage

The protocol used for persistent connection is quite simple. Messages sent to the server are just raw strings. There isn't any specific format they have to be in. Messages sent to the client are more structured. The properties you can find in the message are as follows:

C – message id, present for all non-KeepAlive messages
M – an array containing actual data.

```
{"C":"d-9B7A6976-B,2|C,2","M":["Welcome!"]}
```

OnSignalRResult

When a server method is invoked the server returns a confirmation that the invocation has completed by sending the invocation id to the client and – if the method returned a value – the return value, or – if invoking the method failed – the error.
Here are sample results of a server method call:

{"I":"0"}

A server void method whose invocation identifier was "0" completed successfully.

"{"I":"0", "R":42}

A server method returning a number whose invocation identifier was "0" completed successfully and returned the value 42.

{"I":"0", "E":"Error occurred"}

OnSignalRKeepAlive

This event is raised when a KeepAlive message is received from the server.

API Bittrex

Bittrex

The Bittrex WebSocket API is implemented using **SignalR**.
Bittrex component has a property called Bittrex where you can fill API Keys provided by Bittrex to get access to your account data.

Authenticate

Verifies a user's identity to the server and begins receiving account-level notifications. Needs an ApiKey and ApiSecret, which can be obtained in Bittrex user page.
Updates are received OnBittrexMessage event, where Callback is "OrderDelta" or "BalanceDelta".
JSON Order Response

```
{
    AccountUuid : Guid,
    Nonce       : int,
    Type        : int,
    Order:
    {
        Uuid              : guid,
        Id                : long,
        OrderUuid         : guid,
        Exchange          : string,
        OrderType         : string,
        Quantity          : decimal,
        QuantityRemaining : decimal,
        Limit             : decimal,
        CommissionPaid    : decimal,
        Price             : decimal,
        PricePerUnit      : decimal,
        Opened            : date,
        Closed            : date,
        IsOpen            : bool,
        CancelInitiated   : bool,
        ImmediateOrCancel : bool,
        IsConditional     : bool,
        Condition         : string,
        ConditionTarget   : decimal,
        Updated           : date
    }
}
```

JSON Balance Response

```
{
    Nonce : int,
    Delta :
    {
        Uuid          : guid,
        AccountId     : int,
        Currency      : string,
        Balance       : decimal,
        Available     : decimal,
        Pending       : decimal,
        CryptoAddress : string,
```

```
        Requested     : bool,
        Updated       : date,
        AutoSell      : bool
    }
}
```

QueryExchangeState

Allows the caller to retrieve the full order book for a specific market. There is only one Parameter called MarketName, e.g. BTC-ETH
Updates are received **OnBittrexMessage** event, where Callback is "QueryExchangeState"
JSON Response

```
{
    MarketName : string,
    Nonce      : int,
    Buys:
    [
        {
            Quantity : decimal,
            Rate     : decimal
        }
    ],
    Sells:
    [
        {
            Quantity : decimal,
            Rate     : decimal
        }
    ],
    Fills:
    [
        {
            Id        : int,
            TimeStamp : date,
            Quantity  : decimal,
            Price     : decimal,
            Total     : decimal,
            FillType  : string,
            OrderType : string
        }
    ]
}
```

QuerySummaryState

Allows the caller to retrieve the full state for all markets.
Updates are received **OnBittrexMessage** event, where Callback is "QuerySummaryState"
JSON Response

```
{
    Nonce     : int,
    Summaries :
    [
        {
            MarketName     : string,
            High           : decimal,
```

```
            Low            : decimal,
            Volume         : decimal,
            Last           : decimal,
            BaseVolume     : decimal,
            TimeStamp      : date,
            Bid            : decimal,
            Ask            : decimal,
            OpenBuyOrders  : int,
            OpenSellOrders : int,
            PrevDay        : decimal,
            Created        : date
        }
    ]
}
```

SubscribeToExchangeDeltas

Allows the caller to receive real-time updates to the state of a single market. There is only one Parameter called MarketName, e.g. BTC-ETH
If user is subscribed to the feed, **OnBittrexSubscribed** event will be raised.
Updates are received **OnBittrexMessage** event, where Callback is "MarketDelta"
JSON Response

```
{
    MarketName : string,
    Nonce      : int,
    Buys:
    [
        {
            Type     : int,
            Rate     : decimal,
            Quantity : decimal
        }
    ],
    Sells:
    [
        {
            Type     : int,
            Rate     : decimal,
            Quantity : decimal
        }
    ],
    Fills:
    [
        {
            FillId    : int,
            OrderType : string,
            Rate      : decimal,
            Quantity  : decimal,
            TimeStamp : date
        }
    ]
}
```

SubscribeToSummaryDeltas

Allows the caller to receive real-time updates of the state of all markets.
If the user is subscribed to the feed, **OnBittrexSubscribed** event will be raised.
Updates are received **OnBittrexMessage** event, where Callback is "SummaryDelta"
JSON Response

```
{
    Nonce : int,
    Deltas :
    [
        {
            MarketName     : string,
            High           : decimal,
            Low            : decimal,
            Volume         : decimal,
            Last           : decimal,
            BaseVolume     : decimal,
            TimeStamp      : date,
            Bid            : decimal,
            Ask            : decimal,
            OpenBuyOrders  : int,
            OpenSellOrders : int,
            PrevDay        : decimal,
            Created        : date
        }
    ]
}
```

SubscribeToSummaryLiteDeltas

Allows the caller to receive real-time updates of the state of all markets but only *market name, the last price, and the base currency*
If user is subscribed to the feed, **OnBittrexSubscribed** event will be raised.
Updates are received **OnBittrexMessage** event, where Callback is "LiteSummaryDelta"
JSON Response

```
{
    Deltas :
    [
        {
            MarketName : string,
            Last       : decimal,
            BaseVolume : decimal
        }
    ]
}
```

API Binance

Binance

Binance is an international multi-language cryptocurrency exchange.

Properties

Streams: contains a list of streams.

Methods

AddAggregateTrades

The Aggregate Trade Streams push trade information that is aggregated for a single taker order.

```
{
  "e": "aggTrade",  // Event type
  "E": 123456789,   // Event time
  "s": "BNBBTC",    // Symbol
  "a": 12345,       // Aggregate trade ID
  "p": "0.001",     // Price
  "q": "100",       // Quantity
  "f": 100,         // First trade ID
  "l": 105,         // Last trade ID
  "T": 123456785,   // Trade time
  "m": true,        // Is the buyer the market maker?
  "M": true         // Ignore.
}
```

AddTrades

The Trade Streams push raw trade information; each trade has a unique buyer and seller.

```
{
  "e": "trade",     // Event type
  "E": 123456789,   // Event time
  "s": "BNBBTC",    // Symbol
  "t": 12345,       // Trade ID
  "p": "0.001",     // Price
  "q": "100",       // Quantity
  "b": 88,          // Buyer order Id
  "a": 50,          // Seller order Id
  "T": 123456785,   // Trade time
  "m": true,        // Is the buyer the market maker?
  "M": true         // Ignore.
}
```

AddKline

The Kline/Candlestick Stream push updates to the current klines/candlestick every second.

```
{
  "e": "kline",     // Event type
```

```
      "E": 123456789,   // Event time
      "s": "BNBBTC",    // Symbol
      "k": {
        "t": 123400000, // Kline start time
        "T": 123460000, // Kline close time
        "s": "BNBBTC",  // Symbol
        "i": "1m",      // Interval
        "f": 100,       // First trade ID
        "L": 200,       // Last trade ID
        "o": "0.0010",  // Open price
        "c": "0.0020",  // Close price
        "h": "0.0025",  // High price
        "l": "0.0015",  // Low price
        "v": "1000",    // Base asset volume
        "n": 100,       // Number of trades
        "x": false,     // Is this kline closed?
        "q": "1.0000",  // Quote asset volume
        "V": "500",     // Taker buy base asset volume
        "Q": "0.500",   // Taker buy quote asset volume
        "B": "123456"   // Ignore
      }
}
```

AddTicker

24hr Ticker statistics for a single symbol pushed every second

```
    {
      "e": "24hrTicker",  // Event type
      "E": 123456789,     // Event time
      "s": "BNBBTC",      // Symbol
      "p": "0.0015",      // Price change
      "P": "250.00",      // Price change percent
      "w": "0.0018",      // Weighted average price
      "x": "0.0009",      // Previous day's close price
      "c": "0.0025",      // Current day's close price
      "Q": "10",          // Close trade's quantity
      "b": "0.0024",      // Best bid price
      "B": "10",          // Best bid quantity
      "a": "0.0026",      // Best ask price
      "A": "100",         // Best ask quantity
      "o": "0.0010",      // Open price
      "h": "0.0025",      // High price
      "l": "0.0010",      // Low price
      "v": "10000",       // Total traded base asset volume
      "q": "18",          // Total traded quote asset volume
      "O": 0,             // Statistics open time
      "C": 86400000,      // Statistics close time
      "F": 0,             // First trade ID
      "L": 18150,         // Last trade Id
      "n": 18151          // Total number of trades
    }
```

AddAllMarketTickers

24hr Ticker statistics for all symbols in an array pushed every second

```
    [
      {
        // Same as <symbol>@ticker payload
      }
```

```
]
```

AddPartialBookDepth

Top <levels> bids and asks, pushed every second. Valid <levels> are 5, 10, or 20.

```
{
  "lastUpdateId": 160,  // Last update ID
  "bids": [             // Bids to be updated
    [
      "0.0024",         // price level to be updated
      "10",             // quantity
      []                // ignore
    ]
  ],
  "asks": [             // Asks to be updated
    [
      "0.0026",         // price level to be updated
      "100",            // quantity
      []                // ignore
    ]
  ]
}
```

AddDiffDepth

Order book price and quantity depth updates used to locally manage an order book pushed every second.

```
{
  "e": "depthUpdate", // Event type
  "E": 123456789,     // Event time
  "s": "BNBBTC",      // Symbol
  "U": 157,           // First update ID in event
  "u": 160,           // Final update ID in event
  "b": [              // Bids to be updated
    [
      "0.0024",       // price level to be updated
      "10",
      []              // ignore
    ]
  ],
  "a": [              // Asks to be updated
    [
      "0.0026",       // price level to be updated
      "100",          // quantity
      []              // ignore
    ]
  ]
}
```

API Bitstamp

Bitstamp

Bitstamp is a bitcoin exchange based in Luxembourg. It allows trading between USD currency and bitcoin cryptocurrency. It allows USD, EUR, bitcoin, litecoin, ethereum, ripple or bitcoin cash deposits and withdrawals.
Bitstamp API inherits from Pusher component.

Methods

- **SubscribeTicker:** get live trades from currency selected (if no currency, by default is BTC/USD). JSON data:

id	*Trade unique ID.*
amount	*Trade amount.*
price	*Trade price.*
type	*Trade type (0 - buy; 1 - sell).*
timestamp	*Trade timestamp.*
buy_order_id	*Trade buy order id.*
sell_order_id	*Trade sell order id.*

- **SubscribeOrderBook:** get order book from currency selected (if no currency, by default is BTC/USD). JSON data:

bids	*List of tops 100 bids.*
asks	*List of top 100 asks.*
timestamp	*Order book timestamp.*

- **SubscribeFullOrderBook:** get full order book from currency selected (if no currency, by default is BTC/USD). JSON data:

bids	*List of changed bids since the last broadcast.*
asks	*List of asks since the last broadcast.*

- **SubscribeOrders:** get live orders from currency selected (if no currency, by default is BTC/USD). JSON data:

id	*Order ID.*
amount	*Order amount.*
price	*Order price.*
order_type	*Order type (0 - buy; 1 - sell).*
datetime	*Order datetime.*

API Huobi

Huobi

Huobi is an international multi-language cryptocurrency exchange.

Methods

SubscribeKLine

K line data, including the opening price, closing price, highest price, lowest price, volume, turnover, number of transactions, etc. per unit time interval $period Optional values: { 1min, 5min, 15min, 30min, 60min, 4hour ,1day, 1mon, 1week, 1year }

```
{
  "status": "ok",
  "rep": "market.btcusdt.kline.1min",
  "tick": [
    {
      "amount": 1.6206,
      "count":  3,
      "id":     1494465840,
      "open":   9887.00,
      "close":  9885.00,
      "low":    9885.00,
      "high":   9887.00,
      "vol":    16021.632026
    },
    {
      "amount": 2.2124,
      "count":  6,
      "id":     1494465900,
      "open":   9885.00,
      "close":  9880.00,
      "low":    9880.00,
      "high":   9885.00,
      "vol":    21859.023500
    }
  ]
}
```

SubscribeMarketDepth

Handbook depth, according to different step aggregation, buy one, buy two, buy three, and sell one, sell two, sell three, etc. type Optional value: { step0, step1, step2, step3, step4, step5, percent10 } (combination depth 0-5); when step0, no depth is merged

```
{
  "ch": "market.btcusdt.depth.step0",
  "ts": 1489474082831,
  "tick": {
    "bids": [
    [9999.3900,0.0098], // [price, amount]
    [9992.5947,0.0560],
    // more Market Depth data here
```

```
      ]
      "asks": [
      [10010.9800,0.0099]
      [10011.3900,2.0000]
      //more data here
      ]
    }
}
```

SubscribeTradeDetail

Transaction record, including transaction price, volume, direction of transaction, etc.

```
{
  "ch": "market.btcusdt.trade.detail",
  "ts": 1489474082831,
  "tick": {
        "id": 14650745135,
        "ts": 1533265950234,
        "data": [
            {
                "amount": 0.0099,
                "ts": 1533265950234,
                "id": 146507451359183894799,
                "price": 401.74,
                "direction": "buy"
            },
            // more Trade Detail data here
        ]
    }

  ]
  }
}
```

SubscribeMarketDetail

Last 24 hours of volume, turnover, opening price, closing price, highest price, lowest price, number of transactions, etc.

```
{
  "rep": "market.btcusdt.detail",
  "status": "ok",
  "id": "id12",
  "tick": {
    "amount": 12224.2922,
    "open":   9790.52,
    "close":  10195.00,
    "high":   10300.00,
    "ts":     1494496390000,
    "id":     1494496390,
    "count":  15195,
    "low":    9657.00,
    "vol":    121906001.754751
  }
}
```

SubscribeMarketTickers

Information on the daily K line, the last 24 hours of trading, etc.

Events

OnHuobiSubscribed: event called after a successful subscription.

OnHuobiUnSubscribed: event called after a successful unsubscription.

OnHuobiUpdate: every time there is an update in data (kline, market depth...) this event is called.

OnHuobiError: if there is an error in Huobi API, this event will provide information about error.

API Cex

Cex

WebSocket API allows getting real-time notifications without sending extra requests, making it a faster way to obtain data from the exchange
Cex component has a property called Cex where you can fill API Keys provided by Cex to get access to your account data.

Message encoding

- All messages are encoded in JSON format.
- Prices are presented as strings to avoid rounding errors at JSON parsing on client side
- Compression of WebSocket frames is not supported by the server.
- Time is presented as integer UNIX timestamp in seconds.

Authentication

To get access to CEX.IO WebSocket data, you should be authorized.

- Log in to CEX.IO account.
- Go to https://cex.io/trade/profile#/api page.
- Select the type of required permissions.
- Click "Generate Key" button and save your secret key, as it will become inaccessible after activation.
- Activate your key.

Connectivity

- If a connected Socket is inactive for 15 seconds, CEX.IO server will send a PING message.
- Only server can be an Initiator of PING request.
- The server sends ping only to the authenticated user.
- The user has to respond with a PONG message. Otherwise, the WebSocket will be DISCONNECTED. This is handled automatically by the library.
- For the authenticated user, in case there is no notification or ping from the server within 15 seconds, it would be safer to send a request like 'ticker' or 'get-balance' and receive a response, in order to ensure connectivity and authentication.

Public Channels

These channels don't require to Authenticate before. Responses from the server are received by OnCexMessage event.

- **SubscribeTickers:** Ticker feed with only price of transaction made on all pairs (deprecated)

```
{
    "e": "tick",
    "data": {
        "symbol1": "BTC",
        "symbol2": "USD",
        "price": "428.0123"
    }
}
```

- **SubscribeChart:** OHLCV chart feeds with Open, High, Low, Close, Volume numbers (deprecated)

```
{
    'e': 'ohlcv24',
    'pair': 'BTC:USD',
    'data': [
        '418.2936',
        '420.277',
        '412.09',
        '416.9778',
        '201451078368'
    ]
}
```

- **Subscribe Pair:** Market Depth feed (deprecated)

```
{
    'e': 'md_groupped',
    'data': {
        'pair': 'BTC:USD',
        'id': 11296131,
        'sell': {
            '427.5000': 1000000,
            '480.0000': 263544334,
            ...
        },
        'buy': {
            '385.0000': 3630000,
            '390.0000': 1452458642,
            ... 400+ pairs togather with 'sell' pairs
        }
    }
}
```

- **Subscribe Pair:** Order Book feed (deprecated)

```
{
    'e': 'md',
    'data': {
```

```
        'pair': 'BTC:USD',
        'buy_total': 63221099,
        'sell_total': 112430315118,
        'id': 11296131,
        'sell': [
            [426.45, 10000000],
            [426.5, 66088429300],
            [427, 1000000],
            ... 50 pairs overaall
        ],
        'buy': [
            [423.3, 4130702],
            [423.2701, 10641168],
            [423.2671, 1000000],
            ... 50 pairs overaall
        ]
    }
}
```

Private Channels

To access these channels, first call Authenticate method. Responses from the server are received OnCexMessage event.

GetTicker

```
{
    "e": "ticker",
    "data": {
        "timestamp": "1471427037",
        "low": "290",
        "high": "290",
        "last": "290",
        "volume": "0.02062068",
        "volume30d": "14.38062068",
        "bid": 240,
        "ask": 290,
        "pair": [
            "BTC",
            "USD"
        ]
    },
    "oid": "1471427036908_1_ticker",
    "ok": "ok"
}
```

GetBalance

```
{
    "e": "get-balance",
    "data": {
           "balance": {
            'LTC': '10.00000000',
            'USD': '1024.00',
            'RUB': '35087.98',
            'EUR': '217.53',
            'GHS': '10.00000000',
            'BTC': '9.00000000'
        },
           "obalance": {
```

```
            'BTC': '0.12000000',
            'USD': "512.00",
         },
      },
      "time": 1435927928597
   "oid": "1435927928274_2_get-balance",
   "ok": "ok"
}
```

SubscribeOrderBook

```
{
"e": "order-book-subscibe",
"data": {
   "timestamp": 1435927929,
   "bids": [
      [
         241.947,
         155.91626
      ],
      [
         241,
         981.1255
      ],
   ],
   "asks": [
      [
         241.95,
         15.4613
      ],
      [
         241.99,
         17.3303
      ],
   ],
   "pair": "BTC:USD",
   "id": 67809
},
"oid": "1435927928274_5_order-book-subscribe",
"ok": "ok"
}
```

UnSubscribeOrderBook

```
{
"e": "order-book-unsubscribe",
"data": {
   "pair": "BTC:USD"
},
"oid": "1435927928274_4_order-book-unsubscribe",
"ok": "ok"
}
```

GetOpenOrders

```
{
"e": "open-orders",
"data": [
   {
      "id": "2477098",
```

```
      "time": "1435927928618",
      "type": "buy",
      "price": "241.9477",
      "amount": "0.02000000",
      "pending": "0.02000000"
   },
   {
      "id": "2477101",
      "time": "1435927928634",
      "type": "sell",
      "price": "241.9493",
      "amount": "0.02000000",
      "pending": "0.02000000"
   }
],
"oid": "1435927928274_9_open-orders",
"ok": "ok"
}
```


PlaceOrder


```
{
"e": "place-order",
"data": {
   "complete": false,
   "id": "2477098",
   "time": 1435927928618,
   "pending": "0.02000000",
   "amount": "0.02000000",
   "type": "buy",
   "price": "241.9477"
},
"oid": "1435927928274_7_place-order",
"ok": "ok"
}
```


CancelReplaceOrder


```
{
"e": "cancel-replace-order",
"data": {
   "complete": false,
   "id": "2689009",
   "time": 1443464955904,
   "pending": "0.04000000",
   "amount": "0.04000000",
   "type": "buy",
   "price": "243.25"
},
"oid": "1443464955209_16_cancel-replace-order",
"ok": "ok"
}
```


GetOrderRequest

In CEX.IO system, orders can be present in the trade engine or in an archive database. There can be time periods (~2 seconds or more), when the order is done/cancelled, but still not moved to the archive database. That means you cannot see it using calls: archived-orders/open-orders.

This call allows getting order information in any case. Responses can have different format depending on orders location.


```
{
"e": "get-order",
"data": {
   "user": "XXX",
   "type": "buy",
   "symbol1": "BTC",
   "symbol2": "USD",
   "amount": "0.02000000",
   "remains": "0.02000000",
   "price": "50.75",
   "time": 1450214742160,
   "tradingFeeStrategy": "fixedFee",
   "tradingFeeBuy": "5",
   "tradingFeeSell": "5",
   "tradingFeeUserVolumeAmount": "nil",
   "a:USD:c": "1.08",
   "a:USD:s": "1.08",
   "a:USD:d": "0.00",
   "status": "a",
   "orderId": "5582060"
},
"oid": "1450214742135_10_get-order",
"ok": "ok"
}
```


CancelOrderRequest


```
{
"e": "cancel-order",
"data": {
   "order_id": "2477098"
   "time": 1443468122895
},
"oid": "1435927928274_12_cancel-order",
"ok": "ok"
}
```


GetArchivedOrders


```
{
"e": "archived-orders",
"data": [
   {
       "type": "buy",
       "symbol1": "BTC",
       "symbol2": "USD",
       "amount": 0,
       "amount2": 5000,
       "remains": 0,
       "time": "2015-04-17T10:46:27.971Z",
       "tradingFeeBuy": "2",
       "tradingFeeSell": "2",
       "ta:USD": "49.00",
       "fa:USD": "0.98",
       "orderId": "2340298",
       "status": "d",
       "a:BTC:cds": "0.18151851",
       "a:USD:cds": "50.00",
```

```
        "f:USD:cds": "0.98"
    },
    {
        "type": "buy",
        "symbol1": "BTC",
        "symbol2": "USD",
        "amount": 0,
        "amount2": 10000,
        "remains": 0,
        "time": "2015-04-08T15:46:04.651Z",
        "tradingFeeBuy": "2.99",
        "tradingFeeSell": "2.99",
        "ta:USD": "97.08",
        "fa:USD": "2.91",
        "orderId": "2265315",
        "status": "d",
        "a:BTC:cds": "0.39869578",
        "a:USD:cds": "100.00",
        "f:USD:cds": "2.91"
    }
],
"oid": "1435927928274    15_archived-orders",
"ok": "ok"
}
```

OpenPosition

```
{
    "e": "open-position",
    "oid": "1435927928274_7_open-position",
    "data": {
        'amount': '1',
        'symbol': 'BTC',
        "pair": [
            "BTC",
            "USD"
        ],
        'leverage': '2',
        'ptype': 'long',
        'anySlippage': 'true',
        'eoprice': '650.3232',
        'stopLossPrice': '600.3232'
    }
}
```

GetPosition

```
{
  "e": "get_position",
  "ok": "ok",
  "data": {
    "user": "ud100036721",
    "pair": "BTC:USD",
    "amount": "1.00000000",
    "symbol": "BTC",
    "msymbol": "USD",
    "omamount": "1528.77",
    "lsymbol": "USD",
    "lamount": "3057.53",
    "slamount": "3380.11",
    "leverage": "3",
```

```
    "stopLossPrice": "3380.1031",
    "dfl": "3380.10310000",
    "flPrice": "3057.53333333",
    "otime": 1513002370342,
    "psymbol": "BTC",
    "ptype": "long",
    "ofee": "10",
    "pfee": "10",
    "cfee": "10",
    "tfeeAmount": "152.88",
    "rinterval": "14400000",
    "okind": "Manual",
    "a:BTC:c": "1.00000000",
    "a:BTC:s": "1.00000000",
    "oorder": "89101551",
    "pamount": "1.00000000",
    "lremains": "3057.53",
    "slremains": "3380.11",
    "oprice": "4586.3000",
    "status": "a",
    "id": "125531",
    "a:USD:cds": "4739.18"
  }
}
```

GetOpenPositions

```
{
    'e': 'open_positions',
    "oid": "1435927928256_7_open-positions",
    'ok': 'ok',
    'data': [
        {
            'user': 'ud100036721',
            'id': '104102',
            'otime': 1475602208467,
            'symbol': 'BTC',
            'amount': '1.00000000',
            'leverage': '2',
            'ptype': 'long',
            'psymbol': 'BTC',
            'msymbol': 'USD',
            'lsymbol': 'USD',
            'pair': 'BTC:USD',
            'oprice': '607.5000'
            'stopLossPrice': '520.3232',
            'ofee': '1',
            'pfee': '3',
            'cfee': '4',
            'tfeeAmount': '3.04',
            'pamount': '1.00000000',
            'omamount': '303.75',
            'lamount': '303.75',
            'oorder': '34106774',
            'rinterval': '14400000',
            'dfl': '520.32320000',
            'slamount': '520.33',
            'slremains': '520.33',
            'lremains': '303.75',
            'flPrice': '303.75000000',
            'a:BTC:c': '1.00000000',
```

```
            'a:BTC:s': '1.00000000',
            'a:USD:cds': '610.54',
        },
        ...
    ]
}
```

ClosePosition

```
{
    'e': 'close_position',
    "oid": "1435927928364_7_close-position",
    'ok': 'ok',
    'data': {
        'id': 104034,
        'ctime': 1475484981063,
        'ptype': 'long',
        'msymbol': 'USD'
        'pair': {
            'symbol1': 'BTC',
            'symbol2': 'USD'
        }
        'price': '607.1700',
        'profit': '-12.48',
    }
}
```

API Bitmex

Bitmex

Is a cryptocurrency exchange and derivative trading platform.

Methods

Subscribe / Unsubscribe

BitMEX allows subscribing to real-time data. This access is not rate-limited once connected and is the best way to get the most up-to-date data to your programs. In some topics, you can pass a Symbol to filter events by symbol, example: trades, quotes...

The following subscription topics are available without authentication:

- **btmAnnouncement**: Site Announcements
- **btmChat**: Trollbox chat
- **btmConnected**: Statistics of connected users/bots
- **btmFunding**: Updates of swap funding rates. Sent every funding interval (usually 8hrs)
- **btmInstrument**: Instrument updates including turnover and bid/ask
- **btmInsurance**: Daily Insurance Fund updates
- **btmLiquidation**: Liquidation orders as they're entered into the book
- **btmOrderBookL2_25**: Top 25 levels of level 2 order book
- **btmOrderBookL2**: Full level 2 order book
- **btmOrderBook10**: Top 10 levels using traditional full book push
- **btmPublicNotifications**: System-wide notifications (used for short-lived messages)
- **btmQuote**: Top level of the book
- **btmQuoteBin1m**: 1-minute quote bins
- **btmQuoteBin5m**: 5-minute quote bins
- **btmQuoteBin1h**: 1-hour quote bins
- **btmQuoteBin1d**: 1-day quote bins
- **btmSettlement**: Settlements
- **btmTrade**: Live trades
- **btmTradeBin1m**: 1-minute trade bins
- **btmTradeBin5m**: 5-minute trade bins
- **btmTradeBin1h**:1-hour trade bins
- **btmTradeBin1d**: 1-day trade bins

The following subjects require authentication:

- **btmAffiliate**: Affiliate status, such as total referred users & payout %
- **btmExecution**: Individual executions; can be multiple per order
- **btmOrder**: Live updates on your orders
- **btmMargin**: Updates on your current account balance and margin requirements
- **btmPosition**: Updates on your positions
- **btmPrivateNotifications**: Individual notifications - currently not used
- **btmTransact**: Deposit/Withdrawal updates
- **btmWallet**: Bitcoin address balance data, including total deposits & withdrawals

Example of messages received:

```
{
      "table":"orderBookL2_25",
      "keys":["symbol","id","side"],
      "types":{"id":"long","price":"float","side":"symbol","
size":"long","symbol":"symbol"}
      "foreignKeys":{"side":"side","symbol":"instrument"},
      "attributes":{"id":"sorted","symbol":"grouped"},
      "action":"partial",
      "data":[
        {"symbol":"XBTUSD","id":17999992000,"side":"Sell","s
ize":100,"price":80},
        {"symbol":"XBTUSD","id":17999993000,"side":"Sell","s
ize":20,"price":70},
        {"symbol":"XBTUSD","id":17999994000,"side":"Sell","s
ize":10,"price":60},
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy","si
ze":10,"price":50},
        {"symbol":"XBTUSD","id":17999996000,"side":"Buy","si
ze":20,"price":40},
        {"symbol":"XBTUSD","id":17999997000,"side":"Buy","si
ze":100,"price":30}
      ]
}

{
      "table":"orderBookL2_25",
      "action":"update",
      "data":[
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy","si
ze":5}
      ]
    }
{
      "table":"orderBookL2_25",
      "action":"delete",
      "data":[
        {"symbol":"XBTUSD","id":17999995000,"side":"Buy"}
      ]
}
{
      "table":"orderBookL2_25",
      "action":"insert",
      "data":[
        {"symbol":"XBTUSD","id":17999995500,"side":"Buy","si
ze":10,"price":45},
      ]
}
```

Authentication

If you wish to subscribe to user-locked streams, you must authenticate first. Note that invalid authentication will close the connection.

BitMEX API usage requires an API Key.

Permanent API Keys can be locked to IP address ranges and revoked at will without compromising your main credentials. They also do not require renewal.

To use API Key auth, you must generate an API Key in your account.

Cal method **Authenticate** before subscribe to any Authenticated Topic.

Events

OnBitmexConnect: after client successfully connects to the server, server sends to client a message to confirm connection to the server.

```
procedure OnBitmexConnect(Sender: TObject; const aMessage:
string);
begin
  Log('#Bitmex Connected: ' + aMessage);
end;
```

OnBitmexSubscribed: event called after a successful subscription.

```
procedure OnBitmexSubscribed(Sender: TObject; const aChannel:
string);
begin
  Log('Bitmex Subscribed: ' + aChannel);
end;
```

OnBitmexUnsubscribed: event called after a successful unsubscription.

```
procedure OnBitmexUnsubscribed(Sender: TObject; const aChannel:
string);
begin
  Log('Bitmex Subscribed: ' + aChannel);
end;
```

OnBitmexAuthenticated: after the client has authenticated against the server using ApiKey and ApiSecret, the client receives a message which confirms authentication.

```
procedure OnBitmexAuthenticated(Sender: TObject);
begin
  Log('#Bitmex Authenticated');
end;
```

OnBitmexMessage: all updates from subscribed topics, will be notified using this event. You can know which Topic is, accessing to aTopic parameter.

```
procedure OnBitmexMessage(Sender: TObject; const aTopic:
TwsBitmexTopics; const aMessage: string);
begin
  Log('Bitmex Message: ' + aMessage);
end;
```

OnBitmexError: if there is an error in Bitmex API, this event will provide information about error.

```
procedure OnBitmexError(Sender: TObject; const aMessage: string);
begin
  Log('Bitmex Error: ' + aMessage);
end;
```

API SignalRCore

SignalRCore

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

- Apps that require high-frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
- Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
- Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
- Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

SignalRCore sgcWebSockets component uses WebSocket as transport to connect to a SignalRCore server, if this transport is not supported, an error will be raised.

Hubs

SignalRCore uses hubs to communicate between clients and servers. SignalRCore provides 2 hub protocols: text protocol based on JSON and binary protocol based on MessagePack. The sgcWebSockets component only implements JSON text protocol to communicate with SignalRCore servers.
To configure which Hub client will use, just set in **SignalRCore/Hub** property the name of the Hub before the client connects to the server.

Connection

When a client opens a new connection to the server, sends a request message which contains format protocol and version. sgcWebSockets always sends format protocol as JSON. The server will reply with an error if the protocol is not supported by the server, this error can be handled using **OnSignalRCoreError** event, and if the connection is successful, **OnSignalRCoreConnect** event will be called.

When a client connects to a SignalRCore server, it can send a ConnectionId which identifies client between sessions, so if you get a disconnection client can reconnect to server passing same prior connection id. In order to get a new connection id, just connect normally to the server and you can know ConnectionId using **OnBeforeConnectEvent**. If you want to reconnect to the server and pass a prior connection id, use **ReConnect** method and pass **ConnectionId** as a parameter.

SignalRCore Protocol

The SignalR Protocol is a protocol for two-way RPC over any Message-based transport. Either party in the connection may invoke procedures on the other party, and procedures can return zero or more results or an error. Example: the client can request a method from the server and

server can request a method to the client. There are the following messages exchanged between server and clients:

- **HandshakeRequest:** the client sends to the server to agree on the message format.
- **HandshakeResponse:** server replies to the client an acknowledgement of the previous HandshakeRequest message. Contains an error if the handshake failed.
- **Close:** called by client or server when a connection is closed. Contains an error if the connection was closed because of an error.
- **Invocation:** client or server sends a message to another peer to invoke a method with arguments or not.
- **StreamInvocation:** client or server sends a message to another peer to invoke a streaming method with arguments or not. The Response will be split into different items.
- **StreamItem:** is a response from a previous StreamInvocation.
- **Completion:** means a previous invocation or StreamInvocation has been completed. Can contain a result if the process has been successful or an error if there is some error.
- **CancelInvocation:** cancel a previous StreamInvocation request.
- **Ping:** is a message to check if the connection is still alive.

Communication between Client and Server

There are three kinds of interactions between server and clients:

- **Invocations:** the Caller sends a message to the Callee and expects a message indicating that the invocation has been completed and optionally a result of the invocation

Example: client invokes SendMessage method and passes as parameters user name and text message. Sends an Invocation Id to get a result message from the server.

```
SignalRCore.Invoke('SendMessage', ['John', 'Hello All.'], 'id-
000001');

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

- **Non-Blocking Invocations:** the Caller sends a message to the Callee and does not expect any further messages for this invocation. Invocations can be sent without an Invocation ID value. This indicates that the invocation is "non-blocking".

Example: client invokes SendMessage method and passes as parameters user name and text message. The client doesn't expect any response from the server about the result of the invocation.

```
SignalRCore.Invoke('SendMessage', ['John', 'Hello All.']);
```

- **Streaming Invocations:** the Caller sends a message to the Callee and expects one or more results returned by the Callee followed by a message indicating the end of invocation.

Example: client invokes Counter method and requests 10 numbers with an interval of 500 milliseconds.

```
SignalRCore.InvokeStream('Counter', [10, 500], 'id-000002');

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

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

Invocations

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

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

- Allocate a unique **Invocation ID** value (arbitrary string, chosen by the Caller) to represent the invocation.
- Call **Invoke** or **InvokeStream** method containing the **Target** being invoked, **Arguments** and **InvocationId** (if you don't send InvocationId, you won't get completion result).
- If the Invocation is marked as non-blocking (see "Non-Blocking Invocations" below), stop here and immediately yield back to the application.
- Handle StreamItem or Completion message with a matching Invocation ID.

```
SignalRCore.InvokeStream('Counter', [10, 500], 'id-000002');

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

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

You can call a single **invocation and wait for** completion.

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

- Allocate a unique **Invocation ID** value (arbitrary string, chosen by the Caller) to represent the invocation.
- Call **InvokeAndWait** or **InvokeStreamAndWait** method containing the **Target** being invoked, **Arguments** and **InvocationId**
- The program will **wait till completion** event is called or **Time out has been exceeded**.

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


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

Cancel Invocation

If the client wants to stop receiving StreamItem messages before the Server sends a Completion message, the client can send a CancelInvocation message with the same InvocationId used for the StreamInvocation message that started the stream.

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

Completion and Results

An Invocation is only considered completed when the Completion message is received. If the client receives an Invocation from the server, **OnSignalRCoreInvocation** event will be called.

```
procedure OnSignalRCoreInvocation(Sender: TObject; Invocation:
TSignalRCore_Invocation);
begin
  if Invocation.Target = 'SendMessage' then
    ... your code here ...
end;
```

Once invocation is completed, call **Completion** method to inform server invocation is finished.

If result is successful, then call **CompletionResult** method:

```
SignalRCore.CompletionResult('id-000002', 'ok');
```

If not, then call **CompletionError** method:

```
SignalRCore.CompletionError('id-000002', 'Error processing
invocation.');
```

Close Connection

Sent by the client when a connection is closed. Contains an error reason if the connection was closed because of an error.

```
SignalRCore.Close('Unexpected message').
```

If the server close connection by any reason, **OnSignalRCoreClose** event will be called.

```
procedure OnSignalRCoreClose(Sender: TObject; Close:
TSignalRCore_Close);
begin
  DoLog('#closed: ' + Close.Error);
end;
```

Ping

The SignalR Hub protocol supports "Keep Alive" messages used to ensure that the underlying transport connection remains active. These messages help ensure:

- Proxies don't close the underlying connection during idle times (when few messages are being sent)
- If the underlying connection is dropped without being terminated gracefully, the application is informed as quickly as possible.

Keep alive behaviour is achieved calling **Ping** method or enabling HeartBeat on WebSocket client.
If the server sends a ping to the client, the client will send automatically a response and **OnSignalRCoreKeepAlive** event will be called.

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

Extensions

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.

** Indy version provided with Rad Studio XE2 raises an exception because of zlib version mismatch with initialization functions, to fix this, just update your Indy version to latest.*

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.

** Indy version provided with Rad Studio XE2 raises an exception because or zlib version mismatch with initialization functions, to fix this, just update your Indy version to latest.*

IoT

IoT

The Internet of things (IoT) refers to the concept of extending Internet connectivity beyond conventional computing platforms such as personal computers and mobile devices, and into any range of traditionally "dumb" or non-internet-enabled physical devices and everyday objects. Embedded with electronics, Internet connectivity, and other forms of hardware (such as sensors), these devices can communicate and interact with others over the Internet, and they can be remotely monitored and controlled.

sgcWebSockets package implements the following IoT clients:

1. Amazon AWS IoT: AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. This enables you to collect telemetry data from multiple devices, and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

Amazon

IoT Amazon MQTT Client

What Is AWS IoT?

AWS IoT provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. This enables you to collect telemetry data from multiple devices, and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

Message broker

Provides a secure mechanism for devices and AWS IoT applications to publish and receive messages from each other. You can use either the MQTT protocol directly or MQTT over WebSocket to publish and subscribe.

The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like Sensor/temp/room1.

The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing. The act of registering to receive messages for a topic filter is referred to as subscribing.

The topic namespace is isolated for each AWS account and region pair. For example, the Sensor/temp/room1 topic for an AWS account is independent from the Sensor/temp/room1 topic for another AWS account. This is true of regions, too. The Sensor/temp/room1 topic in the same AWS account in us-east-1 is independent from the same topic in us-east-2. AWS IoT does not support sending and receiving messages across AWS accounts and regions.

The message broker maintains a list of all client sessions and the subscriptions for each session. When a message is published on a topic, the broker checks for sessions with subscriptions that map to the topic. The broker then forwards the publish message to all sessions that have a currently connected client.

MQTT Client

TsgcIoTAmazon_MQTT_Client is the component used for connect to AWS IoT, one client can connect to only one device. Client connects using plain MQTT protocol and authenticates using a X.509 Client Certificate.

In order to connect to AWS IoT, client needs the following properties:

> **Amazon.Endpoint:** server name where MQTT client will connect.
> **Certificate.CertFile:** path to X.509 client certificate.
> **Certificate.KeyFile:** path to X.509 client key file.

Client can send optionally a ClientId to identifiy client connection, then others clients can subscribe to receive a notification every time this client has connected, subscribed, disconnected...

Amazon MQTT implementation is based on MQTT version 3.1.1 but it deviates from the specification as follows:

- In AWS IoT, subscribing to a topic with Quality of Service (QoS) 0 means a message is delivered zero or more times. A message might be delivered more than once. Messages delivered more than once might be sent with a different packet ID. In these cases, the DUP flag is not set.
- AWS IoT does not support publishing and subscribing with QoS 2. The AWS IoT message broker does not send a PUBACK or SUBACK when QoS 2 is requested.
- When responding to a connection request, the message broker sends a CONNACK message. This message contains a flag to indicate if the connection is resuming a previous session. The value of this flag might be incorrect if two MQTT clients connect with the same client ID simultaneously.
- When a client subscribes to a topic, there might be a delay between the time the message broker sends a SUBACK and the time the client starts receiving new matching messages.
- The MQTT specification provides a provision for the publisher to request that the broker retain the last message sent to a topic and send it to all future topic subscribers. AWS IoT does not support retained messages. If a request is made to retain messages, the connection is disconnected.
- The message broker uses the client ID to identify each client. The client ID is passed in from the client to the message broker as part of the MQTT payload. Two clients with the same client ID are not allowed to be connected concurrently to the message broker. When a client connects to the message broker using a client ID that another client is using, a CONNACK message is sent to both clients and the currently connected client is disconnected.
- On rare occasions, the message broker might resend the same logical PUBLISH message with a different packet ID.
- The message broker does not guarantee the order in which messages and ACK are received.

Connect to AWS IoT

First, you must sign in your AWS console, register a new device and create a X.509 certificate for this device. Once is done, you can create create a new TsgcIoTAmazon_MQTT_Client and connect to AWS IoT Server.
For example:

```
oClient := TsgcIoTAmazon_MQTT_Client.Create(nil);
oClient.Amazon.Endpoint := 'a2ohgdjqitsmij-ats.iot.us-west-
2.amazonaws.com';
oClient.Amazon.ClientId := 'sgcWebSockets';
oClient.Certificate.CertFile := 'amazon-certificate.pem.crt';
oClient.Certificate.KeyFile := 'amazon-private.pem.key';
oClient.OnMQTTConnect := OnMQTTConnectEvent;
oClient.Active := True;

procedure OnMQTTConnect(Connection: TsgcWSConnection; const
Session: Boolean; const ReturnCode: TmqttConnReturnCode);
begin
  ShowMessage('Connected to AWS');
end;
```

Topics

The message broker uses **topics** to **route messages from publishing clients to subscribing clients**. The forward slash (/) is used to separate topic hierarchy.

The following table lists the wildcards that can be used in the topic filter when you subscribe.

Must be the last character in the topic to which you are subscribing. Works as a wildcard by matching the current tree and all subtrees. For example, a subscription to Sensor/# receives messages published to Sensor/, Sensor/temp, Sensor/temp/room1, but not the messages published to Sensor.

+ Matches exactly one item in the topic hierarchy. For example, a subscription to Sensor/+/room1 receives messages published to Sensor/temp/room1, Sensor/moisture/room1, and so on.

```
oClient := TsgcIoTAmazon_MQTT_Client.Create(nil);
...
oClient.OnSubscribe := OnSubscribeEvent;

vPacketIdentifier := oClient.Subscribe('Sensor/moisture/room1');

procedure OnMQTTSubscribe(Connection: TsgcWSConnection;
aPacketIdentifier: Word; aCodes: TsgcWSSUBACKS);
begin
  if vPacketIdentifier = aPacketIdentifier then
    ShowMessage('Subscribed to topic Sensor/moisture/room1');
end;
```

Client, can send a message using **Publish** method.

```
oClient.Publish('Sensor/moisture/room1', '{"temp"=10}');
```

Messages received from server, are dispatched **OnMQTTPublishEvent**.

```
procedure OnMQTTPublish(Connection: TsgcWSConnection; aTopic,
aText: string);
begin
  DoLog('Received Message: ' + aTopic + ' ' + aText);
end;
```

Reserved Topics

Following methods are used to subscribe / publish to reserved topics.

Subscribe_ClientConnected(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID connects to AWS IoT
Subscribe_ClientDisconnected(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID disconnects to AWS IoT
Subscribe_ClientSubscribed(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID subscribes to an MQTT topic
Subscribe_ClientUnSubscribed(const aClientId: String): AWS IoT publishes to this topic when an MQTT client with the specified client ID unsubscribes to an MQTT topic

Publish_Rule(const aRuleName, aText: String): A device or an application publishes to this topic to trigger rules directly

Publish_DeleteShadow(const aThingName, aText: String): A device or an application publishes to this topic to delete a shadow
Subscribe_DeleteShadow(const aThingName: String): A device or an application subscribe to this topic to delete a shadow
Subscribe_ShadowDeleted(const aThingName: String): The Device Shadow service sends messages to this topic when a shadow is deleted
Subscribe_ShadowRejected(const aThingName: String): The Device Shadow service sends messages to this topic when a request to delete a shadow is rejected

Publish_ShadowGet(const aThingName, aText: String): An application or a thing publishes an empty message to this topic to get a shadow

Subscribe_ShadowGet(const aThingName: String): An application or a thing subscribe to this topic to get a shadow

Subscribe_ShadowGetAccepted(const aThingName: String): The Device Shadow service sends messages to this topic when a request for a shadow is made successfully

Subscribe_ShadowGetRejected(const aThingName: String): The Device Shadow service sends messages to this topic when a request for a shadow is rejected

Publish_ShadowUpdate(const aThingName, aText: String): A thing or application publishes to this topic to update a shadow

Subscribe_ShadowUpdateAccepted(const aThingName: String): The Device Shadow service sends messages to this topic when an update is successfully made to a shadow

Subscribe_ShadowUpdateRejected(const aThingName: String): The Device Shadow service sends messages to this topic when an update to a shadow is rejected

Subscribe_ShadowUpdateDelta(const aThingName: String): The Device Shadow service sends messages to this topic when a difference is detected between the reported and desired sections of a shadow

Subscribe_ShadowUpdateDocuments(const aThingName: String): AWS IoT publishes a state document to this topic whenever an update to the shadow is successfully performed

Persistent Sessions

A persistent session represents an ongoing connection to an MQTT message broker. When a client connects to the AWS IoT message broker using a persistent session, the message broker saves all subscriptions the client makes during the connection. When the client disconnects, the message broker stores unacknowledged QoS 1 messages and new QoS 1 messages published to topics to which the client is subscribed. When the client reconnects to the persistent session, all subscriptions are reinstated and all stored messages are sent to the client at a maximum rate of 10 messages per second.

You create an MQTT persistent session setting the **cleanSession** parameter to False **OnMQTTBeforeConnect** event. If no session exists for the client, a new persistent session is created. If a session already exists for the client, it is resumed.

Devices need to look at the **Session** attribute in the **OnMQTTConnect** event to determine if a persistent session is present. If **Session is True**, a persistent session is present and stored messages are delivered to the client. If **Session is False**, no persistent session is present and the client must re-subscribe to its topic filters.

Persistent sessions have a default expiry period of 1 hour. The expiry period begins when the message broker detects that a client disconnects (MQTT disconnect or timeout). The persistent session expiry period can be increased through the standard limit increase process. If a client has not resumed its session within the expiry period, the session is terminated and any associated stored messages are discarded. The expiry period is approximate, sessions might be persisted for up to 30 minutes longer (but not less) than the configured duration.

Library

Library

DLL
The sgcWebSockets.dll will need to be on any machine that your program will run on. It needs to be in the path. Typically, you would install it in the same directory as your executable, or in the C:\Windows\System32 or C:\Windows\SysWOW64 directory.

sgcWebsockets.dll is located inside Lib Directory:

Win32: dll for Windows 32bits.
Win64: dll for Windows 64bits.

Delphi
Include the files sgcWebSocketLib.pas, sgcWebSocketLib_Types.pas and sgcWebSocketLib_Const.pas in your project. These files are Pascal import units which define all of the sgcWebSocket methods and events defined.

Add **sgcWebSocketLib** to your uses clauses.

Example of use: open a new WebSocket client connection to host '127.0.0.1' and port 8080.

```
sgcWebSocketLib.Instance.Client_Initialize('127.0.0.1', 8080);
sgcWebSocketLib.Instance.Client_Start;
```

QuickStart

C# (C Sharp .NET)
Include the file sgcWebSocketLib.cs in your project. That file defines all of the sgcWebSocket methods and events defined.

Add **esegece.sgcWebSockets** and **System.Runtime.InteropServices** to your uses clauses.

Example of use: open a new WebSocket client connection to host '127.0.0.1' and port 8080.

```
sgcWebSocketLib.Instance.Client_Initialize("127.0.0.1", 8080);
sgcWebSocketLib.Instance.Client_Start();
```

QuickStart

Library | Build

If you are a registered customer, you can rebuild sgcWebSockets.dll, to do this, just open sgcWebSockets project located inside Package directory and recompile the library.

QuickStart

QuickStart | Library | Delphi

Let's start with a basic example where we need to create a Client WebSocket.

1. Add sgcWebSocketLib.pas, sgcWebSocketLib_Types.pas and sgcWebSocketLib_Const.pas to your project.

2. Add sgcWebSocketLib to your uses clauses.

3. Create Methods to handle WebSocket events:

```
procedure WSClientConnect(const aGuid: widestring); stdcall;
begin
  // Log('#connected');
end;

procedure WSClientDisconnect(const aGuid: widestring; const Code:
Integer);
    stdcall;
begin
  // Log('#disconnected (' + IntToStr(Code) + ')');
end;

procedure WSClientError(const aGuid, Error: widestring); stdcall;
begin
  // Log('#error: ' + Error);
end;

procedure WSClientMessage(const aGuid, Text: widestring);
stdcall;
begin
  // Log(Text);
end;
```

4. Register event methods:

```
sgcWebSocketLib.Instance.Client_OnConnect(WSClientConnect);
sgcWebSocketLib.Instance.Client_OnDisconnect(WSClientDisconnect);
sgcWebSocketLib.Instance.Client_OnError(WSClientError);
sgcWebSocketLib.Instance.Client_OnMessage(WSClientMessage);
```

5. Start a new WebSocket connection (Host is 127.0.0.1 and Port 80):

```
sgcWebSocketLib.Instance.Client_Initialize('127.0.0.1', 80);
sgcWebSocketLib.Instance.Client_Start;
```

6. Send a message to Server:

```
sgcWebSocketLib.Instance.Client_WriteData('Hello World');
```

7. Stop WebSocket connection:

```
sgcWebSocketLib.Instance.Client_Stop;
```

QuickStart | Library | C#

Let's start with a basic example where we need to create a Client WebSocket.

1. Add sgcWebSocketLib.cs to your project.

2. Add esegece.sgcWebSockets and System.Runtime.InteropServices to your uses clauses.

3. Create Methods to handle WebSocket events:

```
private void OnConnectEvent([MarshalAs(UnmanagedType.LPWStr)]
string ID)
{
  // Log("#connected: " + ID);
}

private void OnDisconnectEvent([MarshalAs(UnmanagedType.LPWStr)]
string ID, int aCode)
{
  // Log("#disconnected: " + ID);
}

private void OnMessageEvent([MarshalAs(UnmanagedType.LPWStr)]
string ID, [MarshalAs(UnmanagedType.LPWStr)] string aText)
{
  // Log(ID + ":" + aText);
}

private void OnError([MarshalAs(UnmanagedType.LPWStr)] string ID,
[MarshalAs(UnmanagedType.LPWStr)] string aError)
{
  // Log("#error: " + aError);
}
```

4. Register event methods:

```
sgcWebSocketLib.Instance.Client_OnConnect(OnConnectEvent);
sgcWebSocketLib.Instance.Client_OnDisconnect(OnDisconnectEvent);
sgcWebSocketLib.Instance.Client_OnError(OnError);
sgcWebSocketLib.Instance.Client_OnMessage(OnMessageEvent);
```

5. Start a new WebSocket connection (Host is 127.0.0.1 and Port 80:

```
sgcWebSocketLib.Instance.Client_Initialize('127.0.0.1', 80);
sgcWebSocketLib.Instance.Client_Start();
```

6. Send a message to Server:

```
sgcWebSocketLib.Instance.Client_WriteData('Hello World');
```

7. Stop WebSocket connection:

```
sgcWebSocketLib.Instance.Client_Stop();
```

Client

Library | Client

Create WebSocket Clients to connect to WebSocket Servers. Library client is based on VCL WebSocket Client, more info.

Methods

Client_Create(aID: WideString = '')

Creates a new WebSocket client. aID parameter is used if you want to create multiple instances of a client.

Client_Clear(aID: WideString = '')

Destroys an existing WebSocket client.

Client_Initialize(aHost: WideString = '127.0.0.1'; aPort: Integer = 80; aID: WideString = '')

This method needs to be called before a new connection is started. You can define which is the Server Host and Port where you want to connect.

Client_Finalize

Used internally to delete all clients created.

Client_Start(aID: WideString = '')

Starts a new WebSocket connection.

Client_Stop(aID: WideString = '')

Stops a WebSocket connection.

Client_LoadOptions(aOptions: WideString; aID: WideString = '')

Load WebSocket client options from a string.

Client_WriteData(aText: WideString; aID: WideString = '')

Sends a Text message.

Client_WriteData(aStream: TMemoryStream; aID: WideString = '')

Sends a Binary Message.

Events

Client_OnConnect(aOnConnect: TsgcWSClient_OnConnect;

aID: WideString = '')

The event raises when a new WebSocket connection is opened.

Client_OnDisconnect(aOnDisconnect: TsgcWSClient_OnDisconnect;
aID: WideString = '')

The event raises when a WebSocket connection is closed.

Client_OnMessage(aOnMessage: TsgcWSClient_OnMessage;
aID: WideString = '')

The event raises when the client receives a new text message from the server.

Client_OnBinary(aOnBinary: TsgcWSClient_OnBinary;
aID: WideString = '')

The event raises when the client receives a new Binary message from the server.

Client_OnError(aOnError: TsgcWSClient_OnError; aID: WideString = '')

The event raises when there is an error on WebSocket connection.

Types

```
TsgcWSClient_OnConnect = procedure(const aGuid: WideString);
TsgcWSClient_OnMessage = procedure(const aGuid: WideString; const aText: WideString);
TsgcWSClient_OnBinary = procedure(const aGuid: WideString;
  const aStream: TMemoryStream);
TsgcWSClient_OnBinary_Bytes = procedure(const aGuid: WideString;
  const aBytes: TBytes; const aContinue: Integer);
TsgcWSClient_OnDisconnect = procedure(const aGuid: WideString;
  const aCode: Integer);
TsgcWSClient_OnError = procedure(const aGuid: WideString; const aError: WideString);
```

Library Client | SocketIO

Create SocketIO clients to connect to SocketIO WebSocket Servers. Library client is based on VCL WebSocket SocketIO Client, more info.

Methods

Client_SocketIO_Create(aID: WideString = '')

Creates a new WebSocket client. aID parameter is used if you want to create multiple instances of a client.

Client_SocketIO_Clear(aID: WideString = '')

Destroys an existing WebSocket client.

Client_SocketIO_Initialize(aHost: WideString = '127.0.0.1'; aPort: Integer = 80; aID: WideString = '')

This method needs to be called before a new connection is started. You can define which is the Server Host and Port where you want to connect.

Client_SocketIO_Finalize

Used internally to delete all clients created.

Client_SocketIO_Start(aID: WideString = '')

Starts a new WebSocket connection.

Client_SocketIO_Stop(aID: WideString = '')

Stops a WebSocket connection.

Client_SocketIO_LoadOptions(aOptions: WideString; aID: WideString = '')

Load WebSocket client options from a string.

Client_SocketIO_WriteData(aText: WideString; aID: WideString = '')

Sends a Text message.

Client_SocketIO_WriteData(aStream: TMemoryStream; aID: WideString = '')

Sends a Binary Message.

Client_SocketIO_SendDisconnect(aEndPoint: WideString = ''; aID: WideString = '')

Signals disconnection. If no endpoint is specified, disconnects the entire socket.

Client_SocketIO_SendConnect(aEndPoint: WideString = ''; aID: WideString = '')

Only used for multiple sockets. Signals a connection to the endpoint. Once the server receives it, it's echoed back to the client.

Client_SocketIO_SendHeartBeat(aID: WideString = '')

Sends a heartbeat. Heartbeats must be sent within the interval negotiated with the server. It's up to the client to decide the padding (for example, if the heartbeat timeout negotiated with the server is 20s, the client might want to send a heartbeat every 15s).

Client_SocketIO_SendTextMessage(aText, aMessageId, aEndPoint: string; aID: WideString = '')

Sends a Text message.

Client_SocketIO_SendJSONMessage(aJSON, aMessageId, aEndPoint: string; aID: WideString = '')

Sends a JSON encoded message.

Client_SocketIO_SendEvent(aEventName, aEventArgs, aMessageId, aEndPoint: string; aID: WideString = '')

An event is like a JSON message, but has a mandatory name and args fields. name is a string and args an array.
The event names: 'message', 'connect', 'disconnect', 'open', 'close', 'error', 'retry', 'reconnect' are reserved, and cannot be used by clients or servers with this message type.

Client_SocketIO_SendACK(aMessage, aData: WideString; aID: WideString = '')

An acknowledgement contains the message id as the message data. If a + sign follows the message id, it's treated as an event message packet.

Client_SocketIO_SendError(aReason, aAdvice, aEndPoint: WideString; aID: WideString = '')

For example, if a connection to a sub-socket is unauthorized.

Client_SocketIO_SendNoop(aID: WideString = '')

No operation. Used for example to close a poll after the polling duration times out.

Events

Client_SocketIO_OnConnect(aOnConnect: TsgcWSClient_OnConnect; aID: WideString = '')

The event raises when a new WebSocket connection is opened.

Client_SocketIO_OnDisconnect(aOnDisconnect: TsgcWSClient_OnDisconnect; aID: WideString = '')

The event raises when a WebSocket connection is closed.

Client_SocketIO_OnMessage(aOnMessage: TsgcWSClient_OnMessage; aID: WideString = '')

The event raises when client receives a new text message from server.

**Client_SocketIO_OnBinary(aOnBinary: TsgcWSClient_OnBinary;
aID: WideString = '')**

The event raises when client receives a new Binary message from server.

Client_SocketIO_OnError(aOnError: TsgcWSClient_OnError; aID: WideString = '')

The event raises when there is any error on WebSocket connection.

These events are only raised if "RawMessages" property is disabled.

**Client_SocketIO_OnMessageDisconnect(aOnMessageDisconnect:
TsgcWSClient_SocketIO_OnMessageDisconnect; aID: WideString = '')**

The event raises when a client receives a Disconnect message.

**Client_SocketIO_OnMessageConnect(aOnMessageConnect:
TsgcWSClient_SocketIO_OnMessageConnect; aID: WideString = '')**

The event raises when a client receives a Connect message.

**Client_SocketIO_OnMessageHeartBeat(aOnMessageHeartBeat:
TsgcWSClient_SocketIO_OnMessageHeartBeat; aID: WideString = '')**

The event raises when a client receives a HeartBeat message.

**Client_SocketIO_OnMessageEvent(aOnMessageEvent:
TsgcWSClient_SocketIO_OnMessageEvent; aID: WideString = '')**

The event raises when a client receives an Event Message.

**Client_SocketIO_OnMessageText(aOnMessageText:
TsgcWSClient_SocketIO_OnMessageText; aID: WideString = '')**

The event raises when a client receives a Text message.

**Client_SocketIO_OnMessageJSON(aOnMessageJSON:
TsgcWSClient_SocketIO_OnMessageJSON; aID: WideString = '')**

The event raises when a client receives a JSON message.

**Client_SocketIO_OnMessageACK(aOnMessageACK:
TsgcWSClient_SocketIO_OnMessageACK; aID: WideString = '')**

The event raises when a client receives an Acknowledgment message.

**Client_SocketIO_OnMessageError(aOnMessageError:
TsgcWSClient_SocketIO_OnMessageError; aID: WideString = '')**

The event raises when a client receives an Error message.

**Client_SocketIO_OnMessageNoop(aOnMessageNoop:
TsgcWSClient_SocketIO_OnMessageNoop; aID: WideString = '')**

The event raises when a client receives a Noop message.

Types

```
  TsgcWSClient_OnConnect = procedure(const aGuid: WideString);
  TsgcWSClient_OnMessage = procedure(const aGuid: WideString; const aText: WideString);
  TsgcWSClient_OnBinary = procedure(const aGuid: WideString;
    const aStream: TMemoryStream);
  TsgcWSClient_OnDisconnect = procedure(const aGuid: WideString;
    const aCode: Integer);
  TsgcWSClient_OnError = procedure(const aGuid: WideString; const aError: WideString);

  TsgcWSClient_SocketIO_OnMessageDisconnect = procedure(const aGuid: WideString;
      const Text: WideString; var Handled: Boolean) stdcall;
  TsgcWSClient_SocketIO_OnMessageConnect = procedure(const aGuid: WideString; const
      Text: WideString; const EndPoint: WideString) stdcall;
  TsgcWSClient_SocketIO_OnMessageHeartBeat = procedure(const aGuid: WideString; const
Text:
      WideString; var Handled: Boolean) stdcall;
  TsgcWSClient_SocketIO_OnMessageEvent = procedure(const aGuid: WideString; const
      Text, MsgId, MsgEndPoint, EventName, EventArgs, JSON: WideString) stdcall;
  TsgcWSClient_SocketIO_OnMessageText = procedure(const aGuid: WideString; const
      Text, MsgId, MsgEndPoint, MsgText: WideString) stdcall;
  TsgcWSClient_SocketIO_OnMessageJSON = procedure(const aGuid: WideString; const
      Text, MsgId, MsgEndPoint, JSON: WideString) stdcall;
  TsgcWSClient_SocketIO_OnMessageACK = procedure(const aGuid: WideString; const Text,
      MsgData: WideString) stdcall;
  TsgcWSClient_SocketIO_OnMessageError = procedure(const aGuid: WideString; const
      Text, MsgEndPoint, MsgError: WideString) stdcall;
  TsgcWSClient_SocketIO_OnMessageNoop = procedure(const aGuid: WideString; const
      Text: WideString) stdcall;
```

Library | Client Protocol SGC

Create WebSocket Clients to connect to WebSocket Servers. Library client is based on VCL WebSocket Client using protocol SGC, more info.

Methods

Client_Create(aID: WideString = '')

Creates a new WebSocket client. aID parameter is used if you want to create multiple instances of a client.

Client_Clear(aID: WideString = '')

Destroys an existing WebSocket client.

Client_Initialize(aHost: WideString = '127.0.0.1'; aPort: Integer = 80; aID: WideString = '')

This method needs to be called before a new connection is started. You can define which is the Server Host and Port where you want to connect.

Client_Finalize

Used internally to delete all clients created.

Client_Start(aID: WideString = '')

Starts a new WebSocket connection.

Client_Stop(aID: WideString = '')

Stops a WebSocket connection.

Client_LoadOptions(aOptions: WideString; aID: WideString = '')

Load WebSocket client options from a string.

Client_WriteData(aText: WideString; aID: WideString = '')

Sends a Text message.

Client_WriteData(aStream: TMemoryStream; aID: WideString = '')

Sends a Binary Message.

Client_sgc_Subscribe(aChannel: WideString; aID: WideString = '')

Subscribe to a new channel.

Client_sgc_UnSubscribe(aChannel: WideString; aID: WideString = '')

Unsubscribe from a channel.

Client_sgc_Broadcast(aText, aChannel: WideString; aID: WideString = '')

Sends a Text message to all clients connected to a specified channel.

Client_sgc_RPC(aIdMethod, aMethod, aParams: WideString; aID: WideString = '')

Sends a Remote Procedure Call.

Client_sgc_Notify(aMethod, aParams: WideString; aID: WideString = '')

Notify a message to the server.

Client_sgc_Publish(aText, aChannel: WideString; aID: WideString = '')

Sends a Text message to all clients connected to a specified channel.

Client_sgc_GetSession(aID: WideString = '')

Requests Session ID from Server.

Client_sgc_StartTransaction(aChannel: WideString = ''; aID: WideString = '')

Starts a New Transaction.

Client_sgc_Commit(aChannel: WideString = ''; aID: WideString = '')

Commits all messages of a Transaction.

Client_sgc_RollBack(aChannel: WideString = ''; aID: WideString = '')

RollBack all messages of a Transaction.

Client_sgc_Broker(aID_Broker: WideString = ''; aGUID_Broker: WideString = ''; aID: WideString = '')

Connect sgc protocol to a broker.

Events

Client_sgc_OnConnect(aOnConnect: TsgcWSClient_OnConnect; aID: WideString = '')

The event raises when a new WebSocket connection is opened.

Client_sgc_OnDisconnect(aOnDisconnect: TsgcWSClient_OnDisconnect; aID: WideString = '')

The event raises when a WebSocket connection is closed.

Client_sgc_OnMessage(aOnMessage: TsgcWSClient_OnMessage; aID: WideString = '')

The event raises when the client receives a new text message from server.

Client_sgc_OnBinary(aOnBinary: TsgcWSClient_OnBinary; aID: WideString = '')

The event raises when client receives a new Binary message from the server.

Client_sgc_OnError(aOnError: TsgcWSClient_OnError; aID: WideString = '')

The event raises when there is any error on WebSocket connection.

Client_sgc_OnSubscription(aOnSubscription: TsgcWSP_sgc_OnSubscription; aID: WideString = '')

The event raises if a new subscription is received from the server.

Client_sgc_OnUnSubscription(aOnUnSubscription: TsgcWSP_sgc_OnUnSubscription; aID: WideString = '')

The event raises if an unsubscription is received from the server.

Client_sgc_OnEvent(aOnEvent : TsgcWSPClient_sgc_OnEvent; aID: WideString = '')

The event raises when a new event is received.

Client_sgc_OnSession(aOnSession : TsgcWSPClient_sgc_OnSession; aID: WideString = '')

The event raises when a Session ID is received from the server.

Client_sgc_OnAcknowledgment(aOnAcknowledgment : TsgcWSPClient_sgc_OnAcknowledgment; aID: WideString = '')

The event raises when the client receives an Acknowledgment from server that message has been received.

Client_sgc_OnRPCError(aOnRPCError: TsgcWSPClient_sgc_OnRPCError; aID: WideString = '')

The event raises if the server sends an error as response to a Remote Procedure Call.

Server_sgc_OnRPCResult(aOnRPCResult : TsgcWSPClient_sgc_OnRPCResult; aID: WideString = '')

The event raises when the server sends a response to the client after a Remote Procedure Call.

Types

```
TsgcWSClient_OnConnect = procedure(const aGuid: WideString);
TsgcWSClient_OnMessage = procedure(const aGuid: WideString; const aText: WideString);
TsgcWSClient_OnBinary = procedure(const aGuid: WideString;
  const aStream: TMemoryStream);
TsgcWSClient_OnDisconnect = procedure(const aGuid: WideString;
  const aCode: Integer);
TsgcWSClient_OnError = procedure(const aGuid: WideString; const aError: WideString);
TsgcWSP_sgc_OnSubscription = procedure(const aGuid, aChannel: string);
TsgcWSP_sgc_OnUnSubscription = procedure(const aGuid, aChannel: string);
TsgcWSPClient_sgc_OnRPCResult = procedure(const aGuid, aId, aResult: string);
TsgcWSPClient_sgc_OnRPCError = procedure(const aGuid, aId: string;
  aErrorCode: Integer; const ErrorMessage, ErrorData: string);
TsgcWSPClient_sgc_OnEvent = procedure(const aGuid, aChannel, aText: string);
```

Server

Library | Server

Create WebSocket Servers. Library server is based on VCL WebSocket Server, more info.

Methods

Server_Create(aID: WideString = '')

Creates a new websocket server. aID parameter is used if you want to create multiple instances of a server.

Server_Clear(aID: WideString = '')

Destroys an existing websocket server.

Server_Initialize(aPort: Integer = 80; aID: WideString = '')

This method needs to be called before a new connection is started. You can define which is the Port to listen client connections.

Server_Finalize

Used internally to delete all servers created.

Server_Start(aID: WideString = '')

Starts a server.

Server_Stop(aID: WideString = '')

Stops a server.

Server_LoadOptions(aOptions: WideString; aID: WideString = '')

Load websocket server options from a WideString.

**Server_Broadcast(aText: WideString; aID: WideString = '';
aExclude: WideString = ''; aInclude: WideString = '')**

Sends a Text message.

**Server_Broadcast(aStream: TMemoryStream; aID: WideString = '';
aExclude: WideString = ''; aInclude: WideString = '')**

Sends a Binary message.

Events

**Server_OnConnect(aOnConnect: TsgcWSServer_OnConnect;
aID: WideString = '')**

Event raised when a new websocket connection is opened.

**Server_OnDisconnect(aOnDisconnect: TsgcWSServer_OnDisconnect;
aID: WideString = '')**

Event raised when a websocket connection is closed.

**Server_OnMessage(aOnMessage: TsgcWSServer_OnMessage;
aID: WideString = '')**

Event raised when server receives a new text message from client.

**Server_OnBinary(aOnBinary: TsgcWSServer_OnBinary;
aID: WideString = '')**

Event raised when server receives a new Binary message from client.

Server_OnError(aOnError: TsgcWSServer_OnError; aID: WideString = '')

Event raised when there is any error on a websocket connection.

**Server_OnAuthentication(aOnAuthentication
: TsgcWSServer_OnAuthentication; aID: WideString = '')**

Event raised when a client tries to authenticate agains server.

Types

```
TsgcWSServer_OnConnect = procedure(const aGuid: WideString);
TsgcWSServer_OnMessage = procedure(const aGuid: WideString; const aText: WideString);
TsgcWSServer_OnBinary = procedure(const aGuid: WideString;
  const aStream: TMemoryStream);
TsgcWSServer_OnBinary_Bytes = procedure(const aGuid: WideString;
  const aBytes: TBytes; const aContinue: Integer);
TsgcWSServer_OnDisconnect = procedure(const aGuid: WideString;
  const aCode: Integer);
TsgcWSServer_OnError = procedure(const aGuid: WideString; const aError: WideString);
TsgcWSServer_OnAuthentication = procedure(const aGuid: WideString;
  aUser, aPassword: WideString; var Authenticated: Boolean);
```

Library | Server Protocol SGC

Create WebSocket Servers. Library server is based on VCL WebSocket Server using SGC Protocol, more info.

Methods

Server_Create(aID: WideString = '')

Creates a new websocket server. aID parameter is used if you want to create multiple instances of a server.

Server_Clear(aID: WideString = '')

Destroys an existing websocket server.

Server_Initialize(aPort: Integer = 80; aID: WideString = '')

This method needs to be called before a new connection is started. You can define which is the Port to listen client connections.

Server_Finalize

Used internally to delete all servers created.

Server_Start(aID: WideString = '')

Starts a server.

Server_Stop(aID: WideString = '')

Stops a server.

Server_LoadOptions(aOptions: WideString; aID: WideString = '')

Load websocket server options from a WideString.

**Server_Broadcast(aText: WideString; aID: WideString = '';
aExclude: WideString = ''; aInclude: WideString = '')**

Sends a Text message.

**Server_Broadcast(aStream: TMemoryStream; aID: WideString = '';
aExclude: WideString = ''; aInclude: WideString = '')**

Sends a Binary message.

**Server_sgc_Publish(aMessage, aChannel: WideString; aExclude: WideString = '';
aInclude: WideString = ''; aID: WideString = '')**

Publish a new message to all clients subscribed to a custom channel.

**Server_sgc_RPCResult(aIdMethod, aMethod: WideString;
aID: WideString = '')**

Sends result as response of Remote Procedure Call.

**Server_sgc_RPCError(aIdMethod: WideString; aCode: Integer;
aMessage, aData: WideString; aID: WideString = '')**

Sends an error as response of Remote Procedure Call.

Events

**Server_sgc_OnConnect(aOnConnect: TsgcWSServer_OnConnect;
aID: WideString = '')**

Event raised when a new websocket connection is opened.

**Server_sgc_OnDisconnect(aOnDisconnect: TsgcWSServer_OnDisconnect;
aID: WideString = '')**

Event raised when a websocket connection is closed.

**Server_sgc_OnMessage(aOnMessage: TsgcWSServer_OnMessage;
aID: WideString = '')**

Event raised when server receives a new text message from client.

**Server_sgc_OnBinary(aOnBinary: TsgcWSServer_OnBinary;
aID: WideString = '')**

Event raised when server receives a new Binary message from client.

Server_sgc_OnError(aOnError: TsgcWSServer_OnError; aID: WideString = '')

Event raised when there is any error on a websocket connection.

**Server_sgc_OnSubscription
(aOnSubscription: TsgcWSP_sgc_OnSubscription; aID: WideString = '')**

Event raised when a client requests a new subscription to a channel.

**Server_sgc_OnUnSubscription
(aOnUnSubscription: TsgcWSP_sgc_OnUnSubscription; aID: WideString = '')**

Event raised when a client requests an unsubscription from a channel.

**Server_sgc_OnNotification(aOnNotification
: TsgcWSPServer_sgc_OnNotification; aID: WideString = '')**

Event raised when server receives a notification from a client.

**Server_sgc_OnRPC(aOnRPC: TsgcWSPServer_sgc_OnRPC;
aID: WideString = '')**

Event raised when server receives a Remote Procedure Call from a client.

**Server_sgc_OnRPCAuthentication(aOnRPCAuthentication
: TsgcWSPServer_sgc_OnRPCAuthentication; aID: WideString = '')**

Event raised when server receives a RCP which requires authentication.

Types

```
TsgcWSServer_OnConnect = procedure(const aGuid: WideString);
TsgcWSServer_OnMessage = procedure(const aGuid: WideString; const aText: WideString);
TsgcWSServer_OnBinary = procedure(const aGuid: WideString;
  const aStream: TMemoryStream);
TsgcWSServer_OnDisconnect = procedure(const aGuid: WideString;
  const aCode: Integer);
TsgcWSServer_OnError = procedure(const aGuid: WideString; const aError: WideString);
TsgcWSServer_OnAuthentication = procedure(const aGuid: WideString;
  aUser, aPassword: WideString; var Authenticated: Boolean);

TsgcWSP_sgc_OnSubscription = procedure(const aGuid, aChannel: string);
TsgcWSP_sgc_OnUnSubscription = procedure(const aGuid, aChannel: string);
TsgcWSPServer_sgc_OnNotification = procedure(const aGuid, aMethod,
  aParams: string);
TsgcWSPServer_sgc_OnRPC = procedure(const aGuid, aId, aMethod,
  aParams: string);
TsgcWSPServer_sgc_OnRPCAuthentication = procedure(const aGuid, aMethod,
  aUser, Password: string; var Authenticated: Boolean);
```

Reference

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

JSON

JSON or **JavaScript Object Notation**, is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

More Information

JSON-RPC 2.0

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over http, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

Example: client call method subtract with 2 params (42 and 23). Server sends a result of 19.

Client To Server --> `{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}`

Server To Client<-- `{"jsonrpc": "2.0", "result": 19, "id": 1}`

Parsers

sgcWebSockets provides a built-in JSON component, but you can use your own JSON parser. Just implement following interfaces located at sgcJSON.pas:

IsgcJSON
IsgcObjectJSON

There are 3 implementations of theses interfaces

- **sgcJSON.pas:** default JSON parser provided.
- **sgcJSON_System.pas:** uses JSON parser provided with latest versions of delphi.
- **sgcJSON_XSuperObject.pas:** uses JSON library written by Onur YILDIZ, you can download sources from: https://github.com/onryldz/x-superobject

To use your own JSON parser or use some of the JSON parsers provided, just call **SetJSONClass** in your initialization method. For example: if you want use XSuperObject JSON parser, just call:

```
SetJSONClass(TsgcXSOJSON)
```

If you don't call this method, sgcJSON will be used by default.

More information

WAMP

The WebSocket Application Messaging Protocol (WAMP) is an open WebSocket subprotocol that provides two asynchronous messaging patterns: RPC and PubSub.

The WebSocket Protocol is already built into modern browsers and provides bidirectional, low-latency message-based communication. However, as such, WebSocket it 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

License

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

A
API Binance 179
API Bitfinex 164
API Bitmex 196
API Bitstamp 182
API Bittrex 175
API Blockchain 162
API Cex 187
API Huobi 184
API Pusher 169
API SignalR 173
API SignalRCore 199
API SocketIO 160
APIs 159, 160, 162, 164, 169, 173, 175, 179, 182, 184, 187, 196, 199
Authentication 31
B
Bindings 38
C
Compression 36
Configure Install 11
Custom Objects 33
Custom Sub 26
D
Deflate-Frame 206
E
Editions 5
Extensions 204
F
Fast Performance Server 25
Files 56
Flash 37
Fragmented Messages 43
H
HeartBeat 54
HTTP 48
I
Install Errors 19
Install Package 13
Installation 7
Introduction 1
IoT 207
IoT Amazon MQTT Client 208
J
JSON 232
L
Library 213, 214, 217, 219, 223, 226, 228
Library Build 214
Library Client 217, 219, 223
Library Client Protocol SGC 223
Library Client SocketIO 219
Library Server 226, 228
Library Server Protocol SGC 228
License 239
LoadBalancing 53
Logs 58
M
MQTT 208, 236
O
Overview 21
P
PerMessage-Deflate 205
Protocol AppRTC 141
Protocol Dataset 104, 110
Protocol Dataset Javascript 110
Protocol Default 94, 100
Protocol Default Javascript 100
Protocol Files 113
Protocol MQTT 129
Protocol Presence 118, 126
Protocol Presence Javascript 126
Protocol STOMP 132
Protocol WAMP 146, 150
Protocol WAMP Javascript 150
Protocol WAMP2 153
Protocol WebRTC 143, 145
Protocol WebRTC Javascript 145
Protocols 26, 88, 90, 94, 100, 104, 110, 113, 118, 126, 129, 132, 141, 143, 145, 146, 150, 153
Protocols Javascript 90
Proxy 57
Q
Quality 39
  Service 39
Quality Of Service 39
Queues 41
QuickStart Library C 216
QuickStart Library Delphi 215
S
Secure Connections 34
Server-Sent Events 50, 237
Service 39
  Quality 39
SubProtocol 47
T
TCP Connections 46
Throttle 49
Transactions 44
TsgcIWWebSocketClient 84
TsgcIWWSPClient_Dataset 109
TsgcIWWSPClient_sgc 99
TsgcWebSocketClient 72
TsgcWebSocketClient_SocketIO 76
TsgcWebSocketClient_WinHTTP 80
TsgcWebSocketHTTPServer 67
TsgcWebSocketLoadBalancerServer 82
TsgcWebSocketProxyServer 83
TsgcWebSocketServer 63
TsgcWebSocketServer_HTTPAPI 70

TsgcWSConnection 86
TsgcWSHTTPWebBrokerBridgeServer 79
TsgcWSMessageFile 117
TsgcWSPClient_Dataset 107
TsgcWSPClient_Files 116
TsgcWSPClient_MQTT 130
TsgcWSPClient_Presence 123
TsgcWSPClient_sgc 97
TsgcWSPClient_STOMP 133
TsgcWSPClient_STOMP_ActiveMQ 138
TsgcWSPClient_STOMP_RabbitMQ 135
TsgcWSPClient_WAMP 148
TsgcWSPClient_WAMP2 154
TsgcWSPresenceMessage 122
TsgcWSPServer_AppRTC 142
TsgcWSPServer_Dataset 105
TsgcWSPServer_Files 114
TsgcWSPServer_Presence 119
TsgcWSPServer_sgc 95
TsgcWSPServer_WAMP 147
TsgcWSPServer_WebRTC 144
U
Using DLL 61
W
WAMP 234
WatchDog 55
Web Browser Test 24
WebRTC 235
WebSocket Events 59
WebSocket Parameters Connection 60
WebSockets 59, 60, 231