sgcWebSockets 3.4

Table of Contents

Introduction

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

- Fully functional **multithreaded WebSocket server** according to **RFC 6455**.
- Supports **Firemonkey** (**Windows** and **MacOS**).
- Supports **NEXTGEN Compiler** (IOS and Android Support).
- Supports **Lazarus / FreePascal.**
- Supports **CBuilder.**
- Supports **Chrome**, **Firefox, Safari, Opera and Internet Explorer** (including **iPhone**, **iPad** and **iPod**)
- Supports **C#.NET** using compiled library (for Windows 32 and 64 bits).
- Multiple Threads Support
- Supports Message Compression using PerMessage_Deflate extension.
- 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.
- 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**.
- **Events** Available: OnConnect, OnDisconnect, OnMessage, OnError, OnHandshake

- **Built-in sub-protocols**: JSON-RPC 2.0, Dataset, WebRTC and WAMP
- **Built-in Javascript libraries** to support browser clients.
- Easy to setup
- **Javascript Events** for a full control
- **Async Events** using Ajax
- **SSL/TLS Support** for Server / Client Components

Main components available are:

- **TsgcWebSocketServer:** Non-visual component, it's used to manage client threaded connections. Supports RFCC 6455.
- **TsgcWebSocketHTTPServer:** Non-visual component, it's used to manage client threaded connections. Supports RFCC 6455. Supports HTTP Requests using an unique port for WebSocket and HTTP Connections.
- **TsgcWebSocketClient:** Non-visual component, used to establish a WebSocket connection with a WebSocket server.
- **TsgcWebsocketProxyServer:** Non-visual component, used to translate websocket connections to normal TCP connections.
- **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

CBuilder supported IDE

- CBuilder 2010
- CBuilder XE
- CBuilder XE2
- CBuilder XE3
- CBuilder XE4
- CBuilder XE5
- CBuilder XE6
- CBuilder XE7
- CBuilder XE8
- CBuilder 10 Seattle

AppMethod supported IDE

- AppMethod 2014

FreePascal supported IDE

- Lazarus

HTML supported IDE

- HTML5 Builder

Free 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

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}\libDXE8\$(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

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

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 1.0.6 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.

History

[*] : Bug
[+] : New
[-] : Deleted
[/] : Breaking changes

3.4: 2015 September

[+] : New Property "AutoEscapeText" on DataSet Protocol: disabled by default. Automatically escape/unescape characters inside field values like "{", "["...
[+] : WriteData Method has a new parameter "size" which is the maximum size of every text packet.
[+] : New Property "ReadStartSSL" on HTTP Server component, max number of times an HTTPS connection tries to start.
[+] : New Property "DisconnectAll" on Server components, disconnects all active connections.

[*] : Fixed Bug Dataset Protocol sending/receiving messages.
[*] : Fixed Bug re-sending messages using qosLevel2 through a broker.
[*] : Fixed Bug when call to disconnect using ssl client connection.
[*] : Fixed Bug when a client try to connect without handshake headers.
[*] : Fixed Bug Files Protocol reading Message File.
[*] : Fixed Bug Dataset Protocol sending updates from client.
[*] : Fixed Bug Dataset Protocol raising dataset events.
[*] : Fixed Bug Files Protocol QoSLevel2 for fragments not received.
[*] : Fixed Bug Reading fragmented text message.
[*] : Fixed Bug Reading SSE message from D2009+.
[*] : Fixed Bug Reading/Writing using Protocols.

[*] : Fixed Bug Client Component not clearing FWSConnection field when a connection was closed by server.
[*] : Fixed Bug Client Component destroying Critical Section.
[*] : Fixed Bug SGC Protocol OnDisconnect Event, an access violation was raised trying to send a message inside this event.

3.3: 2015 May

[+] : Added support for Rad Studio XE8.
[+] : New Property "RaiseDisconnectExceptions": enabled by default, raises an exception every time there is a disconnection by protocol error.
[+] : New Property "IO_HandShakeCustomURL"on Socket.io client, overrides url to get socket.io session.
[+] : New Event, "OnBeforeSubscription" on Server Component Protocols, allows/denies a client subscribe to a channel.
[+] : New Property "FragmentedMessages", allows to handle Fragmented Messages:
frgOnlyBuffer: 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 first packet.
When all data is received, it raises OnBinary or OnMessage event.
[+] : WriteData Method has a new parameter "size" which is the maximum size of every binary packet.
[+] : Added Support on Broker Protocol Components for binary messages.
[+] : New Methods for SGC Client Protocol to retrieve Method and Parameters sent on RPC calls: GetRPCMethodById and GetRPCParamsById.

[+] : Ping Method has been overloaded to accept a TimeOut, first sends a pings and waits a response, if no response after a timeout, returns false.
[+] : New Property "Subscriptions" on Server and Client Protocol Components, returns a list of active subscriptions.
[+] : New Event, "OnStartup" on WebSocket Servers, raised when a server is started.
[+] : New Event, "OnShutdown" on WebSocket Servers, raised when a server is stopped.
[+] : Method "Disconnect" of TsgcWSConnection now it's overloaded to accept close code as a parameter.
[+] : New Method, SetAuthenticationCloseCode on TsgcWSConnectionServer, allows to set close code if not authenticated.
[+] : Added support for broker on SGC Protocol using sgcWebSockets library.
[+] : New Method "UnSubscribeAll" for Client SGC Protocol.

[*] : Fixed Access Violation when Client tries to reconnect to Server.
[*] : Fixed DeadLock when Client disconnects and buffer is not empty.
[*] : Fixed Scape Error when JSON value is an object or array.
[*] : Fixed Bug trying to disconnect a connection several times.
[*] : Fixed Bug building CBuilder XE6-XE7 Package on Win64.
[*] : Fixed Bug SGC Protocol OnMessage Event.
[*] : Fixed Bug SGC Protocol WriteData Method.
[*] : Fixed Bug SGC Protocol Broadcast Method.
[*] : Fixed Bug Server Broadcast method using include / exclude arguments.
[*] : Fixed Bug SGC Protocol using QoSLevel2.
[*] : Fixed Bug SGC Protocol OnRPCResult Event, when result parameter was an array of objects.
[*] : Fixed Bug SGC Dataset Client when attached to a broker.
[*] : Fixed Bugs JSON parser.
[*] : Fixed Bug HeartBeat Timeout.

[*] : Fixed Bug Socket.IO Client on SendEvent method.
[*] : Fixed Bug when using TsgcWSConnection.Data property OnDestroy Event.
[*] : Fixed Bug DataSet Protocol when a DateTime Field IsNull.

[/] : WriteData method now "Stream" parameter is a TStream.

3.2: 2014 September

[+] : Added support for Rad Studio XE7.
[+] : Added support for AppMethod.
[+] : Added support for Server-Sent Events API (SSE), push notifications from server to client using http connection.
[+] : Added suport for SSE on Javascript libraries: automatic fallback to SSE+XHR if WebSocket is not implemented.
[+] : New Component TsgcWebSocketProxyServer: server that translates WebSocket protocol to normal socket, allowing a browser to connect to any application/server.
[+] : New Property "WatchDog"
Server: keeps server active automatically (disabled by default).
Client: reconnects automatically after an unexpected disconnection (disabled by default).
[+] : New Property "LocalIP" on TsgcWSConnection, returns IP Address of Host.
[+] : New Property "LocalPort" on TsgcWSConnection, returns Port Address of Host.
[+] : New Property "HeartBeat" on Server Components: sends a ping every x seconds (disabled by default).
[+] : New Property "IO_API" on TsgcWebSocketClient_SocketIO:
ioAPI0: supports socket.io 0.* servers (selected by default)
ioAPI1: supports socket.io 1.* servers

[*] : Fixed Bug writing data on TsgcWebSocketClient_SocketIO if connection is not Active.
[*] : Fixed Bug on search paths runtime packages from Delphi XE3-XE6.
[*] : Fixed DeadLock writing data on Server and Client components.
[*] : Fixed Bug reading JSON objects passed as parameters in SGC and Dataset Protocols.
[*] : Fixed Memory Leak OnException Event.
[*] : Fixed Bug Broadcast method using Exclude parameter.
[*] : Fixed Bug sending double quoted string inside a JSON parameter.
[*] : Fixed DeadLock when Keep-Alive is active on TsgcWebSocketHTTPServer.
[*] : Fixed Bug retrieving MIME-Type from file, when TsgcWebSocketHTTPServer tries to serve a file from DocumentRoot.
[*] : Fixed DeadLock on TsgcTimer class when compiled as a library.
[*] : Fixed Warnings.

[/] : Deleted property TsgcWSConnectionServer.ServerHost
[/] : Deleted property TsgcWSConnectionServer.ServerPort

3.1: 2014 May

[+] : Added support for Rad Studio XE6
[+] : Added support for C++Builder (2010-XE6).
[+] : Added support for old browsers (using Adobe Flash FallBack) without WebSocket implementation like Internet Explorer 6 to 9.
[+] : New WebSocket Extension: PerMessage-Deflate. Deflate-frame Extension has been deprecated and replaced by PerMessage-Deflate.
[+] : New Property "ReadTimeOut" on Server components, timeout to check if socket has received data.

[+] : New Property "ReadEmptySource" on HTTP Server component, max number of times an HTTP connection is read with no data.
[+] : New Property "Transport" on TsgcWSConnection, returns which transport is used (RFC6455, Hixie76, Flash or Undefinided).
[+] : New Property "SessionList" on TsgcWebSocketHTTPServer
[+] : New Property "ParseParams" on TsgcWebSocketHTTPServer

[*] : Fixed Bug on DCP paths design packages from Delphi XE2-XE5.
[*] : Fixed Bug reading if TCPConnection is connected.
[*] : Fixed Bug detecting connection state on javascript library.
[*] : Fixed Bug detecting type of message on javascript library.
[*] : Fixed Bug setting client log filename.
[*] : Fixed Bug sending a compressed close connection frame.

3.0: 2014 March

[+] : New Library "sgcWebSockets.dll" which allows to handle WebSocket (Server and Client) connections from Delphi or C# .Net (Demos inside Library Folder)
WebSocket Server
WebSocket Client
WebSocket SocketIO Client
WebSocket Server Protocol SGC
WebSocket Client Protocol SGC
[+] : New Property "IO_HandShakeTimestamp" on Socket.io client, if enabled allows to connect to gevent-socket.io servers

[*] : Fixed Bug calculating SendBytes with Binary Messages.
[*] : Fixed Bug TsgcWSConnection.WriteData when NotifyEvents = neNoSync.
[*] : Fixed Bug assigning ssl properties after set active.

[*] : Fixed Memory Leak in built-in libraries on some webbrowsers.
[*] : Fixed Bug OnException Event on NEXTGEN compilers.
[*] : Fixed Bug socket.io client on SendEvent procedure.
[*] : Fixed Bug processing http cache requests.

[/] : If you compile sgcWebSockets inside a DLL or a console application, there is no need to call CheckSynchronize manually.

2.6: 2013 November

[+] : Added support for Rad Studio XE5
[+] : Added support for Android
[+] : Added Basic Authentication support for VCL Websockets and HTTP Requests.
[+] : New for Protocols "sgc" and "Dataset": Added "Queue" param to the following methods: RPC, Publish and Notify:
Level 0: messages are not queued on Server
Level 1: last message is queued on Server, and is sent every time a client subscribes to a new channel or connects to server.
Level 2: all messages are queued on Server, and is sent every time a client subscribes to a new channel or connects to server.
[+] : New "QoS" Level (Quality of Service):
Level 2: messages are assured to arrive exactly once. If the acknowledgement message is not received after a specified period of time, the message is sent again.
[+] : New Property "Throttle": if enabled, limits the amount of bandwith usage.
[+] : New Demo: Server Authentication
[+] : Improved Documentation: new topics about features and general questions.

[*] : Fixed Bug with TsgcWebSocketServer on Windows XP.
[*] : Fixed Bug Connecting using a sub-protocol and using Authentication.

[*] : Fixed Bug in TsgcWebSocketServer causing clients to hang on unauthorized connection.
[*] : Fixed Bug in TsgcWebSocketHTPServer, http sessions were not created.

2.5: 2013 September

[+] : Improved Socket.IO client with new methods and events.
[+] : New Property "RawMessages" on Socket.IO client: if true, socket.io messages are not processed (it works as before 2.5 udpate); if false, it handles socket.io messages (false by default).
[+] : New for Protocols "sgc" and "Dataset": Added Support for transactional messages through server local transactions.
When a client Starts a Transaction on a channel, all messages sent to this channel are queued until client do a commit.
[+] : New for Protocols "sgc" and "Dataset": Added "QoS" (Quality of Service) property with 2 values:
Level 0: messages are delivered according to the best efforts of the underlying TCP/IP network (active by default)
Level 1: messages are assured to arrive but duplicates may occur. If the acknowledgement message is not received after a specified period of time, the message is sent again.
[+] : New for Protocols "sgc" and "Dataset": Added event "OnSession", it raises when server sends client session id.
By default is sent by server after a client connection.
[+] : New for Protocols "sgc" and "Dataset": Added Method GetSession. Allows to get server session id of connection.
[+] : New for Client Protocols "sgc" and "Dataset": Added event "OnAcknowledgment", an acknowledgment is sent by server to client when receives a message from client.
[+] : New for Server Protocol "Dataset": Added property "UpdateMode" [upWhereAll]: updates all fields of a record, [upWhereChanged]: updates only fields that have changed.

[+] : New for Client Protocol "Dataset": Added Event "OnMetaData" to get field structure of server dataset.
[+] : New for All Protocols: Added Event OnRawMessage , allows to handle protocol messages.
[+] : New Property "ConnectTimeout" on Client Components.
[+] : New Property "ReadTimeout" on Client Components.
[+] : New Property "Queue", when "true" queues string/binary messages and when "false" process all messages.
[+] : New Method "QueueClear", clear all queued messages.
[+] : New Events on TsgcWSPServer_Dataset:

+ OnBeforeNewRecord
+ OnBeforeUpdateRecord
+ OnBeforeDeleteRecord
+ OnBeforeDatasetUpdate
+ OnAfterNewRecord
+ OnAfterUpdateRecord
+ OnAfterDeleteRecord
+ OnRPCAuthentication
+ OnNotification
+ OnRPC
+ OnRPCAuthentication

[+] : New Events on TsgcWSPClient_Dataset:

+ OnBeforeNewRecord
+ OnBeforeUpdateRecord
+ OnBeforeDeleteRecord
+ OnBeforeDatasetUpdate
+ OnBeforeSynchronize
+ OnAfterSynchronize
+ OnRPCResult
+ OnRPCError
+ OnEvent

[+] : Property HandShake now is public on TsgcWSConnectionServer class.
[+] : Property Connection now is public on Client Component.
[+] : Added Interfaces directory with source code interfaces.

[*] : Fixed Bug sending Options.Parameters on SocketIO client.
[*] : Fixed Bug Handling Error Event.
[*] : Fixed Bug WAMP Server Protocol using publish method (messages were broadcasted to all clients, subscribed or not).
[*] : Fixed Bug Broker Server Component (events were forwarded to protocol components).
[*] : Fixed Bug WriteData Method if TCPConnection has been destroyed.
[*] : Fixed Bug UnSupported Protocol raised an access violation
[*] : Fixed Bug WAMP Server Protocol, now CallId is unique by client connection.
[*] : Fixed Bug getting IP from TsgcWSConnection.
[*] : Fixed Bug calling Synchronize Method first time on Protocol Dataset Client.
[*] : Fixed Bug Reading JSON Boolean values.
[*] : Fixed Bug Serving DocumentRoot files.
[*] : Fixed Memory Leak on Server Component.
[/] : OnConnectionData event has been removed. Now Connection.Data is read/write.
[/] : OnAfterNewRecord event Added Connection as Parameter.
[/] : OnAfterUpdateRecord event Added Connection as Parameter.
[/] : OnAfterDeleteRecord event Added Connection as Parameter.

2.4: 2013 June

[+] : Added support for Rad Studio XE4
[+] : Added support for NEXTGEN IOS Compiler
[+] : New Method "Disconnect": allows to disconnect a client connection from server side.
[+] : New Property "LogFile" (Server and Client components): save messages to a log file, useful for debugging.
[+] : Improved "Protocol DataSet": now synchronizes in Two ways (from server to client and from client to Server).

[+] : New Property "AutoSynchronize" (Protocol Dataset Server): synchronizes automatically all records OnConnect event.
[+] : New Method "Synchronize" (Protocol Dataset Client): synchronizes automatically all records from Server.
[+] : New Property "NotifyUpdates" (Protocol Dataset): allows to notify updates from server to client or client to server.
[+] : New Property "ApplyUpdates" (Protocol Dataset): allows to apply updates from server to client or client to server.
[+] : New Property "AllowNonAuth": allows to connect to non-authenticated users if authentication enabled
[+] : New Protocol Components: "Broker" allows to use more than one protocol using a single connection.

[*] : Fixed Bug loading resources when is a library
[*] : Fixed Bug OnException Event
[*] : Fixed Bug when a client has more than one protocol assigned
[*] : Fixed Bug handling multithreading messages
[*] : Fixed Bug Firefox Close Code
[-] : Removed all WITH statements

[/] : Removed sgcWebSocket_CS.pas, now it's defined on sgcWebSockets_Classes.pas

2.3: 2013 March

[+] : Added Support to Lazarus / FreePascal.
[+] : Added Support to Delphi 7 (upgrading to Indy 10)
[+] : Improved performance and reliability on websocket connections.
[+] : New Property "DocumentRoot": if defined, automatically publish http response files.
[+] : New Event "DataSession": allows to assign a user session object (database connection, user session data...)
[+] : New Property "Authentication": allows to authenticate WebSocket Connections using a user/password.

[+] : New Property "ThreadPool" on server components
[+] : New Property "AsyncEvents": allows to define which method used to notify websocket events: none, asynchronous (is the default) and synchronous.
[+] : New Property "HeartBeat": sends a ping every x seconds (disabled by default).
[+] : New Property "ValidateUTF8": validates utf8 messages (disabled by default).
[+] : New Property "JavascriptFiles" on server components: allows requests to server built-in javascript libraries.
[+] : New Property "HTMLFiles" on server components: allows requests to server built-in html files.

[*] : Fixed Bug OnHandShake event not fired
[*] : Fixed exception when socket closes connection
[*] : Fixed Bug when receiving fragmented message with control code
[*] : Fixed Bug when optcode is not recognized
[*] : Fixed Bug with RSV bits when no extension is negotiated
[*] : Fixed Bug when payload is empty
[*] : Fixed exception reading Guid
[*] : Fixed Bug with binary Ping Frames
[*] : Fixed Bug with Ping payload greater than 125 octets

[/] : Javascript libraries: binary messages are now received using "stream" event (before, this messages were received on "message" event).
[/] : Protocols are now based on JSON-RPC 2.0 (except WAMP protocol which uses his own protocol).

2.2: 2013 January

[+] : New WebSocket Protocol: WAMP
    Open Source SubProtocol
    Provides two asynchronous messaging patterns:
      - RPC
      - Subscription / Publish
[+] : New WebSocket Protocol: WebRTC

Open Project that enables web browsers with Real-Time Communications (RTC)
Currently Only Chrome it's supported
WebRTC is still on Beta Status
[+] : Added support for read/write JSON lists
[+] : Server Components: Added Arguments Exclude and Include List of guids
[+] : Javascript source is minified to save bandwith.
[+] : Added methods to send stream on sub-protocol base component.

[*] : Fixed Bug on Test Pages using ssl connection
[*] : Fixed Bug on Read UnMasked Frame
[*] : Fixed Bug on Client Component when Error #10054 raised.

2.1: 2012 November

[+] : New WebSocket Extension: Deflate-frame, allows to compress exchanged data.
[+] : Server Components: New Event "OnBinary" fired when a clients sends a binary message.
[+] : Server Components: Overloaded BroadCast and WriteData to allow to send streams.
[+] : Server Components: New Property "SSL", allows to stablish SSL connections with clients.
[+] : TsgcWebSocketClient: New Event "OnBinary" fired when client receives a binary message.
[+] : TsgcWebSocketClient: Overloaded WriteData to allow to send streams.
[+] : TsgcWebSocketClient: New Event "OnException" fired when a Exception is raised outside WebSocket Connection.
[+] : Server Components: New Property "Bindings".
[+] : Server Components: New Property "OriginsAllowed", on new connection, checks if origin is allowed (by default accepts all origins).
[+] : TsgcWebSocketClient: New Property "Options", allows to define which Parameters / Origin are sent on opening HandShake.

[+] : TsgcWSConnection: New Property "Port" (connection peer port).
[+] : TsgcWSConnection: New Property "RecBytes" (connection bytes received).
[+] : TsgcWSConnection: New Property "SendBytes", (connection bytes sent).

[*] : Fixed Bug re-connectiong from a SSL Client connection.
[*] : Fixed some memory leaks.
[*] : Fixed bug on Protocol Hixie76.
[*] : FIxed bug UTF-8 messages.

[/] : TsgcWebSocketClient: Property "Parameters" now is a Property of "Options"

2.0: 2012 October

[+] : RAD [+] : RAD Studio XE3 Supported
[+] : Added Support to Firemonkey (Win32, Win64 and MacOSX)
[+] : Added support to create custom sub-protocols
[+] : TsgcWebSocketHTTPServer: New Component! WebSocket Server that includes HTTP Server to share WebSocket and HTTP connections over the same port.
[+] : TsgcWebSocketClient_SocketIO: New Component! WebSocket Client that allows to connect to Socket.IO Servers
[+] : TsgcWSPServer_sgc: New Component! server that uses JSON to broadcast messages to all clients.
[+] : TsgcWSPClient_sgc: New Component! client that uses JSON messages to communicate with server.
[+] : TsgcWSPServer_dataset: New Component! broadcast dataset changes to all clients connected.
[+] : TsgcWSPClient_dataset: New Component! automatically updates client dataset from server changes.
[+] : TsgcWebSocketServer: Added javascript browser client support.
[+] : TsgcWebSocketServer: Added built-in html pages to test browser websocket connection.

[+] : TsgcWebSocketServer: New Property "MaxConnections"
[+] : TsgcWebSocketClient: New Property "Proxy", now supports WebSocket Protocol over HTTP Proxy Connections.
[+] : TsgcWebSocketClient: New Method "Ping"
[+] : TsgcWebSocketClient: New Property "MaxConnections"
[+] : TsgcWebSocketClient: New Property "Parameters", allows to pass parameters on Handshake

[*] : Fixed RangeCheck Error
[*] : Fixed Unicode Warnings
[*] : Fixed Active Property Error
[*] : TsgcWebSocketClient: OnConnect Event only fired if HandShake is correct

[/] : TsgcWebScocketServer: Deleted property Protocols
[/] : TsgcWebSocketClient: Deleted property Protocol
[/] : Subscription / UnSubscription methods now are implemented on protocol components.

1.2: 2012 June

[+] : TsgcWebSocketServer: Now Supports draft Hixie76, Safari and iOS browsers now are supported from version 4.2
[+] : TsgcIWWebSocketClient: New Property "Mode": [Native, Emulation] Allows to emulate socket connections
on webbrowsers that don't support websockets natively (like internet explorer).
[+] : TsgcWebSocketServer: New Property "Specifications": [RFC6455, Hixie76] defines which specifications are supported.
[+] : TsgcWebSocketClient: New Property "Specifications": [RFC6455, Hixie76] defines which specifications are supported.
[+] : TsgcWebSocketServer: Method "Broadcast" now accepts "Subscription" allowing to send custom messages only to subscribed clients

[+] : TsgcWebSocketServer: New Event "OnSubscription" fired when a client Subscribes to a channel
[+] : TsgcWebSocketServer: New Event "OnUnSubscription" fired when a client UnSubscribes to a channel
[+] : TsgcWebSocketClient: New Methods: "Subscribe" and "Unsubscribe" allowing to subscribe to custom channels
[+] : TsgcIWWebSocketClient: New Methods: "Subscribe" and "Unsubscribe" allowing to subscribe to custom channels

[*] : TsgcWebSocketServer: Fixed bug reading get parameters
[*] : TsgcWebSocketServer: Fixed bug OnMessage Event

1.1: 2012 May

[+] : TsgcWebSocketServer: allows to Send a Message to a single Client
[+] : TsgcWebSocketServer: New Property "Connections" to get access to all client connections
[+] : TsgcWebSocketServer: New Event "OnHandshake" allows to modify handshake before send to client
[+] : TsgcWebSocketClient: New Event "OnHandshake" allows to modify handshake before send to server
[+] : TsgcWebSocketClient: New Property "TLS" allows to stablish secure connections
[+] : TsgcIWWebSocketClient: New Property "TLS" allows to stablish secure connections
[+] : TsgcWebSocketServer: Event "OnDisconnect" now has "Code" to get close reason if applicable

[*] : TsgcWebSocketclient: Fixed error assigning properties at runtime
[*] : TsgcWebSocketServer: Fixed broadcast error on Chrome (v.19)
[*] : TsgcIWWebSocketClient: Fixed close connection error on Chrome (v.19)
[*] : TsgcWebSocketClient: error reading handshake status

[*] : TsgcWebSocketClient: in some environments, OnDisconnect event was not fired
[*] : TsgcIWWebSocketClient: Fixed close connection error

[/] : TsgcWebSocketServer: Event "OnDisconnect" introduced parameter "Code"
[/] : TsgcWebSocketClient: Event "OnDisconnect" introduced parameter "Code"

1.0: 2012 April

[+] : First Version

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.

WebBrowser 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"></scrip
t>
</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 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 server. On Server Side, you will see a message with text sent by Web Browser Client.

WebBrowser Test

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

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 following WebSocket methods:

Open
Close
Status
Send

Topics

Features

Authentication

Supported by

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

WebSockets Specification doesn't has 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 need 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. 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/p
assword
```

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 webbrowsers 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 user is not in AuthUsers list, client don'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 server, you can access to 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 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

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 client with this protocol activated and all messages will be compressed; if Server doesn't has 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 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 Browser doesn't support websockets natively. So, if you enable Flash Fallback on 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 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 Server have more than one IP, if you enable a WebSocket Server and set listening port to 80, when 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 allows to define 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 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
Java script

SGC Default Protocol 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 message to subscribers

Level 1

The receipt of a message by the server is acknowledged by a 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 a Message ID in the message.

The table below shows the QoS level 1 protocol flow.

Client	Message and direction	Server
QoS = 1 Message ID = x **Action:** Store message	PUBLISH ---------->	**Actions:** • Store message • Publish message to subscribers • Delete message
Action: Discard message	ACKNOWLEDGMENT <----------	

If the client does not receive a 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 message to subscribers • Delete message
Action: Discard message	ACKNOWLEDGMENT <----------	Message ID = x

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

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

Level 1

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

The table below shows the Queue level 2 protocol flow.

Client	Message and direction	Server
Queue = 2	PUBLISH ---------->	**Action:** Store message
Action: Process message	NOTIFY <----------	**Action:** Every time a new client subscribes to this

		channel, ALL Messages are sent.

Transactions

Supported by

TsgcWSPServer_sgc
TsgcWSPClient_sgc
Java script

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

StartTransaction

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

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

Commit

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

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

RollBack

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

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

HTTP

Supported by

TsgcWebSocketHTTPServer

TsgcWebSocketHTTPServer is a component that allows to handle WebSocket and HTTP connections using the SAME port. Is very useful when you need to setup 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 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 amount of bits per second sent/received by socket. Indy uses a blocking method, so if a client is limiting its reading, unread data will be inside client socket and server will be blocked from writing new data to 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 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 be 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 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.

<u>retry</u>

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);
```

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 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 to send 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. Progress of file send and received.
4. Authorization of files received.
5. Acknowledgment of packets sent.

Proxy

Supported by

TsgcWebSocketClient
TsgcWebSocketClient_SocketIO

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

Proxy / Enabled

Once set to True, you can setup

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

Logs

Supported by

[TsgcWebSocketServer]
[TsgcWebSocketHTTPServer]
[TsgcWebSocketClient]
[TsgcWebSocketClient_SocketIO]

This is a useful feature that allows to debug 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 file it's not created it will be created but with you can't access until server is closed, if you want to open log file while 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
Event raised when a new connection is established.

OnDisconnect
Event raised when a connection is closed.

OnError
Event raised when a connection has any error.

OnMessage
Event raised when a new text message is received.

OnBinary
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 operating system when is allowed. This behavior 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 you need to handle a lot of messages on a very short period of time, 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 connection is established. If you need to pass some parameters to server, you can use the following property:

Options / Parameters

By default, si 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"></scrip
t>
<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 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 handkshake), 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 server side.

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

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

OnStartup: raised when a server is started.

OnShutdown: raised when a server is 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, message will be sent only to subscribers.

Protocol: if defined, message will be sent only to 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 a 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 connection.

DisconnectAll: disconnects all active connections.

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

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

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

Bindings: used to manage IP and Ports.

Connections: contains a list with 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 browser, you can enable the following fallbacks:

Flash: if enabled, if 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 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.

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 this 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 to handle Fragmented Messages

frgOnlyBuffer: 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 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 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 limit is reached, then connection is closed.

SecurityOptions:

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

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

- **PoolSize:** size of ThreadPool, by default is 32.

WatchDog: if enabled, restart server after unexpected disconnection.

- **Interval:** seconds before reconnects.

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

Throttle: used to limit the amount 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 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 handkshake), 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 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 client and enable/disable Authenticated Variable.

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, message will be sent only to subscribers.

Protocol: if defined, message will be sent only to 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 a 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 with 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, 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 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.

TsgcWebSocketClient

TsgcWebSocketClient implements Client VCL WebSocket Component and can connect to a WebSocket Server. Follow 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.

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

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

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

5. If 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.

HIxie76: 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 server sends a text message, this event is fired

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

OnHandhake: this event is fired when handshake is evaluated on 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, packet will be split if size of message is greater of size.

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

Properties

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

Implements 2 types of WebSocket Authentication

Session: client need 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 server.

HeartBeat: if enabled try to keeps alive websocket a 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 miliseconds before a connection is ready.

ReadTimeout: max time in miliseconds to read messages.

Port: Port used to connect to 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 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 this 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 to customize headers sent on handshake.

FragmentedMessages: allows to handle Fragmented Messages

frgOnlyBuffer: 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 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 message contains UTF8 valid characters, by default is disabled.

Extensions: you can enable compression on messages sent.

Protocol: if exists, shows current protocol used

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

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

> **Interval:** seconds before reconnects.
>
> **Attempts:** max number of reconnects, if zero, then unlimited.

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

TLS: enables secure connection.

TsgcWebSocketClient_SocketIO

TsgcWebSocketClient_SocketIO inherits all properties and methods from TsgcWebSocketClient and allows to connect 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 a 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 evert 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 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.

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

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

SendACK: An acknowledgment 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;
```

Properties

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 to 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 to customize url to get socket.io session.

TsgcWebSocketProxyServer

TsgcWebSocketProxyServer implements a WebSocket Server Component which listens client websocket connections and forward data connections to a normal TCP/IP server. This is specially 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 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.

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 don'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 handkshake), this event is fired

OnAsyncMessage: every time 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 connection is Active, otherwise returns False.

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

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

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

JSError: here you can include JavaScript Code on 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 server side. A "CloseCode" can be specified optionally.

Ping: sends a ping to the client.

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

Properties

Data: user session data object, here you can pass a object and access this every time you need, 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 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 Server side, it will validate if sub-protocol sent by client is supported by 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 same communication interface.

Although 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 Default: implemented using JSON-RPC 2.0 messages, provides 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 record) from server to clients.

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

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

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

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).

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 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%}","use
r":"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 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.
- 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 favorite WebBrowser, 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

Publish: sends a message to all subscribed clients.

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

RPCError: if a call RPC from client it has an error, 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 a single or multiple selected clients.

Properties

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

Methods: is a list of allowed methods. Every time a client sends a 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.

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 server receives a 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 to deny 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: client sends a notification to a server, this notification don'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 client is subscribed, OnSubscription event will be fired.

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

UnsubscribeAll: unsubscribe client from all subscribed channel. If 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 client receives successful response from server after a RPC is sent.

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

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

OnRawMessage: this event is fired before a message is processed by 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 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"></s
cript>
```

Open Connection

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

Get Connection Status

```
<script
src="http://{%host%}:{%port%}/sgcWebSockets.js"></
script>
<script
src="http://{%host%}:{%port%}/esegece.com.js"></sc
ript>
<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"></sc
ript>
<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"></sc
ript>
<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 component to search records on 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 favorite WebBrowser, 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 server receives a dataset update from client, it will be saved on server side.

NotifyUpdates: if enabled, every time dataset server changes, server broadcasts this change 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 server, 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.

Events

These events are specific on 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 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 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 handles this event and if won't be updated (by default this argument is False)

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 server

GetMetaData: requests all dataset fields from server

Events

These events are specific on 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 a synchronization has ended.

OnBeforeDeleteRecord: event fired before a record is deleted from Dataset. If Argument "Handled" is True, means that 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 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 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 client receives a dataset update from server, it will be saved on client side.

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

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

UpdateMode: if umWhereAll (by default) all fields are transmitted to server, if umWhereChanged only Fields that have changed will be transmitted to 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 to send 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: object which encapsulates file packet information.

TsgcWSPServer_Files

This is 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 comma, which you don't want to send this file.
- **aInclude:** connection guids separated by 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: 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: message is sent.

qosLevel1: message is sent and you get an acknowledgment if message has been processed.

qosLevel2: message is sent, you get an acknowledgment if message has been processed and packets are requested by 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 file can be received.

OnFileReceivedError: fired when an error occurs receiving a file.

OnFileReceivedFragment: fired when a fragment file is received. Useful to show a progress.

OnFileSent: fired when a file is successfully sent.

OnFileSentAcknowledgment: fired when a fragment is sent and receiver has processed.

OnFileSentError: fired when an error occurs sending a file.

OnFileSentFragment: fired when a fragment file is sent. Useful to show a progress.

TsgcWSPClient_Files

This is 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 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 server.

aQoS: type of quality of service.

Properties

Files: files properties

BufferSize: default size of every packet sent, in bytes.

SaveDirectory: 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: message is sent.

qosLevel1: message is sent and you get an acknowledgment if message has been processed.

qosLevel2: message is sent, you get an acknowledgment if message has been processed and packets are requested by 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 file can be received.

OnFileReceivedError: fired when an error occurs receiving a file.

OnFileReceivedFragment: fired when a fragment file is received. Useful to show a progress.

OnFileSent: fired when a file is successfully sent.

OnFileSentAcknowledgment: fired when a fragment is sent and receiver has processed.

OnFileSentError: fired when an error occurs sending a file.

OnFileSentFragment: fired when a fragment file is sent. Useful to show a progress.

TsgcWSMessageFile

This object is passed as a parameter every time a file protocol event is raised.

Properties

- BufferSize: default size of 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 user can set whatever text.
- FileName: name of 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 message.
- Streaming: for internal use.
- Text: for internal use.

WebRTC

Protocol WebRTC

WebRTC (Web Real-Time Communication) is an API definition being drafted by the World Wide Web Consortium (W3C) to enable browser to browser applications for voice calling, video chat and P2P file sharing without plugins.The RTC in WebRTC stands for Real-Time Communications, 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 favorite WebBrowser, 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.j
s"></script>
<script>
  var socket = new
sgcws_webrtc('ws://{%host%}:{%port%}');
</script>
```

Open WebRTC Channel
When a browser has accessed 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.j
s"></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.j
s"></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 it's 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 favorite WebBrowser, 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 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 client when request 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 server receives RPC called by client

- **CallId:** this is the ID generated by client when request 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, 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 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 client when request 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 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 client receive 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>
```

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 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 to customize Server and Client size of 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 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 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 zlib version mismatch with initialization functions, to fix this, just update your Indy version to latest.*

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 defines 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(WSCli
entConnect);
sgcWebSocketLib.Instance.Client_OnDisconnect(WS
ClientDisconnect);
sgcWebSocketLib.Instance.Client_OnError(WSClien
tError);
sgcWebSocketLib.Instance.Client_OnMessage(WSCli
entMessage);
```

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('Hell
o 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.LPWS
tr)] 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(OnCon
nectEvent);
sgcWebSocketLib.Instance.Client_OnDisconnect(On
DisconnectEvent);
sgcWebSocketLib.Instance.Client_OnError(OnError
);
sgcWebSocketLib.Instance.Client_OnMessage(OnMes
sageEvent);
```

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('Hell
o 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 = '')**

Event raised when a new websocket connection is opened.

**Client_OnDisconnect(aOnDisconnect: TsgcWSClient_OnDisconnect;
aID: WideString = '')**

Event raised when a websocket connection is closed.

**Client_OnMessage(aOnMessage: TsgcWSClient_OnMessage;
aID: WideString = '')**

Event raised when client receives a new text message from server.

**Client_OnBinary(aOnBinary: TsgcWSClient_OnBinary;
aID: WideString = '')**

Event raised when client receives a new Binary message from server.

Client_OnError(aOnError: TsgcWSClient_OnError; aID: WideString = '')

Event raised when there is any 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_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 evert 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 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 acknowledgment 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 = '')**

Event raised when a new websocket connection is opened.

**Client_SocketIO_OnDisconnect(aOnDisconnect: TsgcWSClient_OnDisconnect;
aID: WideString = '')**

Event raised when a websocket connection is closed.

**Client_SocketIO_OnMessage(aOnMessage: TsgcWSClient_OnMessage;
aID: WideString = '')**

Event raised when client receives a new text message from server.

**Client_SocketIO_OnBinary(aOnBinary: TsgcWSClient_OnBinary;
aID: WideString = '')**

Event raised when client receives a new Binary message from server.

Client_SocketIO_OnError(aOnError: TsgcWSClient_OnError; aID: WideString = '')

Event raised 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 = '')

Event raised when a client receives a Disconnect message.

Client_SocketIO_OnMessageConnect(aOnMessageConnect: TsgcWSClient_SocketIO_OnMessageConnect; aID: WideString = '')

Event raised when a client receives a Connect message.

Client_SocketIO_OnMessageHeartBeat(aOnMessageHeartBeat: TsgcWSClient_SocketIO_OnMessageHeartBeat; aID: WideString = '')

Event raised when a client receives a HeartBeat message.

Client_SocketIO_OnMessageEvent(aOnMessageEvent: TsgcWSClient_SocketIO_OnMessageEvent; aID: WideString = '')

Event raised when a client receives an Event Message.

Client_SocketIO_OnMessageText(aOnMessageText: TsgcWSClient_SocketIO_OnMessageText; aID: WideString = '')

Event raised when a client receives a Text message.

Client_SocketIO_OnMessageJSON(aOnMessageJSON:

TsgcWSClient_SocketIO_OnMessageJSON; aID: WideString = '')

Event raised when a client receives a JSON message.

Client_SocketIO_OnMessageACK(aOnMessageACK: TsgcWSClient_SocketIO_OnMessageACK; aID: WideString = '')

Event raised when a client receives an Acknowledgment message.

Client_SocketIO_OnMessageError(aOnMessageError: TsgcWSClient_SocketIO_OnMessageError; aID: WideString = '')

Event raised when a client receives an Error message.

Client_SocketIO_OnMessageNoop(aOnMessageNoop: TsgcWSClient_SocketIO_OnMessageNoop; aID: WideString = '')

Event raised 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 = '')

Unsuscribe from a channel.

Client_sgc_Broadcast(aText, aChannel: WideString; aID: WideString = '')

Sends a Text message to all clients connected to an 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 server.

Client_sgc_Publish(aText, aChannel: WideString; aID: WideString = '')

Sends a Text message to all clients connected to an 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 = '')**

Event raised when a new websocket connection is opened.

**Client_sgc_OnDisconnect(aOnDisconnect: TsgcWSClient_OnDisconnect;
aID: WideString = '')**

Event raised when a websocket connection is closed.

**Client_sgc_OnMessage(aOnMessage: TsgcWSClient_OnMessage;
aID: WideString = '')**

Event raised when client receives a new text message from server.

**Client_sgc_OnBinary(aOnBinary: TsgcWSClient_OnBinary;
aID: WideString = '')**

Event raised when client receives a new Binary message from server.

Client_sgc_OnError(aOnError: TsgcWSClient_OnError; aID: WideString = '')

Event raised when there is any error on websocket connection.

Client_sgc_OnSubscription(aOnSubscription: TsgcWSP_sgc_OnSubscription; aID: WideString = '')

Event raised if a new subscription is received from server.

Client_sgc_OnUnSubscription(aOnUnSubscription: TsgcWSP_sgc_OnUnSubscription; aID: WideString = '')

Event raised if an unsubscription is received from server.

Client_sgc_OnEvent(aOnEvent : TsgcWSPClient_sgc_OnEvent; aID: WideString = '')

Event raised when a new event is received.

Client_sgc_OnSession(aOnSession : TsgcWSPClient_sgc_OnSession; aID: WideString = '')

Event raised when a Session ID is received from server.

Client_sgc_OnAcknowledgment(aOnAcknowledgment : TsgcWSPClient_sgc_OnAcknowledgment; aID: WideString = '')

Event raised when client receives an Acknowledgment from server that message has been received.

Client_sgc_OnRPCError(aOnRPCError: TsgcWSPClient_sgc_OnRPCError; aID: WideString = '')

Event raised if server sends an error as response to a Remote Procedure Call.

Server_sgc_OnRPCResult(aOnRPCResult : TsgcWSPClient_sgc_OnRPCResult; aID: WideString = '')

Event raised when server sends a response to 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_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}

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, realtime multimedia applications (think video chat) on the web, without requiring plugins, downloads or installs. It's purpose is to help build a strong RTC platform that works across multiple web browsers, across multiple platforms.

More Information

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 is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The eSeGeCe components is licensed, not sold.
The eSeGeCe components is a freeware. You may evaluate it for free and You can use the eSeGeCe components for commercial purpose. 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 is 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 was designed [e.g., Windows 95, Windows NT, Windows 98, Windows 2000, Windows 2003, Windows XP, Windows ME, Windows Vista, Windows 7].
(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 is 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 is 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