# WebSocket-Sharp for Unity

Provides the **WebSocket** protocol client and server for your Unity apps.

Supports:

- **WebSocket Client** and **Server**
- **RFC 6455**
- **Per-message Compression** extension
- **Secure Connection**
- **HTTP Authentication**
- .NET **3.5** or later (includes compatible)

## Before Starting

If you succeed in downloading and importing **WebSocket-Sharp for Unity**, could you see if **WebSocket-Sharp** menu item exists under **Window** menu item in the Unity Editor?

If it exists, could you try both **Echo Back Test** and **About WebSocket-Sharp** under the WebSocket-Sharp menu item?

And then if you obtain the same results as the screenshots of **WebSocket-Sharp for Unity** on the Unity Asset Store, it's available.

But if it doesn't, could you see if you succeeded in downloading and importing **WebSocket-Sharp for Unity**?

## Getting Started

### WebSocket Client

```
using System;
using UnityEditor;
using UnityEngine;
using WebSocketSharp;

namespace WebSocketSharp.Unity.Editor
{
  public class MenuExtension : MonoBehaviour
```

```
  {
    [MenuItem ("websocket-sharp/Echo Back Test")]
    private static void EchoBack ()
    {
      string res = null;
      using (var ws = new WebSocket ("ws://localhost:4649/Echo")) {
        var ver = Application.unityVersion;
        ws.OnOpen += (sender, e) =>
          ws.Send (String.Format ("Hello, Unity {0}!", ver));

        ws.OnMessage += (sender, e) =>
          res = e.Data;

        ws.OnError += (sender, e) =>
          Debug.LogError (e.Message);

        ws.Connect ();
      }

      if (!res.IsNullOrEmpty ())
        EditorUtility.DisplayDialog ("Echo Back Successfully!", res, "OK");
    }
  }
}
```

**Step 1** Required namespace.

```
using WebSocketSharp;
```

The `WebSocket` class exists in the `WebSocketSharp` namespace.

**Step 2** Creating an instance of the `WebSocket` class with the WebSocket URL to connect.

```
using (var ws = new WebSocket ("ws://example.com")) {
  ...
}
```

The `WebSocket` class inherits the `System.IDisposable` interface, so you can use the `using` statement.

**Step 3** Setting the `WebSocket` events.

**WebSocket.OnOpen Event** A `WebSocket.OnOpen` event occurs when the WebSocket connection has been established.

```
ws.OnOpen += (sender, e) => {
  ...
};
```

`e` has passed as the `System.EventArgs.Empty`, so you don't use `e`.

**WebSocket.OnMessage Event** A `WebSocket.OnMessage` event occurs when the `WebSocket` receives a message.

```
ws.OnMessage += (sender, e) => {
  ...
};
```

`e` has passed as a `WebSocketSharp.MessageEventArgs`.

`e.Type` property returns either `WebSocketSharp.Opcode.TEXT` or `WebSocketSharp.Opcode.BINARY` that represents the type of the received message. So by checking it, you determine which item you should use.

If `e.Type` is `Opcode.TEXT`, you should use `e.Data` property (returns a `string`) that represents the received **Text** message.

Or if `e.Type` is `Opcode.BINARY`, you should use `e.RawData` property (returns a `byte []`) that represents the received **Binary** message.

```
if (e.Type == Opcode.TEXT) {
  // Do something with e.Data
  return;
}

if (e.Type == Opcode.BINARY) {
  // Do something with e.RawData
  return;
}
```

**WebSocket.OnError Event** A `WebSocket.OnError` event occurs when the `WebSocket` gets an error.

```
ws.OnError += (sender, e) => {
  ...
};
```

`e` has passed as a `WebSocketSharp.ErrorEventArgs`.

`e.Message` property returns a `string` that represents the error message. So you should use it to get the error message.

**WebSocket.OnClose Event** A `WebSocket.OnClose` event occurs when the WebSocket connection has been closed.

```
ws.OnClose += (sender, e) => {
  ...
};
```

`e` has passed as a `WebSocketSharp.CloseEventArgs`.

`e.Code` property returns a `ushort` that represents the status code indicating the reason for closure, and `e.Reason` property returns a `string` that represents the reason for closure. So you should use them to get the reason for closure.

**Step 4** Connecting to the WebSocket server.

```
ws.Connect ();
```

If you would like to connect to the server asynchronously, you should use the `WebSocket.ConnectAsync ()` method.

**Step 5** Sending a data to the WebSocket server.

```
ws.Send (data);
```

The `WebSocket.Send` method is overloaded.

You can use the `WebSocket.Send (string)`, `WebSocket.Send (byte [])`, and `WebSocket.Send (System.IO.FileInfo)` methods to send a data.

If you would like to send a data asynchronously, you should use the `WebSocket.SendAsync` method.

```
ws.SendAsync (data, completed);
```

And if you would like to do something when the send is complete, you should set `completed` to any `Action<bool>`.

**Step 6** Closing the WebSocket connection.

```
ws.Close (code, reason);
```

If you would like to close the connection explicitly, you should use the `WebSocket.Close` method.

The `WebSocket.Close` method is overloaded.

You can use the `WebSocket.Close ()`, `WebSocket.Close (ushort)`, `WebSocket.Close (WebSocketSharp.CloseStatusCode)`, `WebSocket.Close (ushort, string)`, or `WebSocket.Close (WebSocketSharp.CloseStatusCode, string)` method to close the connection.

If you would like to close the connection asynchronously, you should use the `WebSocket.CloseAsync` method.

### WebSocket Server

```
using System;
using UnityEditor;
using UnityEngine;
using WebSocketSharp.Server;

namespace WebSocketSharp.Unity.Editor
{
  public class ServerMonitor : EditorWindow
  {
    WebSocketServer _server;

    ServerMonitor ()
    {
      _server = new WebSocketServer (4649);
      _server.AddWebSocketService<Echo> ("/Echo");
      _server.AddWebSocketService<Chat> ("/Chat");
      _server.Start ();
    }

    void OnDestroy ()
    {
      if (_server != null)
        _server.Stop ();
    }

    void OnGUI ()
    {
```

```
      GUILayout.Label ("A WebSocket server started!", EditorStyles.boldLabel);
      if (GUILayout.Button ("Close", GUILayout.Width (100)))
        Close ();
    }
  }
}
```

**Step 1** Required namespace.

```
using WebSocketSharp.Server;
```

The `WebSocketServer` and `WebSocketService` classes exist in the `WebSocketSharp.Server` namespace.

**Step 2** Creating the class that inherits the `WebSocketService` class.

For example, if you would like to provide an echo service,

```
using System;
using WebSocketSharp;
using WebSocketSharp.Server;

public class Echo : WebSocketService
{
  protected override void OnMessage (MessageEventArgs e)
  {
    Send (e.Data);
  }
}
```

And if you would like to provide a chat service,

```
using System;
using WebSocketSharp;
using WebSocketSharp.Server;

public class Chat : WebSocketService
{
  private string _suffix;

  public Chat ()
    : this (null)
  {
```

```
  }

  public Chat (string suffix)
  {
    _suffix = suffix ?? String.Empty;
  }

  protected override void OnMessage (MessageEventArgs e)
  {
    Sessions.Broadcast (e.Data + _suffix);
  }
}
```

If you override the `WebSocketService.OnMessage (MessageEventArgs)` method, it's called when the `OnMessage` event of the `WebSocket` used by the current session in the WebSocket service occurs.

And if you override the `WebSocketService.OnOpen ()`, `WebSocketService.OnError (ErrorEventArgs)`, and `WebSocketService.OnClose (CloseEventArgs)` methods, each of them is called when each event of the `WebSocket` (the `OnOpen`, `OnError`, and `OnClose`) occurs.

The `WebSocketService.Send` method sends a data to the client on the current session in the WebSocket service.

If you would like to access the sessions in the WebSocket service, you should use the `WebSocketService.Sessions` property (returns a `WebSocketSharp.Server.WebSocketSessionManager`).

The `WebSocketService.Sessions.Broadcast` method broadcasts a data to every client in the WebSocket service.

**Step 3** Creating an instance of the `WebSocketServer` class.

```
var wssv = new WebSocketServer (4649);
wssv.AddWebSocketService<Echo> ("/Echo");
wssv.AddWebSocketService<Chat> ("/Chat");
wssv.AddWebSocketService<Chat> ("/ChatWithNiceBoat", () => new Chat (" Nice boat."));
```

You can add any WebSocket service to your `WebSocketServer` with the specified path to the service, using the `WebSocketServer.AddWebSocketService<TWithNew> (string)` or `WebSocketServer.AddWebSocketService<T> (string, Func<T>)` method.

The type of `TWithNew` must inherit the `WebSocketService` class and must have a public parameterless constructor.

The type of `T` must inherit the `WebSocketService` class.

So you can use the classes created in **Step 2** to add the WebSocket service.

**Step 4** Starting the WebSocket server.

```
wssv.Start ();
```

**Step 5** Stopping the WebSocket server.

```
wssv.Stop (code, reason);
```

The `WebSocketServer.Stop` method is overloaded.

You can use the `WebSocketServer.Stop ()`, `WebSocketServer.Stop (ushort, string)`, or `WebSocketServer.Stop (WebSocketSharp.CloseStatusCode, string)` method to stop the server.

### HTTP Server with the WebSocket

I modified the `System.Net.HttpListener`, `System.Net.HttpListenerContext`, and some other classes of **Mono** to create the HTTP server that allows to accept the WebSocket connection requests.

So websocket-sharp provides the `WebSocketSharp.Server.HttpServer` class.

You can add any WebSocket service to your `HttpServer` with the specified path to the service, using the `HttpServer.AddWebSocketService<TWithNew> (string)` or `HttpServer.AddWebSocketService<T> (string, Func<T>)` method.

```
var httpsv = new HttpServer (4649);
httpsv.AddWebSocketService<Echo> ("/Echo");
httpsv.AddWebSocketService<Chat> ("/Chat");
httpsv.AddWebSocketService<Chat> ("/ChatWithNiceBoat", () => new Chat (" Nice boat."));
```

### WebSocket Extensions

**Per-message Compression** websocket-sharp supports the **Per-message Compression** extension. (But it doesn't support with the extension parameters.)

If you would like to enable this extension as a WebSocket client, you should set like the following.

```
ws.Compression = CompressionMethod.DEFLATE;
```

And then your client sends the following header with the connection request to the server.

```
Sec-WebSocket-Extensions: permessage-deflate
```

If the server supports this extension, it returns the same header. And when your client receives that header, it enables this extension.

### Secure Connection

websocket-sharp supports the **Secure Connection (SSL)**.

As a **WebSocket Client**, you should create an instance of the `WebSocket` class with the **wss** scheme WebSocket URL to connect.

```
using (var ws = new WebSocket ("wss://example.com")) {
  ...
}
```

And if you would like to set the custom validation for the server certificate, you should set the `WebSocket.ServerCertificateValidationCallback` property.

```
ws.ServerCertificateValidationCallback = (sender, certificate, chain, sslPolicyErrors) => {
  // Do something to validate the server certificate.
  return true; // If the server certificate is valid.
};
```

If you set this property to nothing, the validation does nothing with the server certificate and returns valid.

As a **WebSocket Server**, you should create an instance of the `WebSocketServer` or `HttpServer` class with some settings for the secure connection. It's like the following.

```
var wssv = new WebSocketServer (4649, true);
wssv.Certificate = new X509Certificate2 ("/path/to/cert.pfx", "password for cert.pfx");
```

### HTTP Authentication

websocket-sharp supports the **HTTP Authentication (Basic/Digest)**.

As a **WebSocket Client**, you should set a pair of user name and password for the HTTP authentication, using the `WebSocket.SetCredentials (string, string, bool)` method before connecting.

```
ws.SetCredentials (username, password, preAuth);
```

If `preAuth` is `true`, the `WebSocket` sends the Basic authentication credentials with the first connection request to the server.

Or if `preAuth` is `false`, the `WebSocket` sends either the Basic or Digest authentication (determined by the unauthorized response to the first connection request) credentials with the second connection request to the server.

As a **WebSocket Server**, you should set an HTTP authentication scheme, a realm, and any function to find the user credentials before starting. It's like the following.

```
wssv.AuthenticationSchemes = AuthenticationSchemes.Basic;
wssv.Realm = "WebSocket Test";
wssv.UserCredentialsFinder = identity => {
  var expected = "nobita";
  return identity.Name == expected
         ? new NetworkCredential (expected, "password", "gunfighter") // User name, passwor
         : null; // If the user credentials not found.
};
```

If you would like to provide the Digest authentication, you should set like the following.

```
wssv.AuthenticationSchemes = AuthenticationSchemes.Digest;
```

### Logging

The `WebSocket` class includes the own logging function.

You can access it with the `WebSocket.Log` property (returns a `WebSocketSharp.Logger`).

So if you would like to change the current logging level (`WebSocketSharp.LogLevel.ERROR` as the default), you should set the `WebSocket.Log.Level` property to any of the `LogLevel` enum values.

```
ws.Log.Level = LogLevel.DEBUG;
```

This means a log with less than `LogLevel.DEBUG` cannot be outputted.

And if you would like to output a log, you should use any of the output methods. The following outputs a log with `LogLevel.DEBUG`.

```
ws.Log.Debug ("This is a debug message.");
```

The `WebSocketServer` and `HttpServer` classes include the same logging function.

## Using in the Unity Web Player

As a **WebSocket Client**, it requires a **Socket Policy Server** for the Unity Web Player.

### If you access to the local WebSocket server from your apps in the Unity Web Player

Unity includes a simple Socket Policy Server (sockpol.exe).

So you should start the server before starting your apps.

The following is starting the server in command prompt at my environment (Unity 4.2, Win 8).

```
>cd C:\Program Files (x86)\Unity\Editor\Data\Tools\SocketPolicyServer\
>sockpol.exe --all
```

And you should call the `Security.PrefetchSocketPolicy` method in your apps code before calling the `WebSocket.Connect` method, like the following.

```
Security.PrefetchSocketPolicy ("localhost", 843);
...
ws.Connect ();
```

### If you access to the remote WebSocket server from your apps in the Unity Web Player

It requires a Socket Policy Server on the same remote host as the WebSocket server.

And you should call the `Security.PrefetchSocketPolicy` method in your apps code before calling the `WebSocket.Connect` method, like the following.

```
Security.PrefetchSocketPolicy (
  "remote host domain name or ip address", port_number_of_socket_policy_server);
...
ws.Connect ();
```

As a **WebSocket Server**, it isn't available because Unity doesn't allow to listen the TCP Sockets in the Unity Web Player.

See also: **Security Sandbox of the Webplayer**

## Documentation

- **websocket-sharp Library Reference**

## Supported WebSocket Specifications

websocket-sharp supports **RFC 6455** and is based on the following WebSocket references.

- **The WebSocket Protocol**
- **The WebSocket API**
- **Compression Extensions for WebSocket**

## websocket-sharp Project Site

- **websocket-sharp by sta**