WebSocket Server in C#

Dave Haig

24 Dec 2017 CPOL

Web Socket Client and Server implemented in C# for the modern version 13 of the WebSocket protocol

Download WebSockets - 47 KB (Relates to this article)

Latest async version on Github (Targets .NetStandard 2.0)

Set `WebSocketsCmd` as the startup project.

Introduction

NEW - Complete refactor to target .NetStandard 2.0 (See GitHub link above)

NEW - Better logging for SSL errors

NEW - SSL support added. Secure your connections

NEW - C# client support added. Refactor for a better API

No external libraries. Completely standalone.

A lot of the Web Socket examples out there are for old Web Socket versions and included complicated code (and external libraries) for fall back communication. All modern browsers support at least **version 13 of the Web Socket protocol** so I'd rather not complicate things with backward compatibility support. This is a bare bones implementation of the web socket protocol in C# with no external libraries involved. You can connect using standard HTML5 JavaScript or the C# client.

This application serves up basic HTML pages as well as handling WebSocket connections. This may seem confusing but it allows you to send the client the HTML they need to make a web socket connection and also allows you to share the same port. However, the `HttpConnection` is very rudimentary. I'm sure it has some glaring security problems. It was just made to make this demo easier to run. Replace it with your own or don't use it.

Background

There is nothing magical about Web Sockets. The spec is easy to follow and there is no need to use special libraries. At one point, I was even considering somehow communicating with Node.js but that is not necessary. The spec can be a bit fiddly but this was probably done to keep the overheads low. This is my first CodeProject article and I hope you will find it easy to follow. The following links offer some great advice:

Step by step guide:

- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

The official Web Socket spec:

- http://tools.ietf.org/html/rfc6455

Some useful stuff in C#:

- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_server

Alternative C# implementations:

- Microsoft ASP.NET SignalR - http://www.asp.net/signalr
- Nuget available library by sta - https://github.com/sta/websocket-sharp

Using the Code

When you first run this app, you should get a Windows firewall warning popup message. Just accept the warning and add the automatic firewall rule. Whenever a new app listens on a port (which this app does), you will get this message and it is there to point out nefarious applications potentially sending and receiving unwanted data over your network. All the code is there for you to see so you can trust what's going on in this project.

A good place to put a breakpoint is in the `WebServer` class in the `HandleAsyncConnection` function. Note that this is a multithreaded server so you may want to freeze threads if this gets confusing. The console output prints the thread id to make things easier. If you want to skip past all the plumbing, then another good place to start is the `Respond` function in the `WebSocketConnection` class. If you are not interested in the inner workings of Web Sockets and just want to use them, then take a look at the `OnTextFrame` in the `ChatWebSocketConnection` class. See below.

Implementation of a chat web socket connection is as follows:

```
internal class ChatWebSocketService : WebSocketService
{
    private readonly IWebSocketLogger _logger;

    public ChatWebSocketService(NetworkStream networkStream,
                                TcpClient tcpClient, string header, IWebSocketLogger logger)
        : base(networkStream, tcpClient, header, true, logger)
    {
        _logger = logger;
    }

    protected override void OnTextFrame(string text)
    {
        string response = "ServerABC: " + text;
        base.Send(response);
    }
}
```

The factory used to create the connection is as follows:

```
internal class ServiceFactory : IServiceFactory
{
    public ServiceFactory(string webRoot, IWebSocketLogger logger)
    {
        _logger = logger;
        _webRoot = webRoot;
    }

    public IService CreateInstance(ConnectionDetails connectionDetails)
    {
        switch (connectionDetails.ConnectionType)
        {
            case ConnectionType.WebSocket:
                // you can support different kinds of web socket connections using a different path
                if (connectionDetails.Path == "/chat")
                {
                    return new ChatWebSocketService(connectionDetails.NetworkStream,
                       connectionDetails.TcpClient, connectionDetails.Header, _logger);
                }
                break;
            case ConnectionType.Http:
                // this path actually refers to the relative location of some html file or image
                return new HttpService(connectionDetails.NetworkStream,
                                       connectionDetails.Path, _webRoot, _logger);
        }

        return new BadRequestService
                (connectionDetails.NetworkStream, connectionDetails.Header, _logger);
    }
}
```

HTML5 JavaScript used to connect:

```
// open the connection to the Web Socket server
var CONNECTION = new WebSocket('ws://localhost/chat');

// Log messages from the server
CONNECTION.onmessage = function (e) {
    console.log(e.data);
};

CONNECTION.send('Hellow World');
```

However, you can also write your own test client in C#. There is an example of one in the command line app. Starting the server and the test client from the command line app:

```
private static void Main(string[] args)
{
    IWebSocketLogger logger = new WebSocketLogger();

    try
    {
        string webRoot = Settings.Default.WebRoot;
        int port = Settings.Default.Port;

        // used to decide what to do with incoming connections
        ServiceFactory serviceFactory = new ServiceFactory(webRoot, logger);

        using (WebServer server = new WebServer(serviceFactory, logger))
        {
            server.Listen(port);
            Thread clientThread = new Thread(new ParameterizedThreadStart(TestClient));
            clientThread.IsBackground = false;
            clientThread.Start(logger);
            Console.ReadKey();
        }
    }
    catch (Exception ex)
    {
        logger.Error(null, ex);
        Console.ReadKey();
    }
}
```

The test client runs a short self test to make sure that everything is fine. Opening and closing handshakes are tested here.

Web Socket Protocol

The first thing to realize about the protocol is that it is, in essence, a basic duplex TCP/IP socket connection. The connection starts off with the client connecting to a remote server and sending HTTP header text to that server. The header text asks the web server to upgrade the connection to a web socket connection. This is done as a handshake where the web server responds with an appropriate HTTP text header and from then onwards, the client and server will talk the Web Socket language.

Server Handshake

```
Regex webSocketKeyRegex = new Regex("Sec-WebSocket-Key: (.*)");
Regex webSocketVersionRegex = new Regex("Sec-WebSocket-Version: (.*)");

// check the version. Support version 13 and above
const int WebSocketVersion = 13;
int secWebSocketVersion =
Convert.ToInt32(webSocketVersionRegex.Match(header).Groups[1].Value.Trim());
if (secWebSocketVersion < WebSocketVersion)
{
    throw new WebSocketVersionNotSupportedException(string.Format("WebSocket Version {0} not
supported.
                Must be {1} or above", secWebSocketVersion, WebSocketVersion));
}

string secWebSocketKey = webSocketKeyRegex.Match(header).Groups[1].Value.Trim();
string setWebSocketAccept = base.ComputeSocketAcceptString(secWebSocketKey);
string response = ("HTTP/1.1 101 Switching Protocols\r\n"
                   + "Connection: Upgrade\r\n"
                   + "Upgrade: websocket\r\n"
                   + "Sec-WebSocket-Accept: " + setWebSocketAccept);

HttpHelper.WriteHttpHeader(response, networkStream);
```

NOTE: Don't use Environment.Newline, use \r\n because the HTTP spec is looking for carriage return line feed (two specific ascii characters) and not whatever your environment deems to be equivalent.

This computes the `accept string`:

```
/// <summary>
/// Combines the key supplied by the client with a guid and returns the sha1 hash of the
combination
/// </summary>
public static string ComputeSocketAcceptString(string secWebSocketKey)
{
    // this is a guid as per the web socket spec
    const string webSocketGuid = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";

    string concatenated = secWebSocketKey + webSocketGuid;
    byte[] concatenatedAsBytes = Encoding.UTF8.GetBytes(concatenated);
    byte[] sha1Hash = SHA1.Create().ComputeHash(concatenatedAsBytes);
    string secWebSocketAccept = Convert.ToBase64String(sha1Hash);
    return secWebSocketAccept;
}
```

Client Handshake

```
Uri uri = _uri;
WebSocketFrameReader reader = new WebSocketFrameReader();
Random rand = new Random();
```

```
byte[] keyAsBytes = new byte[16];
rand.NextBytes(keyAsBytes);
string secWebSocketKey = Convert.ToBase64String(keyAsBytes);

string handshakeHttpRequestTemplate = "GET {0} HTTP/1.1\r\n" +
                                        "Host: {1}:{2}\r\n" +
                                        "Upgrade: websocket\r\n" +
                                        "Connection: Upgrade\r\n" +
                                        "Origin: http://{1}:{2}\r\n" +
                                        "Sec-WebSocket-Key: {3}\r\n" +
                                        "Sec-WebSocket-Version: 13\r\n\r\n";

string handshakeHttpRequest = string.Format(handshakeHttpRequestTemplate, uri.PathAndQuery,
uri.Host, uri.Port, secWebSocketKey);
byte[] httpRequest = Encoding.UTF8.GetBytes(handshakeHttpRequest);
networkStream.Write(httpRequest, 0, httpRequest.Length);
```

Reading and Writing

After the handshake as been performed, the server goes into a `read` loop. The following two classes convert a stream of bytes to a web socket frame and visa versa: `WebSocketFrameReader` and `WebSocketFrameWriter`.

```
// from WebSocketFrameReader class
public WebSocketFrame Read(Stream stream, Socket socket)
{
    byte byte1;

    try
    {
        byte1 = (byte) stream.ReadByte();
    }
    catch (IOException)
    {
        if (socket.Connected)
        {
            throw;
        }
        else
        {
            return null;
        }
    }

    // process first byte
    byte finBitFlag = 0x80;
    byte opCodeFlag = 0x0F;
    bool isFinBitSet = (byte1 & finBitFlag) == finBitFlag;
    WebSocketOpCode opCode = (WebSocketOpCode) (byte1 & opCodeFlag);

    // read and process second byte
    byte byte2 = (byte) stream.ReadByte();
    byte maskFlag = 0x80;
    bool isMaskBitSet = (byte2 & maskFlag) == maskFlag;
    uint len = ReadLength(byte2, stream);
    byte[] payload;

    // use the masking key to decode the data if needed
    if (isMaskBitSet)
    {
        byte[] maskKey = BinaryReaderWriter.ReadExactly(WebSocketFrameCommon.MaskKeyLength,
stream);
        payload = BinaryReaderWriter.ReadExactly((int) len, stream);
        // apply the mask key to the payload (which will be mutated)
        WebSocketFrameCommon.ToggleMask(maskKey, payload);
    }
    else
    {
        payload = BinaryReaderWriter.ReadExactly((int) len, stream);
    }

    WebSocketFrame frame = new WebSocketFrame(isFinBitSet, opCode, payload, true);
    return frame;
}
```

```
// from WebSocketFrameWriter class
public void Write(WebSocketOpCode opCode, byte[] payload, bool isLastFrame)
{
    // best to write everything to a memory stream before we push it onto the wire
    // not really necessary but I like it this way
    using (MemoryStream memoryStream = new MemoryStream())
    {
        byte finBitSetAsByte = isLastFrame ? (byte) 0x80 : (byte) 0x00;
        byte byte1 = (byte) (finBitSetAsByte | (byte) opCode);
        memoryStream.WriteByte(byte1);

        // NB, set the mask flag if we are constructing a client frame
        byte maskBitSetAsByte = _isClient ? (byte)0x80 : (byte)0x00;

        // depending on the size of the length we want to write it as a byte, ushort or ulong
        if (payload.Length < 126)
        {
            byte byte2 = (byte)(maskBitSetAsByte | (byte) payload.Length);
            memoryStream.WriteByte(byte2);
        }
        else if (payload.Length <= ushort.MaxValue)
        {
            byte byte2 = (byte)(maskBitSetAsByte | 126);
            memoryStream.WriteByte(byte2);
            BinaryReaderWriter.WriteUShort((ushort) payload.Length, memoryStream, false);
        }
        else
        {
            byte byte2 = (byte)(maskBitSetAsByte | 127);
            memoryStream.WriteByte(byte2);
            BinaryReaderWriter.WriteULong((ulong) payload.Length, memoryStream, false);
        }

        // if we are creating a client frame then we MUST mack the payload as per the spec
        if (_isClient)
        {
            byte[] maskKey = new byte[WebSocketFrameCommon.MaskKeyLength];
            _random.NextBytes(maskKey);
            memoryStream.Write(maskKey, 0, maskKey.Length);

            // mask the payload
            WebSocketFrameCommon.ToggleMask(maskKey, payload);
        }

        memoryStream.Write(payload, 0, payload.Length);
        byte[] buffer = memoryStream.ToArray();
        _stream.Write(buffer, 0, buffer.Length);
    }
}
```

NOTE: C**lient frames MUST contain masked payload data**. This is done to prevent primitive proxy servers from caching the data, thinking that it is static HTML. Of course, using SSL gets around the proxy issue but the authors of the protocol chose to enforce it regardless.

Points of Interest

Problems with Proxy Servers:
Proxy servers which have not been configured to support Web sockets will not work well with them.
I suggest that you use transport layer security (using an SSL certificate) if you want this to work across the wider internet especially from within a corporation.

Using SSL - Secure Web Sockets

To enable ssl in the demo, you need to do these things:

1. Get a valid signed certificate (usually a *.pfx* file)
2. Fill in the `CertificateFile` and `CertificatePassword` settings in the application (or better still, modify the `GetCertificate` function to get your certificate more securely)
3. Change the port to 443
4. (for JavaScript client) Change the *client.html* file to use "`wss`"instead of "`ws`" in the web socket URL.
5. (for command line client) Change the client URL to "`wss`" instead of "`ws`".

I suggest that you get the demo chat to work before you attempt to use a JavaScript client since there are many things that can go wrong and the demo exposes more logging information. If you are getting certificate errors (like name mismatch or expiries), then you can always disable the check by making the `WebSocketClient.ValidateServerCertificate` function always return `true`.

If you are having trouble creating a certificate, I strongly recommend using `LetsEncrypt` to get yourself a free certificate that is signed by a proper root authority. You can use this certificate on your localhost (but your browser will give you certificate warnings).

History

- Version 1.01 - WebSocket
- Version 1.02 - Fixed endianness bug with length field
- Version 1.03 - Major refactor and added support for C# Client
- Version 1.04 - SSL support added. You can now secure your websocket connections with an SSL certificate
- Version 1.05 - Bug fixes

What has changed since December 2015

Thanks to all the feedback from the comments below, and after using the protocol in various projects, I have a much better understanding of how it works and have made a more robust version available from nuget - Ninja.WebSockets. Since it targets .NetStandard 2.0 it even works from a cell phone if you use Xamarin! Or linux or OSx. I will continue to monitor this page and please keep the comments coming. However, my main focus will be on the Ninja.WebSockets version which implements a standard WebSocket abstract class (System.Net.WebSockets.WebSocket) that Microsoft has put a lot of thought into. Ninja.WebSockets does a better job at handling errors and closing handshakes. It is also specifically designed to be as lightweight as possible (by being async and using buffer pools) and it will be interesting to see how far it can be pushed on a decent server.

License

About the Author

Dave Haig

Software Developer (Senior)

United Kingdom

Works as a senior software developer in investment banking

Comments and Discussions

83 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1063910/WebSocket-Server-in-Csharp** to post and view comments on this article, or click **here** to get a print view with messages.