mdn web docs

Writing a WebSocket server in C#

If you would like to use the WebSocket API, it is useful if you have a server. In this article I will show you how to write one in C#. You can do it in any server-side language, but to keep things simple and more understandable, I chose Microsoft's language.

This server conforms to RFC 6455 , so it will only handle connections from Chrome version 16, Firefox 11, IE 10 and over.

First steps

WebSockets communicate over a TCP (Transmission Control Protocol) connection. Luckily, C# has a TcpListener class which does as the name suggests. It is in the `System.Net.Sockets` namespace.

> **Note:** It is a good idea to include the namespace with the `using` keyword in order to write less. It allows usage of a namespace's classes without typing the full namespace every time.

TcpListener

Constructor:

CS

```
TcpListener(System.Net.IPAddress localaddr, int port)
```

`localaddr` specifies the IP of the listener, and `port` specifies the port.

> **Note:** To create an `IPAddress` object from a `string`, use the `Parse` static method of `IPAddress`.

Methods:

- `Start()`
- `System.Net.Sockets.TcpClient AcceptTcpClient()` Waits for a Tcp connection, accepts it and returns it as a TcpClient object.

Here's a barebones server implementation:

CS

```
using System.Net.Sockets;
using System.Net;
using System;

class Server {
    public static void Main() {
        TcpListener server = new TcpListener(IPAddress.Parse("127.0.0.1"), 80);

        server.Start();
```

```
        Console.WriteLine("Server has started on 127.0.0.1:80.{0}Waiting for a connection…", Environment.NewLine);

        TcpClient client = server.AcceptTcpClient();

        Console.WriteLine("A client connected.");
    }
}
```

TcpClient

Methods:

- `System.Net.Sockets.NetworkStream GetStream()` Gets the stream which is the communication channel. Both sides of the channel have reading and writing capability.

Properties:

- `int Available` This Property indicates how many bytes of data have been sent. The value is zero until `NetworkStream.DataAvailable` is *true*.

NetworkStream

Methods:

- Writes bytes from buffer, offset and size determine length of message.

  CS

```
Write(byte[] buffer, int offset, int size)
```

- Reads bytes to `buffer`. `offset` and `size` determine the length of the message.

  CS

  ```
  Read(byte[] buffer, int offset, int size)
  ```

Let us extend our example.

CS

```
TcpClient client = server.AcceptTcpClient();

Console.WriteLine("A client connected.");

NetworkStream stream = client.GetStream();

//enter to an infinite cycle to be able to handle every change in stream
while (true) {
    while (!stream.DataAvailable);

    byte[] bytes = new byte[client.Available];

    stream.Read(bytes, 0, bytes.Length);
}
```

Handshaking

When a client connects to a server, it sends a GET request to upgrade the connection to a WebSocket from a simple

HTTP request. This is known as handshaking.

This sample code can detect a GET from the client. Note that this will block until the first 3 bytes of a message are available. Alternative solutions should be investigated for production environments.

CS

```cs
using System.Text;
using System.Text.RegularExpressions;

while(client.Available < 3)
{
   // wait for enough bytes to be available
}

byte[] bytes = new byte[client.Available];

stream.Read(bytes, 0, bytes.Length);

//translate bytes of request to string
String data = Encoding.UTF8.GetString(bytes);

if (Regex.IsMatch(data, "^GET")) {

} else {

}
```

The response is easy to build, but might be a little difficult to understand. The full explanation of the Server handshake

can be found in RFC 6455, section 4.2.2. For our purposes, we'll just build a simple response.

You must:

1. Obtain the value of the "Sec-WebSocket-Key" request header without any leading or trailing whitespace
2. Concatenate it with "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" (a special GUID specified by RFC 6455)
3. Compute SHA-1 and Base64 hash of the new value
4. Write the hash back as the value of "Sec-WebSocket-Accept" response header in an HTTP response

CS

```
if (new System.Text.RegularExpressions.Regex("^GET").IsMatch(data))
{
    const string eol = "\r\n"; // HTTP/1.1 defines the sequence CR LF as the end-of-line marker

    byte[] response = Encoding.UTF8.GetBytes("HTTP/1.1 101 Switching Protocols" + eol
        + "Connection: Upgrade" + eol
        + "Upgrade: websocket" + eol
        + "Sec-WebSocket-Accept: " + Convert.ToBase64String(
            System.Security.Cryptography.SHA1.Create().ComputeHash(
                Encoding.UTF8.GetBytes(
                    new System.Text.RegularExpressions.Regex("Sec-WebSocket-Key: (.*)").Match(data).Groups[1].Value.Trim() +
"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
                )
            )
        ) + eol
        + eol);
```

```
    stream.Write(response, 0, response.Length);
}
```

Decoding messages

After a successful handshake, the client will send encoded messages to the server.

If we send "MDN", we get these bytes:

```
129 131 61 84 35 6 112 16 109
```

Let's take a look at what these bytes mean.

The first byte, which currently has a value of 129, is a bitfield that breaks down as such:

FIN (Bit 0)	RSV1 (Bit 1)	RSV2 (Bit 2)	RSV3 (Bit 3)	Opcode (Bit 4:7)
1	0	0	0	0×1=0001

- FIN bit: This bit indicates whether the full message has been sent from the client. Messages may be sent in frames, but for now we will keep things simple.
- RSV1, RSV2, RSV3: These bits must be 0 unless an extension is negotiated which supplies a nonzero value to them.
- Opcode: These bits describe the type of message received. Opcode 0×1 means this is a text message. Full list of Opcodes

The second byte, which currently has a value of 131, is another bitfield that breaks down as such:

MASK (Bit 0)	Payload Length (Bit 1:7)
1	0×83=0000011

- MASK bit: Defines whether the "Payload data" is masked. If set to 1, a masking key is present in Masking-Key, and this is used to unmask the "Payload data". All messages from the client to the server have this bit set.
- Payload Length: If this value is between 0 and 125, then it is the length of message. If it is 126, the following 2 bytes (16-bit unsigned integer) are the length. If it is 127, the following 8 bytes (64-bit unsigned integer) are the length.

> **Note:** Because the first bit is always 1 for client-to-server messages, you can subtract 128 from this byte to get rid of the MASK bit.

Note that the MASK bit is set in our message. This means that the next four bytes (61, 84, 35, and 6) are the mask bytes used to decode the message. These bytes change with every message.

The remaining bytes are the encoded message payload.

Decoding algorithm

D_i = *E_i* XOR *M_*(*i* mod 4)

where *D* is the decoded message array, *E* is the encoded message array, *M* is the mask byte array, and *i* is the index of

the message byte to decode.

Example in C#:

CS

```cs
byte[] decoded = new byte[3];
byte[] encoded = new byte[3] {112, 16, 109};
byte[] mask = new byte[4] {61, 84, 35, 6};

for (int i = 0; i < encoded.Length; i++) {
    decoded[i] = (byte)(encoded[i] ^ mask[i % 4]);
}
```

Put together

wsserver.cs

CS

```cs
//
// csc wsserver.cs
// wsserver.exe

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Text.RegularExpressions;
```

```
class Server {
    public static void Main() {
        string ip = "127.0.0.1";
        int port = 80;
        var server = new TcpListener(IPAddress.Parse(ip), port);

        server.Start();
        Console.WriteLine("Server has started on {0}:{1}, Waiting for a connection…", ip, port);

        TcpClient client = server.AcceptTcpClient();
        Console.WriteLine("A client connected.");

        NetworkStream stream = client.GetStream();

        // enter to an infinite cycle to be able to handle every change in stream
        while (true) {
            while (!stream.DataAvailable);
            while (client.Available < 3); // match against "get"

            byte[] bytes = new byte[client.Available];
            stream.Read(bytes, 0, bytes.Length);
            string s = Encoding.UTF8.GetString(bytes);

            if (Regex.IsMatch(s, "^GET", RegexOptions.IgnoreCase)) {
                Console.WriteLine("=====Handshaking from client=====\n{0}", s);

                // 1. Obtain the value of the "Sec-WebSocket-Key" request header without any leading or trailing whitespace
                // 2. Concatenate it with "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" (a special GUID specified by RFC 6455)
                // 3. Compute SHA-1 and Base64 hash of the new value
                // 4. Write the hash back as the value of "Sec-WebSocket-Accept" response header in an HTTP response
```

```
                string swk = Regex.Match(s, "Sec-WebSocket-Key: (.*)").Groups[1].Value.Trim();
                string swka = swk + "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
                byte[] swkaSha1 = System.Security.Cryptography.SHA1.Create().ComputeHash(Encoding.UTF8.GetBytes(swka));
                string swkaSha1Base64 = Convert.ToBase64String(swkaSha1);

                // HTTP/1.1 defines the sequence CR LF as the end-of-line marker
                byte[] response = Encoding.UTF8.GetBytes(
                    "HTTP/1.1 101 Switching Protocols\r\n" +
                    "Connection: Upgrade\r\n" +
                    "Upgrade: websocket\r\n" +
                    "Sec-WebSocket-Accept: " + swkaSha1Base64 + "\r\n\r\n");

                stream.Write(response, 0, response.Length);
            } else {
                bool fin = (bytes[0] & 0b10000000) != 0,
                    mask = (bytes[1] & 0b10000000) != 0; // must be true, "All messages from the client to the server have
this bit set"
                int opcode = bytes[0] & 0b00001111, // expecting 1 - text message
                    offset = 2;
                ulong msglen = bytes[1] & 0b01111111;

                if (msglen == 126) {
                    // bytes are reversed because websocket will print them in Big-Endian, whereas
                    // BitConverter will want them arranged in little-endian on windows
                    msglen = BitConverter.ToUInt16(new byte[] { bytes[3], bytes[2] }, 0);
                    offset = 4;
                } else if (msglen == 127) {
                    // To test the below code, we need to manually buffer larger messages — since the NIC's autobuffering
                    // may be too latency-friendly for this code to run (that is, we may have only some of the bytes in this
                    // websocket frame available through client.Available).
```

```
                    msglen = BitConverter.ToUInt64(new byte[] { bytes[9], bytes[8], bytes[7], bytes[6], bytes[5], bytes[4],
bytes[3], bytes[2] },0);
                    offset = 10;
                }

                if (msglen == 0) {
                    Console.WriteLine("msglen == 0");
                } else if (mask) {
                    byte[] decoded = new byte[msglen];
                    byte[] masks = new byte[4] { bytes[offset], bytes[offset + 1], bytes[offset + 2], bytes[offset + 3] };
                    offset += 4;

                    for (ulong i = 0; i < msglen; ++i)
                        decoded[i] = (byte)(bytes[offset + i] ^ masks[i % 4]);

                    string text = Encoding.UTF8.GetString(decoded);
                    Console.WriteLine("{0}", text);
                } else
                    Console.WriteLine("mask bit not set");

                Console.WriteLine();
            }
        }
    }
}
```

client.html

HTML

```
<!doctype html>
```

```
<html lang="en">
  <style>
    textarea {
      vertical-align: bottom;
    }
    #output {
      overflow: auto;
    }
    #output > p {
      overflow-wrap: break-word;
    }
    #output span {
      color: blue;
    }
    #output span.error {
      color: red;
    }
  </style>
  <body>
    <h2>WebSocket Test</h2>
    <textarea cols="60" rows="6"></textarea>
    <button>send</button>
    <div id="output"></div>
  </body>
  <script>
    // http://www.websocket.org/echo.html
    const button = document.querySelector("button");
    const output = document.querySelector("#output");
    const textarea = document.querySelector("textarea");
    const wsUri = "ws://127.0.0.1/";
```

```
const websocket = new WebSocket(wsUri);

button.addEventListener("click", onClickButton);

websocket.onopen = (e) => {
  writeToScreen("CONNECTED");
  doSend("WebSocket rocks");
};

websocket.onclose = (e) => {
  writeToScreen("DISCONNECTED");
};

websocket.onmessage = (e) => {
  writeToScreen(`<span>RESPONSE: ${e.data}</span>`);
};

websocket.onerror = (e) => {
  writeToScreen(`<span class="error">ERROR:</span> ${e.data}`);
};

function doSend(message) {
  writeToScreen(`SENT: ${message}`);
  websocket.send(message);
}

function writeToScreen(message) {
  output.insertAdjacentHTML("afterbegin", `<p>${message}</p>`);
}
```

```
    function onClickButton() {
      const text = textarea.value;

      text && doSend(text);
      textarea.value = "";
      textarea.focus();
    }
  </script>
</html>
```

Related

- Writing WebSocket servers

This page was last modified on Jul 25, 2023 by MDN contributors.