mdn web docs_

Writing WebSocket servers

A WebSocket server is nothing more than an application listening on any port of a TCP server that follows a specific protocol. The task of creating a custom server tends to scare people; however, it can be straightforward to implement a simple WebSocket server on your platform of choice.

A WebSocket server can be written in any server-side programming language that is capable of Berkeley sockets , such as C(++), Python, PHP, or server-side JavaScript. This is not a tutorial in any specific language, but serves as a guide to facilitate writing your own server.

This article assumes you're already familiar with how HTTP works, and that you have a moderate level of programming experience. Depending on language support, knowledge of TCP sockets may be required. The scope of this guide is to present the minimum knowledge you need to write a WebSocket server.

> **Note:** Read the latest official WebSockets specification, RFC 6455 . Sections 1 and 4-7 are especially interesting to server implementors. Section 10 discusses security and you should definitely peruse it before exposing your server.

A WebSocket server is explained on a very low level here. WebSocket servers are often separate and specialized servers (for load-balancing or other practical reasons), so you will often use a reverse proxy (such as a regular HTTP server) to detect WebSocket handshakes, pre-process them, and send those clients to a real WebSocket server. This means that you don't have to bloat your server code with cookie and authentication handlers (for example).

The WebSocket handshake

First, the server must listen for incoming socket connections using a standard TCP socket. Depending on your platform, this may be handled for you automatically. For example, let's assume that your server is listening on `example.com`, port 8000, and your socket server responds to `GET` requests at `example.com/chat`.

> **Warning:** The server may listen on any port it chooses, but if it chooses any port other than 80 or 443, it may have problems with firewalls and/or proxies. Browsers generally require a secure connection for WebSockets, although they may offer an exception for local devices.

The handshake is the "Web" in WebSockets. It's the bridge from HTTP to WebSockets. In the handshake, details of the connection are negotiated, and either party can back out before completion if the terms are unfavorable. The server must be careful to understand everything the client asks for, otherwise security issues can occur.

> **Note:** The request-uri (`/chat` here) has no defined meaning in the spec. So, many people use it to let one server handle multiple WebSocket applications. For example, `example.com/chat` could invoke a multiuser chat app, while `/game` on the same server might invoke a multiplayer game.

Client handshake request

Even though you're building a server, a client still has to start the WebSocket handshake process by contacting the server and requesting a WebSocket connection. So, you must know how to interpret the client's request. The **client** will send a pretty standard HTTP request with headers that looks like this (the HTTP version **must** be 1.1 or greater, and the method **must** be `GET`):

BASH

```
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

The client can solicit extensions and/or subprotocols here; see Miscellaneous for details. Also, common headers like `User-Agent`, `Referer`, `Cookie`, or authentication headers might be there as well. Do whatever you want with those; they don't directly pertain to the WebSocket. It's also safe to ignore them. In many common setups, a reverse proxy has already dealt with them.

> **Note:** All **browsers** send an `Origin` header. You can use this header for security (checking for same origin, automatically allowing or denying, etc.) and send a 403 Forbidden if you don't like what you see. However, be warned that non-browser agents can send a faked `Origin`. Most applications reject requests without this header.

If any header is not understood or has an incorrect value, the server should send a `400` ("Bad Request") response and immediately close the socket. As usual, it may also give the reason why the handshake failed in the HTTP response body, but the message may never be displayed (browsers do not display it). If the server doesn't understand that version of WebSockets, it should send a `Sec-WebSocket-Version` header back that contains the version(s) it does understand. In the example above, it indicates version 13 of the WebSocket protocol.

The most interesting header here is `Sec-WebSocket-Key`. Let's look at that next.

> **Note:** Regular HTTP status codes can be used only before the handshake. After the handshake succeeds, you have to use a different set of codes (defined in section 7.4 of the spec).

Server handshake response

When the **server** receives the handshake request, it should send back a special response that indicates that the protocol will be changing from HTTP to WebSocket. That header looks something like the following (remember each header line ends with `\r\n` and put an extra `\r\n` after the last one to indicate the end of the header):

BASH

```bash
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Additionally, the server can decide on extension/subprotocol requests here; see Miscellaneous for details. The `Sec-WebSocket-Accept` header is important in that the server must derive it from the `Sec-WebSocket-Key` that the client sent to it. To get it, concatenate the client's `Sec-WebSocket-Key` and the string "`258EAFA5-E914-47DA-95CA-C5AB0DC85B11`" together (it's a "magic string "), take the SHA-1 hash of the result, and return the base64 encoding of that hash.

> **Note:** This seemingly overcomplicated process exists so that it's obvious to the client whether the server supports WebSockets. This is important because security issues might arise if the server accepts a WebSockets connection but interprets the data as a HTTP request.

So if the Key was " `dGhlIHNhbXBsZSBub25jZQ==` ", the `Sec-WebSocket-Accept` header's value is " `s3pPLMBiTxaQ9kYGzzhZRbK+xOo=` ". Once the server sends these headers, the handshake is complete and you can start swapping data!

> **Note:** The server can send other headers like `Set-Cookie` , or ask for authentication or redirects via other status codes, before sending the reply handshake.

Keeping track of clients

This doesn't directly relate to the WebSocket protocol, but it's worth mentioning here: your server must keep track of clients' sockets so you don't keep handshaking again with clients who have already completed the handshake. The same client IP address can try to connect multiple times. However, the server can deny them if they attempt too many connections in order to save itself from Denial-of-Service attacks .

For example, you might keep a table of usernames or ID numbers along with the corresponding `WebSocket` and other data that you need to associate with that connection.

Exchanging data frames

Either the client or the server can choose to send a message at any time — that's the magic of WebSockets. However, extracting information from these so-called "frames" of data is a not-so-magical experience. Although all frames follow the same specific format, data going from the client to the server is masked using XOR encryption (with a 32-bit key). Section 5 of the specification describes this in detail.

Format

Each data frame (from the client to the server or vice versa) follows this same format:

BASH

```
Frame format:

   0                   1                   2                   3
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
  +-+-+-+-+-------+-+-------------+-------------------------------+
  |F|R|R|R| opcode|M| Payload len |    Extended payload length    |
  |I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
  |N|V|V|V|       |S|             |   (if payload len==126/127)   |
  | |1|2|3|       |K|             |                               |
  +-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
  |     Extended payload length continued, if payload len == 127  |
  + - - - - - - - - - - - - - - - +-------------------------------+
  |                               |Masking-key, if MASK set to 1  |
  +-------------------------------+-------------------------------+
  | Masking-key (continued)       |          Payload Data         |
  +-------------------------------- - - - - - - - - - - - - - - - +
  :                     Payload Data continued ...                :
  + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
  |                     Payload Data continued ...                |
  +---------------------------------------------------------------+
```

This means that a frame contains the following bytes:

- First byte:
  - bit 0: FIN
  - bit 1: RSV1
  - bit 2: RSV2
  - bit 3: RSV3
  - bits 4-7 OPCODE
- Bytes 2-10: payload length (see Decoding Payload Length)
- If masking is used, the next 4 bytes contain the masking key (see Reading and unmasking the data)
- All subsequent bytes are payload

The MASK bit tells whether the message is encoded. Messages from the client must be masked, so your server must expect this to be 1. (In fact, section 5.1 of the spec says that your server must disconnect from a client if that client sends an unmasked message.) When sending a frame back to the client, do not mask it and do not set the mask bit. We'll explain masking later. *Note: You must mask messages even when using a secure socket*. RSV1-3 can be ignored, they are for extensions.

The opcode field defines how to interpret the payload data: `0x0` for continuation, `0x1` for text (which is always encoded in UTF-8), `0x2` for binary, and other so-called "control codes" that will be discussed later. In this version of WebSockets, `0x3` to `0x7` and `0xB` to `0xF` have no meaning.

The FIN bit tells whether this is the last message in a series. If it's 0, then the server keeps listening for more parts of the message; otherwise, the server should consider the message delivered. More on this later.

Decoding Payload Length

To read the payload data, you must know when to stop reading. That's why the payload length is important to know. Unfortunately, this is somewhat complicated. To read it, follow these steps:

1. Read bits 9-15 (inclusive) and interpret that as an unsigned integer. If it's 125 or less, then that's the length; you're **done**. If it's 126, go to step 2. If it's 127, go to step 3.
2. Read the next 16 bits and interpret those as an unsigned integer. You're **done**.
3. Read the next 64 bits and interpret those as an unsigned integer. (The most significant bit *must* be 0.) You're **done**.

Reading and unmasking the data

If the MASK bit was set (and it should be, for client-to-server messages), read the next 4 octets (32 bits); this is the masking key. Once the payload length and masking key is decoded, you can read that number of bytes from the socket. Let's call the data `ENCODED`, and the key `MASK`. To get `DECODED`, loop through the octets (bytes a.k.a. characters for text data) of `ENCODED` and XOR the octet with the (i modulo 4)th octet of `MASK`. In pseudocode (that happens to be valid JavaScript):

JS

```
const MASK = [1, 2, 3, 4]; // 4-byte mask
const ENCODED = [105, 103, 111, 104, 110]; // encoded string "hello"

// Create the byte Array of decoded payload
const DECODED = Uint8Array.from(ENCODED, (elt, i) => elt ^ MASK[i % 4]); // Perform an XOR on the mask
```

Now you can figure out what **DECODED** means depending on your application.

Message Fragmentation

The FIN and opcode fields work together to send a message split up into separate frames. This is called message fragmentation. Fragmentation is only available on opcodes `0x0` to `0x2` .

Recall that the opcode tells what a frame is meant to do. If it's `0x1` , the payload is text. If it's `0x2` , the payload is binary data. However, if it's `0x0,` the frame is a continuation frame; this means the server should concatenate the frame's payload to the last frame it received from that client. Here is a rough sketch, in which a server reacts to a client sending text messages. The first message is sent in a single frame, while the second message is sent across three frames. FIN and opcode details are shown only for the client:

```
Client: FIN=1, opcode=0x1, msg="hello"
Server: (process complete message immediately) Hi.
Client: FIN=0, opcode=0x1, msg="and a"
Server: (listening, new message containing text started)
Client: FIN=0, opcode=0x0, msg="happy new"
Server: (listening, payload concatenated to previous message)
Client: FIN=1, opcode=0x0, msg="year!"
Server: (process complete message) Happy new year to you too!
```

Notice the first frame contains an entire message (has `FIN=1` and `opcode!=0x0`), so the server can process or respond as it sees fit. The second frame sent by the client has a text payload (`opcode=0x1`), but the entire message has not arrived yet (`FIN=0`). All remaining parts of that message are sent with continuation frames (`opcode=0x0`), and the final frame of the message is marked by `FIN=1`. Section 5.4 of the spec describes message fragmentation.

Pings and Pongs: The Heartbeat of WebSockets

At any point after the handshake, either the client or the server can choose to send a ping to the other party. When the ping is received, the recipient must send back a pong as soon as possible. You can use this to make sure that the client is still connected, for example.

A ping or pong is just a regular frame, but it's a **control frame**. Pings have an opcode of `0x9`, and pongs have an opcode of `0xA`. When you get a ping, send back a pong with the exact same Payload Data as the ping (for pings and pongs, the max payload length is 125). You might also get a pong without ever sending a ping; ignore this if it happens.

> **Note:** If you have gotten more than one ping before you get the chance to send a pong, you only send one pong.

Closing the connection

To close a connection either the client or server can send a control frame with data containing a specified control sequence to begin the closing handshake (detailed in Section 5.5.1). Upon receiving such a frame, the other peer sends a Close frame in response. The first peer then closes the connection. Any further data received after closing of connection is then discarded.

Miscellaneous

> **Note:** WebSocket codes, extensions, subprotocols, etc. are registered at the IANA WebSocket Protocol Registry .

WebSocket extensions and subprotocols are negotiated via headers during the handshake. Sometimes extensions and subprotocols are very similar, but there is a clear distinction. Extensions control the WebSocket *frame* and *modify* the payload, while subprotocols structure the WebSocket *payload* and *never modify* anything. Extensions are optional and generalized (like compression); subprotocols are mandatory and localized (like ones for chat and for MMORPG games).

Extensions

Think of an extension as compressing a file before emailing it to someone. Whatever you do, you're sending the *same* data in different forms. The recipient will eventually be able to get the same data as your local copy, but it is sent differently. That's what an extension does. WebSockets defines a protocol and a simple way to send data, but an extension such as compression could allow sending the same data but in a shorter format.

> **Note:** Extensions are explained in sections 5.8, 9, 11.3.2, and 11.4 of the spec.

Subprotocols

Think of a subprotocol as a custom XML schema or doctype declaration . You're still using XML and its syntax, but you're additionally restricted by a structure you agreed on. WebSocket subprotocols are just like that. They do not introduce anything fancy, they just establish structure. Like a doctype or schema, both parties must agree on the subprotocol; unlike a doctype or schema, the subprotocol is implemented on the server and cannot be externally referred to by the client.

> **Note:** Subprotocols are explained in sections 1.9, 4.2, 11.3.4, and 11.5 of the spec.

A client has to ask for a specific subprotocol. To do so, it will send something like this *as part of the original handshake*:

BASH

```bash
GET /chat HTTP/1.1
...
Sec-WebSocket-Protocol: soap, wamp
```

or, equivalently:

BASH

```bash
...
Sec-WebSocket-Protocol: soap
Sec-WebSocket-Protocol: wamp
```

Now the server must pick one of the protocols that the client suggested and it supports. If there is more than one, send the first one the client sent. Imagine our server can use both `soap` and `wamp`. Then, in the response handshake, it sends:

BASH

```
Sec-WebSocket-Protocol: soap
```

> **Warning:** The server can't send more than one `Sec-Websocket-Protocol` header. If the server doesn't want to use any subprotocol, ***it shouldn't send any `Sec-WebSocket-Protocol` header***. Sending a blank header is incorrect. The client may close the connection if it doesn't get the subprotocol it wants.

If you want your server to obey certain subprotocols, then naturally you'll need extra code on the server. Let's imagine we're using a subprotocol `json`. In this subprotocol, all data is passed as JSON . If the client solicits this protocol and the server wants to use it, the server needs to have a JSON parser. Practically speaking, this will be part of a library, but the server needs to pass the data around.

> **Note:** To avoid name conflict, it's recommended to make your subprotocol name part of a domain string. If you are building a custom chat app that uses a proprietary format exclusive to Example Inc., then you might use this: `Sec-WebSocket-Protocol: chat.example.com`. Note that this isn't required, it's just an optional convention, and you can use any string you wish.

Related

- Writing WebSocket client applications
- Tutorial: Websocket server in C#
- Tutorial: Websocket server in Java

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