

JAMIE PATEL

WebSockets From Scratch

MAY 21, 2015

I have been at Pusher for almost 6 months and, mainly working on customer-facing developer work, parts of our deeper infrastructure have seemed a bit of a black box to me. Pusher, a message service that lets you send realtime data from server to client or from client to client, has the [WebSocket protocol](#) at its core. I was aware of how the HTTP protocol worked, but not WebSocket – aside from the fact it lets you do some nifty realtime stuff.

Therefore I decided to dig a little deeper and try to build a WebSocket server from scratch – and by ‘scratch’, I mean using only Ruby’s built-in libraries. This blog post is to partly share what I’ve learnt and partly act as a tutorial, given that I couldn’t find many that would lead me through the process step-by-step. That said, there were plenty of awesome

This guide is aimed at people who are new to WebSocket, or just wish to know more about what's under the hood. What I'll cover, in around 100 lines of Ruby, is:

- The HTTP handshake that initiates a WebSocket connection.
- Listening to messages on the server.
- Sending messages from the server.

A lot of very important features will be left out for the sake of brevity, such as ping/pong heartbeats, types of messages that aren't UTF-8 text data, security, proxying, handling different WebSocket protocol versions, and message fragmentation. So let's get to it.

An Overview to the WebSocket Protocol

WebSocket, like HTTP, is a layer upon the [TCP protocol](#). A high-level difference between the two is that a classic HTTP response closes the TCP socket, whereas in the WebSocket protocol, the connection stays open. This allows bi-directional communication between the server and client, and is great for the realtime functionality you are used to: chat applications, data visualization, activity streams and so on.

A WebSocket connection begins with a HTTP GET request from the client to the server, called the 'handshake'. This request carries with it a `Connection: upgrade` and `Upgrade: websocket` header to tell the server that it wants to begin a WebSocket connection, and a `Sec-WebSocket-Version` header that indicates the kind of response it wants. In this guide we'll only focus on version 13 of the protocol.

The request headers also include a `Sec-WebSocket-Key` field.

later.

Once this handshake is made, each party is free to exchange messages, which are wrapped in ‘frames’. Each frame consists of information about:

- Whether this frame is or isn’t part of a continuation. In this guide, we’ll only deal with frames that contain a complete message (not fragmented).
- The content-type. In this post, we’ll only deal with UTF8-encoded text.
- Whether the frame is encoded, or ‘masked’. Frames from the client always have to be masked; frames from the server do not have to be.
- The payload length.
- The masking ‘key’ with which to decode the message – if the frame is masked.
- The payload of the frame.

The Guide

What We’ll Build

During this post we’ll build a simple echo server that takes messages from a client and sends them back with a thank you, simply as a basic implementation of a WebSocket server.

```
server = WebSocketServer.new

loop do
  Thread.new(server.accept) do |connection|
    puts "Connected"
    while (message = connection.recv)
      puts "Received #{message}"
      connection.send("Received #{message}. Thanks!")
    end
  end
end
```

Getting Started

Let's start with two classes: our `WebSocketServer` and our `WebSocketConnection`. Create them in files called `websocket_server.rb` and `websocket_connection.rb` respectively.

The `WebSocketServer`

The `WebSocketServer` will be initialized with options, such as the path of the WebSocket endpoint, the port and the host – these will default to `'/'`, `4567` and `localhost` respectively.

```
require 'socket'

class WebSocketServer

  def initialize(options={path: '/', port: 4567, host: 'localhost',
    @path, port, host = options[:path], options[:port],
    @tcp_server = TCPServer.new(host, port)
  end
  ...
end
```

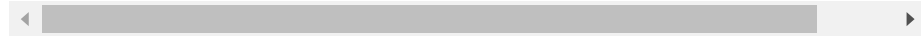
Upon initialization, a `TCPServer` object will be created with our host and port options – though it will not run until we ‘accept’ it. Remember to require the built-in `socket` library that lets you create TCP connections.

On calling `#accept`, our `WebSocketServer` will be responding to any incoming WebSocket requests. It will be responsible for validating incoming HTTP requests, and sending back a handshake. If a handshake can and has been made – that is, if `send_handshake` returns `true` – it will return a new `WebSocketConnection`, as shown in the example below.

...

```
def accept
  socket = @tcp_server.accept
  send_handshake(socket) && WebSocketConnection.new(sc
end
```

end



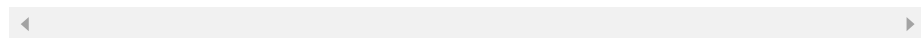
The WebSocketConnection

The WebSocketConnection will be our API for sending and receiving messages. We initialize it with the TCP socket made upon firing up the TCPServer in WebSocketServer#accept.

```
class WebSocketConnection
```

```
  attr_reader :socket
```

```
  def initialize(socket)
    @socket = socket
  end
end
```



The connection object will read and write to this socket as it listens for and sends messages.

The Handshake

Going back to our WebSocketServer class, a WebSocketServer#send_handshake method is where everything begins. Firstly, let's get the request_line (e.g. 'GET / HTTP/1.1') and request header from the socket, using the socket#gets method. This will block if there is nothing yet available, and will also get a line at a time.

```
def send_handshake(socket)
  request_line = socket.gets
  header = get_header(socket)
  ...
end

# this gets the header by recursively reading each line
def get_header(socket, header = "")
  (line = socket.gets) == "\r\n" ? header : get_header(socket, header + line)
end
```

If we have not received a GET request at the specified path, or there is no Sec-WebSocket-Key in the header, let's write a 400 error to the socket. We can use the << operator, and then close the socket to end the request. By returning false, we make sure a WebSocketConnection is not created and returned to the application.

```
def send_handshake(socket)
  request_line = socket.gets
  header = get_header(socket)
  if (request_line =~ /GET #{@path} HTTP\/1.1/) && (header =~ /Sec-WebSocket-Key/)
    ... # complete the handshake
  else
    send_400(socket)
    false # reject the handshake
  end
end

def send_400(socket)
  socket << "HTTP/1.1 400 Bad Request\r\n" +
    "Content-Type: text/plain\r\n" +
    "Connection: close\r\n" +
    "\r\n" +
    "Incorrect request"
  socket.close
end
```

If there is a value to Sec-WebSocket-Key, according to the

by taking the value of the Sec-WebSocket-Key and concatenating it with "258EAF5-E914-47DA-95CA-C5AB0DC85B11", a 'magic string', defined in the [protocol specification](#). It takes this concatenation, creates a SHA1 digest of it, then encodes this digest in Base64. We can do this using the built-in `digest/sha1` and `base64` libraries.

```
def send_handshake(socket)
  request_line = socket.gets
  header = get_header(socket)
  if (request_line =~ /GET #{@path} HTTP\/1.1/) && (header =~ /Sec-WebSocket-Key/)
    ws_accept = create_ws_accept($1)
    ...
  end
  send_101(socket)
  true
end

WS_MAGIC_STRING = "258EAF5-E914-47DA-95CA-C5AB0DC85B11"

require 'digest/sha1'
require 'base64'

def create_ws_accept(key)
  digest = Digest::SHA1.digest(key + WS_MAGIC_STRING)
  Base64.encode64(digest)
end
```



Now we can take this key, and write our expected response to the socket. This response includes the status code "101 Switching Protocols", to indicate that the server and client will now be speaking via a WebSocket. This also includes the same Upgrade and Connection headers sent to us by the client, and also the appropriate Sec-WebSocket-Accept key and value.

```
def send_handshake(socket)
```

```

    if (request_line =~ /^GET #{@path} HTTP\/1.1/) && (neac
      ws_accept = create__accept($1)
      send_handshake_response(socket, ws_accept)
      return true
    end
    send_400(socket)
    false
  end

  def send_handshake_response(socket, ws_accept)
    socket << "HTTP/1.1 101 Switching Protocols\r\n"
      "Upgrade: websocket\r\n"
      "Connection: Upgrade\r\n"
      "Sec-WebSocket-Accept: #{ws_accept}\r\n"
  end

```

Now that we've sent the handshake and returned true, a new `WebSocketConnection` will be returned to our application. So, test it out! Let's create a loop that constantly listens for new requests. Using `Thread.new` and yielding the result of `server.accept` – which should be our new connection – we can handle concurrent requests.

In your Ruby app, write this:

```

server = WebSocketServer.new

loop do
  Thread.new(server.accept) do |connection|
    puts "Connected"
  end
end

```

Run this app and while this code is running, open up your browser console (on a page *not* served via HTTPS) and create a WebSocket connection to your server:

```

var socket = new WebSocket("ws://localhost:4567");

```


You should see that a connection has been made upon handshake, and printed "Connected" to your terminal window. If not, you can check out the source code [here](#).

Listening For Messages

Now that clients can connect to us and the user has access to the `WebSocketConnection` object, we can start listening to messages from the client.

By the end of this section, here is what we want to have:

```
loop do
  Thread.new(server.accept) do |connection|
    puts "Connected"
    while (message = connection.recv)
      puts message
    end
  end
end
```

And if from our browser console, we type –

```
var socket = new WebSocket("ws://localhost:4567");
socket.send("hello");
```

- we should hope to see "hello" in our terminal window. Of course if you try this out now you'll get an error.

So let's create `WebSocketConnection#recv` method. In it we will read from the socket if there are bytes available.

```
class WebSocketConnection
```

```
...
```

```
def recv
```

end

As mentioned in the overview above, WebSocket messages are wrapped in frames, which are a sequence of bytes carrying information about the message. Our `#recv` method will parse the bytes of a frame and yield the message's content to the application thread.

Let's have a look at what we'll receive if, as in the example above, we send "hello" over the socket.

Byte value	129	133	32	2
Binary representation	10000001	10000101	00100000	00000001
Meaning	Fin + opcode	Mask indicator + Length indicator	Key	Mask

The first byte indicates whether this is the complete message. If the first bit is 1 (as it is) then yes, otherwise it is 0. The next 3 bytes are reserved. And the remainder of the byte (0001) indicates that the content type is text.

Using the `TCPSocket#read` method, we can read `n` bytes at a time:

```
def recv
  fin_and_opcode = socket.read(1).bytes[0] # get the 0
  ...
end
```

always will be. It cannot be if it's from a server. If it is masked, the first bit will be 1.

The remainder of the byte indicates the content's length. Firstly, we need to remove the first bit out of the equation by subtracting 128 (or calling `mask_and_length_indicator & 0x7f`, if you are comfortable with bitwise operators – which I'm not).

```
def recv
  fin_and_opcode = socket.read(1).bytes[0]
  mask_and_length_indicator = socket.read(1).bytes[0]
  length_indicator = mask_and_length_indicator - 128
  ...
end
```

If the result is smaller or equal to 125, that is the content length.

```
def recv
  fin_and_opcode = socket.read(1).bytes
  mask_and_length_indicator = socket.read(1).bytes[0]
  length_indicator = mask_and_length_indicator - 128

  length = if length_indicator <= 125
            length_indicator
            ...
          end
end
```

If the `length_indicator` is equal to 126, the next two bytes need to be parsed into a 16-bit unsigned integer to get the numeric value of the length. We do this by using Ruby's `Array#unpack` method, passing in `"n"` to show we want a 16-bit unsigned integer, as per [Ruby's documentation here](#).

```

mask_and_length_indicator = socket.read(1).bytes[0]
length_indicator = mask_and_length_indicator - 128

length = if length_indicator <= 125
  length_indicator
elsif length_indicator == 126
  socket.read(2).unpack("n")[0]
...
end
end

```

If the `length_indicator` is equal to 127, the next eight bytes will need to be parsed into a 64-bit unsigned integer to get the length. "Q>" is passed to `unpack` to indicate this.

```

def recv
  fin_and_opcode = socket.read(1).bytes
  mask_and_length_indicator = socket.read(1).bytes[0]
  length_indicator = mask_and_length_indicator - 128

  length = if length_indicator <= 125
    length_indicator
  elsif length_indicator == 126
    socket.read(2).unpack("n")[0]
  else
    socket.read(8).unpack("Q>")[0]
  end

  ...
end

```

The mask-key itself – what we use to decode the content – will be the next 4 bytes. Then, the encoded content will be the next `n`th bytes, where `n` is the content-length we extracted.

```

def recv
  fin_and_opcode = socket.read(1).bytes
  mask_and_length_indicator = socket.read(1).bytes[0]

```

```

length = if length_indicator <= 125
  length_indicator
elsif length_indicator == 126
  socket.read(2).unpack("n")[0]
else
  socket.read(8).unpack("Q>")[0]
end

keys = socket.read(4).bytes
encoded = socket.read(length).bytes
...
end

```

Let's again use the mask-key to decode the content by using this magic function that loops through the bytes and XORs the octet with the $(i \% 4)$ th octet of the mask. This is defined in the specification [here](#).

```

def recv
  fin_and_opcode = socket.read(1).bytes
  mask_and_length_indicator = socket.read(1).bytes[0]
  length_indicator = mask_and_length_indicator - 128

  length = if length_indicator <= 125
    length_indicator
  elsif length_indicator == 126
    socket.read(2).unpack("n")[0]
  else
    socket.read(8).unpack("Q>")[0]
  end

  keys = socket.read(4).bytes
  encoded = socket.read(length).bytes

  decoded = encoded.each_with_index.map do |byte, index|
    byte ^ keys[index % 4]
  end
  ...
end

```

turn it into a string and return it:

```
def recv
  fin_and_opcode = socket.read(1).bytes
  mask_and_length_indicator = socket.read(1).bytes[0]
  length_indicator = mask_and_length_indicator - 128

  length = if length_indicator <= 125
            length_indicator
          elsif length_indicator == 126
            socket.read(2).unpack("n")[0]
          else
            socket.read(8).unpack("Q>")[0]
          end

  keys = socket.read(4).bytes
  encoded = socket.read(length).bytes

  decoded = encoded.each_with_index.map do |byte, index|
    byte ^ keys[index % 4]
  end

  decoded.pack("c*")
end
```



Test it out on the example at the top of this section. If you've gotten stuck, you can refer to the code [here](#).

Sending Messages

To complete our echo server and show the bidirectional power of WebSockets, let's implement a message sending method to our `WebSocketConnection` object. This should be a little more straightforward, as messages from a server do not have to be masked.

```
def send(message)
  ...
end
```

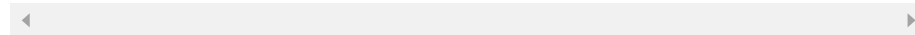
We'll create the initial state of our byte array to send over the socket. This is straightforward as we're sending a complete message and our content is text, so the first value in the array will be 129, i.e. 10000001. The first bit 1, representing that this is a full message, and the last four bits, 0001, showing that the payload is UTF-8 text.

Then we'll get the size of the message and set the length indicator accordingly. Because our frame is not masked, we do not need to add or subtract by 128 (in other words, set the first bit as 1), which the client had done to their messages sent to us.

If the size is smaller or equal to 125, we concatenate this to the byte array.

```
def send(message)
  bytes = [129]
  size = message.bytesize

  bytes += if size <= 125
            [size]
            ...
          end
end
```



If the size is greater than 125 but smaller than 2^{16} , which is the maximum size of two bytes, then we append 126 and the byte array of the length converted from an unsigned 16-bit integer.

```
def send(message)
  bytes = [129]
  size = message.bytesize

  bytes += if size <= 125
            [size]
```

```
...
end
end
```

If the size is greater than 2^{16} , we append 127 to the frame and then the byte array of the length converted from an unsigned 64-bit integer.

```
def send(message)
  bytes = [129]
  size = message.bytesize

  bytes += if size <= 125
    [size]
  elsif size < 2**16
    [126] + [size].pack("n").bytes
  else
    [127] + [size].pack("Q>").bytes
  end

  ...
end
```

Now we can simply append our message as bytes. Then we turn this byte array into chars (using `Array#pack` with the argument `"c*"`). Now we can write this to the socket!

```
def send(message)
  bytes = [129]
  size = message.bytesize

  bytes += if size <= 125
    [size]
  elsif size < 2**16
    [126] + [size].pack("n").bytes
  else
    [127] + [size].pack("Q>").bytes
  end
```



```
    socket << data
  end
```

The Echo Server

Now that we can begin connections, send messages and receive messages, we can write our tiny echo-server application.

```
server = WebSocketServer.new

loop do
  Thread.new(server.accept) do |connection|
    puts "Connected"
    while (message = connection.recv)
      puts "Received #{message} from the browser"
      connection.send("Received #{message}. Thanks!")
    end
  end
end
```

Run this server, and then go into your browser console. Then type:

```
var socket = new WebSocket("ws://localhost:4567");

socket.onmessage = function(event){console.log(event.dat
```

This will set up your WebSocket connection by sending a handshake to your server. Then, if a message is received, it will log it to the console.

Let's send a message and see what we get back:

```
socket.send("hello world!");
```

should have logged out an event whose data is "Received hello world. Thanks!". Meanwhile, your terminal running the server should have logged out "Received hello world from the browser".

That's it! I hope you enjoyed this post and that it was informative for those who were new to WebSocket.

What's Missing?

As I mentioned earlier, there's a lot more one can improve and add to make it a fully-functional WebSocket server – not to mention making it able to handle thousands of concurrent connections. From experience, we've found that developers who implement their own scalable WebSocket solutions have found it tricky to maintain and debug. In addition, developers have to be mindful that WebSocket connections, unlike HTTP, are stateful; managing that state within a properly distributed and load-balanced system is a non-trivial problem.

Thus Pusher's appeal to those for whom realtime is core to their application; we essentially host, maintain and scale these servers for you, and provide an easy-to-use API to interact with them so you can focus on the rest of your application. Hopefully this post has showed you a bit about what goes on underneath.

Further Reading

- [An Introduction to WebSockets](#)
- [WebSocket-Powered Rails Applications](#)
- [Writing WebSocket servers](#)
- [Websockets 101](#)

Getting started with Pusher is *FREE*

Join more than 250,000 happy developers.

Create a Free Account

PREVIOUS POST

← [Update on security](#)

NEXT POST

[What the Hack?](#) →

 COMMENTS


 AUTHOR

2 Comments

Pusher Blog

 Login ▾

 Recommend 7

 Share

Sort by Best ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

Found a typo in one of the code examples:

```
...  
def send_handshake(socket)  
...  
ws_accept = create__accept($1)  
...  
end  
...
```

^ | v • Reply • Share ›



chuanshuo • 3 years ago



```
def get_header(socket, header = "")  
(line = socket.gets) == "rn" ? header : get_header(socket,  
header + line)  
end
```

has a mistake should == "\r\n"

^ | v • Reply • Share ›

ALSO ON PUSHER BLOG

Build an e-commerce application using Laravel and

3 comments • 4 months ago



Nuon Rathana —

AvatarIlluminate\Database\QueryException: SQLSTATE[42000]: Syntax

CSRF in Laravel: how VerifyCsrfToken works and

1 comment • 5 months ago



Said Bakr — However, by a mean of trojan, the attacker may be able to know CSRF

How Laravel implements MVC and how to use it effectively

4 comments • 4 months ago

Optimizing the performance of a Laravel application

2 comments • 2 months ago