Extracted from:

Real-Time Phoenix

Build Highly Scalable Systems with Channels

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Real-Time Phoenix

Build Highly Scalable Systems with Channels

Stephen Bussey

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Sean Dennis
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-719-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2020

Real-time systems are all about getting data from the server to the user, or vice versa, as quickly and efficiently as possible. A critical piece of a real-time system is the communication layer that sits between the server and the user. The user may be on a browser, a mobile app, or even another server. This means that we want to pick a communication layer that can work well in a variety of different circumstances, from high-latency mobile connections to very fast connections.

In this book, we'll use WebSockets as our communication layer; they form the backbone of real-time web applications today. This may change as technology evolves over time, but it's the best solution in the current technology landscape. We'll start building real-time applications in the next chapter, but first we're going to break down how WebSockets work. Understanding WebSockets is crucial in order to build and deliver real-time applications to users. We'll use a "Hello, World!"-style Phoenix application to see the communication of a WebSocket. Once this application is running, we'll look at the different components of a WebSocket to understand how they work.

You can build a real-time system without understanding all the different layers, such as WebSockets, but lacking this knowledge may hurt you in the long run. I remember shipping my first real-time Phoenix application where I didn't fully understand all the layers involved. My WebSockets weren't able to connect! I researched and realized that I needed to understand more about WebSockets in order to get them working with my production load balancer and to reduce my application's memory usage. Learning more about the different layers allowed me to ensure each was working properly.

Let's look at what a WebSocket is and then move into our "Hello WebSocket" application.

Why WebSockets?

It used to be difficult to write real-time systems due to technology limitations at the communication layer. Developers of real-time systems had to make trade-offs between performance, cost, and maintenance; the complicated techniques used often pushed browsers to the limit of their capabilities. Those techniques were highly dependent on the particular web browser used. This meant that a client would be working correctly in one browser but not work in another.

The RFC for the WebSocket protocol emerged with the HTML5 spec in 2011 to solve the challenges of real-time web communication. It took a bit of time for WebSockets to gain support, but they are now supported natively by all major browsers and can be considered mature for application development. We'll be using WebSockets as the primary communication layer in this book because of these strengths:

- WebSockets allow for efficient two-way data communication over a single TCP connection. This helps to minimize message bandwidth and avoids the overhead of creating frequent connections.
- WebSockets have strong support in Elixir with the cowboy web server.[1] They map very well to the Erlang process model which helps to create robust performance-focused applications.
- WebSockets originate with an HTTP request, which means that many standard web technologies such as load balancers and proxies can be used with them.
- WebSockets are able to stay at the edge of our Elixir application. We can change out our communication layer in the future if a better technology becomes available.

WebSockets are powerful. This is evident by the popular and successful applications built using them. Facebook Messenger[2] uses WebSockets to send and receive real-time chats from user clients, allowing Messenger chats to feel snappy. Yahoo Finance[3] uses WebSockets to power their real-time stock ticker across global financial markets. Multiplayer games such as Slither[4] are very popular (not to mention fun!) and are powered completely via WebSockets.

I first dug into the nuts and bolts of WebSockets while developing systems at SalesLoft,[5] an enterprise software as a service (SaaS) company. We use WebSockets to power many important features for our business users, such as real-time notifications and live website information. We send hundreds of millions of events over WebSockets each day.

Enough talk, though, it's time for some action! We'll use a small local Elixir application that exposes a WebSocket in order to see how to connect a WebSocket and how data can be sent over it. You will use this technique to inspect and debug our applications later in the book.

1. https://github.com/ninenines/cowboy
2. https://messenger.com
3. https://finance.yahoo.com
4. https://slither.io
5. https://salesloft.com

Connecting our First WebSocket

To get up and running quickly, we're going to leverage Phoenix's[6] initial project scaffold. This is a good time to go back to Introduction, on page ? in order to make sure that Elixir and Phoenix are set up properly on your system.

We will use mix phx.new to create our first example. You will be prompted to "fetch and install dependencies" during this process. Enter Y in order for the project to be started without manual steps.

```
$ mix phx.new hello_sockets --no-ecto
* creating hello_sockets/config/config.exs
...
Fetch and install dependencies? [Yn] Y
...
```

We'll need to perform one more step to get the sample WebSocket to load. Let's remove the comment on the socket line:

hello_sockets/assets/js/app.js

```
// Import local files
//
// Local files can be imported directly using relative paths, for example:
import socket from "./socket"
```

Run mix phx.server in the hello_sockets folder to start the server. If you get an error when starting the server, double check that you are in the right folder and that you do not already have a program running on port 4000.

Once started, you will see the program running on port 4000:

6. https://phoenixframework.org/

```
$ mix phx.server
Compiling 12 files (.ex)
Generated hello_sockets app
[info] Running HelloSocketsWeb.Endpoint with cowboy 2.6.3 at 0.0.0.0:4000
[info] Access HelloSocketsWeb.Endpoint at http://localhost:4000

Webpack is watching the files…
...
```

We'll use this basic WebSocket application in this chapter to observe how a WebSocket connects and transmits data. It is important to poke around and understand WebSockets so you can debug them more effectively in the future. As you're developing an application, you will spend a fair amount of time looking at what data is being sent to and from the WebSocket.

WebSocket Protocol

WebSockets follow a formal protocol that is implemented by browsers and servers. We will make use of several parts of the WebSocket protocol, but we will not use the entire protocol. In this section, we'll focus on the most basic parts of the protocol. You'll learn how to establish a connection, keep the connection alive, send and receive data, and keep the WebSocket secure.

Using the WebSocket RFC

The RFC for the WebSocket Protocol[a] doesn't make for the most entertaining, or lightest, reading. However, the RFC is highly valuable if you find yourself doing deep debugging into a WebSocket implementation. In this chapter, we'll use Chrome DevTools to inspect how a WebSocket works, but you may benefit from advanced features listed in the RFC.

The RFC can be especially useful if you have extremely tight technical requirements that are not met by the standard WebSocket implementation. However, the standard implementation provided by Phoenix will work for nearly everyone.

a. https://tools.ietf.org/html/rfc6455

We'll use Google Chrome's[7] DevTools to walk through the next example. Any browser with the ability to inspect a WebSocket could be used, although each browser's DevTools vary in look and functionality. WebSockets are supported by all major browsers,[8] which means that you and your users will be able to use WebSockets from any modern device.

7. https://www.google.com/chrome/
8. https://caniuse.com/#feat=websockets

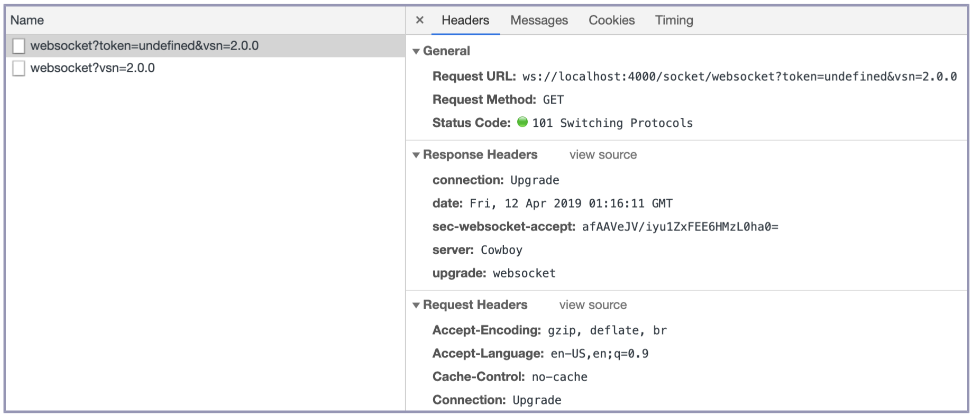
Establishing the Connection

Load the HelloSockets webpage by visiting http://localhost:4000. You will see the default generated Phoenix start screen. What we want to see is hiding from us, and we'll use the DevTools to view it. You can open the DevTools via right-click > Inspect on the webpage. You'll see a variety of tabs, but we want to select the "Network" tab. Once there, reload the webpage in order to capture the connected WebSocket.

Chrome Network Tab Missing Connections

Chrome only shows requests since DevTools was opened. This can lead to a lot of hair-pulling when you're troubleshooting a problem. Reload the webpage if you can't locate your WebSocket connection. Turning it off and on again always works, right?

Select the "WS" tab in order to only show WebSocket connections. Look for the connection labeled websocket?token=undefined&vsn=2.0.0. You may see another connected WebSocket because Phoenix comes with a developer code reloader that operates over a WebSocket, but you can ignore that one. Once you click into the connection, you will see something like this:



```
▼ General
    Request URL: ws://localhost:4000/socket/websocket?token=undefined&vsn=2.0.0
    Request Method: GET
    Status Code: ● 101 Switching Protocols
▼ Response Headers
    connection: Upgrade
    date: Fri, 12 Apr 2019 01:16:11 GMT
    sec-websocket-accept: afAAVeJV/iyu1ZxFEE6HMzL0ha0=
    server: Cowboy
    upgrade: websocket
▼ Request Headers
    Accept-Encoding: gzip, deflate, br
    Accept-Language: en-US,en;q=0.9
    Cache-Control: no-cache
    Connection: Upgrade
```

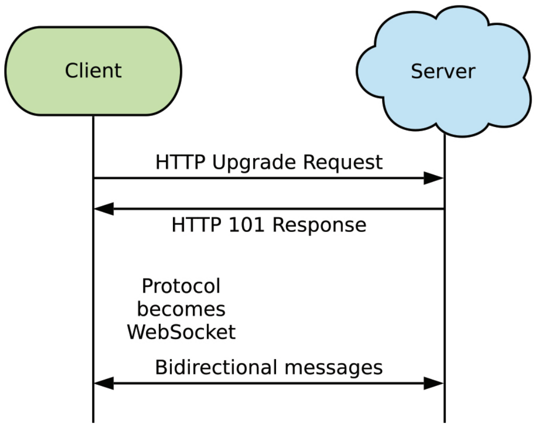
In this image, you can see a few things that reveal how a WebSocket connects. The first is that there are request headers, response headers, and an HTTP method (GET).

A WebSocket starts its life as a normal web request that becomes "upgraded" to a WebSocket. We can see this if we use cURL on the WebSocket endpoint. You'll need several required headers to make this work. The easiest way to generate the cURL request is to right-click the request labeled websocket?token=undefined&vsn=2.0.0 under the "name" column and then select the "copy as cURL" option. This will copy a cURL request to a ws protocol URL. Next, paste the cURL request into your favorite editor and replace ws:// with http://. Run this request in your terminal with the -i flag added. You'll end up with a request that looks like this:

```
# cURL command abbreviated, paste your copied command
# Include all of the headers that came with the copied command
$ curl -i 'http://localhost:4000/socket/websocket?vsn=2.0.0' -H...
HTTP/1.1 101 Switching Protocols
connection: Upgrade
date: Fri, 12 Apr 2019 01:29:18 GMT
sec-websocket-accept: afAAVeJV/iyu1ZxFEE6HMzL0ha0=
server: Cowboy
upgrade: websocket
```

Our web request has received a 101 HTTP response from the server, which indicates that the connection protocol changes from HTTP to a WebSocket. WebSockets operate over a TCP socket using a special data protocol, with the initial HTTP request ensuring that the connection is compatible with browsers and server proxies. The same TCP socket that the HTTP connection request went over becomes the data TCP socket after the upgrade—this allows WebSockets to only use a single socket per connection. WebSockets were designed for allowing browsers to connect to a TCP socket through HTTP, but it is completely acceptable to use them in non-browser environments such as a server or mobile client.

The following figure is a flow diagram of the WebSocket connection process.



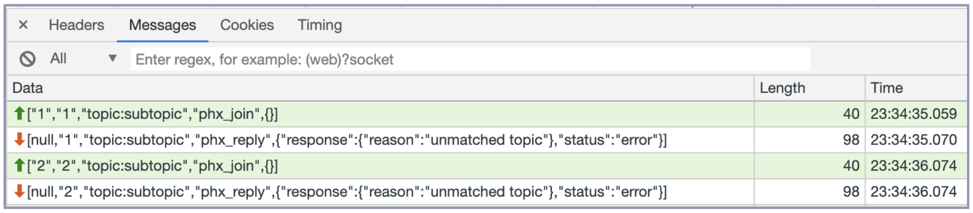
To summarize, a WebSocket connection follows this request flow:

1. Initiate a GET HTTP(S) connection request to the WebSocket endpoint.
2. Receive a 101 or error from the server.
3. Upgrade the protocol to WebSocket if 101 is received.
4. Send/receive frames over the WebSocket connection.

A connection cannot be upgraded with cURL, so we'll move back to DevTools for seeing the data exchange.

Sending and Receiving Data

When you opened the DevTools, you may have noticed a "Messages" tab. This tab shows all messages that are sent to or received from the server. The DevTools for our app looks like this:



Data	Length	Time
⬆["1","1","topic:subtopic","phx_join",{}]	40	23:34:35.059
⬇[null,"1","topic:subtopic","phx_reply",{"response":{"reason":"unmatched topic"},"status":"error"}]	98	23:34:35.070
⬆["2","2","topic:subtopic","phx_join",{}]	40	23:34:36.074
⬇[null,"2","topic:subtopic","phx_reply",{"response":{"reason":"unmatched topic"},"status":"error"}]	98	23:34:36.074

You can ignore the error message for now; the important thing to note is that a WebSocket is capable of sending messages (green background) and receiving messages (white background). This two-way data transmission can happen in both directions simultaneously. A connection which is capable of two-way data transmission is called a full-duplex connection.

WebSockets transmit data through a data framing protocol.[9] We can't see it with the DevTools, but it's worth knowing this provides security benefits and allows WebSocket connections to work properly through different networking layers. These traits allow us to more confidently ship WebSocket-powered applications into production.

The WebSocket protocol contains extensions that provide additional functionality. Extensions are requested by the client using the Sec-WebSocket-Extensions request header. The server can optionally use any of the proposed extensions and return the list of active extensions to the client in a response header named Sec-WebSocket-Extensions. WebSocket data frames are not compressed by default, but can be compressed by using the permessage-deflate extension. This feature allows bandwidth to be reduced at the cost of processing power, which is a benefit for some applications.

Staying Alive, Keep-alive

We have a WebSocket connection that is sending and receiving data, now we have to ensure that the connection stays alive. A disconnected WebSocket is unable to send or receive data. There are things we could do to provide some

9. https://tools.ietf.org/html/rfc6455#section-5

guarantees if a WebSocket disconnects, but we want to base our application on a solid foundation.

The WebSocket protocol specifies Ping and Pong frames[10] which can be used to verify that a connection is still alive. These are optional, though, and you'll soon see that Phoenix doesn't use them. Instead, clients send heartbeat-data messages to the Phoenix Server they're connected to every 30 seconds. The Phoenix WebSocket process will close a connection if it doesn't receive a ping within a timeout period, with 60 seconds the default. With Phoenix, it is possible to use a WebSocket ping control frame to keep the WebSocket connection alive, but the official Phoenix client doesn't use it.

A predictable heartbeat for the connection turns out to be very useful. A connection can be dead but not closed properly; this causes the connection to stay active on the server. A connection that is active but without a client on the other side wouldn't be sending a heartbeat, so it closes gracefully after a short period of time.

It is useful that the client manages the heartbeat rather than the server. If the server is in charge of sending pings to a client, then the server is aware of the connectivity problem but cannot establish a new connection to the client. If a connectivity problem is detected by the client via its ping request, the client can quickly attempt to reconnect and establish the connection again.

10. https://tools.ietf.org/html/rfc6455#section-5.5.2