ably

The WebSocket Handbook

Learn about the technology underpinning the realtime web and build your first web app powered by WebSockets

The WebSocket Handbook

By Alex Diaconu

Version 1, October 2021

Special Thanks

In no particular order: **Jo Franchetti** (for contributing Chapter 4 and building the demo app), **Ramiro Nuñez Dosio** (for encouraging me to write the book in the first place, giving valuable advice, and removing blockers), **Jonathan Mercier-Ganady** (for the technical review), **Jo Stichbury** (for the editorial review), **Leonie Wharton**, **Chris Hipson**, **Jamie Watson** (for all the design work involved).

About the Author

Alex Diaconu is a WebSocket enthusiast who has spent most of his professional career working alongside engineering teams, technical product managers, and system architects, while writing about web technologies. Alex is currently a Technical Content Writer at Ably, where he is exploring the world of realtime tech, and writing about the many challenges of event-driven architectures and distributed systems. In his free time, Alex likes going on hiking trips, watching his favourite football team, playing basketball, and reading sci-fi & history.

Contents

The WebSocket Handbook

Preface

Our everyday digital experiences are in the midst of a realtime revolution. Whether we're talking about virtual events, EdTech, news and financial information, IoT devices, asset tracking and logistics, live score updates, or gaming, consumers increasingly expect realtime digital experiences as standard. And what better to power these realtime interactions than WebSockets?

Until the emergence of WebSockets, the "realtime" web was difficult to achieve and slower than we're used to nowadays; it was delivered by hacking existing HTTP-based technologies that were not designed and optimized for realtime applications.

WebSockets mark a turning point for web development. Designed to be event-driven & full-duplex, and optimized for minimum overhead and low latency, WebSockets have become a preferred choice for many organizations and developers seeking to build interactive, realtime digital experiences that provide delightful user experiences.

Who this book is for

This book is intended for developers (and any other types of technical audiences) who are interested in:

- Learning about the core building blocks of the WebSocket technology, its characteristics, and its advantages.
- Building realtime web apps with WebSockets.

Knowledge of/familiarity with HTML, JavaScript (and Node.js), HTTP, web APIs, and web development is required to get the most out of this book.

What this book covers

Chapter 1: The Road to WebSockets looks at how web technologies evolved since the inception of the World Wide Web, culminating with the emergence of WebSockets, a vastly superior improvement on HTTP for building realtime web apps.

Chapter 2: The WebSocket Protocol covers key considerations related to the WebSocket protocol. You'll find out how to establish a WebSocket connection and exchange messages, what kind of data can be sent over WebSockets, what types of extensions and subprotocols you can use to augment WebSockets.

Chapter 3: The WebSocket API provides details about the constituent components of the WebSocket API — its events, methods, and properties, alongside usage examples for each of them.

Chapter 4: Building a Web App with WebSockets provides detailed, step-by-step instructions on building a realtime web app with WebSockets and Node.js: an interactive cursor position-sharing demo.

Resources — a collection of articles, videos, and WebSocket solutions you might want to explore.

The reader should bear in mind that this is the first version of this book; therefore, it does not intend to be exhaustive. In future versions, we plan to:

- Add more details to the existing chapters.
- Provide more examples and walkthroughs for building apps with WebSockets.
- Cover additional topics that are currently out of scope, such as engineering challenges (for example, scaling and making WebSockets reliable), and alternatives to WebSockets.

CHAPTER 1

The Road to WebSockets

During the 1990s, the web rapidly grew into the dominant way to exchange information. Increasing numbers of users became accustomed to the experience of browsing the web, while browser providers constantly released new features and enhancements.

The first *realtime* web apps started to appear in the 2000s, attempting to deliver responsive, dynamic, and interactive end-user experiences. However, at that time, the realtime web was difficult to achieve and slower than we're used to nowadays; it was delivered by hacking existing HTTP-based technologies that were not designed and optimized for realtime applications. It quickly became obvious that a better alternative was needed.

In this first chapter, we'll look at how web technologies evolved, culminating with the emergence of WebSockets, a vastly superior improvement on HTTP for building realtime web apps.

The World Wide Web is born

In 1989, while working at the European Organization for Nuclear Research (CERN) as a software engineer, Tim Berners-Lee became frustrated with how difficult it was to access information stored on different computers (and, on top of that, running different types of software). This prompted Berners-Lee to develop a project called "WorldWideWeb".

The project proposed a "web" of hypertext documents, which could be viewed by browsers over the Internet using a client-server architecture. The web had the potential to connect the world in a way that was not previously possible, and made it much easier for people everywhere to get information, share, and communicate. Initially used at CERN, the web was soon made available to the world, with the first websites for everyday use starting to appear in 1993-1994.

Berners-Lee managed to create the web by combining two existing technologies: hypertext and the Internet. In the process, he developed three core building blocks:

- **HTML**. The markup (formatting) language of the web.
- **URI**. An “address” (similar to a postal address) that is unique and used to identify each resource on the web.
- **HTTP**. Protocol used for requesting and receiving resources over the web.

This initial version of HTTP[1] (commonly known as HTTP/0.9) that Berners-Lee developed was incredibly basic. Requests consisted of a single line and started with the only supported method, `GET`, followed by the path to the resource:

```
GET /mypage.html
```

The hypertext-only response was extremely simple as well:

```
<HTML>
My HTML page
</HTML>
```

There were no HTTP headers, status codes, URLs, or versioning, and the connection was terminated immediately after receiving the response.

Since interest in the web was skyrocketing, and with HTTP/0.9 being severely limited, both browsers and servers quickly made the protocol more versatile by adding new capabilities. Some key changes:

- Header fields including rich metadata about the request and response (HTTP version number, status code, content type).
- Two new methods — `HEAD` and `POST`.
- Additional content types (e.g., scripts, stylesheets, or media), so that the response was no longer restricted to hypertext.

These modifications were not done in an orderly or agreed-upon fashion, leading to different flavors of HTTP/0.9 in the wild, in turn causing interoperability problems. To resolve these issues, an HTTP Working Group[2] was set up, and in 1996, published HTTP/1.0[3] (defined via RFC 1945). It was an informational RFC, merely documenting all the usages at the time. As such, HTTP/1.0 is not considered a formal specification or an Internet standard.

[1] The Original HTTP as defined in 1991
[2] The IETF HTTP Working Group
[3] RFC 1945: Hypertext Transfer Protocol - HTTP/1.0

In parallel with the efforts made on HTTP/1.0, work to properly standardize HTTP was in progress. The first standardized version of the protocol, HTTP/1.1, was initially defined in RFC 2068[4] and released in January 1997. Several subsequent HTTP/1.1 RFCs[5] have been released since then, most recently in 2014.

HTTP/1.1 introduces many feature enhancements and performance optimizations, including:

- Persistent and pipelined connections.
- Virtual hosting.
- Content negotiation, chunked transfer, compression, and decompression.
- Cache support.
- More methods, bringing the total to seven — `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `OPTIONS`.

JavaScript joins the fold

While HTTP was maturing and being standardized, interest and adoption of the web were growing rapidly. A competition (the so-called "browser wars") for dominance in the usage share of web browsers quickly commenced, initially pitting Microsoft's Internet Explorer against Netscape's Navigator. Both companies wanted to have the best browser, so features and capabilities were inevitably added on a regular basis to their browsers. This competition for supremacy was a catalyst for fast technological breakthroughs.

In 1995, Netscape hired Brendan Eich with the goal of embedding scripting capabilities into their Netscape Navigator browser. Thus, JavaScript was born. The first version of the language was simple, and you could only use it for a few things, such as basic validation of input fields before submitting an HTML form to the server. Limited as it was back then, JavaScript brought dynamic experiences to a web that had been fully static until that point. Progressively, JavaScript was enhanced, standardized, and adopted by all browsers, becoming one of the core technologies of the web as we know it today.

[4] RFC 2068: Hypertext Transfer Protocol - HTTP/1.1

[5] IETF HTTP Working Group, HTTP Documentation, Core Specifications

Hatching the realtime web

The first web applications started to appear in the late ‘90s and used technologies like JavaScript and HTTP. Browsers were already ubiquitous, and users were growing accustomed to the whole experience. Web technologies were constantly evolving, and soon, attempts were made to deliver *realtime* web apps with rich, interactive, and responsive end-user experiences.

We will now look at the main HTTP-centric design models that emerged for developing realtime apps: AJAX and Comet.

AJAX

AJAX (short for Asynchronous JavaScript and XML) is a method of *asynchronously* exchanging data with a server in the background and updating parts of a web page — without the need for an entire page refresh (postback).

Publicly used as a term for the first time in 2005[6], AJAX encompasses several technologies:

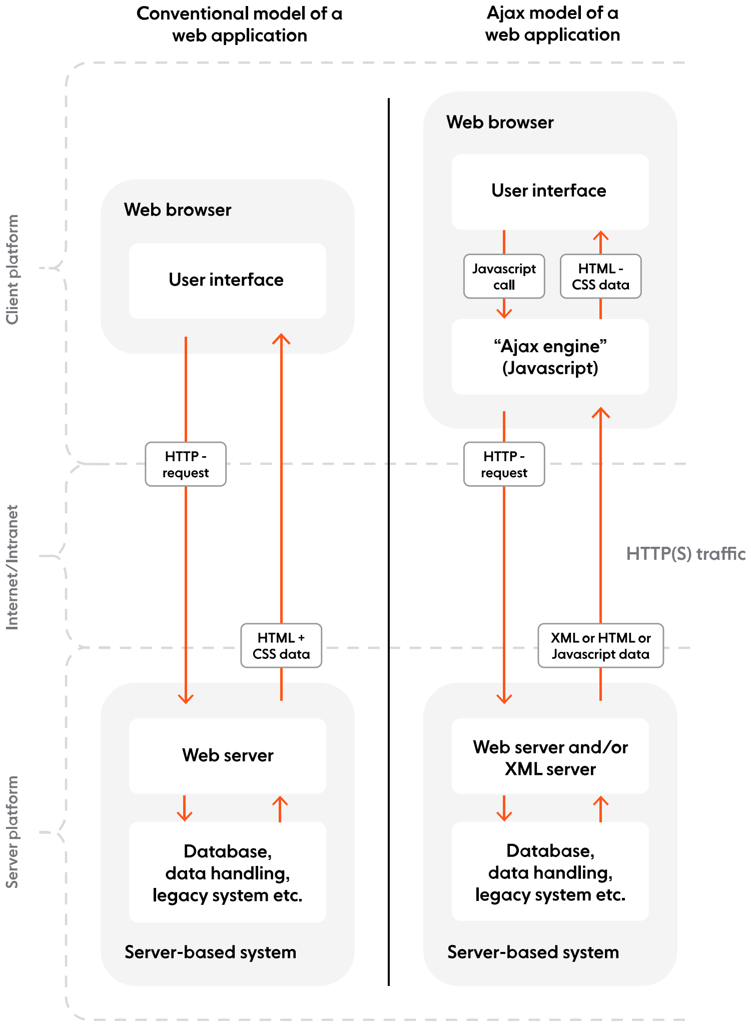
- HTML (or XHTML) and CSS for presentation.
- Document Object Model (DOM) for dynamic display and interaction.
- XML or JSON for data interchange, and XSLT for XML manipulation.
- `XMLHttpRequest`[7] (XHR) object for asynchronous communication.
- JavaScript to bind everything together.

It’s worth emphasizing the importance of `XMLHttpRequest`, a built-in browser object that allows you to make HTTP requests in JavaScript. The concept behind XHR was initially created at Microsoft and included in Internet Explorer 5, in 1999. In just a few years, `XMLHttpRequest` would benefit from widespread adoption, being implemented by Mozilla Firefox, Safari, Opera, and other browsers.

Let’s now look at how AJAX works, by comparing it to the classic model of building a web app.

[6] Jesse James Garrett, Ajax: A New Approach to Web Applications

[7] XMLHttpRequest Living Standard



Conventional model of a web application
Ajax model of a web application
Client platform
Internet/Intranet
Server platform
Web browser
User interface
HTTP - request
HTML + CSS data
Web server
Database, data handling, legacy system etc.
Server-based system
Web browser
User interface
Javascript call
HTML - CSS data
"Ajax engine" (Javascript)
HTTP - request
HTTP(S) traffic
XML or HTML or Javascript data
Web server and/or XML server
Database, data handling, legacy system etc.
Server-based system

Figure 1.1: Classic model of a web app vs. the AJAX model

In a classic model, most user actions in the UI trigger an HTTP request sent to the server. The server processes the request and returns the *entire* HTML page to the client.

In comparison, AJAX introduces an intermediary (an AJAX engine) between the user and the server. Although it might seem counterintuitive, the intermediary significantly improves responsiveness. Instead of loading the webpage, at the start of the session, the client loads the AJAX engine, which is responsible for:

- Regularly polling the server on the client's behalf.
- Rendering the interface the user sees, and updating it with data retrieved from the server.

AJAX (and `XMLHttpRequest` request in particular) can be considered a black swan event for the web. It opened up the potential for web developers to start building truly dynamic, asynchronous, realtime-like web applications that could communicate with the server silently in the background, without interrupting the user's browsing experience. Google was among the first to adopt the AJAX model in the mid-2000s, initially using it for Google Suggest, and its Gmail and Google Maps products. This sparked widespread interest in AJAX, which quickly became popular and heavily used.

Comet

Coined[8] in 2006, Comet is a web application design model that allows a web server to push data to the browser. Similar to AJAX, Comet enables asynchronous communication. Unlike classic AJAX (where the client periodically polls the server for updates), Comet uses long-lived HTTP connections to allow the server to push updates whenever they're available, without the client explicitly requesting them.

The Comet model was made famous by organizations such as Google and Meebo. The former initially used Comet to add web-based chat to Gmail, while Meebo used it for their web-based chat app that enabled users to connect to AOL, Yahoo, and Microsoft chat platforms through the browser. In a short time, Comet became a default standard for building responsive, interactive web apps.

Several different techniques can be used to deliver the Comet model, the most well-known being long polling[9] and HTTP streaming. Let's now quickly review how these two work.

Long polling

Essentially a more efficient form of polling, long polling is a technique where the server elects to hold a client's connection open for as long as possible, delivering a response only after data becomes available or a timeout threshold is reached. Upon receipt of the server response, the client usually issues another request immediately. Long polling is often implemented on the back of `XMLHttpRequest`, the same object that plays a key role in the AJAX model.

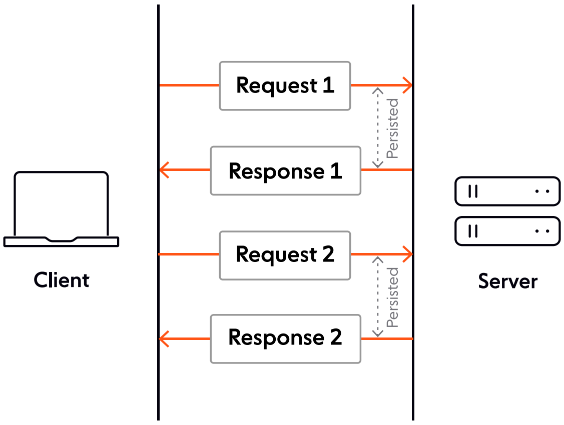


Figure 1.2: High-level overview of long polling

HTTP streaming

Also known as HTTP server push, HTTP streaming is a data transfer technique that allows a web server to continuously send data to a client over a single HTTP connection that remains open indefinitely. Whenever there's an update available, the server sends a response, and only closes the connection when explicitly told to do so.

HTTP streaming can be achieved by using the chunked transfer encoding mechanism available in HTTP/1.1. With this approach, the server can send response data in chunks of newline-delimited strings, which are processed on the fly by the client.

[8] Alex Russell, Comet: Low Latency Data for the Browser
[9] Long Polling - Concepts and Considerations

Here's an example of a chunked response:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

7\r\n
Chunked\r\n
8\r\n
Response\r\n
7\r\n
Example\r\n
0\r\n
\r\n
```

> When chunked transfer encoding is used, each server response includes `Transfer-Encoding: chunked`, while the `Content-Length` header is omitted.

Server-Sent Events[10] (SSE) is another option you can leverage to implement HTTP streaming. SSE is a server push technology commonly used to send message updates or continuous data streams to a browser client. SSE aims to enhance native, cross-browser server-to-client streaming through a JavaScript API called `EventSource`, standardized[11] as part of HTML5 by the World Wide Web Consortium (W3C).

Here's a quick example of opening a stream over SSE:

```
var source = new EventSource('URL_TO_EVENT_STREAM');
source.onopen = function () {
   console.log('connection to stream has been opened');
};
source.onerror = function (error) {
   console.log('An error has occurred while receiving stream', error);
};
source.onmessage = function (stream) {
   console.log('received stream', stream);
};
```

[10] Server-Sent Events (SSE): A Conceptual Deep Dive

[11] Server-sent events, HTML Living Standard

Limitations of HTTP

AJAX and Comet paved the way for creating dynamic, realtime web apps. However — even though they continue to be used nowadays, to a lesser extent — both AJAX and Comet have their shortcomings.

Most of their limitations stem from using HTTP as the underlying transport protocol. The problem is that HTTP was initially designed to serve hypermedia resources in a request-response fashion. It hadn't been optimized to power realtime apps that usually involve high-frequency or ongoing client-server communication, and the ability to react instantly to changes.

Hacking HTTP-based technologies to emulate the realtime web was bound to lead to all sorts of drawbacks. We will now cover the main ones (without being exhaustive).

Limited scalability

HTTP polling, for example, involves sending requests to the server at fixed intervals to see if there's any new update to retrieve. High polling frequencies result in increased network traffic and server demands; this doesn't scale well, especially as the number of concurrent users rises. Low polling frequencies will be less taxing on the server, but they may result in delivery of stale information that has lost (part of) its value.

Although an improvement on regular polling, long polling is also intensive on the server, and handling thousands of simultaneous long polling requests requires huge amounts of resources.

Unreliable message ordering and delivery guarantees

Reliable message ordering can be an issue, since it's possible for multiple HTTP requests from the same client to be in flight simultaneously. Due to various factors, such as unreliable network conditions, there's no guarantee that the requests issued by the client and the responses returned by the server will reach their destination in the right order.

Another problem is that a server may send a response, but network or browser issues may prevent the message from being successfully received. Unless some sort of message receipt confirmation process is implemented, a subsequent call to the server may result in missed messages.

Depending on the server implementation, confirmation of message receipt by one client instance may also cause another client instance to never receive an expected message at all, as the server could mistakenly believe that the client has already received the data it is expecting.

Latency

The time required to establish a new HTTP connection is significant since it involves a handshake with quite a few back and forth exchanges between the client and the server. In addition to the slow start, we must also consider that HTTP requests are issued sequentially. The next request is only sent once the response to the current request has been received. Depending on network conditions, there can be delays before the client gets a response, and the server receives the next request. All of this leads to increased latency for the user — far from ideal in the context of realtime applications.

Although HTTP streaming techniques are better for lower latencies than (long) polling, they are limited themselves (just like any other HTTP-based mechanism) by HTTP headers, which increase message size and cause unnecessary delays. Often, the HTTP headers in the response outweigh the core data being delivered[12].

No bi-directional streaming

A request/response protocol by design, HTTP doesn't support bidirectional, always-on, realtime communication between client and server over the same connection. You can create the illusion of bidirectional realtime communication by using two HTTP connections. However, the maintenance of these two connections introduces significant overhead on the server, because it takes double the resources to serve a single client.

With the web continuously evolving, and user expectations of rich, realtime web-based experiences growing, it was becoming increasingly obvious that an alternative to HTTP was needed.

[12] Matthew O'Riordan, Google — polling like it's the 90s

Enter WebSockets

In 2008, the pain and limitations of using Comet when implementing anything resembling realtime were being felt particularly keenly by developers Michael Carter and Ian Hickson. Through collaboration on IRC[13] and W3C mailing lists[14], they came up with a plan to introduce a new standard for modern, truly realtime communication on the web. Thus, the name "WebSocket" was coined.

In a nutshell, WebSocket is a technology that enables bidirectional, full-duplex communication between client and server over a persistent, single-socket connection. The intent is to provide what is essentially an as-close-to-raw-as-possible TCP communication layer to web application developers while adding a few abstractions to eliminate certain friction that would otherwise exist concerning the way the web works. A WebSocket connection starts as an HTTP request/response handshake; beyond this handshake, WebSocket and HTTP are fundamentally different.

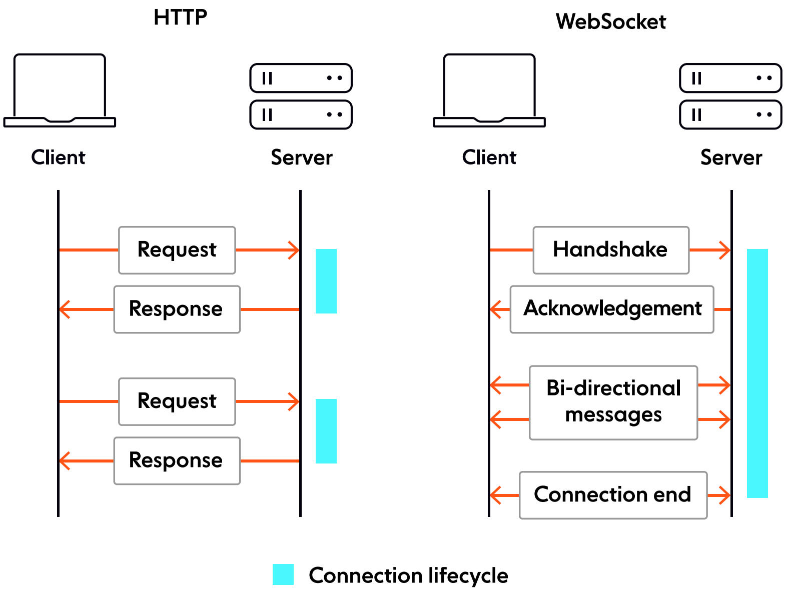


Figure 1.3: WebSockets vs. the traditional HTTP request/response model

The WebSocket technology includes two core building blocks:

- **The WebSocket protocol.** Enables communication between clients and servers over the web, and supports transmission of binary data and text strings. For more details, see Chapter 2: The WebSocket Protocol.
- **The WebSocket API.** Allows you to perform necessary actions, like managing the WebSocket connection, sending and receiving messages, and listening for events triggered by the server. For more details, see Chapter 3: The WebSocket API.

[13] IRC logs, 18.06.2008

[14] W3C mailing lists, TCPConnection feedback

Comparing WebSockets and HTTP

While HTTP is *request-driven*, WebSockets are *event-driven*. The table below illustrates fundamental differences between the two technologies.

WEBSOCKETS	HTTP/1.1
Communication	
Full-duplex	Half-duplex
Message exchange pattern	
Bi-directional	Request-response
Server push	
Core feature	Not natively supported
Overhead	
Moderate overhead to establish the connection, and minimal overhead per message.	Moderate overhead per request/connection.
State	
Stateful	Stateless

Table 1.1: Comparing the characteristics of WebSockets and HTTP/1.1

HTTP and WebSockets are designed for different use cases. For example, HTTP is a good choice if your app relies heavily on CRUD operations, and there's no need for the user to react to changes quickly. On the other hand, when it comes to scalable, low-latency realtime applications, WebSockets are the way to go. More about this in the next section.

Use cases and benefits

The WebSocket technology has broad applicability. You can use it for different purposes, such as streaming data between backend services, or connecting a backend with a frontend via long-lasting, full-duplex connections. In a nutshell, WebSockets are an excellent choice for architecting event-driven systems and building realtime apps and services where it's essential for data to be delivered immediately.

We can broadly group WebSocket use cases into two distinct categories:

- **Realtime updates**, where the communication is unidirectional, and the server is streaming low-latency (and often frequent) updates to the client. Think of live sports updates, alerts, realtime dashboards, or location tracking, to name just a few use cases.
- **Bidirectional communication**, where both the client and the server can send and receive messages. Examples include chat, virtual events, and virtual classrooms (the last two usually involve features like polls, quizzes, and Q&As). WebSockets can also be used to underpin multi-user synchronized collaboration functionality, such as multiple people editing the same document simultaneously.

And here are some of the main benefits of using WebSockets:

- **Improved performance.** Compared to HTTP, WebSockets eliminate the need for a new connection with every request, drastically reducing the size of each message (no HTTP headers). This helps save bandwidth, improves latency, and makes WebSockets more scalable than HTTP and significantly less taxing on the server-side.
- **Extensibility**. Flexibility is ingrained into the design of the WebSocket technology, which allows for the implementation of subprotocols (application-level protocols) and extensions for additional functionality. Learn more about Extensions and Subprotocols.
- **Fast reaction times.** As an event-driven technology, WebSockets allow data to be transferred without the client requesting it. This characteristic is desirable in scenarios where the client needs to react quickly to an event (especially ones it cannot predict, such as a fraud alert).

Adoption

Initially called `TCPConnection`, the WebSocket interface made its way into the HTML5 specification[15], which was first released as a draft in January 2008. The WebSocket protocol was standardized in 2011 via RFC 6455; more about this in Chapter 2: The WebSocket Protocol.

In December 2009, Google Chrome 4 was the first browser to ship full support for WebSockets. Other browser vendors started to follow suit over the next few years; today, all major browsers have full support for WebSockets. Going beyond web browsers, WebSockets can be used to power realtime communication across various types of user agents — for example, mobile apps.

Nowadays, WebSockets are a key technology for building scalable realtime web apps. The WebSocket API and protocol have a thriving community, which is reflected by a variety of client and server options (both open-source and commercial), developer ecosystems, and myriad real-life implementations.

[15] Web sockets, HTML Living Standard

CHAPTER 2

The WebSocket Protocol

In December 2011, the Internet Engineering Task Force (IETF) standardized the WebSocket protocol through RFC 6455[16]. In coordination with IETF, the Internet Assigned Numbers Authority (IANA) maintains the WebSocket Protocol Registries[17], which define many of the codes and parameter identifiers used by the protocol.

This chapter covers key considerations related to the WebSocket protocol, as described in RFC 6455. You'll find out how to establish a WebSocket connection and exchange messages, what kind of data can be sent over WebSockets, what types of extensions and subprotocols you can use to augment WebSockets.

Protocol overview

The WebSocket protocol enables ongoing, full-duplex, bidirectional communication between web servers and web clients over an underlying TCP connection.

In a nutshell, the base WebSocket protocol consists of an opening handshake (upgrading the connection from HTTP to WebSockets), followed by data transfer. After the client and server successfully negotiate the opening handshake, the WebSocket connection acts as a persistent full-duplex communication channel where each side can, independently, send data at will. Clients and servers transfer data back and forth in conceptual units referred to as messages, which, as we describe shortly, can consist of one or more frames. Once the WebSocket connection has served its purpose, it can be terminated via a closing handshake.

[16] RFC 6455: The WebSocket Protocol

[17] IANA WebSocket Protocol Registries

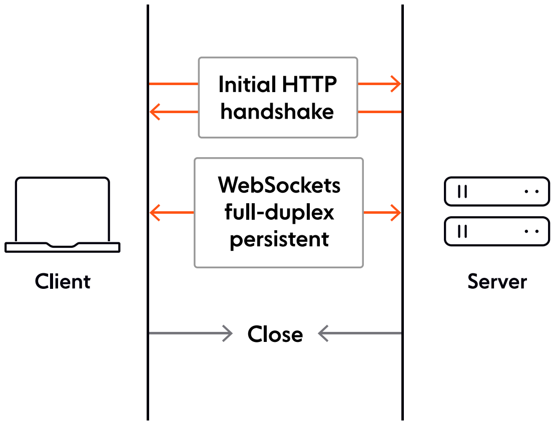


Figure 2.1: High-level overview of a WebSocket connection

URI schemes and syntax

The WebSocket protocol defines two URI schemes for traffic between server and client:

- `ws`, used for unencrypted connections.
- `wss`, used for secure, encrypted connections over Transport Layer Security (TLS).

> The WebSocket URI schemes are analogous to the HTTP ones; the `wss` scheme uses the same security mechanism as `https` to secure connections, while `ws` corresponds to `http`.

The rest of the WebSocket URI follows a generic syntax, similar to HTTP. It consists of several components: host, port, path, and query, as highlighted in the example below.

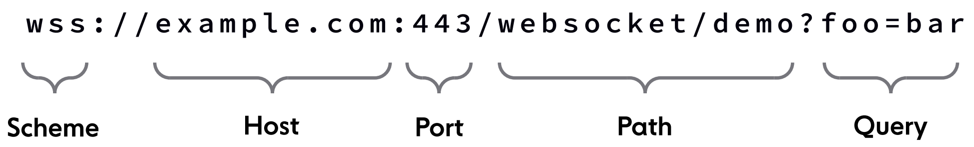


Figure 2.2: WebSocket URI components

It's worth mentioning that:

- The port component is optional; the default is port `80` for `ws`, and port `443` for `wss`.
- Fragment identifiers are not allowed in WebSocket URIs.
- The hash character (`#`) must be escaped as `%23`.

Opening handshake

The process of establishing a WebSocket connection is known as the opening handshake, and consists of an HTTP/1.1 request/response exchange between the client and the server. The client always initiates the handshake; it sends a `GET` request to the server, indicating that it wants to *upgrade* the connection from HTTP to WebSockets. The server must return an `HTTP 101 Switching Protocols` response code for the WebSocket connection to be established. Once that happens, the WebSocket connection can be used for ongoing, bidirectional, full-duplex communications between server and client.

> RFC 8441[18] introduces a separate mechanism that allows you to bootstrap WebSockets with HTTP/2. At the time of writing, this mechanism hasn't been widely adopted by browsers or libraries implementing WebSockets. Consequently, it is out of the scope of this book (but we may cover it in future versions).

Client request

Here's a basic example of a `GET` request made by the client to initiate the opening handshake:

```
GET wss://example.com:8181/ HTTP/1.1
Host: localhost: 8181
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: zy6Dy9mSAIM7GJZNf9rI1A==
```

The request must contain the following headers:

- `Host`
- `Connection`
- `Upgrade`
- `Sec-WebSocket-Version`
- `Sec-WebSocket-Key`

In addition to the required headers, the request may also contain optional ones. See the Opening handshake headers section later in this chapter for more information on headers.

[18] RFC 8441: Bootstrapping WebSockets with HTTP/2

> If any header is not understood or has an incorrect value, the server should stop processing the request and return a response with an appropriate error code, e.g., `400 Bad Request`.

Server response

The server must return an `HTTP 101 Switching Protocols` response code for the WebSocket connection to be successfully established:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept: EDJa7WCAQQzMCYNJM42Syuo9SqQ=
Upgrade: websocket
```

The response must contain several headers: `Connection`, `Upgrade`, and `Sec-WebSocket-Accept`. Other optional headers may be included, such as `Sec-WebSocket-Extensions`, or `Sec-WebSocket-Protocol` (provided they were passed in the client request). See the Opening handshake headers section in this chapter for additional details.

> If the status code returned by the server is anything but `HTTP 101 Switching Protocol`, the handshake will fail, and the WebSocket connection will not be established.

Opening handshake headers

The table below describes the headers used by the client and the server during the opening handshake.

HEADER	REQUIRED	DESCRIPTION
`Host`	Yes	The host name and optionally the port number of the server to which the request is being sent. If no port number is included, a default value is implied (`80` for `ws`, or `433` for `wss`).
`Connection`	Yes	Indicates that the client wants to negotiate a change in the way the connection is being used. Value must be `Upgrade`. Also returned by the server.
`Upgrade`	Yes	Indicates that the client wants to upgrade the connection to alternative means of communication. Value must be `websocket`. Also returned by the server.
`Sec-WebSocket-Version`	Yes	The only accepted value is `13`. Any other version passed in this header is invalid.
`Sec-WebSocket-Key`	Yes	A base64-encoded one-time random value (nonce) sent by the client. Automatically handled for you by most WebSocket libraries or by using the WebSocket class provided in browsers. See the Sec-WebSocket-Key and Sec-WebSocket-Accept section in this chapter for more details.
`Sec-WebSocket-Accept`	Yes	A base64-encoded SHA-1 hashed value returned by the server as a direct response to `Sec-WebSocket-Key`. Indicates that the server is willing to initiate the WebSocket connection. See the Sec-WebSocket-Key and Sec-WebSocket-Accept section in this chapter for more details.

HEADER	REQUIRED	DESCRIPTION
`Sec-WebSocket-Protocol`	No	Optional header field, containing a list of values indicating which subprotocols the client wants to speak, ordered by preference. The server needs to include this field together with one of the selected subprotocol values (the first one it supports from the list) in the response. See the Subprotocols section later in this chapter for more details.
`Sec-WebSocket-Extensions`	No	Optional header field, initially sent from the client to the server, and then subsequently sent from the server to the client. It helps the client and server agree on a set of protocol-level extensions to use for the duration of the connection. See the Extensions section later in this chapter for more details.
`Origin`	No	Header field sent by all browser clients (optional for non-browser clients). Used to protect against unauthorized cross-origin use of a WebSocket server by scripts using the WebSocket API in a web browser. The connection will be rejected if the `Origin` indicated is unacceptable to the server.

Table 2.1: Opening handshake headers

Some common, optional headers like `User-Agent`, `Referer`, or `Cookie` may also be used in the opening handshake. However, we have omitted them from the table above, as they don't directly pertain to WebSockets.

Sec-WebSocket-Key and Sec-WebSocket-Accept

Let's now quickly cover two of the required headers used during the opening handshake: `Sec-WebSocket-Key`, and `Sec-WebSocket-Accept`. Together, these headers are essential in guaranteeing that both the server and the client are capable of communicating over WebSockets.

First, we have `Sec-WebSocket-Key`, which is passed by the client to the server, and contains a 16-byte, base64-encoded one-time random value (nonce). Its purpose is to help ensure that the server does not accept connections from non-WebSocket clients (e.g., HTTP clients) that are being abused (or misconfigured) to send data to unsuspecting WebSocket servers. Here's an example of `Sec-WebSocket-Key`:

```
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
```

In direct relation to `Sec-WebSocket-Key`, the server response includes a `Sec-WebSocket-Accept` header. This header contains a base64-encoded SHA-1 hashed value generated by concatenating the `Sec-WebSocket-Key` nonce sent by the client, and the static value (UUID) `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`.

Based on the `Sec-WebSocket-Key` example provided above, here's the `Sec-WebSocket-Accept` header returned by the server:

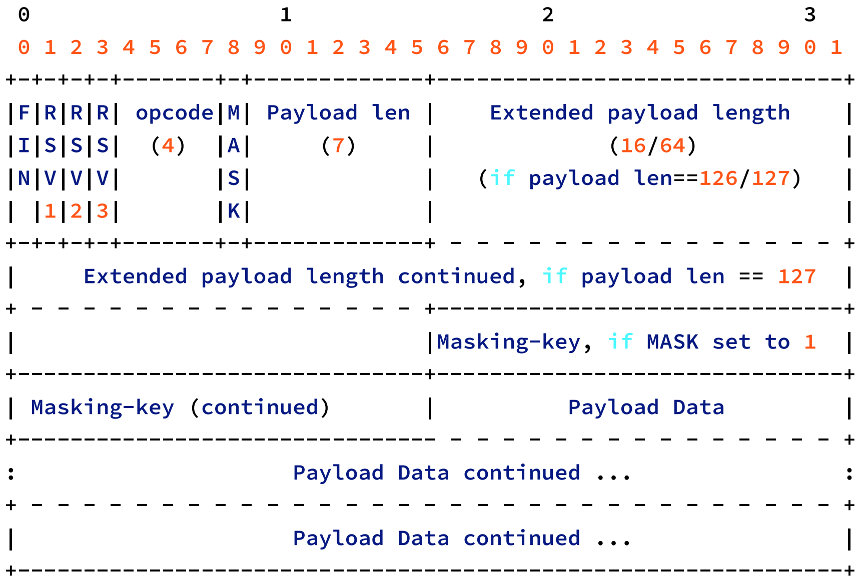
```
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

> If the `Sec-WebSocket-Key` header is missing from the client-initiated handshake, the server will stop processing the request and return an HTTP response with an appropriate error code (`400 Bad Request`, for example). If there's something wrong with the value of `Sec-WebSocket-Accept`, or if the header is missing from the server response, the WebSocket connection will not be established (the client fails the connection).

Message frames

After a successful opening handshake, the client and the server can use the WebSocket connection to exchange messages in full-duplex mode. A WebSocket message consists of one or more frames (see the FIN bit and fragmentation section later in this chapter for details on multi-frame messages).

The WebSocket frame has a *binary* syntax and contains several pieces of information, as shown in the following figure:



```
 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 ...                |
+---------------------------------------------------------------+
```

Figure 2.3: Anatomy of a WebSocket frame

Let's quickly summarize them:

- **FIN bit** - indicates whether the frame is the final fragment in a WebSocket message.
- **RSV 1, 2, 3** - reserved for WebSocket extensions.
- **Opcode** - determines how to interpret the payload data.
- **Mask** - indicates whether the payload is masked or not.
- **Masking key** - key used to unmask the payload data.
- **(Extended) payload length** - the length of the payload.
- **Payload data** - consists of application and extension data.

We will now take a more detailed look at all these constituent parts of a WebSocket frame.

FIN bit and fragmentation

There are numerous scenarios where fragmenting a WebSocket message into multiple frames is required (or at least desirable). For example, fragmentation is often used to improve performance. Without fragmentation, an endpoint would have to buffer the entire message before sending it. With fragmentation, the endpoint can choose a reasonably sized buffer, and when that is full, send subsequent frames as a continuation. The receiving endpoint then assembles the frames to recreate the WebSocket message.

Per RFC 6455[19], another use case for fragmentation is represented by multiplexing, where "[...] it is not desirable for a large message on one logical channel to monopolize the output channel, so the multiplexing needs to be free to split the message into smaller fragments to better share the output channel."

> All data frames that comprise a WebSocket message must be of the same type (text or binary); you can't have a fragmented message that consists of both text and binary frames. However, a fragmented WebSocket message may include control frames. See the Opcodes section later in this chapter for more details about frame types.

Let's now look at some quick examples to illustrate fragmentation. Here's what a single-frame message might look like:

```
0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains "Hello")
```

In comparison, with fragmentation, the same message would look like this:

```
0x01 0x03 0x48 0x65 0x6c (contains "Hel")
0x80 0x02 0x6c 0x6f (contains "lo")
```

The WebSocket protocol makes fragmentation possible via the first bit of the WebSocket frame — the `FIN` bit, which indicates whether the frame is the final fragment in a message. If it is, the `FIN` bit must be set to 1. Any other frame must have the `FIN` bit clear.

[19] RFC 6455: The WebSocket Protocol

RSV 1-3

`RSV1`, `RSV2`, and `RSV3` are reserved bits. They must be 0 unless an extension was negotiated during the opening handshake that defines non-zero values. See the Extensions section in this chapter for more details.

Opcodes

Every frame has an opcode that determines how to interpret that frame's payload data. The standard opcodes currently in use are defined by RFC 6455 and maintained by IANA[20].

OPCODE	DESCRIPTION
`0`	Continuation frame; continues the payload from the previous frame.
`1`	Indicates a text frame (UTF-8 text data).
`2`	Indicates a binary frame.
`3-7`	Reserved for custom data frames.
`8`	Connection close frame; leads to the connection being terminated.
`9`	A ping frame. Serves as a heartbeat mechanism ensuring the connection is still alive. The receiver must respond with a pong frame.
`10`	A pong frame. Serves as a heartbeat mechanism ensuring the connection is still alive. Sent as a response after receiving a ping frame.
`11-15`	Reserved for custom control frames.

Table 2.2: Frame opcodes

> `8` (Close), `9` (Ping), and `10` (Pong) are known as *control frames*, and they are used to communicate *state* about the WebSocket connection.

[20] IANA WebSocket Opcode Registry

Masking

Each WebSocket frame sent by the client to the server needs to be masked with the help of a random `masking-key` (32-bit value). This key is contained within the frame, and it's used to obfuscate the payload data. However, when data flows the other way around, the server must not mask any frames it sends to the client.

> A masking bit set to 1 indicates that the respective frame is masked (and therefore contains a `masking-key`). The server will close the WebSocket connection if it receives an unmasked frame.

On the server-side, frames received from the client must be unmasked before further processing. Here's an example of how you can do that:

```
var unmask = function(mask, buffer) {
   var payload = new Buffer(buffer.length);
   for (var i=0; i<buffer.length; i++) {
       payload[i] = mask[i % 4] ^ buffer[i];
   }
   return payload;
}
```

> Masking is used as a security mechanism that helps prevent cache poisoning.

Payload length

The WebSocket protocol encodes the length of the payload data using a variable number of bytes:

- For payloads <126 bytes, the length is packed into the first two frame header bytes.
- For payloads of 126 bytes, two extra header bytes are used to indicate length.
- If the payload is 127 bytes, eight additional header bytes are used to indicate its length.

Payload data

The WebSocket protocol supports two types of payload data: *text* (UTF-8 Unicode text) and *binary*. In JavaScript, text data is referred to as `String`, while binary data is represented by the `ArrayBuffer` and `Blob` classes. For details on sending and receiving data over WebSockets, together with usage examples, see Chapter 3: The WebSocket API.

> Payload data consists of application data, and potentially extension data (provided extensions were negotiated during the opening handshake).

Each frame's payload type is indicated via a 4-bit opcode (`1` for text or `2` for binary).

Closing handshake

Compared to the opening handshake, the closing handshake is a much simpler process. You initiate it by sending a `close` frame with an opcode of `8`. In addition to the opcode, the close frame may contain a body that indicates the reason for closing. This body consists of a status code (integer) and a UTF-8 encoded string (the reason).

The standard status codes that can be used during the closing handshake are defined by RFC 6455; additional, custom close codes can be registered with IANA[21].

STATUS CODE	NAME	DESCRIPTION
`0-999`	N/A	Codes below `1000` are invalid and cannot be used.
`1000`	Normal closure	Indicates a normal closure, meaning that the purpose for which the WebSocket connection was established has been fulfilled.
`1001`	Going away	Should be used when closing the connection and there is no expectation that a follow-up connection will be attempted (e.g., server shutting down, or browser navigating away from the page).
`1002`	Protocol error	The endpoint is terminating the connection due to a protocol error.

[21] IANA WebSocket Close Code Number Registry

STATUS CODE	NAME	DESCRIPTION
`1003`	Unsupported data	The connection is being terminated because the endpoint received data of a type it cannot handle (e.g., a text-only endpoint receiving binary data).
`1004`	Reserved	Reserved. A meaning might be defined in the future.
`1005`	No status received	Used by apps and the WebSocket API to indicate that no status code was received, although one was expected.
`1006`	Abnormal closure	Used by apps and the WebSocket API to indicate that a connection was closed abnormally (e.g., without sending or receiving a `close` frame).
`1007`	Invalid payload data	The endpoint is terminating the connection because it received a message containing inconsistent data (e.g., non-UTF-8 data within a text message).
`1008`	Policy violation	The endpoint is terminating the connection because it received a message that violates its policy. This is a generic status code; it should be used when other status codes are not suitable, or if there is a need to hide specific details about the policy.
`1009`	Message too big	The endpoint is terminating the connection due to receiving a data frame that is too large to process.
`1010`	Mandatory extension	The client is terminating the connection because the server failed to negotiate an extension during the opening handshake.
`1011`	Internal error	The server is terminating the connection because it encountered an unexpected condition that prevented it from fulfilling the request.
`1012`	Service restart	The server is terminating the connection because it is restarting.
`1013`	Try again later	The server is terminating the connection due to a temporary condition, e.g., it is overloaded.
`1014`	Bad gateway	The server was acting as a gateway or proxy and received an invalid response from the upstream server. Similar to `502 Bad Gateway` HTTP status code.
`1015`	TLS handshake	Reserved. Indicates that the connection was closed due to a failure to perform a TLS handshake (e.g., the server certificate can't be verified).

STATUS CODE	NAME	DESCRIPTION
`1016-1999`	N/A	Reserved for future use by the WebSocket standard.
`2000-2999`	N/A	Reserved for future use by WebSocket extensions.
`3000-3999`	N/A	Reserved for use by libraries, frameworks, and applications. Available for registration at IANA via first-come, first-serve.
`4000-4999`	N/A	Range reserved for private use in applications.

Table 2.3: Closing handshake status codes

Both the client and the server can initiate the closing handshake. Upon receiving a close frame, an endpoint (client or server) has to send a close frame as a response (echoing the status code received).

> Once a `close` frame has been sent, no more data frames can pass over the WebSocket connection.

After an endpoint has both sent and received a `close` frame, the closing handshake is complete, and the WebSocket connection is considered closed.

Subprotocols

Subprotocols (this is the terminology used in RFC 6455) are application-level protocols layered on top of the raw WebSocket protocol. They help define specific formats and higher-level semantics for data exchanges between client and server. Subprotocols can ensure agreement not only about the way the data is structured, but also about the way communication must commence, continue, and eventually terminate.

Subprotocols can be grouped into three main categories:

- **Registered protocols.** This refers to the protocols that are registered with IANA[22].
- **Open protocols.** Open protocols, such as Message Queuing Telemetry Transport (MQTT)[23] or Simple Text Oriented Message Protocol (STOMP)[24].
- **Custom protocols.** Refers to open-source libraries or proprietary solutions introducing their specific flavor of WebSocket-based communications.

Subprotocols are negotiated during the opening handshake. The client uses the `Sec-WebSocket-Protocol` header to pass along one or more comma-separated subprotocols, as shown in this example:

```
Sec-WebSocket-Protocol: amqp, v12.stomp
```

Provided it understands the subprotocols passed in the client request, the server must pick one (and only one) and return it alongside the `Sec-WebSocket-Protocol` header. From this point onwards, the client and server can communicate over the negotiated subprotocol.

> If the server doesn't agree with any of the subprotocols suggested, the `Sec-WebSocket-Protocol` header won't be included in the response.

[22] IANA WebSocket Subprotocol Name Registry
[23] Kayla Matthews, MQTT: A Conceptual Deep-Dive
[24] The Simple Text Oriented Messaging Protocol (STOMP)

Extensions

Extensions are named as such because they extend the WebSocket protocol. They can be used to add new opcodes, data fields, and additional capabilities (multiplexing, for example) to the standard WebSocket protocol.

At the time of writing, there are only a couple of extensions registered with IANA[25], such as `permessage-deflate`, which compresses the payload data portion of WebSocket frames. If you're interested in developing your own extension, you can use an open-source framework like `websocket-extensions`[26].

Extensions are negotiated during the opening handshake. The client uses the `Sec-Websocket-Extensions` header to pass along the extensions it wishes to use, as shown in this example:

```
Sec-WebSocket-Extensions: permessage-deflate, my-custom-extension
```

Provided it supports the extensions sent in the client request, the server must include them in the response, alongside the `Sec-WebSocket-Extensions` header. From this point onwards, the client and server can communicate over WebSockets using the extensions they've negotiated.

Security

In this section, we will cover some of the mechanisms you can use to secure WebSocket connections, and communication done over the WebSocket protocol. This is by no means an exhaustive section; it only aims to provide a high-level overview of several security-related considerations. More about the complex topic of WebSockets security will be treated in future versions of this book.

Let's start with the `Origin` header, which is sent by all browser clients (optional for non-browsers) to the server during the opening handshake. The `Origin` header is essential for securing cross-domain communication. Specifically, if the `Origin` indicated is unacceptable, the server can fail the handshake (usually by returning an `HTTP 403 Forbidden` status code). This ability can be extremely helpful in mitigating denial of service (DoS) attacks.

[25] IANA WebSocket Extension Name Registry

[26] The websocket-extensions framework

Speaking of headers used during the opening handshake, we must also mention `Sec-WebSocket-Key` and `Sec-WebSocket-Accept`. In a nutshell, the purpose of these headers is to protect unsuspecting WebSocket servers from cross-protocol attacks initiated by non-WebSocket clients. Together, `Sec-WebSocket-Key` and `Sec-WebSocket-Accept` ensure that both the client and the server can, in fact, communicate over WebSockets. If there's any issue involving these two headers (e.g., `Sec-WebSocket-Key` is missing from the client request), the WebSocket connection will not be established.

> The WebSocket protocol doesn't prescribe any particular way that servers can authenticate clients. For example, you can handle authentication during the opening handshake, by using cookie headers. Another option is to manage authentication (and authorization) at the application level, by using techniques such as JSON Web Tokens[27].

So far, we've covered security mechanisms that are used during connection establishment. Now, let's look at some aspects that impact security during data exchange between the client and the server. First of all, to reduce the chance of man-in-the-middle and eavesdropping attacks (especially when exchanging critical, sensitive data), it's recommended to use the `wss` URI scheme — which uses TLS to encrypt the connection, just like `https`.

We've talked about message frames earlier in this chapter, and mentioned that frames sent by the client to the server need to be masked with the help of a random `masking-key` (32-bit value). This key is contained within the frame, and it's used to obfuscate the payload data. Frames need to be unmasked by the server before further processing. Masking makes WebSocket traffic look different from HTTP traffic, which is especially useful when proxy servers are involved. That's because some proxy servers may not "understand" the WebSocket protocol, and, were it not for the mask, they might mistake it for regular HTTP traffic; this could lead to all sorts of problems, such as cache poisoning.

[27] RFC 7519: JSON Web Token (JWT)

CHAPTER 3

The WebSocket API

This chapter introduces you to the WebSocket Application Programming Interface (API), which extends the WebSocket protocol to web applications. The WebSocket API enables event-driven communication over a persistent connection. This allows you to build web apps that are truly realtime and less resource-intensive on both the client and the server compared to HTTP techniques.

In the following sections, we'll look at the constituent components of the WebSocket API — its events, methods, and properties.

Overview

Defined in the HTML Living Standard[28], the WebSocket API is a technology that makes it possible to open a persistent two-way, full-duplex communication channel between a web client and a web server. The WebSocket interface enables you to send messages asynchronously to a server and receive event-driven responses without having to poll for updates.

Almost all modern browsers support the WebSocket API[29]. Additionally, there are plenty of frameworks and libraries — both open-source and commercial solutions — that implement WebSocket APIs. See the Resources section in this book for more details.

For the rest of this chapter, we will cover the core capabilities of the WebSocket API. As you will see, it's intuitive, designed with simplicity in mind, and trivial to use.

[28] Web sockets, HTML Living Standard

[29] Can I use WebSockets?

The WebSocket server

A WebSocket server can be written in any server-side programming language that is capable of Berkeley sockets[30]. The server listens for incoming WebSocket connections using a standard TCP socket. Once the opening handshake has been negotiated, the server must be able to send, receive and process WebSocket messages.

See Chapter 4: Building a Web App with WebSockets to learn how to create your own WebSocket server in Node.js.

The WebSocket constructor

To get started with the WebSocket API on the client-side, the first thing to do is to instantiate a `WebSocket` object, which will automatically attempt to open the connection to the server:

```
const socket = new WebSocket('wss://example.org');
```

The WebSocket constructor contains a required parameter — the `url` to the WebSocket server. Additionally, the optional `protocols` parameter may also be included, to indicate one or more WebSocket subprotocols (application-level protocols) that can be used during the client-server communication:

```
const socket = new WebSocket('wss://example.org', 'myCustomProtocol');
```

Once the `WebSocket` object is created and the connection is established, the client can start exchanging data with the server.

> Instantiating the `WebSocket` object essentially initiates the opening handshake we mentioned in the previous chapter.

[30] Berkeley sockets

Events

WebSocket programming follows an asynchronous, event-driven programming model. As long as a WebSocket connection is open, the client and the server simply listen for events in order to handle incoming data and changes in connection status (with no need for polling).

The WebSocket API supports four types of events:

- `open`
- `message`
- `error`
- `close`

> In JavaScript, WebSocket events can be handled by using “onevent” properties (for example, `onopen` is used to handle `open` events). Alternatively, you can use the `addEventListener()` method. Either way, your code will provide callbacks that will execute every time an event is fired.

Open

The `open` event is raised when a WebSocket connection is established. It indicates that the opening handshake between the client and the server was successful, and the WebSocket connection can now be used to send and receive data. Here’s a usage example:

```
// Create WebSocket connection
const socket = new WebSocket('wss://example.org');

// Connection opened
socket.onopen = function(e) {
   console.log('Connection open!');
};
```

Message

The `message` event is fired when data is received through a WebSocket. Messages might contain string (plain text) or binary data, and it's up to you how that data will be processed and visualized.

Here's an example of how to handle a `message` event:

```
socket.onmessage = function(msg) {
   if(msg.data instanceof ArrayBuffer) {
      processArrayBuffer(msg.data);
   } else {
      processText(msg.data);
   }
 }
```

Error

The `error` event is fired in response to unexpected failures or issues (for example, some data couldn't be sent). Here's how you listen for `error` events:

```
socket.onerror = function(e) {
   console.log('WebSocket failure', e);
   handleErrors(e);
};
```

> Errors cause the WebSocket connection to close, so an `error` event is always shortly followed by a close event.

Close

The `close` event fires when the WebSocket connection closes. This can be for a variety of reasons, such as a connection failure, a successful closing handshake, or TCP errors. Once the connection is terminated, the client and server can no longer send or receive messages.

This is how you listen for a `close` event:

```
socket.onclose = function(e) {
   console.log('Connection closed', e);
};
```

> You can manually trigger calling the `close` event by executing the `close()` method.

Methods

The WebSocket API supports two methods: `send()` and `close()`.

send()

Once the connection has been established, you're almost ready to start sending and receiving messages to and from the WebSocket server. But before doing that, you first have to ensure that the connection is open and ready to receive messages. You can achieve this in two main ways.

The first option is to trigger the `send()` method from within the `onopen` event handler, as demonstrated in the following example:

```
socket.onopen = function(e) {
   socket.send(JSON.stringify({'msg': 'payload'}));
}
```

The second way is to check the `readyState` property and choose to send data only when the WebSocket connection is open:

```
function processEvent(e) {
   if(socket.readyState === WebSocket.OPEN) {
       // Socket open, send!
       socket.send(e);
   } else {
       // Show an error, queue it for sending later, etc
   }
}
```

The two code snippets above show how to send text (string) messages. However, in addition to strings, you can also send binary data (`Blob` or `ArrayBuffer`), as shown in this example:

```
var buffer = new ArrayBuffer(128);
socket.send(buffer);

var intview = new Uint32Array(buffer);
socket.send(intview);

var blob = new Blob([buffer]);
socket.send(blob);
```

After sending one or more messages, you can leave the WebSocket connection open for further data exchanges, or call the `close()` method to terminate it.

close()

The `close()` method is used to close the WebSocket connection (or connection attempt). It's essentially the equivalent of the closing handshake we covered previously, in Chapter 2. After this method is called, no more data can be sent or received over the WebSocket connection.

> If the connection is already closed, calling the `close()` method does nothing.

Here's the most basic example of calling the `close()` method:

```
socket.close();
```

Optionally, you can pass two arguments with the `close()` method:

- `code`. A numeric value indicating the status code explaining why the connection is being closed. See the Closing handshake section in Chapter 2 for more details about all the status codes that can be used.
- `reason`. A human-readable string explaining why the connection is closing.

Here’s an example of calling the `close()` method with the two optional parameters:

```
socket.close(1003, 'Unsupported data type!');
```

Properties

The `WebSocket` object exposes several properties containing details about the WebSocket connection.

binaryType

The `binaryType` property controls the type of binary data being received over the WebSocket connection. The default value is `blob`; additionally, WebSockets also support `arraybuffer`.

bufferedAmount

Read-only property that returns the number of bytes of data queued for transmission but not yet sent. The value of `bufferedAmount` resets to zero once all queued data has been sent.

`bufferedAmount` is most useful particularly when the client application transports large amounts of data to the server. Even though calling send() is instant, actually transmitting that data over the Internet is not. Browsers will buffer outgoing data on behalf of your client application. The `bufferedAmount` property is useful for ensuring that all data is sent before closing a connection, or performing your own throttling on the client-side.

Below is an example of how to use bufferedAmount to send updates every second, and adjust accordingly if the network cannot handle the rate:

```
// 10k max buffer size.
var THRESHOLD = 10240;

// Create a New WebSocket connection
const socket = new WebSocket('wss://example.org');

// Listen for the opening event
socket.onopen = function () {
   // Attempt to send update every second.
   setInterval(function () {
       // Send only if the buffer is not full
       if (socket.bufferedAmount < THRESHOLD) {
           socket.send(getApplicationState());
       }
   }, 1000);
};
```

extensions

Read-only property that returns the name of the WebSocket extensions that were negotiated between client and server during the opening handshake.

> If no extensions were negotiated during connection establishment, the extensions property returns an empty string.

“onevent” properties

These properties are called to run associated handler code whenever a WebSocket event is fired. There are four types of “onevent” properties, one for each type of event:

PROPERTY	DESCRIPTION
`onopen`	Called when the WebSocket connection’s `readyState` property changes to 1; this indicates that the connection is open and ready to send and receive data.
`onmessage`	Called when a message is received from the server.
`onerror`	Gets called when an error event occurs, impacting the WebSocket connection.
`onclose`	Called with a close event when the WebSocket’s connection `readyState` property changes to 3; this indicates that the connection is closed.

Table 3.1: “onevent” properties

protocol

Read-only property returning the name of the WebSocket subprotocol that was negotiated for communication between the client and server during the opening handshake.

> Subprotocols are specified via the `protocols` parameter when creating the `WebSocket` object (see The WebSocket constructor section earlier in this chapter for details). If no protocol is specified during connection establishment, the protocol property will return an empty string.

readyState

Read-only property that returns the current state of the WebSocket connection. The table below shows the values you can see reflected by this property, and their meaning:

VALUE	STATE	DESCRIPTION
`0`	`CONNECTING`	Socket has been created, but the connection is not yet open.
`1`	`OPEN`	The connection is open and ready to communicate.
`2`	`CLOSING`	The connection is in the process of closing.
`3`	`CLOSED`	The connection is closed or couldn’t be opened.

Table 3.2: WebSocket connection states

> The value of `readyState` will change over time. It’s recommended to check it periodically to understand the lifespan and life cycle of the WebSocket connection.

url

Read-only property that returns the absolute URL of the WebSocket, as resolved by the constructor.

CHAPTER 4

Building a Web App with WebSockets

The initial version of this chapter was written and published by Jo Franchetti[31]

In this chapter, we will look at how to build a realtime web app with WebSockets and Node.js: *an interactive cursor position-sharing demo*. It's the kind of project that requires bidirectional, instant communication between client and server — the type of use case where the WebSocket technology truly shines.

WebSocket clients and servers

To leverage the WebSocket technology on the server-side, a backend application is required. For our demo, we'll use Node.JS, a lightweight and efficient asynchronous event-driven JavaScript runtime. Node.js is an excellent choice for building scalable realtime web applications and maintaining many hundreds of concurrent WebSocket connections. We'll look at how to implement two different Node.js libraries as the WebSocket server: *ws* and *SockJS*.

Using WebSockets in the frontend is fairly straightforward, via the WebSocket API built into all modern browsers (we'll use this API on the client-side in the first part of the demo, alongside ws on the server-side). Additionally, there are plenty of libraries and solutions implementing the WebSocket technology on both the client-side and the server-side. This includes SockJS, which we will cover in the second part of the demo.

For more details about WebSocket client and server implementations, see the Resources section.

[31] Jo Franchetti, WebSockets and Node.js — testing WS and SockJS by building a web app

ws — a Node.js WebSocket library

ws[32] is a WebSocket server for Node.js. It's quite low-level: you listen to incoming connection requests and respond to raw messages as either strings or byte buffers.

In order to demonstrate how to set up WebSockets with Node.js and ws, we will build a demo app that shares users' cursor positions in realtime. We walk through building it below.

Building an interactive cursor position-sharing demo with ws

This is a demo to create a colored cursor icon for every connected user. When they move their mouse around, their cursor icon moves on the screen and is also shown moving on the screen of every connected user. This happens in realtime, as the mouse is being moved.

Setting up the WebSocket server

First, require the ws library and use the `WebSocket.Server` method to create a new WebSocket server on port 7071 (or any other port of your choosing):

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 7071 });
```

> For brevity's sake, we call it `wss` in our code. Any resemblance to secure WebSockets (often referred to as `wss`) is a coincidence.

Next, create a `Map` to store a client's metadata (any data we wish to associate with a WebSocket client):

```
const clients = new Map();
```

[32] ws: a Node.js WebSocket library

Subscribe to the `wss` connection event using the `wss.on` function, which provides a callback. This will be fired whenever a new WebSocket client connects to the server:

```
wss.on('connection', (ws) => {
    const id = uuidv4();
    const color = Math.floor(Math.random() * 360);
    const metadata = { id, color };

    clients.set(ws, metadata);
```

Every time a client connects, we generate a new unique ID, which is used to identify them. Clients are also assigned a cursor color by using `Math.random()`; this generates a number between 0 and 360, which corresponds to the hue value of an HSV color. The ID and cursor color are then added to an object that we'll call `metadata`, and we're using the `Map` to associate them with our `ws` WebSocket instance.

The `Map` is a dictionary — we can retrieve this metadata by calling `get` and providing a WebSocket instance later on.

Using the newly connected WebSocket instance, we subscribe to that instance's `message` event, and provide a callback function that will be triggered whenever this specific client sends a message to the server.

```
ws.on('message', (messageAsString) => {
```

> This event is on the WebSocket instance (`ws`) itself, and **not** on the `WebSocket.Server` instance (`wss`).

Whenever our server receives a message, we use `JSON.parse` to get the message contents, and load our client metadata for this socket from our `Map` using `clients.get(ws)`.

We're going to add our two metadata properties to the message as `sender` and `color`:

```
const message = JSON.parse(messageAsString);
const metadata = clients.get(ws);

message.sender = metadata.id;
message.color = metadata.color;
```

Then we `stringify` our message again, and send it out to every connected client:

```
 const outbound = JSON.stringify(message);

  [...clients.keys()].forEach((client) => {
    client.send(outbound);
  });
});
```

Finally, when a client closes its connection, we remove its `metadata` from our `Map`:

```
   ws.on("close", () => {
      clients.delete(ws);
    });
});
```

At the bottom we have a function to generate a unique ID:

```
function uuidv4() {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c)
{
    var r = Math.random() * 16 | 0, v = c == 'x' ? r : (r & 0x3 | 0x8);
    return v.toString(16);
  });
}
console.log('wss up');
```

This server implementation multicasts, sending any message it has received to *all connected clients*.

We now need to write some client-side code to connect to the WebSocket server, and transmit the user's cursor position as it moves.

WebSockets on the client-side

We're going to start with some standard HTML5 boilerplate:

```
<!DOCTYPE html>
<html lang='en'>
<head>
    <meta charset='UTF-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <meta name='viewport' content='width=device-width, initial-scale=1.0'>
    <title>Document</title>
```

Next, we add a reference to a style sheet, and an `index.js` file that we're adding as an *ES Module* (using `type=”module”`).

```
    <link rel='stylesheet' href='style.css'>
    <script src='index.js' type='module'></script>
</head>
```

The body contains a single HTML `template` which contains an `SVG` image of a pointer. We're going to use JavaScript to clone this template whenever a new user connects to our server.

```
<body id='box'>
    <template id='cursor'>
        <svg viewBox='0 0 16.3 24.7' class='cursor'>
            <path stroke='#000' stroke-linecap='round' stroke-
linejoin='round'
            stroke-miterlimit='10' d='M15.6 15.6L.6.6v20.5l4.6-4.5 3.2 7.5
            3.4-1.3-3-7.2z' />
        </svg>
    </template>
</body>
</html>
```

Next, we need to use JavaScript to connect to the WebSocket server:

```
(async function() {
    const ws = await connectToServer();
    ...
```

We call the `connectToServer()` function, which resolves a promise containing the connected WebSocket (the function definition will be written later.)

Once connected, we add a handler for `onmousemove` to the `document.body`. The `messageBody` is very simple: it consists of the current `clientX` and `clientY` properties from the mouse movement event (the horizontal and vertical coordinates of the cursor within the application's viewport).

We `stringify` this object, and send it via our now connected `ws` WebSocket instance as the message text:

```
document.body.onmousemove = (evt) => {
     const messageBody = { x: evt.clientX, y: evt.clientY };
     ws.send(JSON.stringify(messageBody));
 };
```

Now we need to add another handler, this time for an `onmessage` event to the WebSocket instance `ws`. Remember that every time the WebSocket server receives a message, it'll forward it to *all connected clients*.

You might notice that the syntax here differs slightly from the server-side WebSocket code. That's because we're using the browser's native `WebSocket` class, rather than the `ws` library.

```
ws.onmessage = (webSocketMessage) => {
     const messageBody = JSON.parse(webSocketMessage.data);
     const cursor = getOrCreateCursorFor(messageBody);
     cursor.style.transform = `translate(${messageBody.x}px,
     ${messageBody.y}px)`;
 };
```

When we receive a message over the WebSocket, we parse the data property of the message, which contains the stringified data that the `onmousemove` handler sent to the WebSocket server, along with the additional `sender` and `color` properties that the server-side code adds to the message.

Using the parsed `messageBody`, we call `getOrCreateCursorFor`. This function returns an HTML element that is part of the DOM. We'll look at how it works later.

We then use the `x` and `y` values from the `messageBody` to adjust the cursor position using a `CSS transform`.

Our code relies on two utility functions. The first is `connectToServer`, which opens a connection to our WebSocket server, and then returns a `Promise` that resolves when the WebSocket `readyState` property is `1 - CONNECTED`.

This means that we can just `await` this function, and we'll know that we have a connected and working WebSocket connection.

```
async function connectToServer() {
     const ws = new WebSocket('ws://localhost:7071/ws');
     return new Promise((resolve, reject) => {
         const timer = setInterval(() => {
             if (ws.readyState === 1) {
                  clearInterval(timer)
                  resolve(ws);
             }
         }, 10);
     });
 }
```

We also use our `getOrCreateCursorFor` function. This function first attempts to find any existing element with the HTML data attribute `data-sender` where the value is the same as the `sender` property in our message. If it finds one, we know that we've already created a cursor for this user, and we just need to return it so the calling code can adjust its position.

```
function getOrCreateCursorFor(messageBody) {
    const sender = messageBody.sender;
    const existing = document.querySelector(`[data-sender='${sender}']`);
    if (existing) {
        return existing;
    }
```

If we can't find an existing element, we clone our HTML `template`, add the data attribute with the current `sender` ID to it, and append it to the `document.body` before returning it:

```
        const template = document.getElementById('cursor');
        const cursor = template.content.firstElementChild.cloneNode(true);
        const svgPath = cursor.getElementsByTagName('path')[0];

        cursor.setAttribute('data-sender', sender);
        svgPath.setAttribute('fill', `hsl(${messageBody.color}, 50%, 50%)`);
        document.body.appendChild(cursor);

        return cursor;
    }

}) ();
```

Now when you run the web application, each user viewing the page will have a cursor that appears on everyone's screens because we are sending the data to all the clients using WebSockets.

Running the demo

If you've been following along with the tutorial, then you can run:

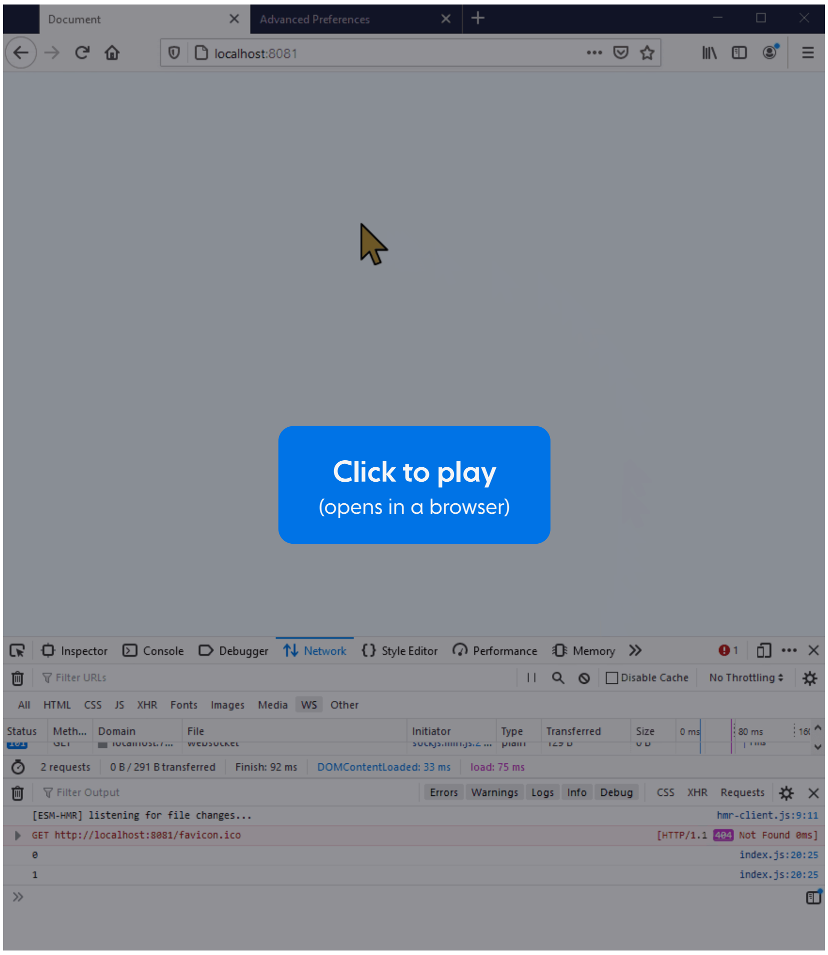
```
> npm install
> npm run start
```

If not, you can clone a working version of the demo from: https://github.com/ably-labs/websockets-cursor-sharing.

```
> git clone https://github.com/ably-labs/WebSockets-cursor-sharing.git
> npm install
> npm run start
```

This demo includes two applications: a web app that we serve through Snowpack[33], and a Node.js web server. The NPM start task spins up both the API and the web server.

The demo should look as depicted below:



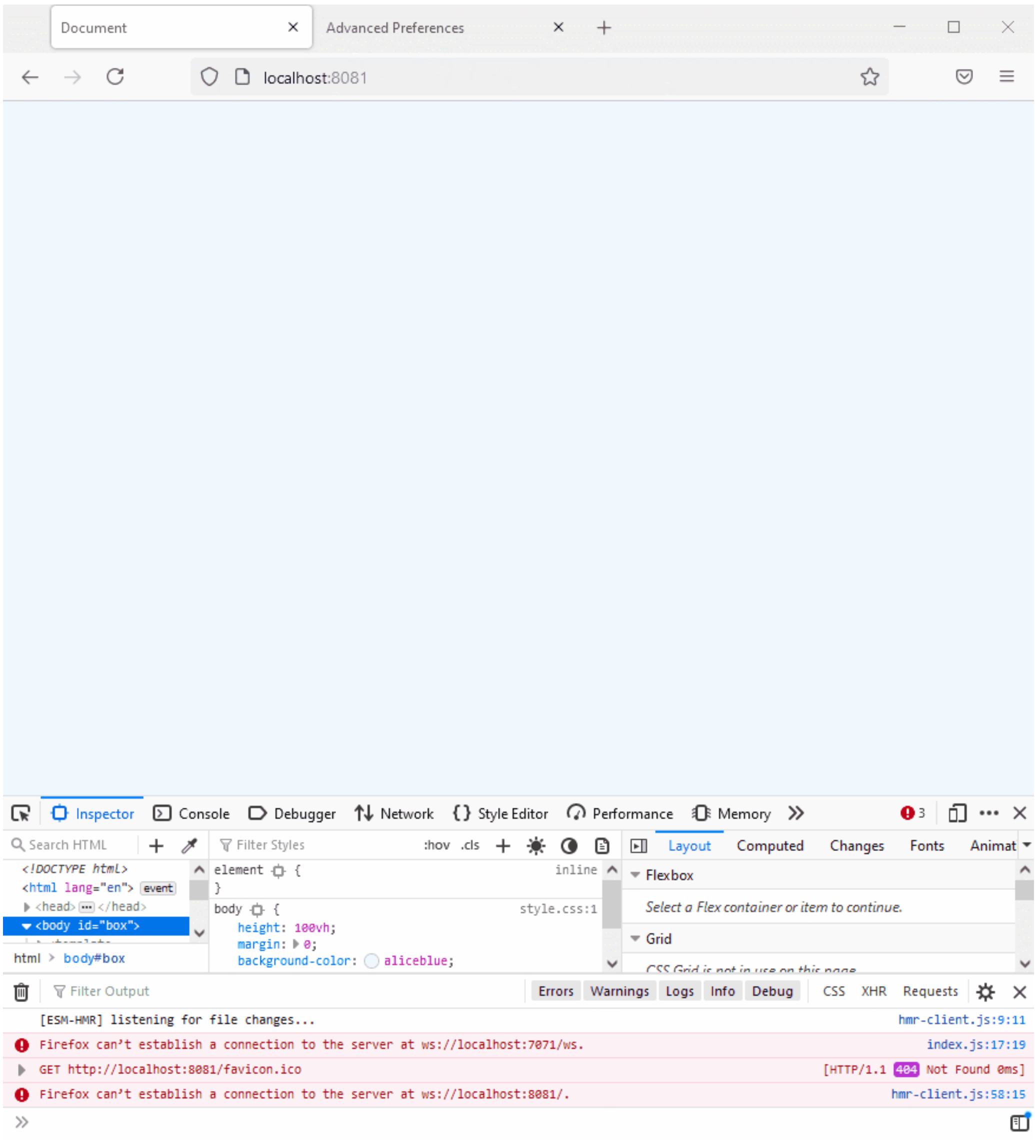
```
2 requests   0 B / 291 B transferred   Finish: 92 ms   DOMContentLoaded: 33 ms   load: 75 ms
```

```
[ESM-HMR] listening for file changes...                          hmr-client.js:9:11
GET http://localhost:8081/favicon.ico              [HTTP/1.1 404 Not Found 0ms]
0                                                                index.js:20:25
1                                                                index.js:20:25
```

Figure 4.1: Realtime cursor movement powered by the ws WebSockets library

[33] Snowpack

However, if you are running the demo in a browser that does not support WebSockets (e.g., IE9 or below), or if you are restricted by particularly tight corporate proxies, you will get an error saying that the browser can’t establish a connection:



```
<!DOCTYPE html>
<html lang="en"> event
  <head> … </head>
  <body id="box">
```

```
element {
}
body {                                    style.css:1
  height: 100vh;
  margin: 0;
  background-color: aliceblue;
```

```
[ESM-HMR] listening for file changes...                                     hmr-client.js:9:11
Firefox can’t establish a connection to the server at ws://localhost:7071/ws.    index.js:17:19
GET http://localhost:8081/favicon.ico                          [HTTP/1.1 404 Not Found 0ms]
Firefox can’t establish a connection to the server at ws://localhost:8081/.   hmr-client.js:58:15
```

Figure 4.2: Error message returned by the browser when a WebSocket connection can’t be established

This is because the ws library offers no fallback transfer protocols if WebSockets are unavailable. If this is a requirement for your project, or you want to have a higher level of reliability of delivery for your messages, then you will need a library that offers multiple transfer protocols, such as SockJS.

SockJS — a JavaScript library to provide WebSocket-like communication

SockJS is a library that mimics the native WebSocket API in browsers. Additionally, it will fall back to HTTP whenever a WebSocket fails to connect, or if the browser being used doesn't support WebSockets. Like ws, SockJS requires a server counterpart; its maintainers provide both a JavaScript client library[34], and a Node.js server library[35].

Using SockJS in the client is similar to the native WebSocket API, with a few small differences. We can swap out ws in the demo built previously and use SockJS instead to include fallback support.

Updating the interactive cursor position sharing demo to use SockJS

To use SockJS in the client, we first need to load the SockJS JavaScript library from their CDN. In the head of the `index.html` document we built earlier, add the following line above the script include of `index.js`:

```
<script src='https://cdn.jsdelivr.net/npm/sockjs-client@1/dist/sockjs.min.js' defer></script>
```

Note the `defer` keyword — it ensures that the SockJS library is loaded before `index.js` runs.

In the `app/script.js` file, we then update the JavaScript to use SockJS. Instead of the `WebSocket` object, we'll now use a `SockJS` object. Inside the `connectToServer` function, we'll establish the connection with the SockJS server:

```
const ws = new SockJS('http://localhost:7071/ws');
```

> SockJS requires a prefix path on the server URL. The rest of the `app/script.js` file requires no change.

[34] SockJS-client

[35] SockJS-node

Next, we have to update the `API/script.js` file to make our server use SockJS. This means changing the names of a few event hooks, but the API is very similar to ws.

First, we need to install `sockjs-node`. In your terminal run:

```
> npm install sockjs
```

Then we need to require the `sockjs` module and the built-in HTTP module from Node. Delete the line that requires ws and replace it with the following:

```
const http = require('http');
const sockjs = require('sockjs');
```

We then change the declaration of `wss` to become:

```
const wss = sockjs.createServer();
```

At the very bottom of the API/index.js file we'll create the HTTPS server and add the SockJS HTTP handlers:

```
const server = http.createServer();
wss.installHandlers(server, {prefix: '/ws'});
server.listen(7071, '0.0.0.0');
```

We map the handlers to a prefix supplied in a configuration object (`'/ws'`). We tell the HTTP server to listen on port 7071 (arbitrarily chosen) on all the network interfaces on the machine.

The final job is to update the event names to work with SockJS:

```
ws.on('message',           will become      ws.on('data',
client.send(outbound);     will become      client.write(outbound);
```

And that's it, the demo will now run with WebSockets where they are supported; and where they aren't, it will use Comet long polling. This latter fallback option will show a slightly less smooth cursor movement, but it is more functional than no connection at all!

Running the demo with SockJS

If you've been following along with the tutorial, then you can run:

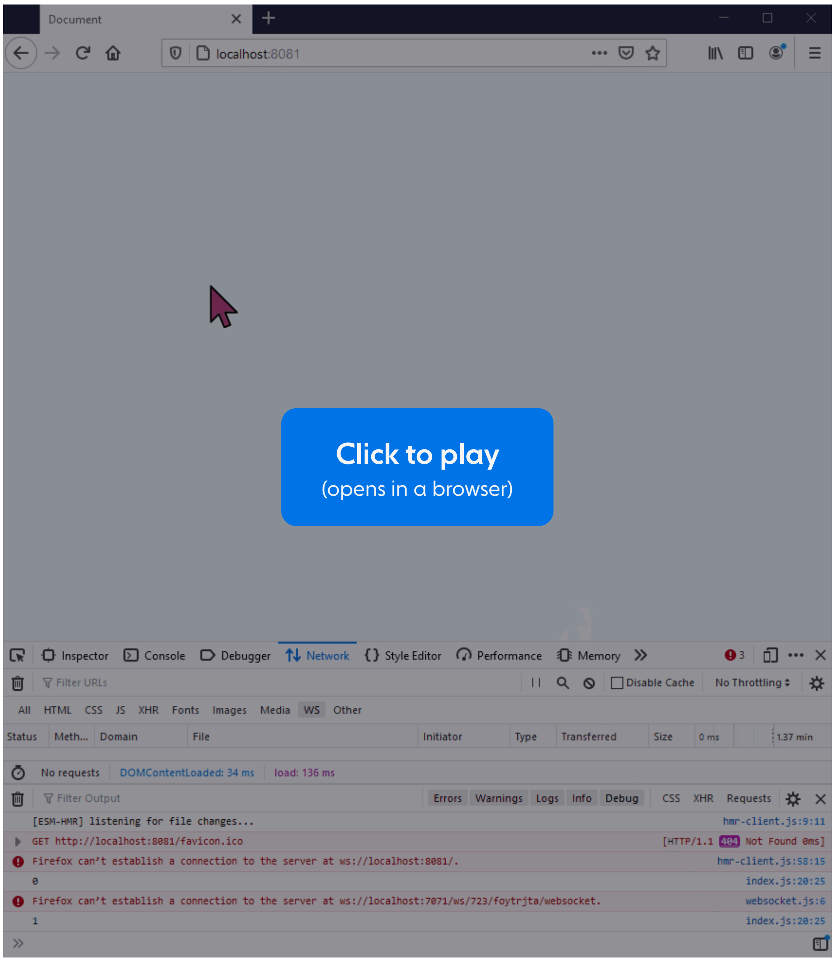
```
> npm install
> npm run start
```

If not, you can clone a working version of the demo from: https://github.com/ably-labs/websockets-cursor-sharing/tree/sockjs.

```
> git clone - b sockjs https://github.com/ably-labs/WebSockets-cursor-
sharing.git
> npm install
> npm run start
```

This demo includes two applications: a web app that we serve through Snowpack[36], and a Node.js web server. The NPM start task spins up both the API and the web server.

The demo should look as depicted below:



```
[ESM-HMR] listening for file changes...                                    hmr-client.js:9:11
GET http://localhost:8081/favicon.ico                       [HTTP/1.1 404 Not Found 0ms]
Firefox can't establish a connection to the server at ws://localhost:8081/.  hmr-client.js:58:15
0                                                                          index.js:20:25
Firefox can't establish a connection to the server at ws://localhost:7071/ws/723/foytrjta/websocket.  websocket.js:6
1                                                                          index.js:20:25
```

Figure 4.3: Realtime cursor movement powered by the SockJS WebSockets library

[36] Snowpack

Scaling WebSockets

You might notice that in both examples we're storing the state in the Node.js WebSocket server — there is a `Map` that keeps track of connected WebSockets and their associated metadata. This means that for the solution to work, and for every user to see one another, they have to be connected to the same WebSocket server.

The number of active users you can support is thus directly related to how much hardware your server has. Node.js is pretty good at managing concurrency, but once you reach a few hundred to a few thousand users, you're going to need to scale your hardware vertically to keep all the users in sync.

Scaling vertically is often an expensive proposition, and you'll always be faced with a performance ceiling of the most powerful piece of hardware you can procure. Additionally, vertical scaling is not elastic, so you have to do it ahead of time.

Once you've run out of vertical scaling options, you'll have to consider horizontal scaling — which is better in the long run, but also significantly more difficult.

What makes WebSockets hard to scale?

To scale regular application servers that don't require persistent connections, the standard approach is to introduce a load balancer in front of them. Load balancers route traffic to whichever node is currently available (either by measuring node performance, or by using techniques such as round-robin).

WebSockets are fundamentally hard to scale because connections to your WebSocket server need to be persistent. And even once you've scaled out your server nodes both vertically *and* horizontally, you also need to provide a solution for sharing data between the nodes. Any state needs to be stored *out-of-process* — this usually involves using something like Redis[37], or a traditional database, to ensure that all the nodes have the same view of state.

In addition to having to share state using additional technology, *broadcasting* to all subscribed clients becomes difficult, because any given WebSocketServer node knows only about the clients connected to itself.

There are multiple ways to solve this: either by using some form of direct connection between the cluster nodes that are handling the traffic, or by using an external pub/sub[38] mechanism. This is sometimes called "adding a backplane" to your infrastructure, and is yet another moving part that makes scaling WebSockets difficult.

We'll dive deeper into the many challenges of scaling WebSockets in future versions of this book.

[37] Redis

[38] Everything You Need To Know About Publish/Subscribe

Resources

References

- Alex Russell, Comet: Low Latency Data for the Browser
- Berkeley sockets
- Can I use WebSockets?
- Everything You Need To Know About Publish/Subscribe
- IANA WebSocket Close Code Number Registry
- IANA WebSocket Extension Name Registry
- IANA WebSocket Opcode Registry
- IANA WebSocket Protocol Registries
- IANA WebSocket Subprotocol Name Registry
- IETF HTTP Working Group, HTTP Documentation, Core Specifications
- IRC logs, 18.06.2018
- Jesse James Garrett, Ajax: A New Approach to Web Applications
- Kayla Matthews, MQTT: A Conceptual Deep-Dive
- Long Polling — Concepts and Considerations
- Matthew O'Riordan, Google — polling like it's the 90s
- Redis
- RFC 1945: Hypertext Transfer Protocol - HTTP/1.0
- RFC 2068: Hypertext Transfer Protocol - HTTP/1.1
- RFC 6455: The WebSocket Protocol
- RFC 7519: JSON Web Token (JWT)
- RFC 8441: Bootstrapping WebSockets with HTTP/2
- Server-Sent Events (SSE): A Conceptual Deep Dive
- Server-sent events, HTML Living Standard
- Snowpack
- SockJS-client
- SockJS-node
- The IETF HTTP Working Group
- The Original HTTP as defined in 1991
- The Simple Text Oriented Messaging Protocol (STOMP)

- The websocket-extensions framework
- W3C mailing lists, TCPConnection feedback
- Web sockets, HTML Living Standard
- ws: a Node.js WebSocket library
- XMLHttpRequest Living Standard

Videos

- A Beginner's Guide to WebSockets
- The Complete Guide to WebSockets
- WebSockets Crash Course - Handshake, Use-cases, Pros & Cons and more

Further reading

- WebSockets Security: Main Attacks and Risks
- WebSocket Security - Cross-Site Hijacking (CSWSH)
- The Future of Web Software Is HTML-over-WebSockets
- Implementing a WebSocket server with Node.js
- Migrating Millions of Concurrent Websockets to Envoy (Slack Engineering)
- The Periodic Table of Realtime

Open-source WebSocket libraries

- Socket.IO
- Nodejs-websocket
- WebSocket-Node
- SockJS-node
- SockJS-client
- ws
- websocket-as-promised
- faye-websocket-node
- Sockette
- rpc-websockets

Final thoughts

We hope this book has helped you gain a good understanding of how WebSockets came to be and how they work, and has enabled you to easily build your first realtime WebSocket-based app. In future versions, we plan to walk you through building additional web apps powered by WebSockets, and cover hard engineering challenges, such as reliably scaling WebSockets, handling reconnections with continuity, or syncing state across multiple WebSocket servers.

We treasure any feedback from our readers. If you've spotted a mistake, if you have any suggestions for what we should include in future versions of the ebook, or if you simply want to chat about WebSockets, reach out to us!

Contact us

About Ably

Ably is the platform that powers synchronized digital experiences in realtime for millions of concurrently connected devices around the world. Whether attending an event in a virtual venue, receiving realtime financial information, or monitoring live car performance data – consumers simply expect realtime digital experiences as standard.

Ably provides a suite of APIs to build, extend, and deliver powerful digital experiences in realtime – primarily over WebSockets – for more than 250 million devices across 80 countries each month. Organizations like Bloomberg, HubSpot, Verizon, and Hopin depend on Ably's platform to offload the growing complexity of business-critical realtime data synchronization at global scale.

Sign up for a free account