O'REILLY®

WebSocket

LIGHTWEIGHT CLIENT-SERVER COMMUNICATIONS

Andrew Lombardi

Websockets

Andrew Lombardi

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Websockets

by Andrew Lombardi

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Brian MacDonald
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January 2015: First Edition

Revision History for the First Edition:

2014-10-14: Early release revison 1

2015-01-12: Early release revison 2

2015-03-28: Early release revison 3

2015-05-29: Early release revison 4

2015-07-22: Early release revison 5

See *http://oreilly.com/catalog/errata.csp?isbn=9781449369279* for release details.

ISBN: 978-1-449-36927-9

[?]

For Joaquín

Table of Contents

Preface

The web has grown up.

In the old days, we used to code design-rich websites using an endless mess of nested tables. Today we can use the standards-based approach with Cascading Style Sheets (CSS) to achieve designs not possible in the web's infancy. Just as CSS ushered in a new era of ability and readability to the design aspects of a site, WebSocket can do that for bi-directional communication with the back-end.

WebSockets provide a standards-based approach to coding for full-duplex bi-directional communication that replaces the age-old hacks like Comet and long polling. Today we have the ability to create desktop-like applications in a browser without resorting to methods that exhaust server-side resources.

Learn the simple ways to deliver to your clients on bi-directional communication between server and client, and do so without making the IT guy cry.

Who Should Read This Book

This book is for programmers who want to create web applications that communicate between server and client in a bi-directional way and are looking to avoid using some of the hacks that are prevalent on the web today. The promise of WebSocket is a better way, based on standards and supported by all of the modern browsers of today with sensible fallback options for those who need to support it. For those who haven't considered WebSockets, put down the Comet tutorial you have been reading.

This book is appropriate for novices and experienced users. We assume that you have a programming background and are familiar with JavaScript. It would also help to have used Node.js, but certainly is not required. It would also benefit those who may be charged with maintaining the servers that run WebSocket code, and responsible for ensuring the security of the infrastructure. You need to know the potential pitfalls of integrating WebSocket and what that means for you. The earlier chapters may be of less

use to you, but the last three chapters should give you enough knowledge to know what is coming across your network.

Goals of this book

I've been in the trenches, and have had to implement acceptable hacks to achieve bi-directional communication for clients who needed the functionality. It is my hope that throughout this book I can show you a better way, one based on standards that proves simple to implement. For several clients over the years, we have successfully deployed the approach we discuss in this book for communicating with the backend which utilized WebSocket rather than long polling and achieved the goals we were after.

Navigating This Book

The most common way I read a book, is to skim and pull out the relevant pieces and use as a reference while coding. If you actually read the preface, the list below should give you a rough idea of each chapters goals:

- Chapters 1 and 2 provide a quickstart with instructions on dependencies needed throughout the book, and introduces you to the JavaScript API
- Chapter 3 brings a full example with client and server code using chat
- In Chapter 4 we write our own implementation of a standard protocol and layer it on top of WebSocket
- Chapter 5 is essential for those who need to support older browsers
- Finally Chapters 6 through 8 dive into aspects of security, debugging, and an overview of the protocol itself which should help

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

A lot of people made this book possible including my wonderful and patient editor Brian MacDonald. To everyone at O'Reilly who helped make this book happen, a deep and profound thanks.

I would also like to thank my technical reviewers for the book for their invaluable input and advice: Joe Ottinger and Arun Gupta. And thanks to those of you who sent in errata on the preview of the book so we could get it solved before going to production.

Thanks to mom and dad, for putting a computer in front of me and opening up an ever expanding universe of creativity and wonder.

CHAPTER 1
Quickstart

The WebSocket API and protocol is defined in RFC 6455. WebSocket gives you a bi-directional full-duplex communications channel that operates over HTTP through a single socket.

Existing hacks which run over HTTP (like long polling) send requests at intervals regardless of whether messages are available without any knowledge of the state of server nor client. The WebSocket API however, is different, the server and client have an open connection from which they can send messages back and forth. For the security minded, WebSocket also operates over TLS (Transport Layer Security or SSL) and is the preferred method as we will see in Chapter 6.

This book is designed to help you understand the Websocket API, the protocol, and how to use it in your web applications today.

In this book, JavaScript is used for all code examples, and Node.js for all server code, and occasional client code or tests when a browser is extraneous to getting a sense of functionality. Some level of proficiency with JavaScript therefore is expected if the examples are to be understood. If you'd like to study up on JavaScript I recommend Douglas Crockfords Javascript: The Good Parts.

Node.js has been so prevalant in the last several years, that the barriers to entry for the examples in this book are remarkably low. If you've done any development for the web, chances are very high that you've developed in JavaScript or at least understand it. The usage of Node.js and JavaScript throughout then, is meant only to simplify the teaching process, and should not be construed as a requirement for a WebSocket project.

There are libraries and servers that support WebSocket in nearly every possible configuration. We'll talk more in Chapter 5 about several of the options you have for deploying a WebSocket-capable server, including fallback methods for clients that don't offer support for this technology yet.

Getting Node and npm

To ensure you can run all of the examples in the book, I strongly recommend that you install Node.js and npm in your development environment. While it is possible to learn all about WebSockets without touching any code, we don't recommend it. Below we'll annotate a few simple ways to get Node in your environment.

Installing on Windows

We will only cover downloading and installing the pre-compiled binary available on Windows. If you are masochistic , and would like to compile it yourself, you can follow these instructions (*https://github.com/joyent/node/wiki/Installation#installing-on-windows*).

For the rest of us, download the stand-alone Windows executable (*http://nodejs.org/dist/latest/node.exe*). Then grab the latest .zip archive of npm (*http://nodejs.org/dist/npm/*). Put node.exe and unpack the npm zip in a directory you add to your `PATH`. You'll be able to run scripts and install modules using node and npm respectively.

Installing on OS X

Two of the easiest methods of installing Node and npm are via a pre-compiled downloadable package, or via a package manager. Our preference is to use a package manager like Homebrew (*http://mxcl.github.io/homebrew/*) to get Node.js onto your machine. This allows for quick and easy updating without having to re-download a package from the web. Assuming you have Homebrew installed:

```
brew install node
```

And if you'd rather use the available pre-compiled binaries you can find the download at (*http://nodejs.org/#download*). When you'd like to install an updated version of Node.js, just download and install the latest package and it will overwrite the existing binaries.

Installing on Linux

Since there are more flavors of Linux than stars in the sky, we'll only outline how to compile it yourself, and how to get it via apt on Ubuntu. If you're running another distro and would like to use the package manager available on your particular flavor, visit the Node.js wiki for instructions on installing (*https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager*).

So getting it using apt requires a few simple steps:

```
sudo apt-get update
sudo apt-get install python-software-properties python g++ make
sudo add-apt-repository ppa:chris-lea/node.js
```

```
sudo apt-get update
sudo apt-get install nodejs
```

This installs the current stable Node.js onto your Ubuntu distro, ready to free JavaScript from the browser and write some server-side code.

If you'd like to compile it yourself, assuming Git is already installed and available on your machine.

```
git clone git://github.com/joyent/node.git
cd node
git checkout v0.10.7
./configure && make && make install
```

Check *http://nodejs.org/* for the latest version of Node.js to checkout onto your system.

Hello, World! Example

When tackling a new topic in development, I prefer to start with an example fairly quickly. So we'll utilize the battle tested example across languages "hello, world!" to initiate a connection to a WebSocket capable Node.js server, and receive the greeting upon connection. `Echo` is the first app to be written in the networking space.

History of hello, world!

The initial incarnation of everyones first application in a new language / technology was first written in Kernighan's 1972 A Tutorial Introduction to the Language B, and it was used to illustrate external variables in the language. (source: *http://en.wikipedia.org/wiki/Hello_world_program*)

We start by writing code that starts up a Websocket capable server on port `8181`. First things first, we will use the CommonJS idiom and **require** the websocket module. and assign that class to `WebSocketServer` object. Then we call the constructor with our initialization object consisting of the port definition.

The Websocket Protocol is essentially a message passing facility, to begin, you will listen for an event called *connection*. Upon receiving a connection event, the provided WebSocket object will be used to send back the "Hello, World!" greeting.

To make life a little bit simpler, and because I don't fancy reinventing the wheel, the wonderful WebSocket library called *ws* will be used. The ws library can take a lot of the

headache out of writing a WebSocket server (or client) by offering a simple, clean API for your Node.js application.

Install it using `npm`:

```
npm install ws
```

Another popular option is to use Websocket Node Library.

All of the examples in this book will assume that they exist in a folder denoted by the abbreviated chapter name, so create a directory called `ch1`. Now create a new file in your editor of choice called `server.js` and add the following code for our application:

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({port: 8181});

wss.on('connection', function(ws) {
    console.log('client connected');
    ws.on('message', function(message) {
        console.log(message);
    });
});
```

Short and to the point. Next, run the server so it's listening for the client you're about to code:

```
node server.js
```

Next for the client, create a file called `client.html` and place in the same directory as the server file. With this simple example, the client can be hosted anywhere, even run from the `file://` protocol. In later chapters, you'll use HTTP libraries and require a more web-centric focus for file and directory management.

In this first pass however, you're going to use a very basic HTML page to call the WebSocket server. The structure of the HTML page will be a simple form, with a text field and a button to initiate the send. The two methods of sending our message will be a form submit (via return/enter) or clicking on the "Send!" button. Then you'll add an action on the form submit and onclick of the button to call the `sendMessage` JavaScript function. One thing to note is the code returns false in the Form's `onsubmit` so the page doesn't refresh.

The WebSocket initialization is rather simple, you initiate a connection to a WebSocket server on port `8181` on `localhost`. Next, since the WebSocket API is event-based (more about this later), you define a function for the onopen event to output a status message for a successful connection to the server. The `sendMessage` function then, merely has to call the send function on the variable `ws` and grab the value inside the `message` textfield.

And *voila!* You have your first WebSocket example.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>WebSocket Echo Demo</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
    href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<link rel="stylesheet"
    href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
    <script>
        var ws = new WebSocket("ws://localhost:8181");
        ws.onopen = function(e) {
          console.log('Connection to server opened');
        }

        function sendMessage() {
            ws.send($('#message').val());
        }
    </script>
</head>
<body lang="en">
    <div class="vertical-center">
    <div class="container">
        <p> </p>
    <form role="form" id="chat_form" onsubmit="sendMessage(); return false;">
        <div class="form-group">
        <input class="form-control" type="text" name="message" id="message"
          placeholder="Type text to echo in here" value="" autofocus/>
        </div>
        <button type="button" id="send" class="btn btn-primary"
          onclick="sendMessage();">Send!</button>
    </form>
    </div>
    </div>
<script
    src="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
</body>
</html>
```

Throughout the book we will utilize the two wonderful libraries prevalent throughout the web for display and interaction:

- Bootstrap 3
- jQuery

In later examples, we'll dispense with including the script and CSS style tags in favor of brevity. You can use the HTML above as a template for future examples, and just remove the contents of our custom `<script>` tag, and the contents between the `<body>` tags while keeping the bootstrap javascript include intact.

With that, open up the HTML page in your favorite browser (I suggest Google Chrome or Mozilla Firefox). Send a message and watch it show up in your servers console output.

If you're using Chrome, it has an excellent facility for viewing WebSocket connections from the frontmost page. Let's do this now, open up Developer Tools (on Windows: F12, Ctrl + Shift + I; on Mac ⌥ + ⌘ + I) or from the **Hot dog menu Tools → Developer tools**.

Figure 1-1 shows the Google Chrome Developer Tool filtered for WebSocket calls

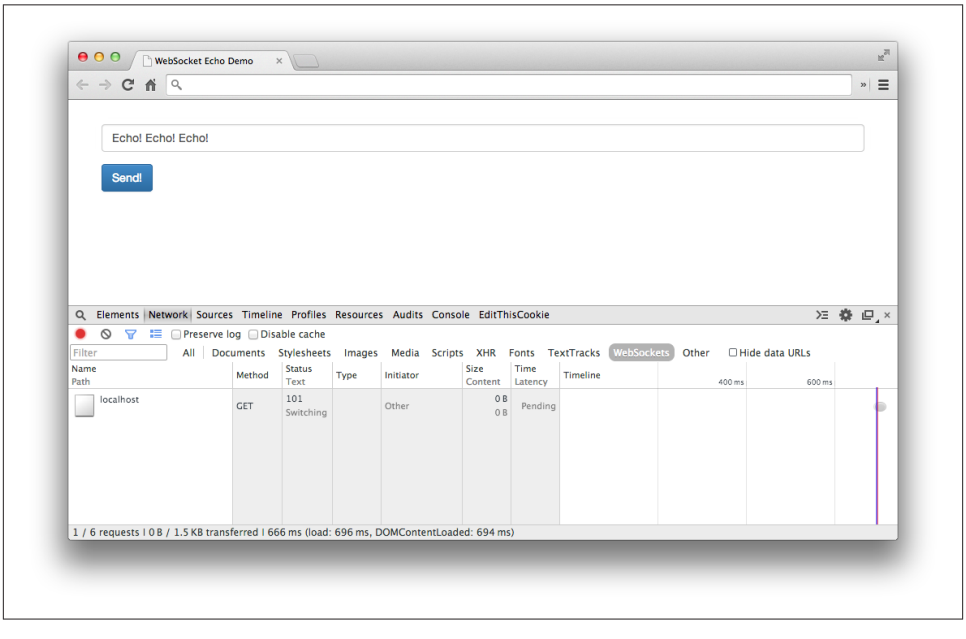


Figure 1-1. Chrome Developer Tool - Network tab

Select the Network tab and refresh the example html. In the table you should see an entry for the HTML, and an entry for the WebSocket connection with status of "101 Switching Protocols". If you select it, you'll see the Request Headers, and Response Headers for this connection.

Request Headers.

```
GET ws://localhost:8181/ HTTP/1.1
Pragma: no-cache
Origin: null
Host: localhost:8181
Sec-WebSocket-Key: qalODNsUoRp+2K9FJty55Q==
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3)...
Upgrade: websocket
Sec-WebSocket-Extensions: x-webkit-deflate-frame
Cache-Control: no-cache
Connection: Upgrade
Sec-WebSocket-Version: 13
```

Response Headers.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept: nambQ7W9imtAIYpzsw4hNNuGD58=
Upgrade: websocket
```

If you're used to seeing HTTP headers, this should look no different. We do have a few extra headers here including `Connection: Upgrade`, `Sec-Websocket-Key`, `Upgrade: websocket`, which we'll go into further detail on in Chapter 8. For now, relish in your first WebSocket example, and lets learn about why WebSockets should be on your radar for your next project.

Why Websockets?

The ability today to create desktop-like applications in the browser is achieved primarily using Comet and AJAX. In order to achieve either solution, developers have used hacks in servers and clients alike to keep connections open longer and fake a long-running connection.

While the above hacks technically work, they open up resource allocation issues on servers. With existing methods, the perceived latency to the end-user may be low, but the efficiency on the backend leaves a lot to be desired. Long polling makes unnecessary requests and keeps a constant stream of opening and closing connections for your servers to deal with. There is no facility for layering other protocols on top of Comet or AJAX, and even if you could the simplicity is just not there.

WebSockets give you the ability to utilize an upgraded HTTP request (we'll discuss the particulars of this in Chapter 8), and send data in a message-based way similar to UDP with all the reliability of TCP. This means a single-connection, and the ability to send data back and forth between client and server with negligible penalty in resource utilization. You can also layer another protocol on top of WebSocket, and provide it in a secure way over TLS. In later chapters we'll dive deeper into these and other features such as heartbeating, origin domain, and more.

One of the common pitfalls of the choice between WebSocket and long polling was the sad state of browser support. Today, the state of browser support for WebSockets is much brighter for the end-user.

The table below shows the current state of browser support for WebSocket. For the most up to date information on support for WebSocket you can reference *http://caniuse.com/websocket*

Table 1-1. Table shows the state of WebSocket browser support

Browser	No support	Partial Support	Full Support
IE	version 8.0, 9.0		version 10.0 and up
Firefox			version 27.0 and up
Chrome			version 31.0 and up
Safari			version 7 and up
Opera			version 20.0 and up
iOS Safari	version 3.2, 4.0-4.1	version 4.2-4.3, 5.0-5.1	version 6.0 and up
Opera Mini	version 5.0-7.0		
Android Browser	version 2.1-4.3		version 4.4
Blackberry Browser			version 7.0, 10.0
IE Mobile			version 10.0

As we'll discover in Chapter 5, we can mitigate the lack of support in older browsers for native WebSocket using framework libraries such as SockJS or Socket.IO.

Summary

In this chapter, we were introduced to WebSockets and how to build a simple `echo` server using Node.js. We also saw how to build a simple client for testing out our WebSocket server, along with one simple way to test our WebSocket server using Chrome Developer Tools. In the next chapters, we will explore the WebSocket API, the protocol, and see how to layer other protocols on-top of WebSocket to give us even more power.

CHAPTER 2
WebSocket API

In this chapter we'll expose the details behind using the WebSocket Application Programming Interface (API). WebSocket is an event-driven full duplex asynchronous communications channel for your web applications. It has the ability to give you real-time updates that today you would use long polling or other hacks to achieve. The primary benefit is reducing resource needs on both the client and more importantly, the server.

While WebSocket utilizes HTTP as the initial transport mechanism, the communication doesn't end after a response is received by the client. Using the WebSocket API, you can be freed from the constraints of the typical HTTP request / response cycle. This also means that as long as the connection stays open, the client and server can freely send messages asynchronously without polling for anything new.

Throughout this chapter you'll build a simple stock ticker client using WebSocket as data transport and learn about it's simple API in the process. So we're going to create a new project folder for `ch2` to store all of our code for this chapter. Our client code will be in a file named `client.html`, and our server code in a file named `server.js`.

Initializing

By default, to initiate a connection to the server, the constructor for the WebSocket object requires a URL. It will connect by default via `port 80` (the `HTTP` port), or any other specified after the host using the same format as the HTTP protocol.

If you're running a traditional webserver on port 80 already, you'll have to use a server that understands and can proxy the WebSocket connection, or can pass the connection through to your custom-written application. In a later chapter we'll discuss one popular option using nginx for passing through an upgraded connection to your Node.js based server.

For now, since you'll be running the WebSocket server locally, without a webserver proxying the connection, you can simply initialize the browser native WebSocket object with the below:

```
var ws = new WebSocket("ws://localhost:8181");
```

We now have a WebSocket object `ws` which we can use to listen for events. In the next section on "WebSocket Events" we'll talk in detail about the various events available to listen for.

Table 2-1. WebSocket Constructor parameters

Parameter Name	Description
URL	URL as `ws://` or `wss://` (if using TLS)
protocol (optional)	parameter specifying sub-protocols that may be used as Array or single string

The second optional parameter in the WebSocket constructor is `protocols` passed in headers as `Sec-WebSocket-Protocol`. This can be either a single protocol string or an array of protocol strings. They indicate sub-protocols so a single server can implement multiple WebSocket sub-protocols. If nothing is passed, an empty string is assumed. If subprotocols are supplied and the server does not accept any of these subprotocols, the connection will not be established. In Chapter 4 you'll build a subprotocol for the STOMP protocol and learn how to utilize that over WebSocket.

If there is an attempt to initiate a WebSocket connection while using HTTPS at the origin website, but using the non-TLS protocol method of `ws://` a `SECURITY_ERR` will be thrown. In addition you'll receive the same error if attempting to connect to a WebSocket server over a port the user agent blocks access (typically 80 and 443 are always allowed).

Registered Protocols
: In the spec for WebSocket RFC 6455, section 11.5 defines the Subprotocol Name Registry for IANA maintained registrations.

Open Protocols
: In addition, you may use open protocols that are unregistered such as Extensible Messaging and Presence Protocol (XMPP) or Simple Text Oriented Message Protocol (STOMP), and various others.

Custom
: You are free to design any protocol you like, as long as your server and client both support it. Recommended is to use names that contain the ASCII version of the domain name of the subprotocol's originator. "chat.acme.com"

Stock Example UI

The example you'll build out relies on some static data to make life easier. Our server will have a list of stock symbols with pre-defined values and randomize the price changes across a spectrum of small positive/negative values.

In order to show a cleaner looking UI and ease the CSS modification process we'll utilize Twitter's Bootstrap and jQuery. In the following code snippet you can copy and paste the contents into your `client.html` file.

```
<!DOCTYPE html>
<html lang="en"><head>
<title>Stock Chart over WebSocket</title>

        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet"
                href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
        <link rel="stylesheet"
                href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
        <script
                src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
        <script language="text/javascript">
        // code from chapter goes here
        </script>
</head>
<body lang="en">
        <div class="vertical-center">
                <div class="container">

                        <h1>Stock Chart over WebSocket</h1>
                        <table class="table" id="stockTable">
                        <thead>
                                <tr>
                                <th>Symbol</th>
                                <th>Price</th>
                                </tr>
                        </thead>
                        <tbody id="stockRows">
                        <tr>
                                <td><h3>AAPL</h3></td>
                                <td id="AAPL">
                                        <h3><span class="label label-default">95.00</span></h3>
                                </td>
                        </tr>
                        <tr>
                                <td><h3>MSFT</h3></td>
                                <td id="MSFT">
                                        <h3><span class="label label-default">50.00</span></h3>
                                </td>
```

```
                        </tr>
                        <tr>
                                <td><h3>AMZN</h3></td>
                                <td id="AMZN">
                                        <h3><span class="label label-default">300.00</span></h3>
                                </td>
                        </tr>
                        <tr>
                                <td><h3>GOOG</h3></td>
                                <td id="GOOG">
                                        <h3><span class="label label-default">550.00</span></h3>
                                </td>
                        </tr>
                        <tr>
                                <td><h3>YHOO</h3></td>
                                <td id="YHOO">
                                        <h3><span class="label label-default">35.00</span></h3>
                                </td>
                        </tr>
                        </tbody>
                        </table>

                </div>
        </div>
    <script
            src="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
    </body></html>
```

Below is what the completed UI will look like after we hook up the server and client.

Figure 2-1 Shows what our stock chart application looks like.

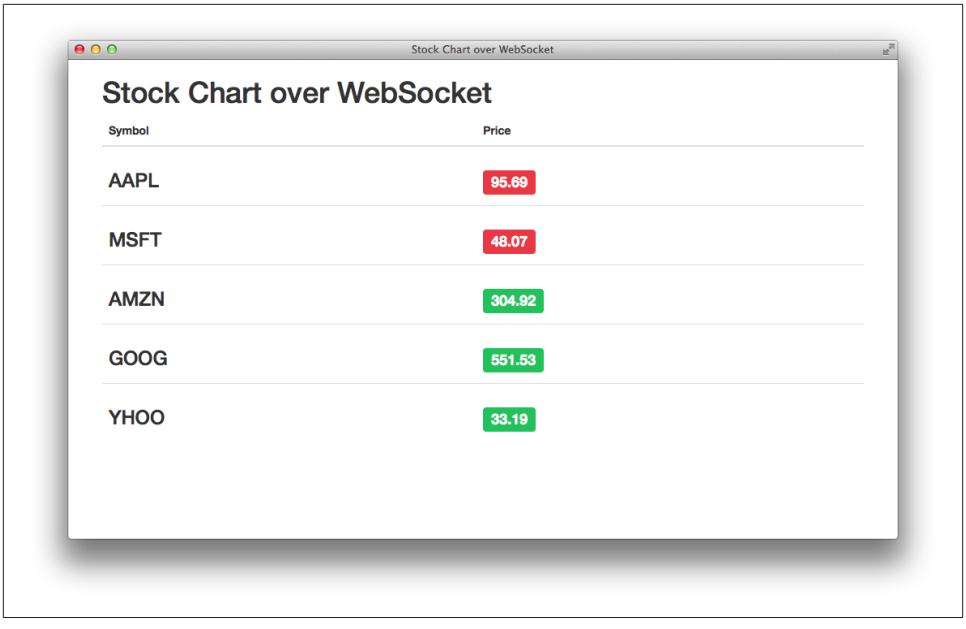


Figure 2-1. Stock Chart over WebSocket

WebSocket Events

The API for WebSockets is all based around events. In this section we'll be talking about the four events that your stock ticker code can listen for. We'll give descriptions of each, describe how to handle situations you'll see in the field, and build our example using what you learn. For our example, you'll need to define a few bits of sample data to pass to the server:

```
var stock_request = {"stocks": ["AAPL", "MSFT", "AMZN", "GOOG", "YHOO"]};

var stocks = {"AAPL": 0,
                        "MSFT": 0,
                        "AMZN": 0,
                        "GOOG": 0,
                        "YHOO": 0};
```

The first structure `stock_request` is passed after the successful connection between client and server and asks that the server keep telling us about the updated pricing on these specific stocks. The second structure `stocks` is a simple associative array which will hold the changing values passed back from the server and then used to modify the text in the table and colors as you'll see.

WebSocket fires four different events which are available from the JavaScript API and defined by the W3C at *http://www.w3.org/TR/websockets/*:

- open
- message
- error
- close

With JavaScript the way to listen for these events to fire, is either the handler `on<event name>`, or the `addEventListener()` method. Your code will provide a callback which will execute every time that event gets fired.

Event: Open

When the WebSocket server responds to the connection request, and the handshake is complete, the open event fires and the connection is established. Once this happens, the server has completed the handshake and is ready to send and receive messages from your client application.

Event Handler for Open.

```
// WebSocket connection established
ws.onopen = function(e) {
        console.log("Connection established");
        ws.send(JSON.stringify(stock_request));
};
```

From within this handler you can send messages to the server, output status to the screen, the connection is ready and available for bidirectional communication. The initial message being sent to the server over WebSocket will be the `stock_request` structure as a JSON string. Our server now knows what stocks we want to get updates on and will send them back to the client in one second intervals.

Event: Message

After you've established a connection to the WebSocket server, it will be available to send messages to (we'll look at that in the section on WebSocket methods), and receive messages. The WebSocket API will prepare complete messages to be processed in the `onmessage` handler.

We will discuss in more detail about the WebSocket protocol in a later chapter, which includes frames and more detail about how data flows back and forth between the server and client. For now, the only thing to remember is, when the server has data, the WebSocket API will call the `onmessage` handler.

```
// UI update function
var changeStockEntry = function(symbol, originalValue, newValue) {
        var valElem = $('#' + symbol + ' span');
        valElem.html(newValue.toFixed(2));
```

```
        if(newValue < originalValue) {
                valElem.addClass('label-danger');
                valElem.removeClass('label-success');
        } else if(newValue > originalValue) {
                valElem.addClass('label-success');
                valElem.removeClass('label-danger');
        }
}

// WebSocket message handler
ws.onmessage = function(e) {
        var stocksData = JSON.parse(e.data);
        for(var symbol in stocksData) {
                if(stocksData.hasOwnProperty(symbol)) {
                        changeStockEntry(symbol, stocksData[symbol], stocks[symbol]);
                        stocks[symbol] = stocksData[symbol];
                }
        }
};
```

We can see from this short snippet, that the handler is receiving a message from the server via a callback `onmessage`. When querying for data, the attribute `data` will contain updated stock values. In the code snippet above you are:

1. Parsing the JSON response within `e.data`
2. Iterating over the associative array
3. Ensuring the key exists in the array
4. Calling our UI update fragment
5. Assigning the new stock values to our local array

We're passing around regular strings here, but WebSocket has full support for sending text and binary data.

Event: Error

When a failure happens for any reason at all, the handler you've attached to the `error` event will get fired. When an error occurs, it can be assumed that the WebSocket connection will close and a close event will fire. In some instances since the close event happens shortly after an error, the `code` and `reason` attributes can give you some indication as to what happened. Here's a sample of how to handle the error case, and possibly reconnect to the WebSocket server as well.

```
ws.onerror = function(e) {
        console.log("WebSocket failure, error", e);
        handleErrors(e);
};
```

Event: PING / PONG

The WebSocket protocol calls out two different frame types which are PING and PONG. There is no ability from the WebSocket JavaScript client API to send a PING frame to the server. PING frames get sent out by the server only, and browser implementations should send back PONG frames in response.

Event: Close

The `close` event will fire when the WebSocket connection closes, and the callback `onerror` will be executed. You can manually trigger calling the `onclose` event by executing the `close()` method on a WebSocket object which will terminate the connection with the server. Once the connection is closed, communication between client and server will not continue. In our example below, zeroing out the `stocks` array upon a `close` event being fired to show cleaning up resources.

```
ws.onclose = function(e) {
        console.log(e.reason + " " + e.code);
        for(var symbol in stocks) {
                if(stocks.hasOwnProperty(symbol)) {
                        stocks[symbol] = 0;
                }
        }
}

ws.close(1000, 'WebSocket connection closed')
```

As was mentioned briefly in the `error` event section, there are two attributes `code` and `reason` which are conveyed by the server and could indicate an error condition to be handled and/or reason for the close event (other than normal expectation). Either side may terminate the connection via the `close()` method on the WebSocket object as we have shown in the code above. Your code can also use the boolean attribute `wasClean` to find out if the termination was clean, or the result of an error state.

The `readyState` value will move from closing: `2` to closed: `3`. Now let's move on to talking about the methods available to our WebSocket object.

WebSocket Methods

They've kept it pretty simple with WebSocket methods, there are only two: `send()` and `close()`.

Method: Send

When your connection has been established, you're ready to start sending (and receiving) messages to/from the WebSocket server. The client application can specify what

type of data is being passed in and will accept several including `string` and `binary` values. As was shown above, in the code for our client is sending a JSON string of listed stocks.

```
ws.send(JSON.stringify(stock_request));
```

Of course, performing this send just anywhere, won't be appropriate. As we've discussed, WebSockets are event-driven, so we will need to ensure that the connection is open and ready to receive messages. There are two main ways of achieving this.

Perform your send from within the `onopen` event.

```
var ws = new WebSocket("ws://localhost:8181");
ws.onopen = function(e) {
        ws.send(JSON.stringify(stock_request));
}
```

Or we can check the `readyState` attribute to ensure that the WebSocket object is ready to receive messages.

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

In later chapters we'll talk about Blob objects, ArrayBuffers and many other facets of sending data over WebSockets including layering.

Method: Close

Closing the WebSocket connection or terminating an attempt at connection is done via the `close()` method. After this method is called no more data can be sent or received from this connection. And calling it multiple times has no effect.

Here's an example of calling the close() method without arguments.

```
// Close WebSocket connection
ws.close();
```

Optionally you can pass a numeric code and a human-readable reason through the `close()` method. This will give some indication to the server as to why the connection was closed on the client end. The following code shows how you would pass those values, note that if you don't pass a `code`, the status 1000 is assumed which means `CLOSE_NORMAL`.

```
// Close the WebSocket connection with reason.
ws.close(1000, "Goodbye, World!");
```

Here's a list of status codes you can use in the WebSocket close method:

Table 2-2. WebSocket close codes

Status code	Name	Description
0-999		Reserved and not used.
1000	CLOSE_NORMAL	Normal closure; the connection successfully completed
1001	CLOSE_GOING_AWAY	The endpoint is going away, either because of a server failure or because the browser is navigating away from the page that opened the connection
1002	CLOSE_PROTOCOL_ERROR	The endpoint is terminating the connection due to a protocol error
1003	CLOSE_UNSUPPORTED	The connection is being terminated because the endpoint received data of a type it cannot accept
1004	CLOSE_TOO_LARGE	The endpoint is terminating the connection because a data frame was received that is too large
1005	CLOSE_NO_STATUS	**Reserved.** Indicates that no status code was provided even though one was expected.
1006	CLOSE_ABNORMAL	**Reserved.** Used to indicate that a connection was closed abnormally
1007-1999		Reserved for future use by the WebSocket standard*
2000-2999		Reserved for use by WebSocket extensions.
3000-3999		Available for use by libraries and frameworks. **May not** be used by applications
4000-4999		Available for use by applications

WebSocket Attributes

When the event for `open` is fired the WebSocket object can have several possible attributes which can be read in your client applications. In this section we'll discuss the attributes, and the best-practices for using them in your client code.

Attribute: readyState

The state of the WebSocket connection can be checked via the read-only WebSocket Object Attribute `readyState`. The value of `readyState` will change and is a good idea to check before committing to send any data to the server.

The following table shows the different values you will see reflected in the `readyState` attribute.

Table 2-3. readyState Constants

Attribute Name	Attribute Value	Description
WebSocket.CONNECTING	0	The connection is not yet open
WebSocket.OPEN	1	The connection is open and ready to communicate
WebSocket.CLOSING	2	The connection is in the process of closing
WebSocket.CLOSED	3	The connection is closed or couldn't be opened

Each of these values can be checked at different points for debug, and for understanding the lifecycle of your connection to the server.

Attribute: bufferedAmount

Also included with the attributes is the amount of data buffered for sending to the server. While this is most used when sending binary data, as the data size tends to be much larger, the browser will take care of properly queueing the data for send. Since we're dealing only with the client code at this point (the next chapter will deal with the protocol), much of the behind the scenes is hidden from our view. Usage of the `bufferedAmount` attribute can be useful for ensuring all data is sent before closing a connection, or performing your own throttling on the client side.

Attribute: protocol

Reflecting back to the constructor for WebSocket, the optional `protocol` argument allows you to send one or many sub-protocols that the client is asking for. The server decides which protocol it chooses, and this will be reflected in this attribute for the WebSocket connection. The handshake when completed should contain a selection from one that was sent by the client, or empty if none were chosen or offered.

Stock Example Server

Now that you have a working client that will connect to a WebSocket server to retrieve stock quotes, it's time to show what the server will look like.

```
var WebSocketServer = require('ws').Server,
        wss = new WebSocketServer({port: 8181});

var stocks = {
        "AAPL": 95.0,
        "MSFT": 50.0,
        "AMZN": 300.0,
        "GOOG": 550.0,
        "YHOO": 35.0
}

function randomInterval(min, max) {
        return Math.floor(Math.random()*(max-min+1)+min);
}

var stockUpdater;
var randomStockUpdater = function() {
        for (var symbol in stocks) {
                if(stocks.hasOwnProperty(symbol)) {
                        var randomizedChange = randomInterval(-150, 150);
                        var floatChange = randomizedChange / 100;
```

```
                        stocks[symbol] += floatChange;
                }
        }
        var randomMSTime = randomInterval(500, 2500);
        stockUpdater = setTimeout(function() {
                randomStockUpdater();
        }, randomMSTime)

}

randomStockUpdater();

wss.on('connection', function(ws) {
        var clientStockUpdater;
        var sendStockUpdates = function(ws) {
                if(ws.readyState == 1) {
                        var stocksObj = {};

                        for(var i=0; i<clientStocks.length; i++) {
                                symbol = clientStocks[i];
                                stocksObj[symbol] = stocks[symbol];
                        }

                        ws.send(JSON.stringify(stocksObj));
                }
        }
        clientStockUpdater = setInterval(function() {
                sendStockUpdates(ws);
        }, 1000);

        var clientStocks = [];

        ws.on('message', function(message) {
            var stock_request = JSON.parse(message);
            clientStocks = stock_request['stocks'];
            sendStockUpdates(ws);
        });

        ws.on('close', function() {
                if(typeof clientStockUpdater !== 'undefined') {
                        clearInterval(clientStockUpdater);
                }
        });
});
```

The server code after execution will run a function a variable amount of time between 0.5s and 2.5s and update the stock prices. It does this to appear as random as possible in a book example without requiring code to go out and retrieve real stock prices (see Chapter 4 for that). Our frontend is expecting to receive a static list of 5 stocks retrieved from the server. Simple. The server after receiving the connection event from the client will setup a function to run every second and send back our list of 5 stocks with

randomized prices once a second. The server can accept requests for different stocks as long as those stock symbols and a starting price are added to the `stocks` JavaScript object defined in the server.

Testing for WebSocket Support

It should come as no surprise, if you've coded anything for the web over the years, that browsers do not always have support for the latest technology. Due to the lack of support for the API in some older browsers, it is important to check for compatibility before utilizing the WebSocket API. In Chapter 5 we'll present some alternatives if the client browsers used by your community of users doesn't offer support for the WebSocket API. For now, here is a quick way to check if the API is supported on the client:

```
if (window.WebSocket) {
        console.log("WebSocket: supported");
        // ... code here for doing WebSocket stuff
} else {
        console.log("WebSocket: unsupported");
        // ... fallback mode, or error back to user
}
```

Summary

In this chapter we went over essential details of the WebSocket API and how to use each of them within your client application. We discussed the API's events, messages, attributes and methods along with some sample code along the way.

In Chapter 3, you'll write a bi-directional chat application example, learning how to pass messages back and forth with multiple connected clients.

CHAPTER 3
Bi-directional Chat

Our first full-fledged example is to build a bi-directional chat using WebSockets. The end result will be a server that accepts WebSocket connections and messages for our "chat room" and fans the messages out to connected clients. The WebSocket protocol itself is simple, so to write our chat application you will manage the collection of message data in an array and hold the socket and unique UUID for the client in locally scoped variables.

Long polling

The process of long polling is keeping a connection to the server alive without having data immediately sent back to the client. Long polling (or long held HTTP request) sends a server request which is kept open until it has data, and the client will receive it and re-open a connection soon after receiving data from the server. This in effect, allows for a persistent connection with the server to send data back and forth.

In practice, there are two common techniques for achieving this. An XMLHttpRequest is initiated and then held open waiting for a response from the server. Once this is received, another request is made to the server and held open awaiting more data. The other involves writing out custom script tags possibly pointing to a different domain (cross-domain requests are not allowed with the first method). Requests are then handled in a similar manner and re-opened in the typical long polling fashion.

Long polling is the most common way of implementing this type of application on the web today. What you will see in this chapter, is a much simpler, and efficient method of implementation. In subsequent chapters you will tackle the compatibility issue of older browsers that may not yet support WebSockets.

Writing a basic chat

Our first chapter showed a basic server that accepted a Websocket connection and sent any received message from a connected client to the console. Let's take another look at that code, and add features that will be required for implementing our bi-directional chat.

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({port: 8181});

wss.on('connection', function(socket) {
        console.log('client connected');
        socket.on('message', function(message) {
        console.log(message);
    });
});
```

The WebSocketServer provided by the popular `ws` Node module gets initialized and starts listening on port 8181. You can follow this by listening for a client `connection` event and the subsequent `message` events that follow. Our `connection` event accepts a callback function where you pass a `socket` object to be used for listening to messages after a successful connection has occurred.

This works well to show off a simple connection for our purposes, and now we're going to build on top of that by tracking the clients that connect, and sending those messages out to all other connected clients.

The Websocket protocol does not provide any of this functionality by default, the responsibility for creation and tracking is ours. In later chapters, you will dive into libraries such as Socket.IO which extend the functionality of Websockets and provides a richer API and backwards compatibility with older browsers.

Figure 3-1 Shows what our chat application looks like currently.

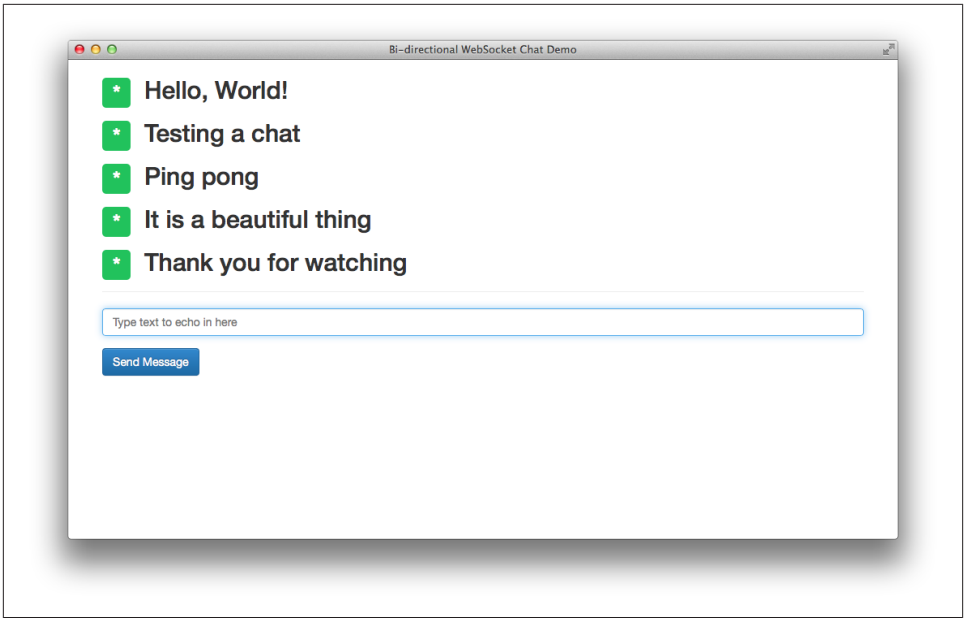


Figure 3-1. Your First WebSocket Chat

Building on the code from our first chapter let's import a node module for generating a UUID. First things first, we'll use npm to install `node-uuid`.

```
% npm install node-uuid
```

```
var uuid = require('node-uuid');
```

A UUID is utilized to identify each client that has connected to the server and add them to a collection. A UUID will allow you to target messages from specific users, operate on those users, and provide data targeted for those users as needed.

Universally Unique IDentifier

A UUID is a standardized identifier commonly used in building distributed systems which can be assumed "practically unique". Generally speaking, you won't run into collisions but it isn't guaranteed. Therefore for our simple chat application we should be fine using this as our identifier.

Next, you'll enhance the connection to the server with identification and logging.

```
var clients = [];

wss.on('connection', function(ws) {
  var client_uuid = uuid.v4();
```

```
    clients.push({"id": client_uuid, "ws": ws});
    console.log('client [%s] connected', client_uuid);
```

Assigning the result of the `uuid.v4` function to the `client_uuid` variable allows you to reference it later when identifying message sends, and any close event. A simple metadata object in the form of JSON contains the client UUID along with the WebSocket object.

When the server receives a message from the client, it will iterate over all known connected clients using the `clients` collection, and send back a JSON object containing the `message` and `id` of the message sender. You may notice that this will also send back the message to the client that initiated, and this simplicity is by design. On the frontend client we don't update the list of messages unless it is returned by the server.

```
ws.on('message', function(message) {
  for(var i=0; i<clients.length; i++) {
      var clientSocket = clients[i].ws;
      console.log('client [%s]: %s', clients[i].id, message);
      clientSocket.send(JSON.stringify({
          "id": client_uuid,
          "message": message
      }));
  }
});
```

The WebSocket server now receives `message` events from any of the connected clients. After receiving the message, it iterates through the connected clients and sends a JSON string which includes the unique identifier for the client who sent the message, and the message itself. Every connected client will receive this JSON string and can show this to the end-user.

A server must handle error states gracefully, and still continue to work. In the event of a WebSocket close event, you haven't yet defined what to do in the case of a WebSocket close event, but there is something missing that needs to be addressed in the `message` event code. The collection of connected clients needs to account for the possibility that the client has gone away, and ensure that before you send a `message`, there is still an open WebSocket connection.

The new code is as follows:

```
ws.on('message', function(message) {
  for(var i=0; i<clients.length; i++) {
      var clientSocket = clients[i].ws;
      if(clientSocket.readyState === WebSocket.OPEN) {
        console.log('client [%s]: %s', clients[i].id, message);
        clientSocket.send(JSON.stringify({
            "id": client_uuid,
            "message": message
        }));
      }
```

```
  }
});
```

You now have a server that will accept connections from WebSocket clients, and will rebroadcast received messages to all connected clients. The final thing to handle is the `close` event.

```
ws.on('close', function() {
  for(var i=0; i<clients.length; i++) {
      if(clients[i].id == client_uuid) {
          console.log('client [%s] disconnected', client_uuid);
          clients.splice(i, 1);
      }
  }
});
```

Our server listens for a `close` event, and upon receiving it for this client, iterates through the collection and removes the client. Couple the above with our check of the `readyState` flag for our WebSocket object and you've got a server that will work with our new client.

Later in this chapter you will broadcast the state of disconnected and connected clients along with our chat messages.

WebSocket Client

Our simple echo client from the first chapter can be used as a jumping off point for our chat webapp. All the connection handling will work as specified, and you'll need to listen for the `onmessage` event which was being ignored previously:

```
ws.onmessage = function(e) {
  var data = JSON.parse(e.data);
  var messages = document.getElementById('messages');
  var message = document.createElement("li");
  message.innerHTML = data.message;
  messages.appendChild(message);
}
```

The client receives a message from the server in the form of a JSON object. Utilizing JavaScript's built-in parsing function returns an object that can be used to extract the message field. Let's add a simple unordered list above the form so messages can be appended using the DOM methods shown in the function. Above the form element you add:

```
<ul id="messages"></ul>
```

Messages will be appended to the list using the DOM method `appendChild`, and shown in every connected client. So far we have only scratched the surface of functionality that shows off the seamless messaging provided by the WebSocket protocol. In the next section you will implement a method of identifying clients by a nickname.

Client Identity

The Websocket specification has been left relatively simplistic in terms of implementation and lacks some of the features seen in alternatives. In your code so far, you have already gone a long way toward identifying each client individually, now you can add nickname identity to the client and server code.

```
var nickname = client_uuid.substr(0, 8);
clients.push({"id": client_uuid, "ws": ws, "nickname": nickname});
```

The server gets modified to add the field `nickname` to a locally stored JSON object for this client. In order to uniquely identify a connected client who hasn't identified his nickname choice, you can utilize the first 8 characters of the UUID and assign that to the nickname variable. All of this will be sent back over an open WebSocket connection between server and all of its connected clients.

You will utilize a convention used with Internet Relay Chat clients (IRC) and accept `/nick new_nick` as the command for changing the client nickname from the random string.

```
if(message.indexOf('/nick') == 0) {
  var nickname_array = message.split(' ')
  if(nickname_array.length >= 2) {
    var old_nickname = nickname;
    nickname = nickname_array[1];
    for(var i=0; i<clients.length; i++) {
      var clientSocket = clients[i].ws;
      var nickname_message = "Client " + old_nickname + " changed to " + nickname;
      clientSocket.send(JSON.stringify({
        "id": client_uuid,
        "nickname": nickname,
        "message": nickname_message
      }));
    }
  }
}
```

The above code checks for the existence of the `/nick` command followed by a string of characters that represents a nickname. Update your nickname variable, and you can build a notification string to send out to all connected clients over the existing open connection.

The clients don't yet know about this new field, as the JSON we originally sent over only included `id`, and `message`. So add it:

```
clientSocket.send(JSON.stringify({
    "id": client_uuid,
    "nickname": nickname,
    "message": message
}));
```

The `appendLog` function within the client frontend needs to be modified to support the addition of the `nickname` variable.

```
function appendLog(nickname, message) {
  var messages = document.getElementById('messages');
  var messageElem = document.createElement("li");
  var message_text = "[" + nickname + "] - " + message;
  messageElem.innerHTML = message_text;
  messages.appendChild(messageElem);
}
```

Figure 3-2 Shows our chat application with the addition of identity

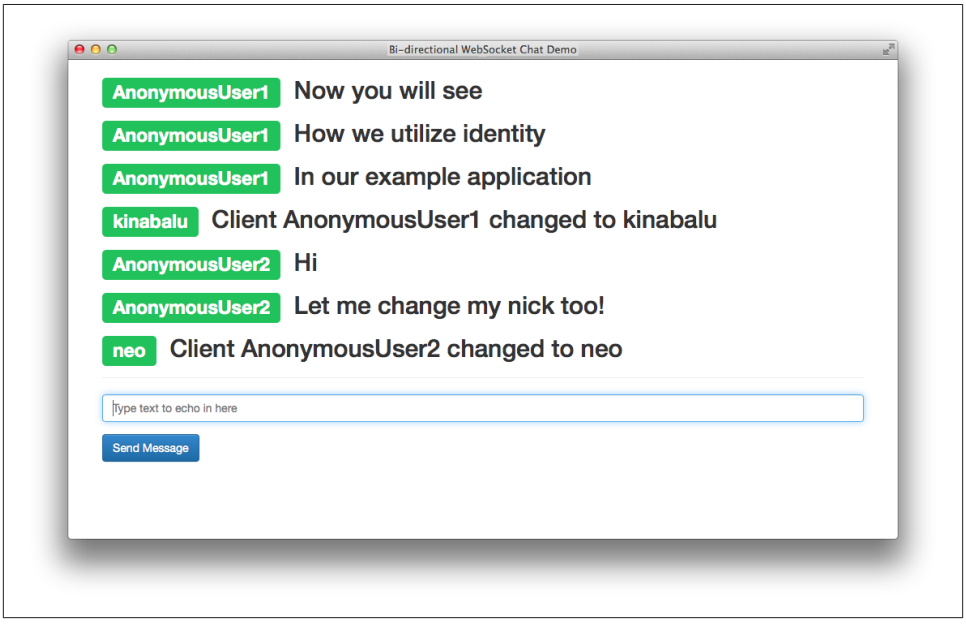


Figure 3-2. Identity-Enabled Chat

Our new function signature includes `nickname` along with `message`, and you can preface every message now with the client nickname. At the clients request, you can see a nickname preceding messages rather than a random string of characters before each message.

Events and notifications

If you were in the middle of a conversation and another person magically appeared in front of you and started talking, that would be odd. To alleviate this, you can add notification of connection or disconnection and send that back to all connected clients.

In your code now, you have several instances where you've gone through the trouble of iterating over all connected clients, checking the `readyState` of the socket, and sending a similar JSON encoded string with varying values. For good measure, we'll extract this into a generic function, and call it from several places in our code instead:

```
function wsSend(type, client_uuid, nickname, message) {
  for(var i=0; i<clients.length; i++) {
    var clientSocket = clients[i].ws;
    if(clientSocket.readyState === WebSocket.OPEN) {
      clientSocket.send(JSON.stringify({
        "type": type,
        "id": client_uuid,
        "nickname": nickname,
        "message": message
      }));
    }
  }
}
```

With the generic function above, you can send notifications to all connected clients, handle the connection state, and encode the string as the client expects it, like so:

```
wss.on('connection', function(ws) {
    ...
      wsSend("message", client_uuid, nickname, message);
    ...
});
```

Sending messages to all clients post connection is now simple. Connection messages, disconnection messages, any notification you need is now handled with your new function.

The Server

Here is the complete code for the server.

```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server,
    wss = new WebSocketServer({port: 8181});
var uuid = require('node-uuid');

var clients = [];

function wsSend(type, client_uuid, nickname, message) {
  for(var i=0; i<clients.length; i++) {
    var clientSocket = clients[i].ws;
    if(clientSocket.readyState === WebSocket.OPEN) {
      clientSocket.send(JSON.stringify({
        "type": type,
        "id": client_uuid,
        "nickname": nickname,
```

```
        "message": message
      }));
    }
  }
}

var clientIndex = 1;

wss.on('connection', function(ws) {
  var client_uuid = uuid.v4();
  var nickname = "AnonymousUser"+clientIndex;
  clientIndex+=1;
  clients.push({"id": client_uuid, "ws": ws, "nickname": nickname});
  console.log('client [%s] connected', client_uuid);

  var connect_message = nickname + " has connected";
  wsSend("notification", client_uuid, nickname, connect_message);

  ws.on('message', function(message) {
    if(message.indexOf('/nick') === 0) {
      var nickname_array = message.split(' ');
      if(nickname_array.length >= 2) {
        var old_nickname = nickname;
        nickname = nickname_array[1];
        var nickname_message = "Client " + old_nickname + " changed to " + nickname;
        wsSend("nick_update", client_uuid, nickname, nickname_message);
      }
    } else {
      wsSend("message", client_uuid, nickname, message);
    }
  });

  var closeSocket = function(customMessage) {
    for(var i=0; i<clients.length; i++) {
        if(clients[i].id == client_uuid) {
            var disconnect_message;
            if(customMessage) {
                disconnect_message = customMessage;
            } else {
                disconnect_message = nickname + " has disconnected";
            }
          wsSend("notification", client_uuid, nickname, disconnect_message);
          clients.splice(i, 1);
        }
    }
  }
  ws.on('close', function() {
      closeSocket();
  });

  process.on('SIGINT', function() {
      console.log("Closing things");
```

```
        closeSocket('Server has disconnected');
        process.exit();
    });
});
```

The Client

Here is the complete code for the client.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Bi-directional WebSocket Chat Demo</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
  href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<link rel="stylesheet"
  href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
<script
  src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

    <script>
            var ws = new WebSocket("ws://localhost:8181");
            var nickname = "";
            ws.onopen = function(e) {
              console.log('Connection to server opened');
            }
            function appendLog(type, nickname, message) {
              var messages = document.getElementById('messages');
              var messageElem = document.createElement("li");
              var preface_label;
              if(type==='notification') {
                  preface_label = "<span class=\"label label-info\">*</span>";
              } else if(type=='nick_update') {
                  preface_label = "<span class=\"label label-warning\">*</span>";
              } else {
                  preface_label = "<span class=\"label label-success\">" + nickname + "</span>";
              }
              var message_text = "<h2>" + preface_label + "  " + message + "</h2>";
              messageElem.innerHTML = message_text;
              messages.appendChild(messageElem);
            }

            ws.onmessage = function(e) {
              var data = JSON.parse(e.data);
              nickname = data.nickname;
              appendLog(data.type, data.nickname, data.message);
              console.log("ID: [%s] = %s", data.id, data.message);
            }
            ws.onclose = function(e) {
              appendLog("Connection closed");
```

```
            console.log("Connection closed");
          }
          function sendMessage() {
             var messageField = document.getElementById('message');
             if(ws.readyState === WebSocket.OPEN) {
                 ws.send(messageField.value);
             }
             messageField.value = '';
             messageField.focus();
          }
          function disconnect() {
            ws.close();
          }
      </script>

  </head>
  <body lang="en">
      <div class="vertical-center">
      <div class="container">
      <ul id="messages" class="list-unstyled">

      </ul>
      <hr />
      <form role="form" id="chat_form" onsubmit="sendMessage(); return false;">
          <div class="form-group">
          <input class="form-control" type="text" id="message" name="message" id="message"
            placeholder="Type text to echo in here" value="" autofocus/>
          </div>
  <button type="button" id="send" class="btn btn-primary"
    onclick="sendMessage();">Send Message</button>
    </form>
    </div>
    </div>
  <script
    src="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
  </body>
  </html>
```

Figure 3-3 Shows our chat application with the addition of notifications

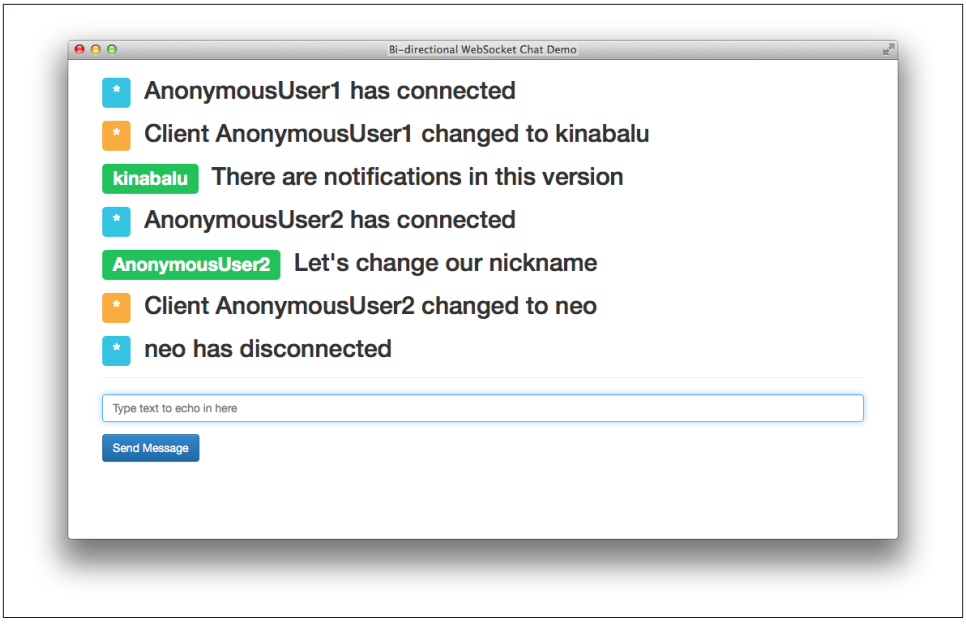


Figure 3-3. Notification-Enabled Chat

Summary

In this chapter you've built out a complete chat client and server using the WebSocket protocol. You steadily built from a very simplistic chat application, to something more robust with only the WebSocket API as our technology of choice. Effective and optimized experiences between internal applications, live chat, layering other protocols over HTTP are all possibilities and native with WebSockets.

All of the above is of course possible with other technology, and as you've probably learned before, there's more than one way to solve a problem. Comet and AJAX are both battle tested to deliver similar experiences to the end-user as provided by WebSockets. Using them however, is rife with inefficiency, latency, unnecessary requests and unneeded connections to the server. Only WebSocket removes the overhead inherent in the previous methods and gives you a socket, full-duplex, bidirectional, and ready to rock 'n roll.

In the next chapter you'll take a look at a popular protocol for layering on top of WebSocket to provide transport without the overhead of HTTP.

CHAPTER 4
STOMP over WebSocket

In previous chapters you've built simple applications using the WebSocket API both on the server-side and on the client. You built a multi-client chat application with WebSocket as the communication layer. In Chapter 2 we briefly discussed the idea of subprotocols with WebSockets. Now you'll take everything learned thus far and layer another protocol on top of WebSocket.

STOMP is an acronym for Simple Text Oriented Messaging Protocol[*https://stomp.github.io/*]. It defines a simple HTTP-like protocol for interacting with any STOMP message broker. Any STOMP client can interact with the message broker and be interoperable among languages and platforms.

In this chapter we'll create a client and server that communicates using STOMP over WebSocket that speaks the protocol of STOMP on top of WebSocket rather than TCP. You will learn how to connect to RabbitMQ using the Web-Stomp plugin. The RabbitMQ Web-Stomp plugin uses WebSocket as it's underlying wire protocol.

As in previous chapters, you'll create a new project folder for Chapter 4 examples with the abbreviated chapter name named `ch4`. The examples in this chapter will again be a stock ticker, and will utilize messaging to subscribe for stock updates. In addition, there are two examples in this chapter tackled so create a subdirectory named `proxy`. You'll be creating several files in this example to build a real working table of stock prices powered by STOMP over WebSocket. Here are the files that will be used:

- `client.html` - the frontend codebase; as before copy the template used in Chapter 1
- `server.js` - our WebSocket proxy that talks to RabbitMQ using AMQP while listening for STOMP over WebSocket with client
- `stomp_helper.js` - a convenience library you'll build for sending and receiving STOMP requests

- `daemon.js` - a daemon that pulls stocks from Yahoo Finance using YQL and pulls and pushes messages to RabbitMQ

Implementing the STOMP Protocol

The STOMP protocol is a simple text protocol which is similar to the HTTP convention of an uppercase command such as `CONNECT`, followed by a list of header key/value pairs, and then optional content which in the case of STOMP is null-terminated. It is also possible and highly recommended to pass `content-length` as a parameter to any commands and the server will use that value instead as the length of passed content.

Getting connected

As you saw in Chapter 2, the native browser API for connecting to a WebSocket server takes two parameters: `URL` and `protocol`. Of those two parameters only the URL is required, but now you will be making use of the second. If you research for registered protocols in the WebSocket Subprotocol Name Registry you'll find an entry for STOMP 1.0 which uses the identifier `v10.stomp`. As we'll discuss in Chapter 8, you are not required to use a registered subprotocol with WebSocket. The subprotocol does need to be supported by the client and the server. In your client then, you'll open a connection the following way:

```
var ws;

var connect = function() {
    if(!ws || ws.readyState !== 1) {
        ws = new WebSocket("ws://localhost:8181", "v10.stomp");
        ws.addEventListener('message', onMessageHandler);
        ws.addEventListener('open', onOpenHandler);
        ws.addEventListener('close', onCloseHandler);
    }
}

connect();
```

As with the previous examples, you open a connection to a WebSocket server on `port 8181`, but in addition here you'll pass a second parameter in the constructor which can either be a string, or an array of strings identifying requested subprotocols from the server. You'll notice that in addition there is a `connect` function now which adds the event listeners for `open`, `message`, and `close` using the `addEventListener` method. This is the essential method of connecting. If you need to reconnect upon a lost connection, the event handlers will not automatically reattach if using the `ws.on<eventname>` method.

After opening the WebSocket connection, an `open` event is fired, and you can officially send and receive messages from the server. If you reference the STOMP 1.0 protocol doc, the following will be shown as the method of initial connection to a STOMP-capable server:

```
CONNECT
login: <username>
passcode: <passcode>

^@
```

For the purposes of the example in this chapter, you'll use `websockets` as username and `rabbitmq` as password for all authentication with the STOMP server and RabbitMQ. So within your code, you can pass the following with the WebSocket `send` function.

```
var frame = "CONNECT\n"
            + "login: websockets\n";
            + "passcode: rabbitmq\n";
            + "nickname: anonymous\n";
            + "\n\n\0";
ws.send(frame);
```

You can see in the STOMP 1.0 protocol doc every frame sent is ended with the null terminator `^@`, or if the `content-length` header is passed, this will be used instead. Because of the simplicity of WebSocket, we're carefully mapping STOMP frames on top of WebSocket frames in these examples. If the server has accepted the connection and authentication information it will pass back the following to the client which will include a `session-id` to be used in later calls to the server.

```
CONNECTED
session: <session-id>

^@
```

In the overview of the chapter the `stomp_helper.js` was mentioned, and before we get to the server code, let's review the library which will assist in sending and receiving STOMP-compatible frames.

Example 4-1. STOMP Library Code

```
(function(exports){
    exports.process_frame = function(data) {
        var lines = data.split("\n");
        var frame = {};
        frame['headers'] = {};
        if(lines.length>1) {
            frame['command'] = lines[0];
            var x = 1;
            while(lines[x].length>0) {
                var header_split = lines[x].split(':');
                var key = header_split[0].trim();
```

```
            var val = header_split[1].trim();
            frame['headers'][key] = val;
            x += 1;
        }
        frame['content'] = lines
                           .splice(x + 1, lines.length - x)
                           .join("\n");

        frame['content'] = frame['content']
                           .substring(0, frame['content'].length - 1);
    }
    return frame;
};

exports.send_frame = function(ws, frame) {
    var data = frame['command'] + "\n";
    var header_content = "";
    for(var key in frame['headers']) {
        if(frame['headers'].hasOwnProperty(key)) {
            header_content += key
                              + ": "
                              + frame['headers'][key]
                              + "\n";
        }
    }
    data += header_content;
    data += "\n\n";
    data += frame['content'];
    data += "\n\0";
    ws.send(data);
};

exports.send_error = function(ws, message, detail) {
    headers = {};
    if(message) headers['message'] = message;
    else headers['message'] = "No error message given";

    exports.send_frame(ws, {
        "command": "ERROR",
        "headers": headers,
        "content": detail
    });
};

})(typeof exports === 'undefined'? this['Stomp']={}: exports);
```

The ceremonial items preceding and following the functions in this library are done to allow this to be used within the browser, and on the server-side with Node.js in a require statement.

The first function to describe is `process_frame` which takes a STOMP frame as a parameter `data` and creates a JavaScript object containing everything parsed out for use within your application. As described in the table below, it splits out the command, all the headers, and any content within the frame and returns an object fully parsed.

Table 4-1. Table shows the JavaScript object structure

Key	Description
command	STOMP command passed by the frame
headers	A JavaScript object with key/values for the passed-in headers
content	Any content sent in the frame which was null-terminated or adhering to the content-length header

Next up and equally as important is the `send_frame` function which accepts a WebSocket object, and a STOMP frame in the form of a JavaScript object exactly as you send back from the `process_frame` function. The `send_frame` function takes each of the values passed in and creates a valid STOMP frame and sends it off over the passed in `WebSocket` parameter.

The remaining function is `send_error`, which takes the following parameters:

Table 4-2. Table showing the parameters accepted for our `send_error` call

Name	Description
WebSocket	The active WebSocket connection
message	Error message explaining what went wrong
detail	Optional detail message passed in the body

You'll be able to use the above set of functions to send and receive STOMP frames without any string parsing within your client or server code.

Connection via the server

On the server side, upon receiving a `connection` event, our initial task to get connected will be to parse what is received in the message frame (using the `stomp_helper.js` library, and send back a `CONNECTED` command or an `ERROR` if it failed.

```
wss.on('connection', function(ws) {
    var sessionid = uuid.v4();

    ws.on('message', function(message) {
        var frame = Stomp.process_frame(message);
        var headers = frame['headers'];
        switch(frame['command']) {
            case "CONNECT":
                Stomp.send_frame(ws, {
                    command: "CONNECTED",
                    headers: {
```

```
                        session: sessionid,
                    },
                    content: ""
                });
                break;
            default:
                Stomp.send_error(ws, "No valid command frame");
                break;
        }
    });
    ...
});
```

As you've seen in previous examples, the connection event is received and work begins. There exists an extra layer thanks to STOMP, which is handled somewhat by our library. After assigning a `sessionid` to a UUID, and upon receiving a message event from the client, you'll run it through the `process_frame` function to get a JavaScript object representing the received frame. To process whatever command was sent, the program uses a case statement, and upon receiving the `CONNECT` command, you send back a STOMP frame letting them know the connection was received and is accepted along with the `sessionid` for this session.

Let's take a quick look at Figure 4-1 which shows a completed connection event:

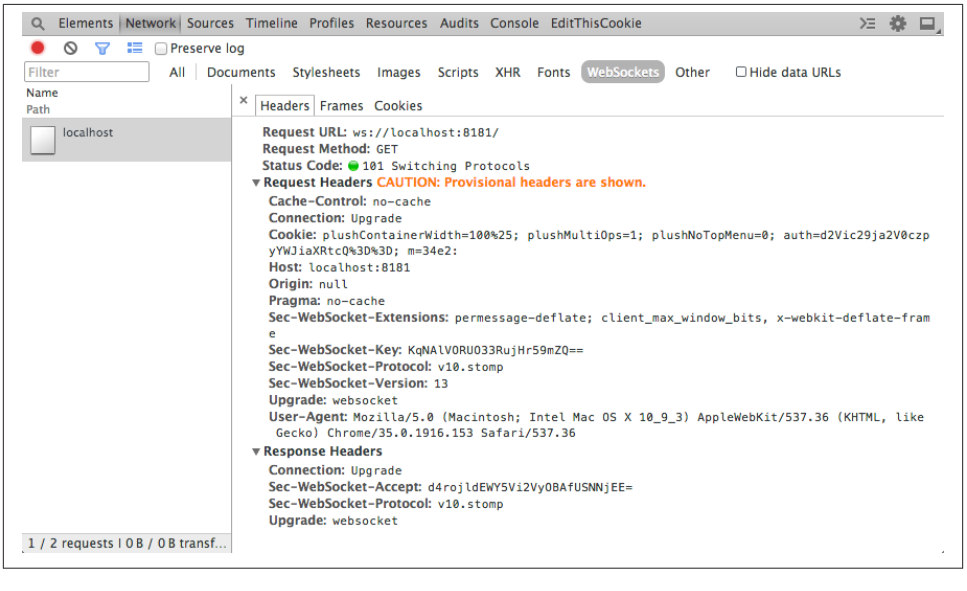


Figure 4-1. Successful WebSocket connection with subprotocol

Looking at the above screen grab, you'll see a new header for the HTTP request and response: `Sec-WebSocket-Protocol`. In Chapter 8 you can read a more in-depth dis-

cussion about the various headers and dive deep into the protocol nitty-gritty. Here in our stocks example, the request sent along includes the subprotocol `v10.stomp`. If the server accepts this subprotocol it will in turn, respond with that subprotocol name, and the client can continue sending and receiving frames to the server. If the server does not speak `v10.stomp` you will receive an error.

The default implementation of the `ws` library will accept any subprotocol that is sent along. Let's write some extra code to ensure that only the `v10.stomp` protocol gets accepted here. In order to do this, you'll write a special handler when initializing the `WebSocketServer` object.

```
var WebSocketServer = require('ws').Server,
wss = new WebSocketServer({port: 8181,
                           handleProtocols: function(protocol, cb) {
    var v10_stomp = protocol[protocol.indexOf("v10.stomp")];
    if(v10_stomp) {
        cb(true, v10_stomp);
        return;
    }
    cb(false);
}});
```

In Chapter 2 the overview of the WebSocket API showed that you could pass in more than one subprotocol. In our handler code, you'll have to unpack an array of subprotocols which includes the one the client is after. Since we're using Node.js you can use conventions like `Array.indexOf` without worrying about things like Internet Explorer not supporting it. With the code above, you've successfully performed a handshake accepting a new subprotocol.

Our initial implemented example for the STOMP protocol as discussed will be the stocks app. You'll send requests over STOMP from the client to server, and the server will send and receive messages with RabbitMQ while our stocks daemon spits out intermittent updates to prices. To get started, let's get a RabbitMQ server in place to queue our messages for the server.

Set Up RabbitMQ

Before continuing, you'll need to get a RabbitMQ node running for your WebSocket server to proxy the requests to. In order to do that, you'll need to have Vagrant setup on your development machine. Vagrant is a very handy tool for creating portable, and light-weight development virtual machines. Installing it as easy as grabbing the proper install binary for your operating system on the download page for Vagrant.

Vagrant

Vagrant is a lightweight tool to create and configure reproducible and portable development environments. It uses VirtualBox or VMWare under the hood for the virtualized instances, and allows for several different providers including: Puppet, Chef, Ansible, and even simple shell scripts.

After you have Vagrant installed successfully, create a new file in your project folder called `Vagrantfile` and include the following:

```
Vagrant.configure("2") do |config|
    config.vm.hostname = "websockets-mq"
    config.vm.box = "precise64"
    config.vm.box_url = "http://bit.ly/ubuntu-vagrant-precise-box-amd64"

    config.vm.network :forwarded_port, guest: 5672, host: 5672
    config.vm.network :forwarded_port, guest: 15672, host: 15672

    config.vm.provision "shell", path: "setup_rabbitmq.sh"

    config.vm.provider :virtualbox do |v|
        v.name = "websockets-mq"
    end
end
```

The configuration file will be used to create a new Vagrant instance using the image at `config.vm.box_url`. It forwards ports `5672` and `15672` to the local machine, and specifies a "shell" based provisioning to be run upon `vagrant up` which is included below:

```
#!/bin/bash

cat >> /etc/apt/sources.list <<EOT
deb http://www.rabbitmq.com/debian/ testing main
EOT

wget http://www.rabbitmq.com/rabbitmq-signing-key-public.asc
apt-key add rabbitmq-signing-key-public.asc

apt-get update

apt-get install -q -y screen htop vim curl wget
apt-get install -q -y rabbitmq-server

# RabbitMQ Plugins
service rabbitmq-server stop
rabbitmq-plugins enable rabbitmq_management
service rabbitmq-server start

# Create our websockets user and remove guest
```

```
rabbitmqctl delete_user guest
rabbitmqctl add_user websockets rabbitmq
rabbitmqctl set_user_tags websockets administrator
rabbitmqctl set_permissions -p / websockets ".*" ".*" ".*"

rabbitmq-plugins list
```

Our shell provisioning script does the following:

- Add a new source for the latest RabbitMQ install
- Installs a few dependencies along with the RabbitMQ Server
- Enables the `rabbitmq_management` plugin
- Removes the guest user and creates our new default user `rabbitmq:websockets`
- Gives that user administrator privileges

Now from the command line, initialize and provision the new vVgrant instance with the following:

```
vagrant up
```

This command will read the `Vagrantfile` and run the provisioning script to install RabbitMQ server on an Ubuntu 12.04 amd64 instance for use in our examples. Below you'll see what should be a similar printout after you complete the command. Immediately after this output, Vagrant will run the provisioning shell script that sets up RabbitMQ.

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'precise64'...
==> default: Matching MAC address for NAT networking...
==> default: Setting the name of the VM: websockets-mq
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 5672 => 5672 (adapter 1)
    default: 15672 => 15672 (adapter 1)
    default: 22 => 2222 (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
```

The included `Vagrantfile` which provides the configuration for Vagrant opens up the following ports:

- `tcp/5672` - the default port for amqp

- `tcp/15672` - the web management interface

Connecting the server to RabbitMQ

After you have the proper dependencies installed, it's time to circle back and get the server talking to RabbitMQ. The connection to RabbitMQ can happen independent of the WebSocket work. Upon execution of the server, you'll open a connection to RabbitMQ and perform two actions with the connection:

- Listen to the `stocks.result` queue for updates on pricing
- Publish stock requests at a set interval to the `stocks.work` queue

In order to do that with our server, we'll need to talk the AMQP protocol with RabbitMQ. There are many libraries out there for node.js to talk AMQP, and the simplest one I've found is *https://github.com/postwait/node-amqp*. Use the command `npm` below to install the library in your project folder.

```
npm install amqp
```

Our initial actions will be upon a valid `CONNECT` request initiated from the client to the server. You'll create a connection to the running RabbitMQ instance, using the authentication information passed in from the client.

Here's how you'll connect to the RabbitMQ instance we installed above:

```
amqp = require('amqp');

var connection = amqp.createConnection(
    { host: 'localhost',
      login: 'websockets',
      password: 'rabbitmq'
});
```

The library being used (amqp) fires events that can be listened for using callbacks. In the snippet below, it listens for the `ready` event and runs the callback function provided. Upon ensuring the connection is ready we start listening to the `stocks.result` queue and subscribe to receive updates to messages that get passed back through it. These messages will contain updated pricing for stocks that have been requested. You'll notice that within the blocks the `stomp_helper.js` library is being used to send `MESSAGE` frames back to the clients that have asked for updates on particular stocks.

```
connection.on('ready', function() {
    connection.queue('stocks.result', {autoDelete: false, durable: true}, function(q) {
        q.subscribe(function(message) {
            var data;
            try {
                data = JSON.parse(message.data.toString('utf8'));
```

```
                } catch(err) {
                    console.log(err);
                }
                for(var i=0; i<data.length; i++) {
                    for(var client in stocks) {
                        if(stocks.hasOwnProperty(client)) {
                            var ws = stocks[client].ws;
                            for(var symbol in stocks[client]) {
                                if(stocks[client].hasOwnProperty(symbol)
                                        && symbol === data[i]['symbol']) {
                                    stocks[client][symbol] = data[i]['price'];
                                    var price = parseFloat(stocks[client][symbol]);
                                    Stomp.send_frame(ws, {
                                        "command": "MESSAGE",
                                        "headers": {
                                            "destination": "/queue/stocks." + symbol
                                        },
                                        content: JSON.stringify({price: price})
                                    });
                                }
                            }
                        }
                    }
                }
            });
        });
    });
```

The payload being received from the `stocks.result` message queue looks like the following:

```
[
    {
        "symbol":"AAPL",
        "price":149.34
    },
    {
        "symbol":"GOOG",
        "price":593.2600000000037
    }
]
```

After parsing the payload, the block of code iterates over the result, and over a master list of stocks being stored across all connected clients. In the process of iterating over a JavaScript object you must check and ensure the value being passed during the iteration is actually part of the object using `myObject.hasOwnProperty(myIteratorValue)`. It maps the updated price with the price being stored and sends a message back to the connected client using STOMP over that specific `destination`.

When the client makes a request for a new stock, they get added to the master list of stocks. A separate block of code runs at an interval to send the master list to a

`stocks.work` queue which gets picked up by our `daemon.js` to find the updated price and send it back over the `stocks.result` queue. One of the prime reasons we do this is that it is easier to scale and the system can process more requests if needed by adding more daemons, without any adverse effect. The updater method looks like the below. It basically creates a string array of stock symbols, and publishes that to the `stocks.work` queue:

```
var updater = setInterval(function() {

    var st = [];
    for(var client in stocks) {
        for(var symbol in stocks[client]) {
            if(symbol !== 'ws') {
                st.push(symbol);
            }
        }
    }
    if(st.length>0) {
        connection.publish('stocks.work',
            JSON.stringify({"stocks": st}),
            {deliveryMode: 2});
    }
}, 10000);
```

Our stock price daemon

The below code is for our daemon which takes in an array of stock symbols, and spits out a JSON object with the up-to-date values using Yahoo YQL. Create a new file called `daemon.js` and insert the following snippet that will be described next.

```
#!/usr/bin/env node

var request = require('request'),
    amqp = require('amqp');

module.exports = Stocks;

function Stocks() {
    var self = this;
}

Stocks.prototype.lookupByArray = function(stocks, cb) {
    var csv_stocks = '"' + stocks.join('","') + '"';

    var env_url = '&env=http%3A%2F%2Fdatatables.org%2Falltables.env&format=json';
    var url = 'https://query.yahooapis.com/v1/public/yql';
    var data = encodeURIComponent('select * from yahoo.finance.quotes where symbol in (' + csv_sto
    var data_url = url
                    + '?q='
                    + data
```

```
                    + env_url;

    request.get({url: data_url, json: true},
        function (error, response, body) {
            var stocksResult = [];
            if (!error && response.statusCode == 200) {
                var totalReturned = body.query.count;
                for (var i = 0; i < totalReturned; ++i) {
                    var stock = body.query.results.quote[i];
                    var stockReturn = {
                        'symbol': stock.symbol,
                        'price': stock.Ask
                    };

                    stocksResult.push(stockReturn);
                }

                cb(stocksResult);
            } else {
                console.log(error);
            }
        });
};

var main = function() {
    var connection = amqp.createConnection({
        host: 'localhost',
        login: 'websockets',
        password: 'rabbitmq'
    });

    var stocks = new Stocks();
    connection.on('ready', function() {
        connection.queue('stocks.work', {autoDelete: false, durable: true}, function(q) {
            q.subscribe(function(message) {
                var json_data = message.data.toString('utf8');
                var data;
                console.log(json_data);
                try {
                    data = JSON.parse(json_data);
                } catch(err) {
                    console.log(err);
                }
                stocks.lookupByArray(data.stocks, function(stocks_ret) {
                    var data_str = JSON.stringify(stocks_ret);
                    connection.publish('stocks.result', data_str, {deliveryMode: 2});
                });

            });
        });
    });
```

```
};

if(require.main === module) {
    main();
}
```

This daemon can be executed using `node daemon.js`, and will connect to RabbitMQ and process the work it pulls from the RabbitMQ message queue. Several conventions should be noticeable from our WebSocket STOMP server, the method of connection, and processing the ready event. The daemon will listen to the `stocks.work` queue however, to get a list of stocks to look up, and in the end push the result back into the `stocks.result` queue. If you take a look at the `Stocks.prototype.lookupByArray` function, it's issuing a Yahoo YQL call for the stocks requested and returning the JSON payload as seen above.

Processing STOMP requests

Previous to diving into the server interaction with RabbitMQ, you saw how to achieve connection with STOMP over WebSocket using our library. Let's continue on and flesh out the rest of the commands necessary to interact with the frontend.

```
wss.on('connection', function(ws) {
    var sessionid = uuid.v4();

    stocks[sessionid] = {};
    connected_sessions.push(ws);
    stocks[sessionid]['ws'] = ws;

    ws.on('message', function(message) {
        var frame = Stomp.process_frame(message);
        var headers = frame['headers'];
        switch(frame['command']) {
            case "CONNECT":
                Stomp.send_frame(ws, {
                    command: "CONNECTED",
                    headers: {
                        session: sessionid
                    },
                    content: ""
                });
                break;
            case "SUBSCRIBE":
                var subscribeSymbol = symbolFromDestination(
                                        frame['headers']['destination']);
                stocks[sessionid][subscribeSymbol] = 0;
                break;
            case "UNSUBSCRIBE":
                var unsubscribeSymbol = symbolFromDestination(
                                        frame['headers']['destination']);
                delete stocks[sessionid][unsubscribeSymbol];
```

```
                break;
            case "DISCONNECT":
                console.log("Disconnecting");
                closeSocket();
                break;
            default:
                Stomp.send_error(ws, "No valid command frame");
                break;
        }
    });

    var symbolFromDestination = function(destination) {
        return destination.substring(destination.indexOf('.') + 1,
                                     destination.length);
    };

    var closeSocket = function() {
        ws.close();
        if(stocks[sessionid] && stocks[sessionid]['ws']) {
            stocks[sessionid]['ws'] = null;
        }
        delete stocks[sessionid];
    };

    ws.on('close', function() {
        closeSocket();
    });

    process.on('SIGINT', function() {
        console.log("Closing via break");
        closeSocket();
        process.exit();
    });
```

Upon a successful connection, as with previous examples a UUID is generated which will act as our sessionid for passing back and forth in the STOMP frame. The frame will get parsed and placed in the JavaScript object. From there we perform different actions based on the frame command passed. You've already seen the code for `CONNECT` and , so we'll focus on `SUBSCRIBE`, `UNSUBSCRIBE`, and `DISCONNECT`.

Both subscribing and unsubscribing modify our `stocks` object. With subscribing, you're adding a new symbol to the existing list of stocks for that `sessionid`. Unsubscribing is met by just removing that symbol from the list so it won't be passed back to the client. Receiving a `DISCONNECT` command from the client is met with closing the `WebSocket` and cleaning up any references to that and the client in the `stocks` object. Since this is an app to be run from the console, there is a chance of receiving a Ctrl-C which would break the connection. To handle this, hook into the `SIGINT` event that gets fired, so you can close the socket gracefully and on your own terms.

Client

The client is a simple interface with stocks that vary in price based on data returned from the server. The form at the very top takes a stock symbol as input, and attempts to `SUBSCRIBE` over STOMP to get updates from the server. While the subscribe request is being sent, a table row gets added for the new symbol and a placeholder of "Retrieving…" while waiting for data to return.

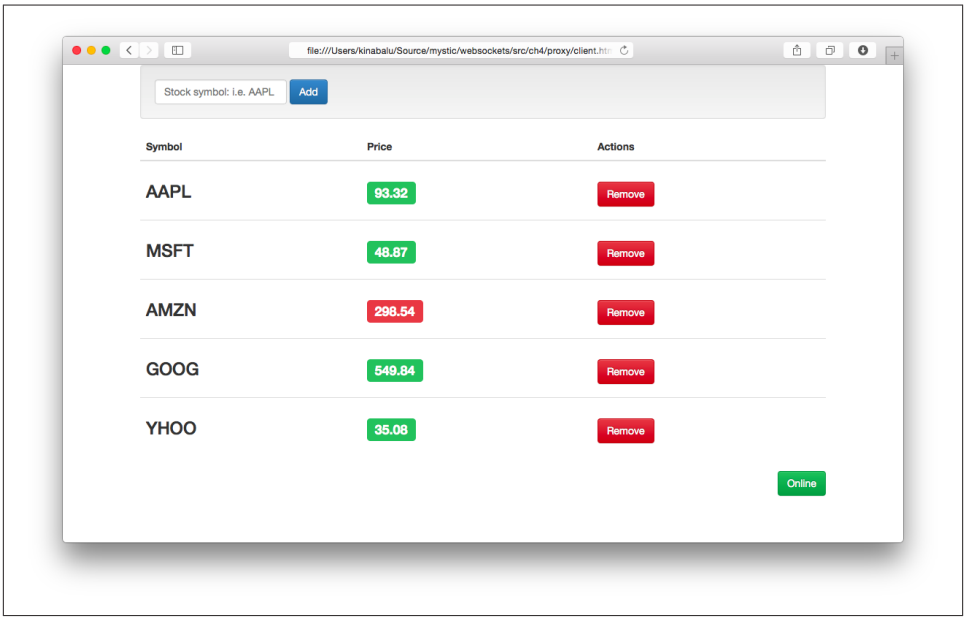


Figure 4-2. Stocks example of STOMP over WebSocket

The markup for our example is listed below. It outlines a simple form which calls the `subscribe` method that will be described next, and the table containing the stock symbols, the up-to-date pricing from the service, and a remove button. In addition a status indicator of connection to the WebSocket server has been added.

```
<div class="vertical-center">
<div class="container">

    <div class="well">

        <form role="form" class="form-inline" id="add_form"
                onsubmit="subscribe($('#symbol').val()); return false;">
            <div class="form-group">
                <input class="form-control" type="text" id="symbol"
                    name="symbol" placeholder="Stock symbol: i.e. AAPL" value=""
                    autofocus />
```

```
            </div>

            <button type="submit" class="btn btn-primary">Add</button>

        </form>

    </div>

        <table class="table" id="stockTable">
          <thead>
            <tr>
              <th>Symbol</th>
              <th>Price</th>
              <th>Actions</th>
            </tr>
          </thead>
          <tbody id="stockRows">
            <tr id="norows">
                <td colspan="3">
                No stocks found, add one above
                </td>
            </tr>
            </tbody>
        </table>

    <div class="text-right">
        <p>
            <a id="connection" class="btn btn-danger"
                    href="#" onclick="connect();">Offline</a>
        </p>
    </div>
</div>
</div>
```

Several functions make up our client app, and they will be described separately in the order they are executed. The first function will be `subscribe` which adds a new symbol to the interface and communicates that to the server.

```
var subscribe = function(symbol) {
    if(stocks.hasOwnProperty(symbol)) {
        alert('You already added the ' + symbol + ' symbol');
        return;
    }

    stocks[symbol] = 0.0;
    Stomp.send_frame(ws, {
        "command": "SUBSCRIBE",
        "headers": {
            "destination": "/queue/stocks." + symbol,
        },
        content: ""
    });
```

```
    var tbody = document.getElementById('stockRows');

    var newRow = tbody.insertRow(tbody.rows.length);
    newRow.id = symbol + '_row';

    newRow.innerHTML = '<td><h3>' + symbol + '</h3></td>' +
                       '<td id="' + symbol + '">' +
                       '<h3>' +
                       '<span class="label label-default">Retrieving..</span>' +
                       '</h3>' +
                       '</td>' +
                       '<td>' +
                       '<a href="#" onclick="unsubscribe(\'' + symbol +
                        '\');" class="btn btn-danger">Remove</a></td>';

    if(!$('#norows').hasClass('hidden')) {
        $('#norows').addClass('hidden');
    }

    $('#symbol').val('');
    $('#symbol').focus();
}
```

First thing to do whenever receiving user input is to perform validation, which is done to check if we already have that symbol in our list and return an error if found. If all is fine, we initialize the symbol to our list of stocks and send a new `SUBSCRIBE` frame to the server. The rest of the code is for the user interface, and adds a table row with default values while waiting for a legitimate value from the server.

If a client can subscribe to a stock update, it should be able to unsubscribe as well. This next snippet does exactly that, and is referenced in the previous code for remove.

```
Object.size = function(obj) {
    var size = 0, key;
    for (key in obj) {
        if (obj.hasOwnProperty(key)) size++;
    }
    return size;
};

var unsubscribe = function(symbol) {
    Stomp.send_frame(ws, {
        "command": "UNSUBSCRIBE",
        "headers": {
            "destination": "/queue/stocks." + symbol,
        },
        content: ""
    });
    $('#' + symbol + '_row').remove();

    delete stocks[symbol];
```

```
    if(Object.size(stocks) === 0) {
        $('#norows').removeClass('hidden');
    }
}
```

To unsubscribe we perform the following tasks:

- Send the `UNSUBSCRIBE` command in a STOMP frame with the symbol as part of the destination.
- Remove the table row in the user interface
- Remove the entry in the `stocks` object
- Check if there are any more symbols in the `stocks` object, and if not, unhide the `#norows` HTML block

The functions in the above two code snippets represent all the actions a user can take with our interface: subscribe, and unsubscribe. Now let's circle back to the `connect()` function which was already shown above, however the handlers were never touched on. The first will be the more elaborate form using the `stomp_helper.js` library for handling `open` events:

```
var onOpenHandler = function(e) {
    Stomp.send_frame(ws, {
        "command": "CONNECT",
        "headers": {
            login: "websockets",
            passcode: "rabbitmq"
        },
        content: ""
    });
}
```

In short, upon getting a connection to our WebSocket server, we send our `CONNECT` command with authentication information over the STOMP frame. To close the connection, you'll follow a similar path, and provide notification for the user interface.

```
var online = false;

var statusChange = function(newStatus) {
    $('#connection').html((newStatus ? 'Online' : 'Offline'));
    $('#connection').addClass((newStatus ? 'btn-success' : 'btn-danger'));
    $('#connection').removeClass((newStatus ? 'btn-danger' : 'btn-success'));
    online = newStatus;
}

var switchOnlineStatus = function() {
    if(online) logoff(); else connect();
}

var logoff = function() {
```

```
        statusChange(false);

        Stomp.send_frame(ws, {
            "command": "DISCONNECT"
            }
        );
        return false;
    }
```

The HTML code contains a status button which when clicked will run the `switchOnlineStatus` function. This will either disconnect you from the server, or reconnect you as seen above. The `logoff` function sends our `DISCONNECT` command using a STOMP frame to tell the server to perform it's own disconnection routines.

All of the work done on the server end to retrieve stocks through RabbitMQ is put into action below. Our `onMessageHandler` as you'll see takes data from the server and updates the frontend with the new values.

```
    var updateStockPrice = function(symbol, originalValue, newValue) {
        var valElem = $('#' + symbol + ' span');
        valElem.html(newValue.toFixed(2));
        var lostValue = (newValue < originalValue);
        valElem.addClass((lostValue ? 'label-danger' : 'label-success'))
        valElem.removeClass((lostValue ? 'label-success' : 'label-danger'))
    }

    var onMessageHandler = function(e) {
        frame = Stomp.process_frame(e.data);
        switch(frame['command']) {
            case "CONNECTED":
                statusChange(true);
                break;
            case "MESSAGE":
                var destination = frame['headers']['destination'];
                var content;
                try {
                    content = JSON.parse(frame['content']);
                } catch(ex) {
                    console.log("exception:", ex);
                }
                var sub_stock = destination.substring(
                        destination.indexOf('.') + 1, destination.length
                );
                updateStockPrice(sub_stock, stocks[sub_stock], content.price);
                stocks[sub_stock] = content.price;
                break;
        }
    }
```

When a new `message` event is passed, the code will process that data as a STOMP frame. The process will be to check for either the `CONNECTED` or `MESSAGE` commands from the frame. Commands that will be processed:

- `CONNECTED`: Call the `statusChange(true)` to change the button status to be "Online"
- `MESSAGE`: Retrieve the destination header, parse the content and update the stock price in the interface

The client has active portions with the subscribe/unsubscribe/disconnect portion, and the passive portions which cater to receiving data from the server. The `MESSAGE` events being fired will be tied to a STOMP `destination`, and the stocks will be updated accordingly based on the data retrieved.

You've successfully implemented the most basic functions available in the STOMP 1.0 protocol. The mapping between STOMP and WebSocket can be simple, and there are a few more commands that we have left unimplemented in our node-based proxy: `BEGIN`, `COMMIT`, `ACK`, and on the server side `RECEIPT`.

Mapping STOMP over WebSocket achieves two things for us: show you how to layer a different protocol over WebSocket using the subprotocol portion of the spec, and enables talking to an AMQP server without specifically needing a server component written. In the next section, you'll learn how to connect to RabbitMQ with SockJS using the Web-Stomp plugin with RabbitMQ. You'll learn more about using SockJS in Chapter 5 talking about compatibility with older browsers. There are several options available for messaging and some of the most popular:

- ActiveMQ (*http://activemq.apache.org/*)
- ActiveMQ Apollo (*http://activemq.apache.org/apollo/*)
- HornetQ (*http://hornetq.jboss.org/*)

RabbitMQ with Web-Stomp

Throughout this chapter you've been writing a server implementation of STOMP to effectively proxy commands to RabbitMQ using the AMQP protocol. This hopefully has shown how easy it can be to layer another protocol on top of WebSocket. Now to round out the end of the chapter, you'll learn how to set up RabbitMQ with Web-Stomp which is a plugin that allows RabbitMQ to accept STOMP. The plugin exposes a SockJS-compatible bridge over HTTP which is an alternative transport library which will be discussed in Chapter 5 in more detail. It enhances compatibility for older browsers that don't have native support for WebSockets.

Advanced Message Queuing Protocol (AMQP)

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security. From Wikipedia[*http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol*]

STOMP client for Web and node.js

For a more complete implementation of what you've worked on in this chapter, download the STOMP Over WebSocket library. It provides a JavaScript client library for accessing servers using STOMP 1.0 and 1.1 over WebSocket and a node.js library for doing the same over WebSocket along with an option for TCP sockets via STOMP.

Installing Web-Stomp plugin

Let's edit that provisioning shell script that was used earlier in the chapter to setup RabbitMQ. In the script, after stopping the rabbitmq server during installation, you'll add the following line:

```
rabbitmq-plugins enable rabbitmq_web_stomp
```

In addition, our virtual machine needs editing so we forward `port 15674` which is opened up by the previously installed plugin to listen for SockJS requests. You'll modify the existing `Vagrantfile` and add the following line with all the other network config options:

```
config.vm.network :forwarded_port, guest: 15674, host: 15674
```

After doing so, if the original virtualbox instance is still running, you can run `vagrant halt` or `vagrant destroy`, and then re-run `vagrant up` to recreate the instance. If you've destroyed, then you're done, and it will open up the new port, and turn on the new plugin. If you've halted, you can perform the following tasks:

```
vagrant ssh
sudo su -
rabbitmq-plugins enable rabbitmq_web_stomp
```

The above enables a new plugin called Web-Stomp and exposes port 15674. Rabbit has standardized on using SockJS for all WebSocket communication, and we will discuss that library further in Chapter 5. In order to continue, you'll want to download the JavaScript STOMP library available at stomp.js. After getting that downloaded, we can continue changing up our client code to use the Web-Stomp endpoint.

Echo Client for Web-Stomp

Let's build a simple echo client that subscribes to a queue `/topic/echo` and then sends and receives messages. At the very top of our HTML file we'll include the following JavaScript statements:

```
<script src="http://cdn.sockjs.org/sockjs-0.3.min.js"></script>
<script src="stomp.min.js"></script>
```

You can choose to download the minimized version as is being referenced above, or the un-minimized version if you're interested. In either case, you can download the Stomp Websocket library on GitHub.

Our HTML will be nearly identical to the previous echo example, and we'll modify the JavaScript to suit our needs using RabbitMQ Web Stomp plugin and the Stomp.js library.

```
<!DOCTYPE html>
<html><head>
    <title>Echo Server</title>
</head>
<body lang="en">
    <h1>Web Stomp Echo Server</h1>

    <ul id="messages">

    </ul>

    <form onsubmit="send_message(); return false;">
        <input type="text" name="message" style="width: 200px;"
            id="message" placeholder="Type text to echo in here"
            value="" autofocus />
        <input type="button" value="Send!" onclick="send_message();" />

    </form>
</body>
</html>
```

Our first task is going to be to initialize the RabbitMQ SockJS endpoint, and then pass that to the STOMP javascript library. The Stomp.js library allows you to use native WebSocket or anything that offers the same API such as SockJS. Since SockJS doesn't offer heartbeat support, you'll keep it turned off. The Stomp.js library offers several opportunities for callback and you can utilize the debug and perform whatever task you'd like on the data that comes back, here we're just outputting the data to console:

```
var ws = new SockJS('http://localhost:15674/stomp');
var client = Stomp.over(ws);

client.heartbeat.outgoing = 0;
client.heartbeat.incoming = 0;

client.debug = function(str) {
```

```
        console.log(str);
    }
```

Connecting to a RabbitMQ queue you'll simply offer login details, and a few callbacks along with the host (or virtualhost in RabbitMQ terms). The `append_log` function will be identical to previously, but implementing the callbacks required for `connect`, `error`, and a new `send_message` function can be seen below:

```
client.connect('websockets', 'rabbitmq', connect_callback, error_callback. '/');

var connect_callback = function(x) {
    id = client.subscribe("/topic/echo", function(message) {
        append_log(message.body);
        console.log(JSON.stringify(message.body));

    });
};

var error_callback = function(error) {
    console.log(error.headers.message);
};
```

In `connect_callback` you will issue a subscribe command for the queue `/topic/echo` so any messages that show up in that bin, will be appended to our UI text area. Our implementation of the `error_callback` simply outputs any error received to the console for debugging as needed.

You now have a client that will echo messages dumped into the queue to a text area. Now you will hook up the submission process to a new `send_message` function that looks very close to the WebSocket version:

```
var send_message = function(data) {
    client.send("/topic/echo", {}, document.getElementById('message').value);
};
```

The major difference here, is rather than just sending through the WebSocket, you provide the queue (destination), and extra headers of which we pass none in this example.

Summary

In this chapter you created a subprotocol over WebSocket for STOMP 1.0. As the server got built, the client evolved to support the different commands needed along the wire to support the protocol. In the end, while the client you built doesn't fully support all of STOMP 1.0, it allowed you to witness how easy it is to layer another protocol on top of WebSocket and connect it to a message broker like RabbitMQ.

As we saw in Chapter 2 implementing STOMP over WebSocket is one of the "Registered Protocols" (and also falls under an "Open Protocol"). Nothing is stopping you from

using the information in this chapter to create your own protocol for communication as the WebSocket spec fully supports this.

In the next chapter we'll explore the compatibility issues facing you when choosing to implement WebSockets, and how to ensure that you can start using the power of WebSockets today.

CHAPTER 5
WebSocket Compatibility

The technology behind WebSocket is to allow bidirectional communication between client and server. A native WebSocket implementation minimizes server resource usage and provides a consistent method of communicating between client and server. As with the adoption of HTML5 in client browsers, the landscape of support is relegated to modern browsers. That means no support for any user with Internet Explorer less than 10, and mobile browser support less than iOS Safari 6 and Chrome for Android.

Here are just some of the versions with RFC 6455 WebSocket support (*http://tools.ietf.org/html/rfc6455*).

- Internet Explorer 10
- Firefox 6
- Chrome 14
- Safari 6.0
- Opera 12.1
- iOS Safari 6.0
- Chrome for Android 27.0

This sentence needs to be inverted: "this chapter outlines options for when you need to support out-of-date browsers, etc."

This chapter outlines options for when you need to support older browsers that predate support for the [WebSocket RFC 6455 spec](*https://tools.ietf.org/html/rfc6455*) and you would like to take advance of bi-directional communication in your application. The platforms we'll look at solve compatibility issues with older client browsers, and add a layer of organization for our messages.

SockJS

SockJS is a JavaScript library that provides a WebSocket-like object in the browser. The library is compatible with many more browsers due to its conditional use of multiple browser transports. It will utilize WebSockets if the option is available as a first choice. If a native connection is not available it can fallback to streaming, and finally polling if that is also unavailable. This provides nearly full browser and restrictive proxy support as can be seen in the table below.

Table 5-1. Supported transports

Browser	WebSockets	Streaming	Polling
IE 6, 7	no	no	jsonp-polling
IE 8, 9 (cookies=no)	no	xdr-streaming	xdr-polling
IE 8, 9 (cookies=yes)	no	iframe-htmlfile	iframe-xhr-polling
IE 10	rfc6455	xhr-streaming	xhr-polling
Chrome 6-13	hixie-76	xhr-streaming	xhr-polling
Chrome 14+	hybi-10 / rfc6455	xhr-streaming	xhr-polling
Firefox <10	no	xhr-streaming	xhr-polling
Firefox 10+	hybi-10 / rfc6455	xhr-streaming	xhr-polling
Safari 5	hixie-76	xhr-streaming	xhr-polling
Opera 10.70+	no	iframe-eventsource	iframe-xhr-polling
Konqueror	no	no	jsonp-polling

In order for you to use the SockJS library completely, it requires a server counterpart. The library has several options for the server counterpart and more being written all the time. A sampling of some of the server libraries available is below.

- SockJS-node
- SockJS-erlang
- SockJS-tornado
- SockJS-twisted
- SockJS-ruby
- SockJS-netty
- SockJS-gevent (SockJS-gevent fork)
- SockJS-go

For our needs, we're going to stick with an all JavaScript solution.

SockJS Chat Server

We're going to revisit our chat application and make changes to use the SockJS libraries for server and client.

As mentioned above, in order to fully utilize the SockJS client library on the browser, we require a valid server component.

```
var express = require('express');
var http = require('http');
var sockjs = require('sockjs');
var uuid = require('uuid');
```

Our list of new libraries now includes [SockJS] (*https://github.com/sockjs/sockjs-node*), `http` from the standard Node.js library, and [Express] (*https://github.com/visionmedia/express*).

Node Package Manager

Node.js has a fully developed package manager with [npm] (*https://npmjs.org*). They are usually installed together, and a simple call to `npm install [package]` will pull down the latest revision. The install will create a `node_modules` directory if it does not exist, and place the modules inside. If you'd like to install the module globally you can use the `-g` flag. For more reading check out [the docs] (*https://www.npmjs.org/doc/*).

These dependencies will not be available in Node.js by default, so simply run the following commands to install them:

```
npm install sockjs
npm install express
```

Your next step is to create a SockJS object and listen for the `connection` event. The events used with SockJS-node are slightly different than similar ones from the WebSocket clients:

- `connection`
- `data` - *equivalent to `message` with WebSocket*
- `close`
- `error`

Express does something interesting with it's library by exporting a function as the interface to it's module. This is used to create a new Express application and can be written a couple of ways:

```
var app = express();
```

or the much more terse

```
var express = require('express')();
```

This creates an Express application and allows you to assign it to the variable right away. Behind the scenes, there's some JavaScript magic happening by assigning the function to `module.exports`:

```
exports = module.exports = createApplication;

...

function createApplication() {
    ...
}
```

Now you can create your new SockJS server by initializing `express`, creating an `httpServer` with the `express` application, and finally creating a SockJS server that listens for the `connection` event:

```
var app = express();
var httpServer = http.createServer(app);
var sockServer = sockjs.createServer();

sockServer.on('connection', function(conn) {
...
  conn.on('message', function(message) {
    if(message.indexOf('/nick') === 0) {
      var nickname_array = message.split(' ');
      if(nickname_array.length >= 2) {
        var old_nickname = nickname;
        nickname = nickname_array[1];
        var nickname_message = "Client " + old_nickname + " changed to " + nickname;
        wsSend("nick_update", client_uuid, nickname, nickname_message);
      }
    } else {
      wsSend("message", client_uuid, nickname, message);
    }
  });
...
}
```

The only change to the event handling from our previous code is listening for an event called `data` now instead of `message`. In addition, we make a slight adjustment to our `wsSend` method to account for differences with the SockJS API:

```
var CONNECTING = 0;
var OPEN = 1;
var CLOSING = 2;
var CLOSED = 3;

function wsSend(type, client_uuid, nickname, message) {
```

```
    for(var i=0; i<clients.length; i++) {
      var clientSocket = clients[i].connection;
      if(clientSocket.readyState === OPEN) {
        clientSocket.write(JSON.stringify({
          "type": type,
          "id": client_uuid,
          "nickname": nickname,
          "message": message
        }));
      }
    }
  }
```

The WebSocket object you had used previously had constants for the `readyState` property, but here you'll have them defined in your client code (to avoid littering the code with integers). The SockJS connection object has the same `readyState` property, and you will check it against the `OPEN` constant which has a value of `1`. The other biggest change is the method for sending data back to the client is `.write(message)` instead of `.send(message)`.

Now that you've converted everything from the WebSocket version to use the SockJS specific code, you'll initialize a new app with Express and bind the prefix `/chat` to your `http.Server` instance.

```
var app = express();
var httpServer = http.createServer(app);

sockServer.installHandlers(httpServer, {prefix:'/chat'});
httpServer.listen(8181, '0.0.0.0');
```

The HTTP server will listen on port `8181` and respond to requests listening on any IP from the machine as `0.0.0.0` denotes.

In the example from Chapter 3 we opened our HTML file without an HTTP server present. With SockJS and the other alternatives in this chapter, we'll opt for serving the client and server from the same HTTP server. Here we set up our `client.html` and `style.css` to be sent back upon a request to `http://localhost:8181/client.html`.

```
express.get('/client.html', function (req, res) {
    res.sendfile(__dirname + '/client.html');
});

express.get('/style.css', function (req, res) {
    res.sendfile(__dirname + '/style.css');
});
```

You have now successfully converted the plain WebSocket server to one that uses the SockJS library.

SockJS Chat Client

Let's walk through how to convert the client to use the SockJS library.

First thing you'll need at the very beginning of any other JavaScript will be to include the SockJS library.

```
<script src="http://cdn.sockjs.org/sockjs-0.3.min.js"></script>
```

This library provides the SockJS object which mimics the WebSocket library included in most modern browsers. The initialization also changes as we are not using the `ws` or `wss` protocol, but instead using `http` as the initial transport.

```
var sockjs = new SockJS("http://127.0.0.1:8181/chat");
```

For our WebSocket client code we used the variable name `ws`. Here it seems more appropriate to rename it to `sockjs`. Find all instances of using `ws` in the code from Chapter 3, and replace them with `sockjs`. That is the extent of the changes required for the client. SockJS delivers nicely on an easy migration from native WebSocket to their library.

They offer support for one or more streaming protocols for every major browser which all work cross-domain and support cookies. Polling transports will be utilized in the event of older browsers and hosts with restrictive proxies as a viable fallback.

In the next section we'll take on changing our chat application to use the Socket.IO platform instead.

Socket.IO

Utilizing WebSocket directly is an easy decision when you can control the clients that are using your system. With most organizations having to cater to a heterogeneous client environment, another alternative is **Socket.IO** (*http://socket.io/*). The development behind Socket.IO looks to make realtime apps possible regardless of browser.

The library is able to perform this feat by gracefully falling back to different technologies that perform similar things. The transports used in the event that WebSocket is not available in the client:

- Adobe Flash Socket
- Ajax long polling
- Ajax multipart streaming
- Forever iframe
- JSONP Polling

Using the native WebSocket implementation would be akin to utilizing TCP directly to communicate. It's certainly possible to do so, and perhaps in most cases the right choice,

but there's no shame in using a framework to do some of the heavy lifting for you. By default, Socket.IO will utilize a native WebSocket connection if browser interrogation deems it possible.

Adobe Flash Socket

One of the alternate transports provided by Socket.IO is the Adobe Flash Socket. This allows a WebSocket-like connection to be used over Adobe Flash in lieu of native support. This has the benefit of a socket connection, with very few drawbacks. One of the drawbacks is requiring another port to be open for the policy server. By default, Socket.IO will check port `10843` and attempt to use that if available.

Connecting

Connecting to a Socket.IO server is first achieved by grabbing the client libraries. If the client you're using is JavaScript, the simplest method of getting this done is simply referencing the Socket.IO server and including the socket.io.js file.

```
<script src="http://localhost:8181/socket.io/socket.io.js"></script>
```

The easiest path of serving the client library, is from the Socket.IO server itself. If your web server and socket.io are both being served by the same host and port, you can omit the host and port from the call and reference it like any other file served from the webserver. In order to serve the Socket.IO client library on the same host and port, you'll have to either configure your web server to forward requests to the Socket.IO server, or you can also clone the `socket.io-client` repository (*https://github.com/LearnBoost/socket.io-client*) and place the files where-ever you'd like.

If you'd like to aggressively cache the Socket.IO client library a further configuration you can do is include the version number in the request like so:

```
<script src="/socket.io/socket.io.v1.0.js"></script>
```

As we discussed in Chapter 2, WebSocket uses four "control frames" or events. With Socket.IO, everything is a lot more open ended in the events department. There are a number of events that are fired from the framework itself:

- **connection** - the initial connection from a client which supplies a `socket` argument which can be used for future communication with client
- **message** - the event that emits when the client invokes `socket.send`.
- **disconnect** - the disconnect event is fired whenever the client-server connection is closed.
- **anything** - any event except for the reserved ones listed, data argument is the data sent, and callback is used to send a reply

First things first, after you include the JavaScript client library we need to open up a connection to the server.

```
var socket = io.connect('http://localhost:8181');
```

Now that you have a Socket.IO connection, you can start listening for specific events that will be emitted from the server. Your client application can listen for any named event coming from the endpoint, and can also emit its own events to be listened to and reacted to from the server-side.

Socket.IO Chat Server

Let's again revisit our chat example. Our code from `SockJS` will be copied mostly verbatim, and we'll do initialization similar to the previous library:

```
var socketio = require('socket.io');

...

var app = express();
var httpServer = http.createServer(app);
var io = socketio.listen(server);
```

Due to *socket.io* having an open-ended naming for events, there is no need to shoe-horn different incoming events within the same `message` construct. Therefore with our *socket.io* code we shall split up messages and the nickname requests into separate events:

```
conn.on('message', function(data) {
    wsSend("message", client_uuid, nickname, message);
});

...

conn.on('nickname', function(nick) {
    var old_nickname = nickname;
    nickname = nick.nickname;
    var nickname_message = "Client " + old_nickname + " changed to " + nickname;
    wsSend('nickname', client_uuid, nickname, nickname_message);
})
```

We've pushed the code for parsing a nickname request to the client, and can also listen for a separate event sent from the server for `nickname` specific messages and logically process them differently if we choose.

Socket.IO Chat Client

When the client wants to communicate with the server, it performs the same API function, and `emit`'s a named event that the server can listen for. Due to the nature of serving the socket.io HTML on the same HTTP server, you are able to reference `socket.io` from the same domain without specifying:

```
<script src="/socket.io/socket.io.js"></script>
```

With SockJS, it closely maps the native WebSocket spec. With socket.io the only similarity is listening for events and sending events back to the server. There are a number of events that are fired from the Socket.IO framework which should help keep you connected and knowledgeable about the connection and status.

- **connect** - when the connection with the server is successful, this `connect` is emitted
- **connecting** - when the connection is being attempted with the server
- **disconnect** - when the connection has been disconnected with the server
- **connect_failed** - when socket.io has failed to establish a connection with any and all transport mechanisms to fallback
- **error** - when an error occurs that isn't handled by other event types
- **message** - when a message is received via a `socket.send` and callback is an optional acknowledgement function.
- **reconnect_failed** - emitted when socket.io failed to re-establish a working connection after the connection dropped
- **reconnect** - emitted when socket.io successfully reconnects to the server
- **reconnecting** - emitted when socket.io is attempting to reconnect with the server
- **anything** - any event except for the reserved ones listed, data argument is the data sent, and callback is used to send a reply

Also, as we discussed earlier, the `socket.io-client` (*https://github.com/LearnBoost/socket.io-client*) is available if you'd like to serve the library without using the regular mechanism.

In our new client codebase, the size grows a bit to handle pushing the nickname command parsing to the frontend, and emitting our new event `nickname`.

```
function sendMessage() {
    var messageField = document.getElementById('message');
    var message = messageField.value;
    if(message.indexOf('/nick') === 0) {
        var nickname_array = message.split(' ');
        if(nickname_array.length >= 2) {
            socket.emit('nickname', {
                nickname: nickname_array[1]
            });
        }
    } else {
        socket.send(messageField.value);
    }
    messageField.value = '';
```

```
        messageField.focus();
    }
```

As you can see from the above, we've moved the code originally in the server, over to the client end, and using the `socket.emit(channel, data)` call from socket.io to send our `nickname` change on to the server.

Everything else on the client, is pretty much the same. We use socket.io's method `on(channel, data)` to listen for specific events (reserved or otherwise), and process them as usual.

Now that you've written your first socket.io project, you can look through the documentation at *http://socket.io* and review the extra features it provides on top of WebSocket and what we've discussed.

Let's move on to one more project, which is of a commercial nature and in the same camp as socket.io in the added features, and value-adds it provides on top of the native WebSocket implementation.

Pusher.com

The final option we will look at is a layer that sits on top of WebSocket and offers the fallbacks we've seen in other solutions. The team behind Pusher has built out an impressive list of features to utilize with your application should you choose to use their service. In the same way as the other two solutions, Pusher has implemented a layer on top of WebSocket via their API along with a method of testing for acceptable fallback methods should the others fail.

Their API is able to perform the fallbacks very similar to Socket.IO by testing for WebSocket support, and in the event that fails utilizing the very popular web-socket.js client (*https://github.com/gimite/web-socket-js*) which substitutes a Flash object for in-browser WebSocket support. In the event that Flash is not installed, or firewalls or proxies prevent a successful connection, the final fallback utilizes HTTP-based transports.

Similar to events which are on-top of Socket.IO's transport, the Pusher API has a few more tricks up its sleeve. They feature channels as a public and private type which allow you to filter and control communication to the server. A special type of channel is also available for `presence` where the client can register member data to show online status.

The major difference here, is that we're including a third-party in our communication between server and client. Our server will receive communication from client code most likely using a simple Ajax call from HTML, and based on that will in turn use the Pusher.com REST API to trigger events. The client will be connected to Pusher.com hopefully over WebSocket if it's available within the browser, or one of the fallback methods, and receive events triggered against the app. Only with several constraints met can a client

trigger events and pass them over the network without going through it's own server API first.

Let's go over some of the particular aspects of the Pusher.com API as they are quite extensive.

Channels

Using native WebSocket is a perfect way to achieve bidirectional communication with the understanding that the clients must support the WebSocket protocol, that you can overcome any proxy issues, and that you'll build out any infrastructure code necessary to make life easier on the backend. You'll get a data stream from the text or binary message frame and it's up to you to parse, make sense of it, and pass it on to whatever handler you've set up in your code.

The Pusher API provides a fair bit of this for you. Channels are a common programming construct and used with this API for filtering data, and controlling access. A channel comes into existence simply by having a client *subscribe* to it, and binding events to it.

Pusher has libraries for a lot of the major frameworks and languages that are popular today. We focus as always, on JavaScript. Here we're showing how to subscribe to a channel `channelName`. Once we have our `channel` variable we can use that to send and receive events.

With most of the channel operations, there is an event you can bind to which will notify you of the subscription success or failure `pusher:subscription_succeeded`.

```
var channel = pusher.subscribe(channelName);
```

In this way, you've just created a public named channel which any client connecting to the server can subscribe to / unsubscribe from. And unsubscribing is also, as simple as they could make it. Just provide the `channelName` and the API will unsubscribe you from listening on that channel.

```
pusher.unsubscribe(channelName);
```

The API also provides for private channel subscription. In order to do so, the permission must be authorized via an HTTP requested authentication url. All private channels are prefixed with `private` as a naming convention as seen in the code sample below. The authentication can happen via Ajax or JSONP.

```
var privateChannelName = "private-mySensitiveChannelName";
var privateChannel = pusher.subscribe(privateChannelName);
```

One of the most needed features when using bidirectional communication is state management for member *presence*. Pusher provides for this with specialized calls for user *presence* along with events to listen for to ensure completeness.

The following events are ones you can listen for to ensure expectations were met.

- **pusher:subscription_succeeded** - common in all channel calls, binding to this event will let you ensure subscription has succeeded
- **pusher:subscription_error** - bind to this event to be notified when a subscription has failed
- **pusher:member_added** - this event gets triggered when a user joins a channel. This event will be fired only once per unique user even if they've joined multiple presence channels.
- **pusher:member_removed** - this event gets triggered when a user has left a channel. Due to the ability to join multiple channels by a user, this event will only be fired when the last channel is closed.

Events

Events in Pusher are the way that messages get passed back and forth from the server and the client. A channel whether public or private can hold events that pass this data down to the client. In essence, the channel is as we discussed earlier, the filtering method, and the events are the data. If you're looking to filter messages in different buckets, events are not the way, but channels are. Events in Pusher are aptly named in the past tense as they are notifications of things that *happened* on the system.

If we have a channel called `chat` then we would want to be aware when new messages were occurring so we could paint that in the GUI.

```
var pusher = new Pusher('APP_KEY');
var channel = pusher.subscribe('chat-websocket');
channel.bind('new-message', function(data) {
    // add any new messages to our collection
  }
);
```

Binding via a channel is not required. Just as easily as we bound to a channel firing events, we can do so using the root pusher variable.

```
var pusher = new Pusher('APP_KEY');
pusher.bind(eventName, function(data) {
    // process eventName's data
});
```

Obviously the API for Pusher and usage patterns we can have are quite vast. The Pusher API is very well designed and able to process an insane amount of messages per day and amount of simultaneous connections. In the next section we'll perform the same exercise we did previously with Socket.IO and build out a small chat application using the Pusher API.

Pusher Chat Server

We've written a simple chat application using Socket.IO and SocksJS, and now it's time to take the knowledge we've gained from Pusher.com's API and way of doing things and rewrite the chat. The major difference is that sending our chat messages from the client will be done via an API we've cooked up on our server. All triggered events to Pusher.com happen via our server, and bound events on channels are passed from Pusher.com to the client using WebSocket or the fallback.

Let's first outline our server including a shell of the API calls and the dependencies we'll need. First thing's first, we'll need to install our node dependencies using npm:

```
$ npm install node-uuid
$ npm install pusher
$ npm install express
$ npm install body-parser
```

We've installed and used `node-uuid` in several other server examples. This section is obviously about the Pusher.com API so we're going to install their nodejs library. In order to listen for and parse the body of messages as JSON we're using `express` and `body-parser` to achieve this.

Here's a shell of what our server looks like:

```
var express = require('express');
var http = require('http');
var Pusher = require('pusher');
var uuid = require('node-uuid');
var bodyParser = require('body-parser');

var app = express();
app.use(bodyParser.json());

var httpServer = http.createServer(app);

var pusher = new Pusher({
  appId: 'YOUR-APP-ID',
  key: 'YOUR-APP-KEY',
  secret: 'YOUR-APP-SECRET'
});

var clients = {};
var clientIndex = 1;

function sendMessage(type, client_uuid, nickname, message) {
}

app.post("/nickname", function(req, res) {
});

app.post("/login", function(req, res) {
```

```
});

app.post("/chat", function(req, res) {
});

app.listen(8181);

app.get('/client.html', function (req, res) {
  res.sendfile(__dirname + '/client.html');
});
```

As you can see from above, we've required and included our dependencies, spun up express with the body-parser and gotten it to listen on port `8181` and serve our client template. Our API consists of the following calls:

Table 5-2. API Calls

HTTP Method	Endpoint	Description
POST	/nickname	Update the client nickname and notify all connected clients
POST	/login	Initial connection which assigns an anonymous nickname and a unique client id
POST	/chat	Messages for the chat are passed along with the nickname and client id

Our `sendMessage` call isn't part of the API but a convenience function used by several of them and triggers an event of `type` on the channel `chat` which we've bound when starting the server. The JSON we're passing back for all messages includes the client `id`, `nickname` if applicable, and `message`.

```
function sendMessage(type, client_uuid, nickname, message) {
    pusher.trigger('chat', type, {
      "id": client_uuid,
      "nickname": nickname,
      "message": message
    });
}
```

The first API call expected to be made by a client is to `login`. The client will receive a unique identifier in the form of a `uuid` and a unique indexed nickname.

```
app.post("/login", function(req, res) {
    var client_uuid = uuid.v4();
    var nickname = "AnonymousUser" + clientIndex;
    clientIndex+=1;

    clients[client_uuid] = {
        'id': client_uuid,
        'nickname': nickname
    };

    res.status(200).send(
        JSON.stringify(clients[client_uuid])
```

```
    );
});
```

Our clients are likely to want their own nicknames represented in the chat application. A call to `/nickname` will make the requested change and trigger an event `nickname` to allow clients to show the change on the frontend:

```
app.post("/nickname", function(req, res) {
    var old_nick = clients[req.body.id].nickname;

    var nickname = req.body.nickname;
    clients[req.body.id].nickname = nickname;

    sendMessage('nickname',
                req.body.id,
                nickname,
                old_nick + " changed nickname to " + nickname);

    res.status(200).send('');
});
```

The simplest of them all, is the chat message. We accept the client `id`, grab the `nick name` from our existing array, and use the `message` passed in the json and trigger a `message` event up to the `chat` Pusher.com channel:

```
app.post("/chat", function(req, res) {
    sendMessage('message',
                req.body.id,
                clients[req.body.id].nickname,
                req.body.message);

    res.status(200).send('');
});
```

Pusher Chat Client

Our server is now awaiting a client to connect. We'll be using the same HTML template as in previous chapters, and using the chat HTML from Chapter 3 to make life simpler. Let's outline what's necessary to get our client synced up with Pusher.com.

First thing we'll need is to include the Pusher.com library in our HTML code:

```
<script src="http://js.pusher.com/2.1/pusher.min.js"></script>
```

Within our JavaScript code we initialize our Pusher object with the app key given in the Pusher dashboard, and immediately subscribe to the `chat` channel.

```
var pusher = new Pusher('YOUR-APP-KEY');
var channel = pusher.subscribe('chat');
var id;

pusher.connection.bind('connected', function() {
```

```
    $.ajax({
        url: 'http://localhost:8181/login',
        type: 'POST',
        dataType: 'json',
        contentType: "application/json",
        complete: function(xhr, status) {
            if(xhr.status === 200) {
                console.log("login successful.");
            }
        },
        success: function(result) {
            appendLog('*', result.nickname + " connected");
            id = result.id;
        }
    })
});

pusher.connection.bind('disconnected', function() {
    appendLog('*', 'Connection closed');
});

function disconnect() {
    pusher.disconnect();
}

channel.bind('message', function(data) {
    appendLog(data.nickname, data.message);
});

channel.bind('nickname', function(data) {
    appendLog('*', data.message);
});
```

The Pusher connection object will emit several events, and we're only concerned with `connected` and `disconnected`. After subscribing to the `chat` channel we bind to a few specific events on that channel: `message` and `nickname`. For each of these we'll show notification messages on the client frontend. When we bind to and receive the `connected` event, we send our `login` request to the server API and receive back our client `id` to be passed in subsequent messages.

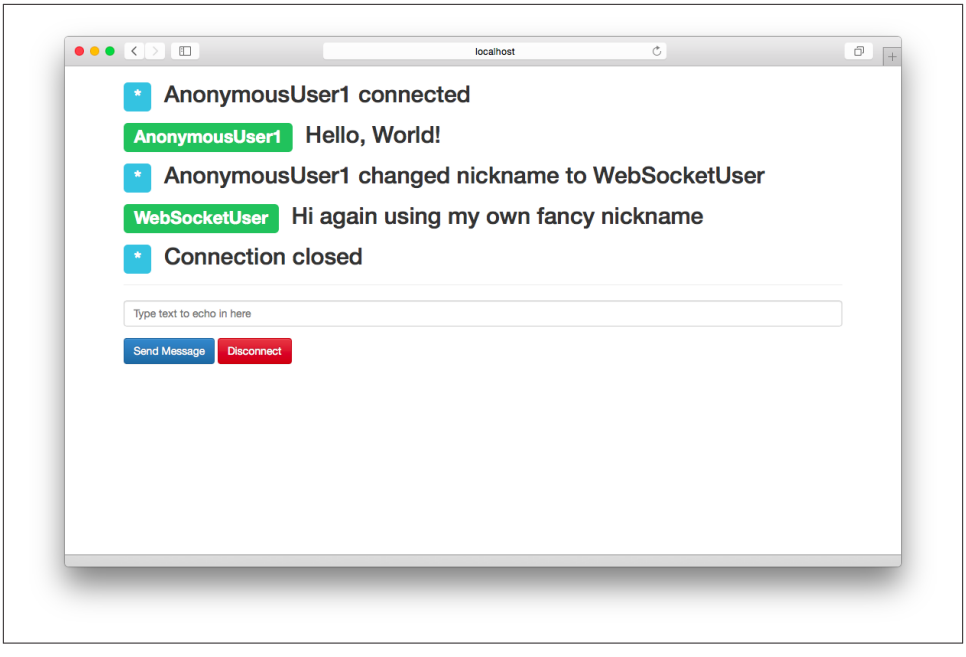


Figure 5-1. Pusher chat example

Above is an example of our chat using Pusher.com. We've given a concrete example of using the basics of the API available to us. Our intention was to show what was possible with alternatives to WebSocket, and Pusher is definitely worthy of consideration as an alternative to pure WebSocket.

Don't forget: Pusher is a Commercial Solution

Unlike WebSocket, Socket.IO and SocksJS, this framework is a commercial service. Evaluation of the solution and the benefits it provides will have to be made by your team. In general the different pricing tiers will be based on connections, messages, and whether the connection is protected via SSL encryption or not. For further evaluation review their pricing page (*http://pusher.com/pricing*).

Reverse Proxy

One of the things you'll likely also be asked to do, is proxy the WebSocket connection behind a webserver. The two most common webservers are nginx and Apache. The setup for these is rather simple, with nginx having the functionality built in to the server itself, and Apache using a module `proxy_wstunnel`. Rather than go into a ton of detail on how to configure both of these servers to proxy the connections, here are two blog articles which discuss them:

- nginx - *http://nginx.com/blog/websocket-nginx/*
- apache - *http://stackoverflow.com/questions/27526281/websockets-and-apache-proxy-how-to-configure-mod-proxy-wstunnel*

Summary

In this chapter we talked about three very popular ways to harness the power of bidirectional communication while dealing with a higher-level API. These solutions give you the power of WebSocket in the event that your client is using a modern client browser, and fallback to Flash socket or other less-optimized solutions for older clients. In addition, the two latter frameworks add features that are not natively supported by WebSocket limiting the amount of code you'll have to write to support your applications communication. In the next chapter we'll look at the methods of securing your WebSocket communication.

CHAPTER 6
WebSocket Security

In this chapter we'll talk at length about the WebSocket security aparatus and the various ways we can use to secure the data being passed over the underlying protocol. We'll also learn why it is always a good idea to communicate over TLS (Transport Layer Security) to avoid ineffective proxies, man in the middle attacks, and ensure frame delivery. Discussion will center around how to setup the WebSocket connection over TLS with `wss://` (WebSocket secure), talk about Origin-based security, frame masking, and specific limits imposed by browsers to ensure messages don't get hijacked.

With any discussion about security, the content of this chapter can be seen as the best known data about how to properly secure your WebSocket communication today. Security is fickle though, and the cat and mouse game played with those who seek to exploit and those who work to block is constant and unending. Data validation and multiple checks are even more important while using WebSocket. You'll begin by setting up WebSocket over TLS in the next section.

TLS and WebSocket

All of our demos so far, have used the unencrypted version of WebSocket communication with the `ws://` connection string. In practice, only in the simplest hierarchies should this ever happen, and all communication via WebSocket should happen over Transport Layer Security (TLS).

Generating a self-signed certificate

A valid TLS-based connection over WebSocket can't be done without a valid certificate. What you'll go over fairly quickly here is a way to generate a self-signed certificate using `OpenSSL`. First thing you'll need to do is ensure that if you don't already have OpenSSL installed, that you follow one of the set of instructions below specific to your platform.

Installing on Windows

We will only cover downloading and installing the pre-compiled binary available on Windows. As we discussed in Chapter 1, for the masochistic among us, download the source and compile yourself (*http://openssl.org/source/*).

For the rest of us, download the stand-alone Windows executable (*http://gnuwin32.sourceforge.net/packages/openssl.htm*). You should be able to run OpenSSL after this via the examples following the instructions on OS X and Linux installs.

Installing on OS X

The easiest method of installing OpenSSL on OS X is via a package manager like Homebrew (*http://mxcl.github.io/homebrew/*). This allows for quick and easy updating without having to re-download a package from the web. Assuming you have Homebrew installed, it is simply:

```
brew install openssl
```

Installing on Linux

There are so many flavors of Linux, it would be impossible to illustrate how to install on all of them. We will reiterate you how to install it via apt on Ubuntu Since there are more flavors of Linux than stars in the sky, we'll only provide simple instructions on how to get it via apt on Ubuntu. If you're running another distro, you can read through the Compilation and Installation[*http://wiki.openssl.org/index.php/Compilation_and_Installation*] instructions from OpenSSL.

So getting it using apt requires a few simple steps:

```
sudo apt-get update
sudo apt-get install openssl
```

Setting up WebSocket over TLS

Now that we have OpenSSL installed we can use it for the purposes of generating a certificate to be used for testing, or to submit to a Certificate Authority.

A Certificate Authority (CA) is an entity that issues digital certificates in a Public Key Infrastructure (PKI). The CA is a trusted entity that certifies the ownership of a public key as the named subject of the certificate. The certificate can be used to validate that ownership, and encrypt all information going over the wire.

Here's what we're going to do in the following block of code:

- Generate a 2048-bit key with passphrase
- Rewrite that key removing the passphrase
- Create a Certificate Signing Request (CSR) from that key
- Generate a self-signed certificate from the key and CSR

First thing to do is generate a 2048-bit key. We'll do this by using the `openssl` command to generate the RSA keypair:

```
% openssl genrsa -des3 -passout pass:x -out server.pass.key 2048

Generating RSA private key, 2048 bit long modulus
.....................................................................+++
........+++
e is 65537 (0x10001)
```

Then we go about generating a private key sans passphrase for eventual creation of a CSR which can be used for a self-signed certificate, or to receive a certificate authorized by a Certificate Authority. After generating the key, we can remove the key with the passphrase as well.

```
% openssl rsa -passin pass:x -in server.pass.key -out server.key
writing RSA key
% rm server.pass.key
```

Now that we have our private key, we can use that to create a Certificate Signing Request (CSR) which will be used to generate the self-signed certificate for sending secure WebSocket communication.

```
% openssl req -new -key server.key -out server.csr
      -subj '/C=US/ST=California/L=Los Angeles/O=Mystic Coders, LLC/
             OU=Information Technology/CN=ws.mysticcoders.com/
             emailAddress=fakeemail AT gmail DOT com/
             subjectAltName=DNS.1=endpoint.com' > server.csr
```

If you're looking to get setup with a proper server certificate, the CSR file will be all you need. You will receive a certificate file from the Certificate Authority which you can then use below. While you wait though, let's get the self-signed certificate for testing and replace it later.

Use the following code below to generate our certificate for use in the server code:

```
% openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
Signature ok
subject=/C=US/ST=California/L=Los Angeles/...
Getting Private key
```

In the directory you have chosen to run everything in, you should now have 3 files:

- `server.key`

- `server.csr`
- `server.crt`

If you decide to send things to a Certificate Authority for a validated Certificate, you'll send the `server.csr` file along with the setup procedure to receive a key. As we're just going to use a self-signed certificate here for testing purposes, we'll continue with our generated certificate `server.crt`. Decide on where you'll keep the private key and certificate files (in this instance we're going to place them in `/etc/ssl/certs`).

WebSocket Server over TLS example

In the below code you'll see an example of using the `https` module to allow the bidirectional WebSocket communication to happen over TLS and listen on `port 8080`:

```
var fs = require('fs');

// you'll probably load configuration from config
var cfg = {
    ssl: true,
    port: 8080,
    ssl_key: '/etc/ssl/certs/server.key',
    ssl_cert: '/etc/ssl/certs/server.crt'
};

var httpsServ = require('https');
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;

var app = null;

// dummy request processing
var processRequest = function( req, res ) {
    res.writeHead(200);
    res.end("Hi!\n");
};

app = httpsServ.createServer({
    key: fs.readFileSync( cfg.ssl_key ),
    cert: fs.readFileSync( cfg.ssl_cert )

}, processRequest ).listen( cfg.port );

var wss = new WebSocketServer( { server: app } );

wss.on( 'connection', function ( wsConnect ) {

    wsConnect.on( 'message', function ( message ) {
        console.log( message );
    });
```

```
});
```

Changing client code to utilize a WebSocket connection over TLS is trivial:

```
var ws = new WebSocket("wss://localhost:8080");
```

When making this connection, the web page being used to load it must also connect over TLS. In fact, if you attempt to load an insecure WebSocket connection from a website using the `https` protocol, it will throw a security error in most modern browsers for attempting to load insecure content. Mixed content is a common attack vector and is rightfully discouraged from being allowed. In most modern browsers, the use of mixed content is not only actively discouraged, it is forbidden. Chrome, Firefox, and Internet Explorer all throw security errors and will refuse to communicate over anything other than WebSocket secure in the event the page being loaded is also served over TLS. Safari unfortunately, does not do the proper thing. Below is an example from Chrome showing the errors in console upon attempting to connect to an insecure WebSocket server.

Figure 6-1. Mixed content Security Error with Chrome

Qualys Labs has a nice chart identifying the browsers that handle mixed content properly, and those that do not available at *https://community.qualys.com/blogs/securitylabs/2014/03/19/https-mixed-content-still-the-easiest-way-to-break-ssl*.

Now that our connection is encrypted, we'll dive into other methods of securing the communications channel in the next section.

Origin-based Security Model

There has always been a race against those who seek to exploit vulnerabilities in a transport mechanism, and those that seek to protect it. The WebSocket protocol is no exception. When `XMLHttpRequest` (XHR) first appeared with Internet Explorer it was limited to the same origin policy (SOP) for all requests to the server. There are innumerable ways that this can be exploited but it worked well enough. As the usage of XHR evolved though, it became necessary to allow for accessing other domains. Cross Origin Resource Sharing (CORS) was the result of the effort to allow this which if used properly, can minimize cross-site scripting attacks while still allowing flexibility.

CORS or Cross-Origin Resource Sharing is a method of access control employed by the browser usually for AJAX requests from a domain outside of the originating domain. For further information about CORS see the Mozilla docs.

WebSocket doesn't place any same origin policy restriction on accessing WebSocket servers. It also doesn't employ CORS. What we're left with in regards to `Origin` validation is server-side verification. In all of our previous examples we use the simple and fast `ws` library with Node.js. We'll continue to do so and show in the initialization how simple it is to employ an origin check to ensure connection from the browser is only the expected `Origin`.

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({
      port: 8181,
      origin: 'http://mydomain.com',
        verifyClient: function(info, callback) {
            if(info.origin === 'http://mydomain.com') {
                callback(true);
                return;
            }
            callback(false);
        }
    }),
```

If we write a `verifyClient` function for the library we're using, we can send a callback with either `true` or `false` indicating a successful validation of any information, including the `Origin` header. Upon success, we will see a valid upgraded HTTP exchange for the `Origin` `http://mydomain.com`.

The HTTP exchange that happens as a result:

Request Header.

```
GET ws://mydomain.com/ HTTP/1.1
Origin: http://mydomain.com
Host: http://mydomain.com
Sec-WebSocket-Key: zy6Dy9mSAIM7GJZNf9rI1A==
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
```

Response Headers.

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

If the `Origin` header doesn't match up, the `ws` library will send back a 401 Unauthorized header. The connection will never complete the handshake and no data can be sent back and forth. If this happens you would receive a response similar to the following:

```
HTTP/1.1 401 Unauthorized
Content-type: text/html
```

It should also be noted that verifying the `Origin` header does not constitute a secure and authorized connection by a valid client. We could just as easily pass the proper `Origin` header from a script run outside of the browser. The `Origin` header can be spoofed with close to no effort at all outside of the browser. Therefore, additional strategies must be employed to ensure our connection is authorized.

The major benefit of requiring the `Origin` header is to combat WebSocket-like Cross-Site Request Forgery (CSRF) attacks coined Cross-Site WebSocket Hijacking (CSWSH) from being possible as the `Origin` header is passed by the user agent, and cannot be modified by JavaScript code. Our implicit trust therefore goes to the client browser in this instance, and the restrictions it places on web-based code.

Clickjacking

One of the other areas of concern with WebSocket and the web at large is termed Clickjacking. The process involves framing the client requested website and executing code in the hidden frame without the users awareness.

In order to combat this, web developers have devised methods called "Framebusting" to ensure that the website their users are visiting is not being framed in any way.

A simple and naive way to bust out of a frame:

```
if (top.location != location) {
    top.location = self.location;
}
```

This tends to fail however, due to inconsistencies in how browsers have handled these properties with JavaScript in the past. Other problems which creep up are availability of JavaScript on the system, or possibly in the `iframe` which can be restricted in certain browsers.

The most thorough JavaScript-based frame busting techniques available out there comes from a study by the Stanford Web Security Research on framebusting which is outlined in the snippet below for effective frame busting:

```
<style>
body { display: none; }
</style>

<script>
if(self===top) {
```

```
    documents.getElementsByTagName("body")[0].style.display = 'block';
  } else {
    top.location = self.location;
  }
  </script>
```

Of all the JavaScript based solutions this allows you to stop the user from viewing your page if it is being framed. The page will also remain blank if JavaScript is turned off and any of the other ways of exploiting the frame busting code is attempted. Due to the nature of WebSocket being JavaScript based, busting any frames will remove any ability for an attacker to hijack the browser and execute code without the users knowledge. Next we'll look at a header-based approach which can be used in conjunction with the script mentioned above, and a proof of concept called Waldo which takes advantage of this attack vector. Using the techniques mentioned here will render the Waldo code moot.

X-Frame-Options for frame busting

The safest method of getting around Clickjacking was introduced by Microsoft with Internet Explorer 8 and involves an HTTP Header option `X-Frame-Options`. The solution caught on and has become popular among all the major browsers out there including Safari, Firefox, Chrome, and Opera and has been officially standardized as RFC 7034. It remains the most effective way of busting out of frames. The table below shows the acceptable values that can be passed by the server to ensure that only acceptable policies are being used for framing the website:

Table 6-1. X-Frame-Options Acceptable Values

Header Value	Description of behavior
DENY	Prevents framing code at all
SAMEORIGIN	Prevents framing by external sites
ALLOW-FROM *origin*	Allows framing only by the specified site

Why does all this matter in regards to WebSocket communication? A proof of concept called Waldo shows how simple it can be for a compromised bit of JavaScript to control and report data back to a WebSocket server. A few of the things Waldo is able to achieve:

- Send back cookies or DOM
- Install and retrieve results of keylogger
- Execute custom JavaScript
- Use in a denial of service attack

Modern browsers all support WebSocket, and the only real defense against this attack vector is anti-framing countermeasures such as `X-Frame-Options` and to a lesser extent the other JavaScript based frame-buster code reviewed above.

If you'd like to test Waldo yourself, the website gives installation instructions. Please be aware that versions of supporting libraries that Waldo uses have advanced.

The supported versions for compiling Waldo:

- websocketpp WebSocket library (version 0.2.x)
- Boost with version 1.47.0 (can use package manager)

After installing `websocketpp` and downloading waldo.zip, simply modify the common.mk file with correct paths for `boost` and `websocketpp` and build. Create a simple html page which includes the compromised JavaScript in a hidden frame and load the regular website in another frame. Ensure you have the waldo c++ app running, and control at will. Waldo is completely relevant as it was released based on RFC 6455 and still works fine on the latest browsers.

For more in depth tools that allow you to use the browser as an attack vector including using WebSocket to perform these tests check out: BeEF which comes with a RESTful and GUI interface and XSSChef which installs as a compromised Google Chrome extension.

Denial of Service (DOS)

WebSocket by its very nature opens connections and keeps them open. An attack vector that has been commonly used with HTTP-based webservers is to open hundreds of connections and keep them open indefinitely by slowly trickling valid data back to the webserver to keep a timeout from occurring and exhausting the available threads used on the server. The term given to the attack is Slowloris, and while more asynchronous servers such as nginx can mitigate the effect, it is not always completely effective. Some best practices to look at to lessen this attack:

- Add IP-based limitation to ensure connections coming from a single source are not overwhelming the number of available connections
- Ensure that any actions being requested by a user are spawned asynchronously on the server-end to lessen the impact of connected clients.

Frame masking

The WebSocket protocol (RFC 6455), discussed in a lot more detail in Chapter 8 defines a 32-bit masking key which is set using the `MASK` bit in the WebSocket frame. The mask

is a random key chosen by the client, and is a best practice that all clients set the `MASK` bit along with passing the obligatory masking key. Each frame must include a random masking key from the client-side to be considered valid. The masking key is then used to XOR the payload data before sending to the server, and the payload data length will be unchanged.

You may be saying to yourself, that's great but why should I care about this when we're talking about security? Two words: cache poisoning. The reality of an app in the wild is you cannot control misbehaving proxy servers, and the relative newness of WebSocket unfortunately means that it can be an attack vector for the malicious.

A paper in 2011 titled "Talking to Yourself for Fun and Profit" outlined multiple methods for fooling a proxy into serving up the attackers JavaScript file. Masking in effect, introduces a bit of variability that is injected into every client message which cannot be exploited by an attackers malicious JavaScript code. Data masking ensures cache poisoning is less likely to happen due to variability in the data packets.

The downside of masking: it also prevents security tools from identifying patterns in the traffic. Unfortunately because WebSocket is still a rather new protocol, a good number of proxies, firewalls, network and endpoint DLP (data loss prevention) software are unable to properly inspect the packets being sent across the wire. In addition, since a lot of the tools out there don't know how to properly inspect WebSocket frames, data can be hidden in reserved flags, buffer overflows or underflows are possible, and hidden in the mask frame as well.

Trust no one.

Validating clients

There are many ways to validate clients attempting connection to your WebSocket server. Due to restrictions on the browser for connection over WebSocket, there is no ability to pass any custom HTTP headers during the handshake. Therefore the two most common methods of implementing auth is using the Basic header, and using form-based auth with a set cookie. In this example we will be employing the latter method and using a simple username/password form, setting and reading the cookie in our WebSocket server.

Setting up dependencies and inits

Since our solution is going to use shared keys via a cookie, we'll need to get some dependencies in place before we continue. The libraries we'll use are all listed below with the npm commands:

```
% npm install ws
% npm install express
```

```
% npm install body-parser
% npm install cookie
```

It is likely that you have the `ws` library installed in your environment from previous examples. We will be utilizing `express` and plugins for parsing form and cookie data: `body-parser` and `cookie`. The remaining dependency is `fs` which we'll use to read our TLS certificate files.

Back to our server code, and the first thing to do is setup our imports with `require`:

```
var fs = require('fs');
var https = require('https');
var cookie = require('cookie');
var bodyParser = require('body-parser');
var express = require('express');
var WebSocket = require('ws');
```

Now that we have all the necessary dependencies in place, we'll set up the self-signed certificate and initialize `express` and the https server backing it. You can use the same self-signed cert that we set up earlier in this chapter.

```
var WebSocketServer = WebSocket.Server;

var credentials = {
    key: fs.readFileSync('server.key', 'utf8'),
    cert: fs.readFileSync('server.crt', 'utf8')};

var app = express();

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true }));

var httpsServer = https.createServer(credentials, app);
httpsServer.listen(8443);
```

Listening for web requests

In order for our form-based auth to work, we will be serving up a login page, and a secured page which will require that a cookie named `credentials` is found, and has the proper key within. For brevity, we will use the username/password combination of `test`/`test` and use a pre-defined key that never changes. In your own code however, this data should be saved to a data source of your choosing that can also be retrieved by the WebSocket server. We'll use stub methods to show where you would insert the retrieval and storage code in whatever data source you decide to use in your own application.

Listed below is the HTML for our login example which we'll serve from our express server. We'll save this in our project directory and name it `login.html`.

```
<html>
<head>
```

```
<title>Login</title>
</head>
<body>
    <h1>Login</h1>
    <form method="POST" action="/login" name="login">
        Username: <input type="text" name="username" />
        Password: <input type="password" name="password" />
        <input type="submit" value="Login" />
    </form>
</body>
</html>
```

We will listen for `GET` and `POST` requests at the URL `/login`, and `GET` request for `/secured` which does a check on the cookie to ensure existence, and redirects if not found.

```
app.get('/login', function (req, res) {
    fs.readFile('./login.html', function(err, html) {
        if(err) {
            throw err;
        }
        res.writeHeader(200, {"Content-Type": "text/html"});
        res.write(html);
        res.end();
    });
});


app.post("/login", function(req, res) {
    if(req.body !== 'undefined') {
        key = validateLogin(req.body['username'], req.body['password']);
        if(key) {
            res.cookie('credentials', key);
            res.redirect('/secured');
            return;
        }
    }
    res.sendStatus(401);
});


var validateLogin = function(username, password) {
    if(username == 'test' && password == 'test') {
        return '591a86e4-5d9d-4bc6-8b3e-6447cd671190';
    } else {
        return null;
    }
}

app.get('/secured', function(req, res) {
    cookies = cookie.parse(req.headers['cookie']);
    if(!cookies.hasOwnProperty('credentials')
            && cookies['credentials'] !== '591a86e4-5d9d-4bc6-8b3e-6447cd671190') {
        res.redirect('/login');
```

```
    } else {
        fs.readFile('./secured.html', function(err, html) {
            if(err) {
                throw err;
            }
            res.writeHeader(200, {"Content-Type": "text/html"});
            res.write(html);
            res.end();
        });
    }
});
```

As you can see, we've stubbed out a `validateLogin` method which in our simple implementation just checks to ensure that the username and password are both `test`. After a successful validation it passes back the `key`. In a production implementation, I would opt for storing this key in a data store which can then be retrieved and validated with the WebSocket server end. We're cheating a bit to not add any unnecessary dependencies to this example. The HTML served up by the `/secured` endpoint is listed below.

```
<html>
<head>
<title>WebSocket Auth Example</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
<script type="text/javascript">

$(function() {
  var ws = new WebSocket("wss://localhost:8443");

  ws.onopen = function(e) {
    console.log('Connection to server opened');
  }
});
</script>
</head>
<body>
Hello, WebSocket.
</body>
</html>
```

Now we have all the web endpoints being served to the user, and a WebSocket client which is connecting securely over TLS to `port 8443`.

WebSocket server

We've handled the web end of the spectrum, and we've got a WebSocket client all loaded up and ready to send messages to our WebSocket endpoint. In our same source file, we'll include code to spin up a secure WebSocket server and use the `verifyClient` to check for our `credentials` cookie. First thing we do is ensure we are talking over a secure channel and if not, return false to the callback failing the connection. Then, we

check the cookie header and call the `checkAuth` function which in production code would look up the key in a data source and validate they indeed can access this service. If all goes well, we return `true` to the callback and allow the connection to proceed.

```
var checkAuth = function(key) {
    return key === '591a86e4-5d9d-4bc6-8b3e-6447cd671190';
}

var wss = new WebSocketServer({
    server: httpsServer,
    verifyClient: function(info, callback) {
        if(info.secure !== true) {
            callback(false);
            return;
        }
        var parsed_cookie = cookie.parse(info.req.headers['cookie']);

        if('credentials' in parsed_cookie) {
            if(checkAuth(parsed_cookie['credentials'])) {
                callback(true);
                return;
            }
        }
        callback(false);
    }

});
wss.on('connection', function( wsConnect ) {
    wsConnect.on('message', function(message) {
        console.log(message);
    });
});
```

As you can see from the above example, this is an end-to-end solution for validating that a WebSocket connection is authorized to continue. You can add other checks as needed depending on the application being built. Just remember that, the client browser cannot set any headers so cookies and the `Basic` header is all you are afforded. This should give you the structure to enable you to build this out into your own applications in a secure and safe manner away from prying eyes.

Summary

In this chapter we looked at various attack vectors and ways of securing your WebSocket application. The primary take-away should be to assume that the client is not a browser and do not trust it.

Three things to remember:

- Always use TLS

- Server code should always verify the `Origin` header
- Verify the request using a random token similar to a CSRF token for AJAX requests.

It is even more important to use the items discussed in this chapter so the possibility of someone hijacking the WebSocket connection for other nefarious purposes is heavily minimized. In the next chapter we'll review several ways of debugging WebSocket and actively measuring the performance benefits over a regular AJAX-based request.

CHAPTER 7
Debugging and Tools

In previous chapters we've gone in depth on building out solutions for using WebSocket in your applications. While in the process of integrating any technology into a new or existing project, perhaps the most vital is learning how to debug when things don't go as originally planned.

Throughout the chapter we will go through several areas of the WebSocket lifecycle and review several tools that can aid in our journey across the WebSocket landscape. Throughout you will get a better sense of what is happening behind the scenes.

Let's take one of our previous examples for a spin, and take a look at what's being passed around and how we can use the tools to see what's going on under the hood.

In a typical WebSocket lifecycle there are three main areas of the lifecycle: opening handshake, sending and receiving frames, and the closing handshake. Each of these can present their own challenges, and it would be impossible to outline each here, but we'll show some methods of investigating should some challenges present themselves while debugging.

The handshake

The expected data the server would receive from a valid client must include several HTTP headers like (`Host`, `Connection`, `Upgrade`, `Sec-WebSocket-Key`, `Sec-WebSocket-Version`) and others that are optional to WebSocket. There are some proxies and security tools on some corporate networks that may modify headers before they are transmitted to the server and could likely cause the handshake to fail. For our testing purposes we can utilize OWASP ZAP. ZAP was designed to assist penetration testers with finding vulnerabilities in web applications, and we can use it to intercept the handshake using its break functionality and remove some of the important headers before the server sees them.

Throughout this chapter we'll be utilizing the identity code example from Chapter 3. The full example for server and client are reproduced below:

The Server

Here is the complete code for the server portion of the identity chat application.

```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server,
    wss = new WebSocketServer({port: 8181});
var uuid = require('node-uuid');

var clients = [];

function wsSend(type, client_uuid, nickname, message) {
  for(var i=0; i<clients.length; i++) {
    var clientSocket = clients[i].ws;
    if(clientSocket.readyState === WebSocket.OPEN) {
      clientSocket.send(JSON.stringify({
        "type": type,
        "id": client_uuid,
        "nickname": nickname,
        "message": message
      }));
    }
  }
}

var clientIndex = 1;

wss.on('error', function(e) {
  console.log("error time");
});

wss.on('connection', function(ws) {
  var client_uuid = uuid.v4();
  var nickname = "AnonymousUser"+clientIndex;
  clientIndex+=1;
  clients.push({"id": client_uuid, "ws": ws, "nickname": nickname});
  console.log('client [%s] connected', client_uuid);

  var connect_message = nickname + " has connected";
  wsSend("notification", client_uuid, nickname, connect_message);

  ws.on('message', function(message) {
    if(message.indexOf('/nick') === 0) {
      var nickname_array = message.split(' ');
      if(nickname_array.length >= 2) {
        var old_nickname = nickname;
        nickname = nickname_array[1];
        var nickname_message = "Client " + old_nickname + " changed to " + nickname;
        wsSend("nick_update", client_uuid, nickname, nickname_message);
```

```
      }
    } else {
      wsSend("message", client_uuid, nickname, message);
    }
  });

  ws.on('error', function(e) {
    console.log("error happens");
  });

  var closeSocket = function(customMessage) {
    for(var i=0; i<clients.length; i++) {
        if(clients[i].id == client_uuid) {
            var disconnect_message;
            if(customMessage) {
                disconnect_message = customMessage;
            } else {
                disconnect_message = nickname + " has disconnected";
            }
          wsSend("notification", client_uuid, nickname, disconnect_message);
          clients.splice(i, 1);
        }
    }
  }
  ws.on('close', function() {
      console.log("closing socket");
      closeSocket();
  });

  process.on('SIGINT', function() {
      console.log("Closing things");
      closeSocket('Server has disconnected');
      process.exit();
  });
});
```

The Client

Here is the complete code for the client portion of the identity chat application.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Bi-directional WebSocket Chat Demo</title>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css"
<link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.mi
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

    <script>
```

```
            var ws = new WebSocket("ws://localhost:8181");
            var nickname = "";
            ws.onopen = function(e) {
              console.log('Connection to server opened');
            }
            function appendLog(type, nickname, message) {
              var messages = document.getElementById('messages');
              var messageElem = document.createElement("li");
              var preface_label;
              if(type==='notification') {
                  preface_label = "<span class=\"label label-info\">*</span>";
              } else if(type==='nick_update') {
                  preface_label = "<span class=\"label label-warning\">*</span>";
              } else {
                  preface_label = "<span class=\"label label-success\">" + nickname + "</span>";
              }
              var message_text = "<h2>" + preface_label + "  " + message + "</h2>";
              messageElem.innerHTML = message_text;
              messages.appendChild(messageElem);
            }

            ws.onmessage = function(e) {
              var data = JSON.parse(e.data);
              nickname = data.nickname;
              appendLog(data.type, data.nickname, data.message);
              console.log("ID: [%s] = %s", data.id, data.message);
            }
            ws.onclose = function(e) {
              appendLog("Connection closed");
              console.log("Connection closed");
            }
            ws.onerror = function(e) {
              appendLog("Error");
              console.log("Connection error");
            }
            function sendMessage() {
              var messageField = document.getElementById('message');
               if(ws.readyState === WebSocket.OPEN) {
                    ws.send(messageField.value);
               }
               messageField.value = '';
               messageField.focus();
            }
            function disconnect() {
              ws.close();
            }
        </script>

    </head>
    <body lang="en">
        <div class="vertical-center">
        <div class="container">
```

```
        <ul id="messages" class="list-unstyled">

        </ul>
        <hr />
        <form role="form" id="chat_form" onsubmit="sendMessage(); return false;">
            <div class="form-group">
            <input class="form-control" type="text" id="message" name="message" id="message"
              placeholder="Type text to echo in here" value="" autofocus/>
            </div>
    <button type="button" id="send" class="btn btn-primary"
      onclick="sendMessage();">Send Message</button>
      </form>
      </div>
      </div>
    <script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
    </body>
    </html>
```

Download and configure ZAP

Best way to follow along with the "bad proxies" test is to download ZAP at *https://code.google.com/p/zaproxy/wiki/Downloads?tm=2* and run it for your specific platform. ZAP can act as a proxy while you browse around your application so you'll need to modify the Network settings for your browser. With so many possible iterations it's best to just link off to ZAP's documentation which talks about a host of browsers and how to configure the proxy *https://code.google.com/p/zaproxy/wiki/HelpStartProxies*. The proxy by default runs on `localhost` port `8080` and can be changed by getting to the `Options` at `Tools > Options > Local proxy`.

In the ZAP client, select "Toggle break on all requests" so you can approve each request before it gets sent out. It is here that we'll modify our handshake and remove some vital and required headers. When visiting the local client by opening up the `client.html` file, it will attempt to make several HTTP connections. Some of these will be for external dependenices on bootstrap for the UI, and there will be one going to `http://localhost:8181` asking for an upgrading connection for WebSocket. There will be next buttons in the header of the UI allowing you to step through each request. When you get to the WebSocket request, stop and lets make some changes.

Here is a request similar to what you should see in ZAP:

```
GET http://localhost:8181/ HTTP/1.1
Host: localhost:8181
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: null
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (Macintosh; ...
```

```
Accept-Encoding: sdch
Accept-Language: en-US,en;q=0.8,de;q=0.6
Sec-WebSocket-Key: BRUZ6wGtxKWln5gToX4MSg==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

In the text area, remove all of the WebSocket-specific headers as a bad proxy or IDS might do:

```
GET http://localhost:8181/ HTTP/1.1
Host: localhost:8181
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Origin: null
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko)
Accept-Encoding: sdch
Accept-Language: en-US,en;q=0.8,de;q=0.6
```

After allowing the request to continue without the proper headers, you will see in the response from ZAP an HTTP 426 error code. This HTTP code indicates that an upgrade is required and was not provided. This can be a common occurrence when interacting with bad proxies which we discuss resolving in the section "WebSocket Secure to the rescue" on page 102 a bit later in this chapter.

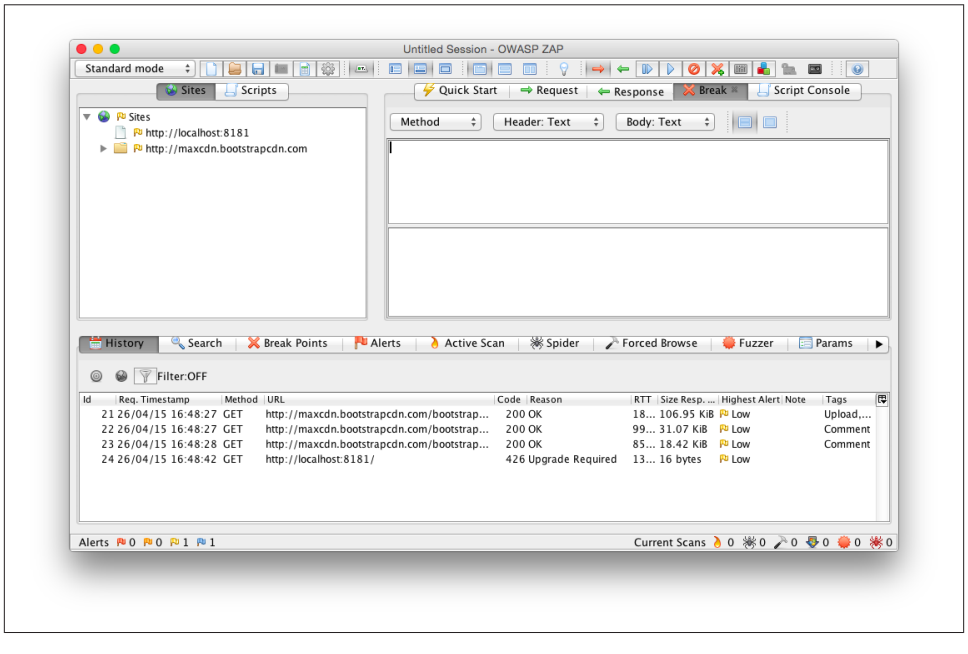


Figure 7-1. OWASP Zed Attack Proxy Breaking WebSocket handshake

Let's look at the Chrome Developer Tools and they should tell us a similar story. You may have to refresh the request and step through again with ZAP after getting to the dev tools > `Network`. You may need to click the "Filter" button and then specify `Web Sockets`. After re-submitting you should also see the `HTTP 426` response code being passed back to the browser.

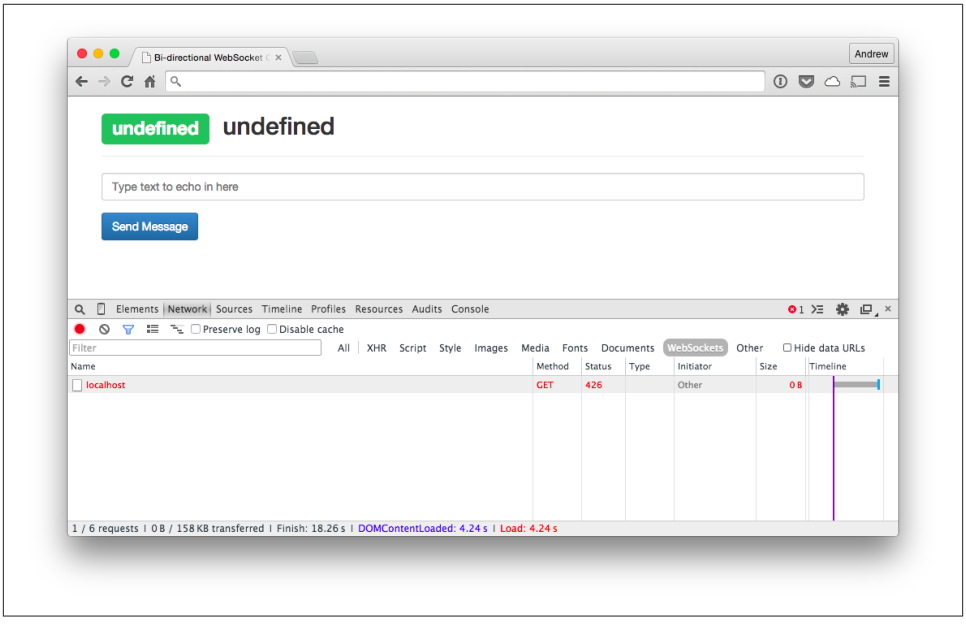


Figure 7-2. Handshake Under Chrome Developer Tools

What would happen if we didn't remove all headers, just something that may be important, like the `Sec-WebSocket-Version`? When the request comes in:

```
GET http://localhost:8181/ HTTP/1.1
Host: localhost:8181
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: null
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko)
Accept-Encoding: sdch
Accept-Language: en-US,en;q=0.8,de;q=0.6
Sec-WebSocket-Key: BRUZ6wGtxKWln5gToX4MSg==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

After submitting the above request back to the server, what will likely come back is an HTTP 400 Bad Request. We're missing some vital information (`Sec-WebSocket-`

`Version`) and it's not going to let us continue. How can we ensure that there is a better chance that our messages are going to get received and sent properly?

WebSocket Secure to the rescue

Thanks to a few neat tools we're able to see what's going on with our connection and why things are looking a bit wonky. How do we get around things like proxies or IDS tools mucking with our precious headers? WebSocket Secure is the answer. As we discussed in Chapter 6, the best way to ensure that your communication will reach its intended destination is to always use `wss://`. If you need help configuring it, refer back to Chapter 6 and follow the instructions in that section. In general, using the secure WebSocket channel can alleviate the issues we've seen above.

Validating the handshake

You will find that most libraries and all browsers with WebSocket RFC 6455 support will implement the simple handshake process without fail. As we will discuss in Chapter 8 the `Sec-WebSocket-Key` is a random nonce that is base64 encoded and sent in the initial handshake from the client. In order to validate your server is sending back the correct value to the client, you could take the example code in Chapter 8 and write a simple script which accepts a `Sec-WebSocket-Key` and spits out a proper response:

```
var crypto = require('crypto');

var SPEC_GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";

var webSocketAccept = function(secWebsocketKey) {
   var sha1 = crypto.createHash("sha1");
   sha1.update(secWebsocketKey + SPEC_GUID, "ascii");
   return sha1.digest("base64");
}

if(process.argv.length <= 2) {
  console.log("You must provide a Sec-WebSocket-Key as the only parameter");
  process.exit(1);
}

var webSocketKey = process.argv[2];

webSocketAccept = webSocketAccept(webSocketKey);
console.log("Sec-WebSocket-Accept:", webSocketAccept);
```

If you are seeing other values when using Wireshark or Chrome Developer Tools, values which would obviously be rejected by the client, run this script against the key first, and then see about fixing whatever may be in error with your server. It could indicate something along the path of communication is inserting itself in the communication and the protocol is doing the right thing by rejecting it.

Inspecting Frames

You will, on more than one occasion be tasked with figuring out why a client is receiving unexpected data coming back from your server. The first reaction to this may be to add some debug logging to your app and ask the client to make another attempt. If your code is in production however, this would be ill advised as it could affect other users and might affect performance or availability. Another option may be to use a network sniffer to watch the communication from the server to the affected client. Let's use our existing chat example in this chapter to see what is coming across along the wire.

Masked payloads

The best way to see each frame coming across the wire is to use our trusty tool Wireshark available at *https://www.wireshark.org/download.html*. That's right kids, Wireshark isn't just for sniffing network connections in a café! The versatile network tool runs well on every platform and allows you to filter and inspect the handshake along with each individual frame getting sent over the wire. As of version 1.9 it runs without needing the X11 dependency as well, which is definitely a bonus.

Getting started with Wireshark is fairly straightforward. After successfully downloading and installing the tool for your specific platform, you'll be greeted with the main screen which will list all network interfaces that Wireshark is ready to listen on.

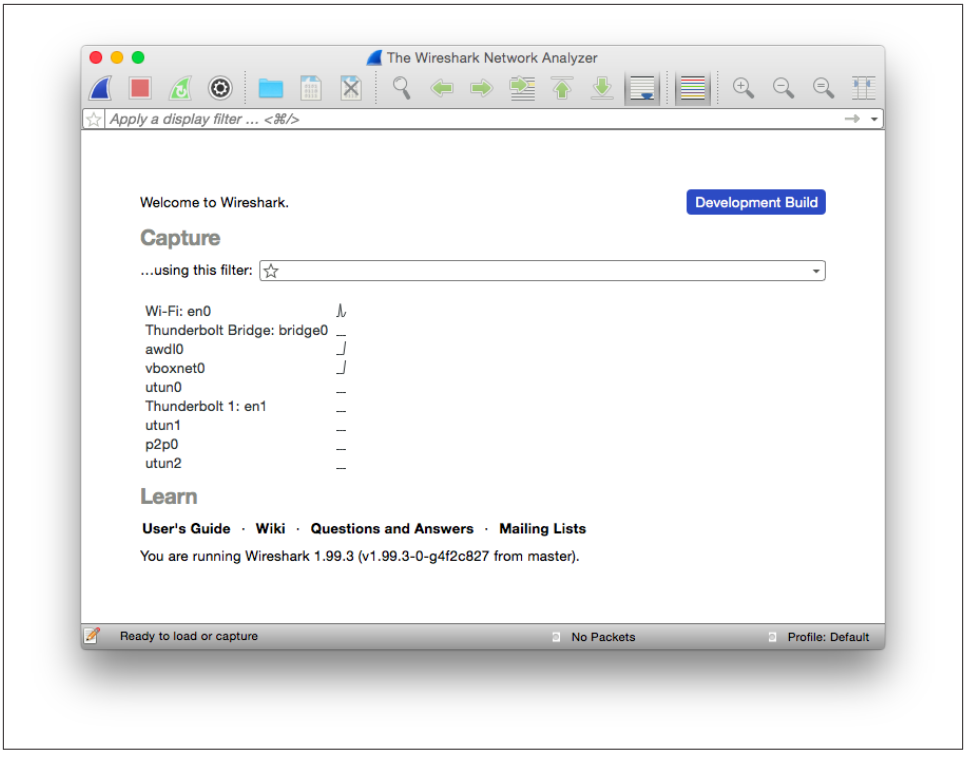


Figure 7-3. WebSocket main screen

Double-click on the interface that you will be browsing for these tests and Wireshark will dutifully start showing you captured packets in the next screen shown below. It shows each captured packet in a table with sortable columns and a filter bar at top to ease in viewing exactly what you're after in the huge stream of data flowing back and forth.

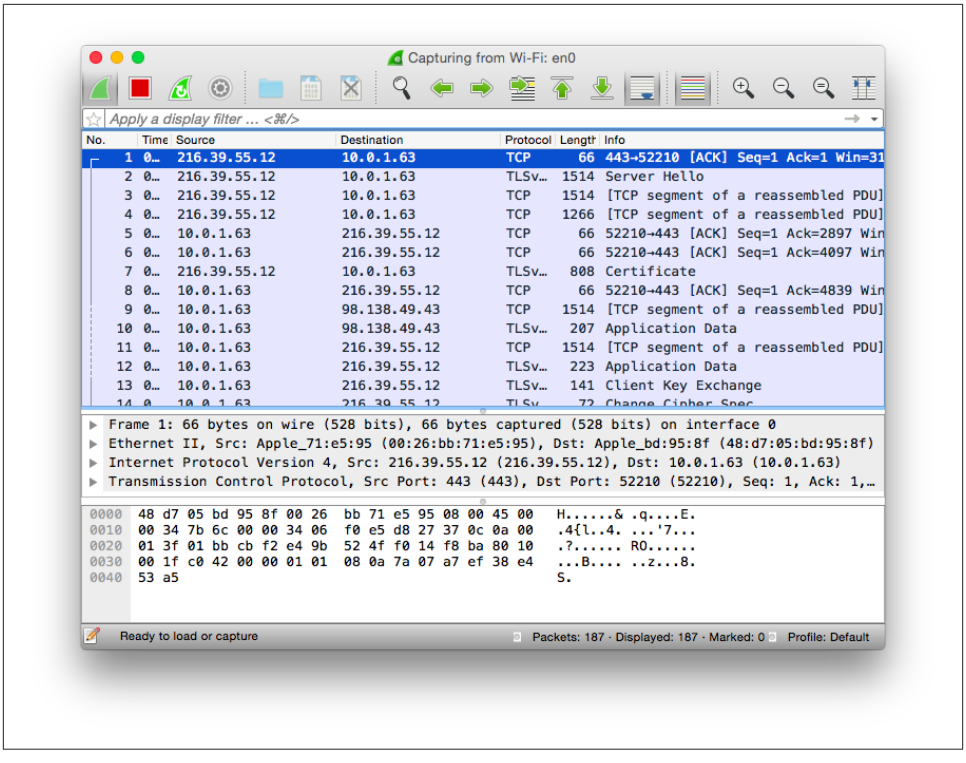


Figure 7-4. WebSocket capture screen

You can choose to follow a TCP, UDP or SSL stream to see the bidirectional communication being sent along the wire. We'll take a look at the handshake below and you'll see a request to the server from the client with the payload after the HTTP header.

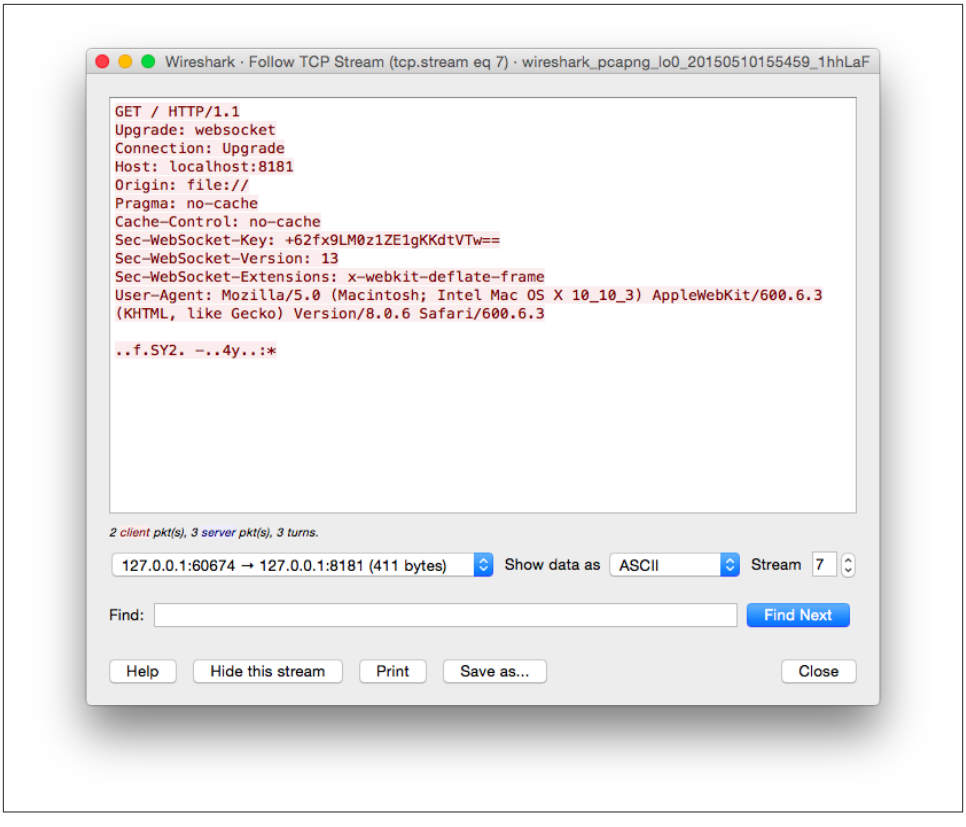


Figure 7-5. WebSocket frame client to server

We discussed briefly in Chapter 6 about frame masking, and if you're looking at the payload thinking it doesn't look like the JSON that you were expecting, you are correct. All clients before sending any message to a server in order to be considered valid, must mask the payload with the 32-bit masking key passed within the frame. You may be saying to yourself, this sucks, how am I supposed to effectively debug what is going on with this client with these antiquated views into my data! Have no fear, the latest versions of Wireshark support automatic unmasking of WebSocket requests to the server. Here's how to view it in Wireshark:

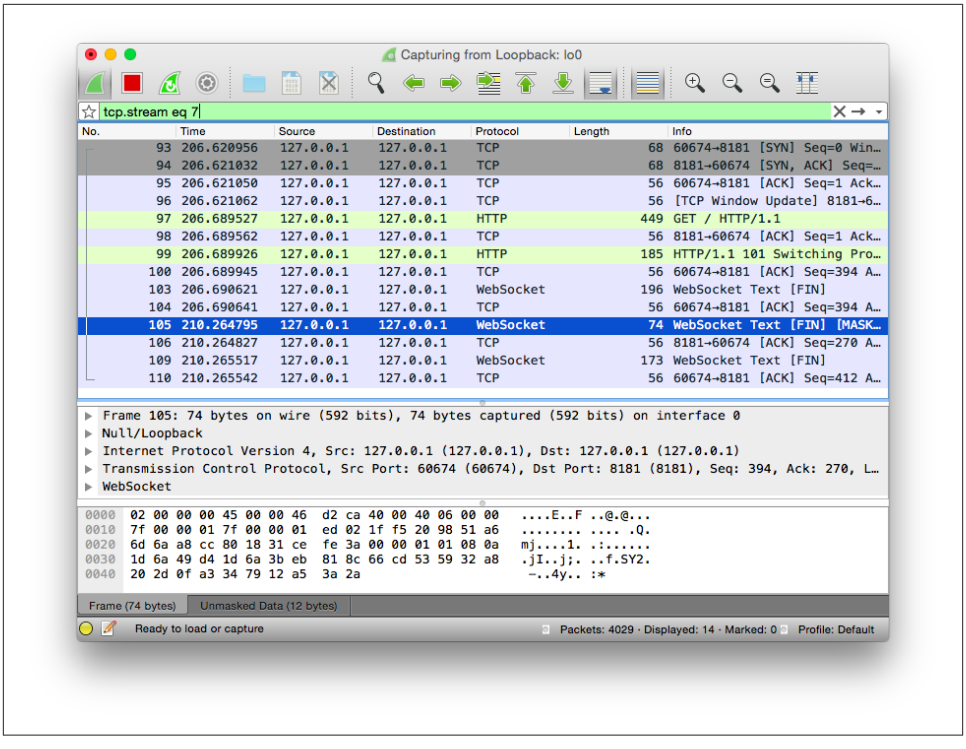


Figure 7-6. WebSocket masked frame

1. Find a frame to select which has [MASK] in the Info column
2. Ensure Packet Details is turned on and viewable
3. Expand the bottom most section labeled "WebSocket"
4. Expand the last section labeled "Unmask Payload" and behold our payload without masking

Below you'll see an example of the Wireshark UI and a sample unmasked payload along with the masked just above it should you need it. This is incredibly powerful for debugging the interactions with multiple clients and your server code.

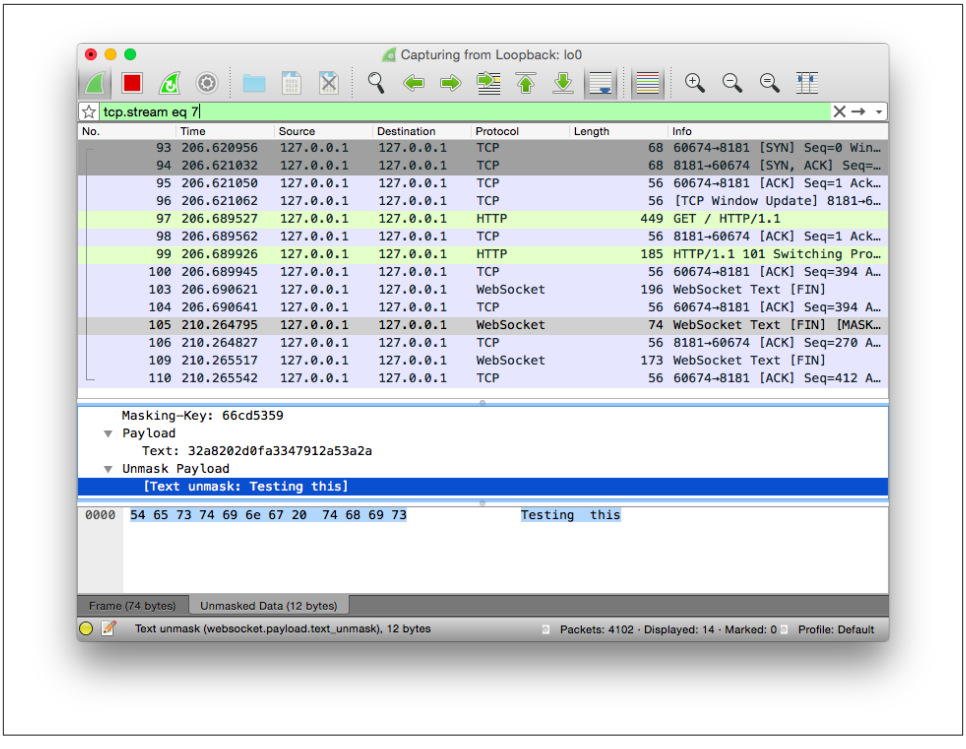


Figure 7-7. WebSocket unmasked payload

Now you can see for a specific message passed from client to server what the unmasked payload looks like without resorting to debugging to stderr/stdout or otherwise hampering performance or availability in your application. When looking at options, Chrome Developer Tools is also a very worthy companion for our efforts, though it does require that **you** are the client and can replicate the errors from your environment. One of the reasons I have found Wireshark to be so powerful in this regard, is the ability to watch the stream without modifying or interrupting other clients in the process.

While viewing specific unmasked payloads is interesting, other times it is most appropriate to see the entire thread of conversation for a specific client. For this, Wireshark is indispensable as well. While still capturing your WebSocket communication you can right-click or cmd-click on any of the WebSocket or the initial HTTP handshake in the communication, select Follow, and then Follow TCP Stream. At the moment of capture, it will show you the entire conversation for that specific client.

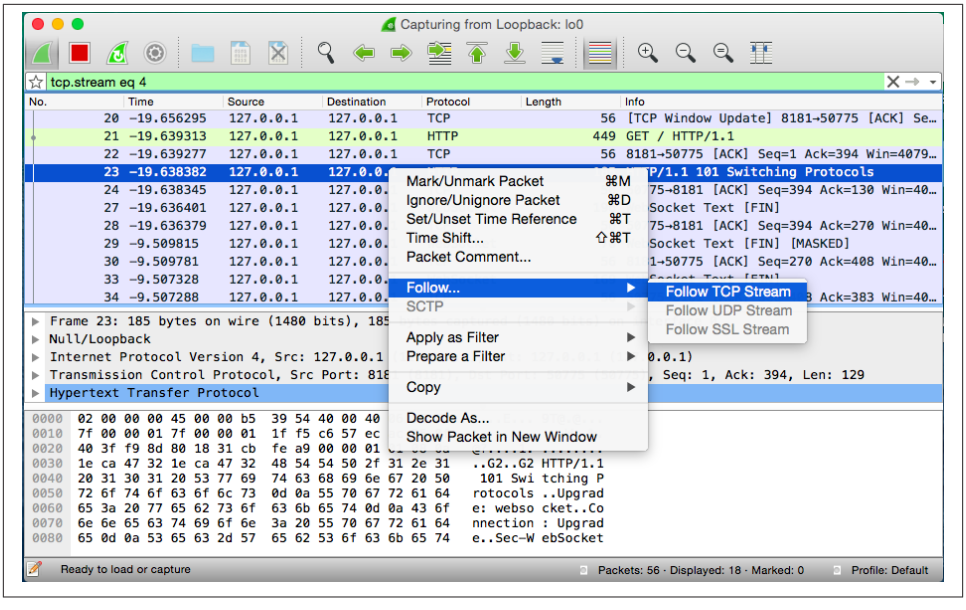


Figure 7-8. Wireshark follow TCP stream

As you look through the conversation being shown in the "Follow TCP Stream" window, browsing back to the main capture screen you should notice that the capture appears filtered. Whenever you follow a stream, it will pick that stream and filter out anything else so you can focus in. In the next section we will cover how to debug and watch for close frames using the Wireshark.

Closing connection

The final thing you will ever see in a WebSocket conversation would be the closing frame. If we were to add a button to our UI allowing the client to send a `close` frame to the server like so:

```
<button type="button" onclick="disconnect();">Disconnect</button>
```

We can then use some of the things we learned above about watching frames in Wireshark. Befitting our task would be to see the masked frame being sent by the client and looking at the very small message with the empty payload for a close request. If we view the frame, and open the WebSocket section from within Wireshark we should see something similar to the following:

```
WebSocket
  1... .... = Fin: True
  .000 .... = Reserved: 0x00
  .... 1000 = Opcode: Connection Close (8)
  1... .... = Mask: True
```

```
    .000 0000 = Payload length: 0
    Masking-Key: 4021df19
```

In Chapter 8 the registered status codes for a close for RFC-specific reasons was defined between 1000 and 1015. The header above indicates a normal close occurred, and no message was passed. If you would like to pass your own status code in the close, and/or a message, you can do so using the JavaScript API as we discussed in Chapter 2 with the following:

```
ws.close(1000, 'Ended normally');
```

The payload would be identical to how message are received normally including being masked, and sent along with the headers in the frame. The opcode would adorn the code passed in the JavaScript call, the mask would be set, and the masking key would be used to mask the message being sent to the client. That is the final communication you'll receive with a WebSocket conversation, and we've learned how to watch it all, and modify things as needed to test out different scenarios.

Summary

In this chapter we went through the opening handshake, the frames, and the closing handshake and learned about tools to watch the interaction between client and server. To recap some of the problems we can identify using these tools:

If you receive a code other than an `HTTP 200` during the opening handshake:

- It's likely that a bad proxy or IDS is involved and removing headers. It could indicate that you didn't use WebSocket Secure which is preferred over ever using the regular non-TLS connection.
- It could also indicate that something went wrong with the `Sec-WebSocket-Key` or the `Sec-WebSocket-Accept` that was sent in response. As we saw earlier in this chapter, we can watch for the request and response using either OWASP, Chrome Developer Tools, or Wireshark and run the values against the script we wrote in "Validating the handshake" on page 102

If a client is complaining of errors but you have no visibility due to masking:

- Use Wireshark and follow the instructions above which detail how to see the server responses and requests made by the client so you can fully understand where things are going wrong.

And finally if a close is occurring:

- Use any of the listed tools to look at what is being sent by the client, or responded to from the server, see the code and/or message and handle accordingly.

We were able to identify some potential pitfalls along the way, and how they could be identified using tools so they won't end up in a days long search during the debugging process. The tools should serve as worthy companions during the development process and during debugging when the app makes its way into production and you need a better view into what is going on.

You may notice that our debugging via browser method was focused exclusively on Chrome Developer Tools. Safari offers no discernible way to debug WebSocket frames at all. Firefox will show you the opening handshake and all headers associated with the connection, but no inspection available of the frame. According to a Mozilla bug 885508 it is still open and looks like there is no implementation available in the otherwise wonderful developer tools. We have plenty of tools available at our disposal though, Chrome along with Wireshark and OWASP ZAP can give us the introspection we need to find out what's going on when things go south.

The final chapter will be a deeper look at the WebSocket protocol itself.

CHAPTER 8
WebSocket Protocol

No discussion about protocols, especially ones that are initiated via an HTTP call would be complete without talking a bit about the history of HTTP. The inception of WebSocket came about because of the massive popularity of Ajax and real-time updates. HTTP which we'll talk about shortly, is a protocol where a client requests a resource, and the server responds with the resource or possibly an error code if something went wrong. This uni-directional nature has been worked around by using technologies like Comet, long polling, etc but come at a cost of computing resources on the server-side. WebSocket seeks to be one of the techniques that solves this problem and allows web developers to implement bidirectional communication over the same HTTP request.

HTTP 0.9 - The web is born

The birth of the World Wide Web brought rise to the first versions of the Hypertext Transfer Protocol (HTTP). The first version of HTTP was conjured up by Tim Berners-Lee in conjunction with the Hypertext Markup Language (HTML). HTTP 0.9 was incredibly simple. A client requests content via the GET method.

```
GET /index.html
```

The simplicity of HTTP 0.9 meant that you could request only two things, plain-text or HTML. This initial version of HTTP didn't have headers, so there was no ability to serve any media. In essence as a client you requested a resource from the server using TCP, and after the server was done sending it, the connection was closed.

HTTP 1.0 and 1.1

The simplicity in 0.9 was not going to last long. With the next version and final version of HTTP, the complexity involved in an HTTP request / response pair has grown. The later versions of HTTP added the ability to send HTTP headers with every request. With

that a growing number of headers to support things such as POST (form) requests, media types, caching, and authentication were added with HTTP 1.0. And in the next and final version, multihomed servers with the `Host` header, content negotiation, persistent connections, chunked responses, and much more which gets used today. The point of all this is, that as HTTP has grown in complexity, the size of headers has grown.

According to a whitepaper from Google talking about SPDY (*http://dev.chromium.org/spdy/spdy-whitepaper*), the average HTTP header is now 800 bytes and often times as large as 2KB. Compression and other techniques are readily available to simplify this situation

```
% curl -I http://www.google.com
HTTP/1.1 200 OK
Date: Wed, 20 May 2015 22:50:00 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=68769f4bb498a69f:FF=0:T...
Set-Cookie: NID=67=D26hM_BKWVnngC-7_1-XGmBR...
P3P: CP="This is not a P3P policy! See http..."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic,p=0
Transfer-Encoding: chunked
Accept-Ranges: none
Vary: Accept-Encoding
```

We have a header from accessing Google shown above. I took the liberty of removing the contents of the `cookie` header, and left how many characters it took up in the header. All told, the header was 850 characters long or just under 1KB. When we're looking to send data back and forth between server and client and vice-versa, having to send a 1KB header on top of that is unnecessary and wasteful. As we'll see after the initial handshake, a WebSocket frame header is miniscule in comparison and akin to opening up a TCP connection over HTTP.

In the sections below there will be code samples showing how to build out portions of the server protocol. Taken together and you can build your own implementation of a RFC compliant WebSocket server.

WebSocket Open Handshake

One of the many benefits of the WebSocket protocol is that it begins its connection to the server as a simple HTTP request. For browsers and clients that support WebSocket they'll send the server a request with specific headers that ask for a `Connection: Upgrade` to utilize WebSocket. The Connection Upgrade header was introduced in HTTP/1.1 to allow the client to notify the server of alternate means of communication available.

It is primarily used at this point as a means of upgrading HTTP to utilize WebSocket and can be used to upgrade to HTTP/2.

According to the WebSocket spec, the only indication that a connection to the WebSocket server has been accepted will be the header field `Sec-WebSocket-Accept`. The value, is a hash of a predefined GUID and the client HTTP header `Sec-WebSocket-Key`.

From RFC 6455

The |Sec-WebSocket-Accept| header field indicates whether the server is willing to accept the connection. If present, this header field must include a hash of the client's nonce sent in |Sec-WebSocket-Key| along with a predefined GUID. Any other value must not be interpreted as an acceptance of the connection by the server.

Sec-WebSocket-Key and Sec-WebSocket-Accept

First thing the spec asks for on the client-side for generating the `Sec-WebSocket-Key` is a nonce, or one-time random value. If you are using a browser that supports WebSocket, the below will be done for you automatically by using the JavaScript API. One of the security restrictions is that an XMLHttpRequest will not be allowed to modify that header. As we discussed in Chapter 6 this ensures that even if the website is compromised, we can trust that the browser will not allow any headers to be modified.

Generating the Sec-WebSocket-Key

The code below will assume running under Node.js and possibly using WebSocket to communicate with another service acting as the WebSocket server. You'll use a GUID generated using the `node-uuid` module which should prove to be random enough for our needs.

The only thing we're required to do at this point is base64 our nonce and include it in the HTTP headers for our WebSocket connection request. We will utilize the `node-uuid` module that we required earlier to create our random string.

```
var uuid = require('node-uuid');

var webSocketKey = function() {
    var wsUUID = uuid.v1();
    return new Buffer(wsUUID).toString('base64');
}
```

Responding with the Sec-WebSocket-Accept

On the server side, first thing we'll do is include the `crypto` module so we can send back our `SHA1` hash of the combined value.

```
var crypto = require('crypto');
```

RFC 6455 (*http://tools.ietf.org/html/rfc6455*) defines a predefined GUID which we'll define as a constant in our code.

```
var SPEC_GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
```

Our next task is to define a function in our JavaScript code which accepts the `Sec-WebSocket-Key` as a parameter, and creates a crypto SHA1 hash object.

```
var webSocketAccept = function(secWebsocketKey) {
   var sha1 = crypto.createHash("sha1");
```

Finally, we'll append the `Sec-WebSocket-Key` together with the predefined GUID, passing that into our SHA1 hash object. The update function will update the hash content with our combined data. We pass in `ascii` to identify the input encoding for the SHA1 update.

```
    sha1.update(secWebsocketKey + SPEC_GUID, "ascii");
    return sha1.digest("base64");
```

Generating the `Sec-WebSocket-Accept` header will usually be the job of a server library. It is a good idea to understand the inner workings and have a way of testing if something should go awry.

WebSocket HTTP Headers

The WebSocket connection must be an HTTP/1.1 GET request, and include the headers:

- **Host**
- **Upgrade**: `websocket`
- **Connection**: `Upgrade`
- **Sec-WebSocket-Key**
- **Sec-WebSocket-Version**

If any of the above is not met, the server should respond with an HTTP error code `400 Bad Request`. Here's an example of a simple HTTP request to upgrade for WebSocket. The arrangement of the headers is not as important as their existence.

```
GET ws://localhost:8181/ HTTP/1.1
Origin: http://localhost:8181
Host: localhost:8181
Sec-WebSocket-Key: zy6Dy9mSAIM7GJZNf9rI1A==
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
```

Table 8-1. Opening Handshake Headers

Header	Required	Value
Host	Yes	header field containing the server's authority
Upgrade	Yes	websocket
Connection	Yes	Upgrade
Sec-WebSocket-Key	Yes	Header field with a base64-encoded value that when decoded, is 16 bytes in length
Sec-WebSocket-Version	Yes	13
Origin	No	Optionally, an `Origin` header field. This header field is sent by all browser clients. A connection attempt lacking this header field SHOULD NOT be interpreted as coming from a browser client. Sending the origin domain in the upgrade is so connections can be restricted to prevent CSRF attacks similar to CORS for XMLHttpRequest
Sec-WebSocket-Accept	Yes (server)	Server sends back an acknowledgement which is described below and must be present for the connection to be valid
Sec-WebSocket-Protocol	No	Optionally, a `Sec-WebSocket-Protocol` header field, with a list of values indicating which protocols the client would like to speak, ordered by preference.
Sec-WebSocket-Extensions	No	Optionally, a `Sec-WebSocket-Extensions` header field, with a list of values indicating which extensions the client would like to speak. The interpretation of this header field is discussed in RFC 6455 Section 9.1.

The server upon receiving a valid upgrade request with all required fields, will decide on accepted protocol, any extensions, and send back an HTTP response with status code 101 along with the `Sec-WebSocket-Accept` handshake acknowledgement. Listed below is a very simple response from the server accepting the WebSocket request and opening up the channel to speak WebSocket.

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

Next we'll go over the actual WebSocket frame header in detail, at the bit level since the protocol is binary and not text.

WebSocket Frame

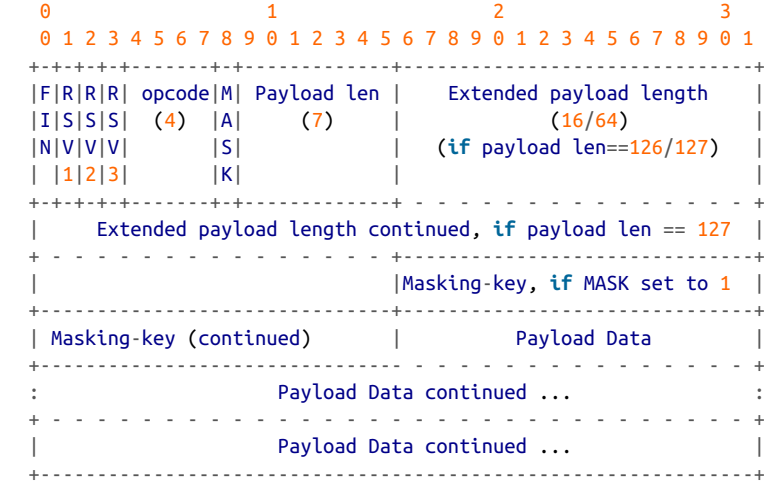
A WebSocket message is composed of one or more frames. The frame is a binary syntax which contains the following pieces of information each of which we will describe in greater detail. If you remember from Chapter 2, the specifics of the frame, fragmentation, masking, are all shielded and kept in the low-level implementation detail of the server and client-side. It is definitely good to understand though, as debugging WebSockets with this information makes things a lot more powerful than without it.

- Fin bit - is this the final frame or is there a continuation?

- Opcode - is this a command frame or data frame?
- Length - how long is the payload?
- Extended Length - if payload is larger than 125 we'll use the next 2 to 8 bytes
- Mask - is this frame masked?
- Masking Key - 4 bytes for the masking key
- Payload Data - the data to send whether binary or UTF-8 string, could be a combination of extension data + payload data

A WebSocket message may make up multiple frames depending on how the server and client decide to send data back and forth. And since the communication between client and server is bidirectional, at any time either side decides, data can be sent back and forth as long as no close frame was previously sent by either side.

Frame Headers.



Let's go through and talk about each of the header elements in greater detail.

Fin Bit

The first bit of the WebSocket header is the FIN bit. If the bit is set, then this fragment is the final in a message. If the bit is clear, the message is not complete with the following fragment. As we'll see in the frame opcode section, the opcode to pass is `0x00`.

Frame Opcodes

Every frame has an opcode that identifies what the frame represents. These opcodes are defined in RFC 6455 (*http://tools.ietf.org/html/rfc6455*). The initial values are as defined

by the IANA in the WebSocket registry (*http://www.iana.org/assignments/websocket/websocket.xml*) and are currently in use, additions to this are possible with WebSocket Extensions. The opcode is placed within the second 4-bits of the first byte of the frame header.

Table 8-2. Opcode Definition

Opcode Value	Description
0x00	Continuation frame, this frame continues the payload from the previous
0x01	Text frame, this frame includes utf-8 text data
0x02	Binary frame, this frame includes binary data
0x08	Connection Close Frame, this frame terminates the connection
0x09	Ping Frame, this frame is a ping
0x0a	Pong Frame, this frame is a pong
0x0b-0x0f	Reserved for future control frames

Masking

By default, all WebSocket frames are to be masked from the client end, and the server is supposed to close the connection if it receives a frame indicating otherwise. As we discovered in "Frame masking" on page 87, the masking introduces variation into the frame to prevent cache poisoning. The second byte of the frame is taken up by the length in the last 7 bits, and the first bit, indicates if the frame is masked or not. The mask to apply, will be the four bytes following the extended length of the WebSocket frame header. All messages received by a WebSocket server must be unmasked before further processing.

```
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;
}
```

Following unmasking, the server can decode UTF-8 for text-based messages (opcode 0x01) and deliver unchanged for binary messages (opcode 0x02).

Length

The payload length is defined by the last seven bits of the second byte of the frame header. The first byte is the opcode defined earlier. Depending on how long the payload ends up being, it may or may not use the extended length bytes that follow the first 2 header bytes.

- For messages under 126 bytes (0 - 125), the length is packed in the last seven bits of the second byte of the frame header
- For messages between 126 and 216, two additional bytes are used in the extended length following the initial length. A value of 126 will be placed within the first 7 bits of the length section to indicate usage of the following 2 bytes for length.
- For messages larger than 216, it will end up using the entire 8 bytes following the length. A value of 127 will be placed within the first 7 bits of the length section to indicate usage of the following 8 bytes for length.

Fragmentation

There are two cases where it may make sense to split up a message into multiple frames.

One case, is that without the ability to fragment messages, the endpoint sending would have to buffer the entire message before sending so it could send back an accurate count. Due to the ability to fragment, the endpoint can choose a reasonable sized buffer and when that is full, send another frame as a continuation until everything is complete.

The second case is in the case of multiplexing where it isn't desirable to fill the pipe with data that is being shared, and instead split up into several chunks before sending. Multiplexing isn't directly supported in the WebSocket protocol but there is a supported extension `x-google-mux` which can offer support for that. To learn more about extensions and how they relate to the WebSocket protocol check out "WebSocket Extensions" on page 123.

If a frame is unfragmented, the FIN bit is set and it contains an opcode other than 0x00. If fragmented, the same opcode must be used when sending each frame until the message has been completed. In addition, the FIN bit would be 0x00 until the final frame which would be empty other than the FIN bit set and an opcode of 0x00 used.

If sending a fragmented message, there must be the ability to interleave control frames when either side is accepting communication (if a large message was sent and a control frame wasn't able to be sent until the end, it would be fairly inefficient). And the last necessary thing to remember, is the fragmented message must be all of the same type, no mix and match of binary and utf-8 string data within a single message.

WebSocket Close Handshake

The closing handshake for a WebSocket connection requires a frame to be sent with the opcode of `0x08`. If the client sends the close frame, it must be masked as is done in all other cases from the client, and not masked coming back from the server. In addition to the opcode, the close frame may contain a body that indicates a reason for closing in the form of a code and a message. The status code is passed in the body of the message

and is a 2-byte unsigned integer. The remainder reason string would follow and as with WebSocket messages, would be a UTF-8 encoded string.

Each of the registered status codes in the RFC are identified and described in the next section.

Table 8-3. WebSocket Registered Status Codes

Status Code	Meaning	Description
1000	Normal Closure	Send this code when your application has successfully completed
1001	Going Away	Send this code when either the server or client application is shuttdng down or closing without expectation of continuing
1002	Protocol error	Send this code when connection is closing with a protocol error
1003	Unsupported Data	Send this code when your application has received a message of an unexpected type that it cannot handle
1004	Reserved	Do not use, this is reserved as per RFC 6455
1005	No Status Rcvd	Do not use, the API will use this to indicate when no valid code was received
1006	Abnormal Closure	Do not use, the API will use this to indicate the connection has closed abnormally
1007	Invalid frame payload data	Send this code if the data in the message received was not consistent with the type of the message (e.g., non-UTF-8)
1008	Policy Violation	Send this code when the message received has violated a policy. This is a generic status code that can be returned when there are no more suitable status codes.
1009	Message Too Big	Send this code when the message received was too large to process
1010	Mandatory Ext.	Send this code if you are expecting an extension from the server but it wasn't returned in the WebSocket handshake
1011	Internal Error	Send this code when the connection is terminated due to an unexpected condition
1012	Service Restart	Send this code indicating that the service is restarted, and a client that reconnects should do so with a randomized delay of 5 - 30s
1013	Try Again Later	Send this code when the server is overloaded and the client should either connect to a different IP (given multiple targets), or reconnect to the same IP when user has performed an action
1014	Unassigned	Do not use, this is unassigned but might be changed in future revisions
1015	TLS handshake	Do not use, this is sent when the TLS handshake has failed

Unlike TCP where connections can be closed at any time without notice, the WebSocket Close is a handshake on both sides. The RFC also identifies the ranges and what they mean categorically for your application. In general you'll be using the defined range for the current version `1000` through `1013`, and given any custom codes necessary in your application the unregistered range `4000` - `4999` are available.

If an endpoint receives a Close frame without sending one, it has to send a Close frame as its response (echoing the status code received). In addition, no more data can pass

over a WebSocket connection that has been sent a Close frame previously. There are certainly cases where an endpoint delays sending a Close frame until all of its current message is sent (in the case of fragmented messages), but the likelihood the other end would process that message is not guaranteed.

When an endpoint (client or server) has sent and received a Close frame, the WebSocket connection is closed and the TCP connection must be closed. A server will always close the connection after receiving and sending immediately, while the client should wait for a server to close, or set up a timeout to close the underlying TCP connection in a reasonable amount of time following a Close frame.

The IANA has a registry of the WebSocket Status Codes to use during the closing handshake. In most of the cases you'll be using the values between `1000` and `1013`. These are the registered codes being used and defined by the RFC.

Table 8-4. WebSocket Close Code Ranges

Status Range	Description
0 - 999	This range is not used for status codes
1000 - 2999	Status codes in this range are either defined by RFC 6455 or will be in future revisions
3000 - 3999	This range is reserved for libraries, frameworks, and applications
4000 - 4999	This range is reserved for private use, and are not registered with the IANA. Feel free to use them in your code between client and server with prior agreement

WebSocket Subprotocols

The RFC for WebSocket defines subprotocols and protocol negotiation between client and server. From the API chapter, you saw how to pass in one or more protocols via the JavaScript WebSocket API. Now that we're in the chapter dedicated to the innards of WebSocket, we can look at how that negotiation actually happens, or doesn't. At the lowest-level the negotiation of which protocol to use for a WebSocket connection happens via the HTTP header `Sec-WebSocket-Protocol`. This header is passed in with the initial upgrade request sent by the client:

```
Sec-WebSocket-Protocol: com.acme.chat, com.acme.anotherchat
```

In this instance, the client is telling the server that the two protocols it would like to speak are `chat` or `anotherchat`. At this point, it is up to the server to decide which protocol it will choose. If the server agrees with none of the protocols, it will return null or won't return that header. If the server agrees with a subprotocol, it will respond with a header such as:

```
Sec-WebSocket-Protocol: com.acme.anotherchat
```

If you remember in the chapter about the API, our JavaScript WebSocket object will have the property `protocol` populated with the value chosen by the server, or none if

nothing was chosen. In this instance, our API will have the value *com.acme.anotherchat* because the handshake response from the server indicates this as an acceptable protocol to communicate with. Any subprotocol doesn't change the underlying WebSocket protocol, but merely layers on top of it, providing a higher-level communication channel on top of the existing protocol. The ability to change the actual definition of a WebSocket frame is available to you however, in the form of "WebSocket Extensions" on page 123 described next.

From Chapter 2, if you remember there were 3 different types of subprotocols that could be used with the subprotocol handshake. The first was the registered protocols, identified in WebSocket RFC 6455, section 11.5 it defined a registry with the IANA (*http://www.iana.org/assignments/websocket/websocket.xml#subprotocol-name*). The second were open protocols such as XMPP or STOMP, although you can see registered protocols for these as well. And the third, which you'll likely use in your application is the Custom Protocols, which usually take the form of the domain name with an identifier for the subprotocol name.

WebSocket Extensions

The WebSocket RFC defines `Sec-WebSocket-Extensions` as an optional HTTP header to be sent by the connecting client asking if the server can support any of the listed extensions. The client will pass one or more extensions with possible parameters via the HTTP header and the server will respond with one or more accepted extensions. The server can choose only from the client-passed in list

Extensions have control to add new opcodes and data fields to the framing format. In essence you can completely change the entire format of a WebSocket frame with a WebSocket extension. In one of the earlier specs `draft-ietf-hybi-thewebsocketprotocol-10` there was even a `deflate-stream` extension mentioned, which would compress the entire WebSocket stream. The effectiveness of this is probably the reason it no longer shows up in later specs, as WebSocket has client-to-server frame masking where mask changes per frame, and with that, deflate would be wholly ineffective.

A few examples of extensions that are available in clients today

- **deflate-frame** which is a better method of deflate (available with Chrome which uses x-webkit-deflate-frame as its name) where frames are compressed at source and extracted at destination (*http://tools.ietf.org/html/draft-tyoshino-hybi-websocket-perframe-deflate-05*)
- **x-google-mux** which is an early stage extension supporting multiplexing (*http://tools.ietf.org/html/draft-tamplin-hybi-google-mux-00*)

The one caveat, and its been an issue with adoption of any new technology attached to browsers as clients, is support must be baked into the browsers used by your clients. The server will parse the extensions passed in by the client, and pass back the list it will support. The order of extensions passed back must coincide with what was passed in by the client. It must only pass back extensions that the client has indicated that it also supports.

Alternate Server Implementations

We have chosen in this book to focus exclusively on using Node.js on the server-side. Implementations of the WebSocket protocol on the server-side is widespread and covered in nearly every language imaginable. Covering any of these other server-side options is certainly outside the scope of this book. The list below is a non-exhaustive list of some of the RFC compliant implementations of WebSocket in the wild today for some of the most popular languages.

- Java API for WebSocket (JSR-356) which is included in any Java EE 7 compatible server such as Glassfish or Jetty
- Python has several options two of which are: pywebsocket available at *https://code.google.com/p/pywebsocket/* and ws4py at *https://ws4py.readthedocs.org/en/latest*
- PHP has a compatible implementation with Ratchet at *http://socketo.me*
- Ruby has an EventMachine based implementation em-websocket at *https://github.com/igrigorik/em-websocket*

These are just a few of the more popular in each language. As with any technical decision on the backend, evaluate the options available in your chosen platform and use the above and the data within this book as a guidepost along the way.

Summary

In this chapter we've gone into a lot of detail about the WebSocket protocol, and hopefully enough to either utilize as is, or extend in the form of subprotocols layered on top of the underlying WebSocket protocol. The WebSocket protocol has taken a long road to get to where it is today, and while there may be changes in the future, it appears that it is a solid way to communicate in a more efficient and powerful manner. It is time to do away with the historically necessary hacks of the past, and embrace the power provided by the WebSocket protocol and its API.