Socket.IO Cookbook

Socket.IO is a JavaScript library that provides you with the ability to implement real-time analytics, binary streaming, instant messaging, and document collaboration. Socket.IO Cookbook is a complete resource, covering topics from WebSocket security to scaling the server-side of a Socket.IO application and everything in between.

This book will provide real-world examples of how secure bi-directional, full-duplex connections that can be created using Socket.IO. It will also explain how the connection vulnerabilities can be resolved for large numbers of users and huge amounts of data/messages. With the help of the examples and real-world solutions, you will learn to create fast, scalable, and dynamic real-time apps.

What this book will do for you...

- Build rich and interactive real-time dashboards using Socket.IO
- Design chat and multiple-person interfaces
- Secure your data by implementing various authentication techniques, locking down the HTTP referrer, and using secure WebSockets
- Load balance across multiple server-side nodes
- Keep your WebSockets in sync using Redis, RabbitMQ or Memcached
- Stream binary data such as audio and video in real-time over a Socket.IO connection
- Create real-time experiences outside of the browser

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Socket.IO is a JavaScript library that provides you with the ability to implement real-time analytics, binary streaming, instant messaging, and document collaboration.

This book will provide real-world examples of how secure bi-directional, full-duplex connections that can be created using Socket.IO. It will also explain how the connection vulnerabilities can be resolved for large numbers of users and huge amounts of data/messages. With the help of the examples and real-world solutions, you will learn to create fast, scalable, and dynamic real-time apps.

Tyson Cadenhead

Over 40 recipes to help you create real-time JavaScript applications using the robust Socket.IO framework
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Wiring It Up'
- A synopsis of the book’s content
- More information on Socket.IO Cookbook
About the Author

Tyson Cadenhead works as a senior JavaScript engineer at Aloompa in Nashville, Tennessee. He has dedicated his professional career to building large-scale applications in JavaScript and Node. Tyson addresses audiences at various conferences and programming meetups on how to build real-time web applications with Socket.IO or Meteor.js. He blogs on topics such as JavaScript and web technologies at http://www.tysoncadenhead.com.

Tyson lives in the greater Nashville area with his wife and two sons, where he enjoys gardening, raising chickens, reading philosophy and economics books, and playing guitar.
Preface

Socket.IO is an excellent library for real-time messaging between the client side and the server side. Whether you want to create a chat room in your browser, reload your hybrid mobile application, or push fresh data to an internal dashboard, this book will show you how to do it.

What this book covers

Chapter 1, Wiring It Up, provides a quick introduction to Socket.IO. It tells you how to get up and running with a Node server. This chapter concludes with debugging tips for the server and the client.

Chapter 2, Creating Real-Time Dashboards, talks about how to stream data from the server to the client. It covers how to emit MongoDB data and how to handle Socket.IO connection timeouts.

Chapter 3, Having Two-Way Conversations, provides several recipes on how to build a two-way communication. From the quintessential chat room example to a fun recipe on how to create a real-time tic-tac-toe game, it includes several other topics.

Chapter 4, Building a Room with a View, explores views and namespaces and how they can be used to target your events to specific consumers.

Chapter 5, Securing Your Data, takes a look at how to secure the Socket.IO communication with various forms of authentication, including how to lock down the HTTP referrer and how to use secure web sockets.

Chapter 6, Performing a Load Balancing Act, covers various techniques for load-balancing Socket.IO, focusing on technologies such as Redis, Memcached, and RabbitMQ.
Chapter 7, *Streaming Binary Data*, explores topics ranging from emitting images as data to streaming video and audio.

Chapter 8, *Integrating With Mobile Applications*, talks about various techniques for using Socket.IO in mobile applications. It also provides a recipe for how to trigger hot deploys from Socket.IO.
In this chapter, we will cover the following recipes:

- Creating a Node HTTP server with Socket.IO
- Creating an Express server with Socket.IO
- Using Socket.IO as a cross-browser WebSocket
- Debugging on the client
- Debugging on the server

**Introduction**

Socket.IO is a powerful tool for creating real-time applications with bidirectional communication between the server side and the client side. It leverages the power of WebSockets along with several fallbacks, including JSON long polling and JSONP long polling through a single unified API. It can be used to create bidirectional interactions, such as real-time dashboards, chat applications, and multiplayer games.

In my previous jobs, I created several real-time JavaScript dashboards predating the Socket.IO library. During that time, I felt the pain of not having a good solution for true real-time communication. I found myself using hacks to obtain new data from the user interface. One method was to pound the server with an Ajax call every few seconds. The server had no way of knowing whether anything had updated since the last request, so it would dump all the data into the huge JSON object. It was up to the client-side JavaScript application to search the data and check whether there were any updates. If there were updates, the client side was responsible for updating the display as needed. This turned out to be difficult to maintain and a nightmare to debug. When Socket.IO was released, I was blown away. Now, I could send only the pieces of data that had actually been updated from the server instead of pushing up everything. Instead of setting an interval to make Ajax calls, I could just send data when new data came in. In short, Socket.IO made my life easier.
Socket.IO is an open source library created by Guillermo Rauch. It is built with Engine.IO, which is a lower-level abstraction on top of the WebSocket technology. Socket.IO is used to communicate bidirectionally between the server side and the client side in a syntax that looks as if you are just triggering and listening to events. The WebSocket API protocol was standardized in 2011. It is a Transmission Control Protocol (TCP) that only relies on HTTP for its initial handshake. After the handshake is complete, the connection is left open so that the server and the client can pass messages back and forth as needed.

For reference, a typical WebSocket connection without Socket.IO will look something similar to the following code on the client side:

```javascript
if ('Websocket' in window) {
    var ws = new WebSocket('ws://localhost:5000/channel');

    ws.onopen = function () {
        ws.send('Hello world');
    };

    ws.onmessage = function (e) {
        console.log(e.data);
    };

    ws.onclose = function () {
        console.warn('WebSocket disconnected');
    }
} else {
    throw new Error('This browser does not support websockets');
}
```

**Downloading the example code**

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Socket.IO goes a step beyond just providing an easy-to-use and more robust API on top of WebSockets. It also provides the ability to seamlessly use other real-time protocols if WebSockets are not available. For example, it will fall back on JSON long polling in the absence of WebSocket support. Long polling is essentially a trick to emulate the WebSocket behavior in browsers that don't support WebSockets. After a long polling request is made, it is held onto by the server instead of immediately responding as a traditional HTTP request would. When data becomes available, the long polling request is resolved, closing the loop of the long request cycle. At this point, a new long polling request will typically be made. This gives the illusion of the continuous connection that WebSockets provides. Although long polling is less than ideal in the landscape of modern technology, it is a perfect fallback when needed. When you send a message with Socket.IO, the API for WebSockets and long polling are identical, so you don't have to deal with the mental overhead of integrating two syntactically different technologies.

Although there are Socket.IO implementations in many server-side languages, we will use Node.js in this book. With Node.js, we can write JavaScript on the server side, which gives us a single syntax on the server and client.

In this chapter, we will create a Node server with Socket.IO and obtain some very basic cross-browser messaging working. We will also look at debugging tools that make working with Socket.IO even easier.

**Creating a Node HTTP server with Socket.IO**

In order to get Socket.IO running, we need to have at least one client and one server set up to talk to each other. In this recipe, we will set up a basic Node HTTP server with the built-in Node.js `http` module.

**Getting ready**

To get started with Socket.IO, you will need to install Node.js. This can be downloaded from https://nodejs.org/. There is a download link on the Node.js website, or you can get one of the binaries at https://nodejs.org/download/.

Once Node.js is installed, you will need to navigate to the directory where your project is located and create a new NPM package by entering `npm init` in your console.

Now, you will need to install Socket.IO. You can use NPM to install Socket.IO by entering `npm install socket.io --save` on your terminal.
How to do it...

To create a Node HTTP server with Socket.IO, follow these steps:

1. Create a new file called `server.js`. This will be your server-side code:

```javascript
var http = require('http'),
    socketIO = require('socket.io'),
    fs = require('fs'),
    server,
    io;

server = http.createServer(function (req, res) {
    fs.readFile(__dirname + '/index.html', function (err, data) {
        res.writeHead(200);
        res.end(data);
    });
});
server.listen(5000);

io = socketIO(server);

io.on('connection', function (socket) {
    socket.emit('greeting-from-server', {
        greeting: 'Hello Client'
    });
    socket.on('greeting-from-client', function (message) {
        console.log(message);
    });
});
```

2. You may see that `server.js` will read a file called `index.html`. You'll need to create this as well, as shown in the following code:

```html
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io('http://localhost:5000');
    socket.on('greeting-from-server', function (message) {
```
3. With your two files in place, you can start your server by entering `node server` on your terminal from the same directory where your files are. This will start a new Node server on port 5000. Node can listen on any port, but we will specifically tell it to listen on port 5000 in our `server.js` file. If you navigate to `http://localhost:5000`, you should see a message that says **Hello Client** in your browser:

4. You should also see a message on your terminal with an object that contains a message that says **'Hello Server'**:  

Congratulations! Your client and your server are now talking to each other.
How it works...

The browser displays a message that originated on the server, whereas the server displays a message that originated on the client. These messages are both relayed by Socket.IO.

The client side also initializes a function, but in the client's case, we need to pass a string containing the server and port number if the server is not running on port 80. In our case, we will run the server on port 5000, so we need to pass http://localhost:5000 in the io function.

The /socket.io/socket.io.js file is served dynamically by Socket.IO, so you don't need to manually add this file anywhere. As long as your server and Socket.IO are set up correctly, the script will be present. There is also a CDN available to host your socket.io.js file. The current version can be found at http://socket.io/download.

The io.on('connection') method in the server-side code listens for any new client-side socket connections. When the client loads a page with Socket.IO on the client side, a new connection will be created here.

When the server gets a new socket connection, it will emit a message to every available socket that says Hello Client. When the client gets this message, it will render it to the DOM. It also emits a message of its own that the server will listen for.

There's more...

Although all the examples in this book use Node.js on the server side, there are server-side libraries for many other languages, including, PHP, C#, Ruby, Python, and so on. Whatever your server-side language of choice happens to be, there is likely to be a library to interface with Socket.IO on your server.

Creating an Express server with Socket.IO

Express is probably the most widely used Node application framework available. Numerous MVC frameworks are written based on Express, but it can also be used on its own. Express is simple, flexible, and unopinionated, which makes it a pleasure to work with.

Socket.IO can be used based on the Express server just as easily as it can run on a standard Node HTTP server. In this section, we will fire the Express server and ensure that it can talk to the client side via Socket.IO.
Getting ready

The Express framework runs on Node, so you will need to have Node installed on your machine. Refer to the previous recipe for instructions on how to install Node and Socket.IO.

In addition to Node and Socket.IO, you will also need to install the Express npm package. Express can be installed by entering `npm install express --save` on your terminal.

How to do it...

Follow these steps to create an Express server using Socket.IO:

1. You will need to create a new server-side JavaScript file called `server.js`. It will contain all of your server instantiation and handle your Socket.IO messaging. The `server.js` file will look similar to the following code:

   ```javascript
   var express = require('express'),
       app = express(),
       http = require('http'),
       socketIO = require('socket.io'),
       server, io;

   app.get('/', function (req, res) {
       res.sendFile(__dirname + '/index.html');
   });

   server = http.createServer(app);
   server.listen(5000);

   io = socketIO(server);

   io.on('connection', function (socket) {
       socket.emit('greeting-from-server', {
           greeting: 'Hello Client'
       });
       socket.on('greeting-from-client', function (message) {
           console.log(message);
       });
   });
   ```
2. The `server.js` file will serve a static HTML file called `index` when the user navigates to the root directory of the server. The HTML file will handle the client-side Socket.IO messaging. It will look similar to the following code:

```html
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io('http://localhost:5000');
    socket.on('greeting-from-server', function (message) {
        document.body.appendChild(document.createTextNode(message.greeting));
    });
    socket.emit('greeting-from-client', { greeting: 'Hello Server' });
</script>
</body>
</html>
```

3. Once both of your files are created, you can start your server by entering `node server` on your terminal.

4. After the server starts, you should be able to navigate to `http://localhost:5000` and see a message that says **Hello Client**:
5. There should be a message that says ‘Hello Server’ on your terminal:

```
$ node server

{ greeting: 'Hello Server' }
```

Awesome! Now you've got Socket.IO running on Express.

**How it works...**

Express is a collection of HTTP utilities and middleware that make it easier to use Node as a web server. Although Express provides a robust API that isn't available out of the box from the built-in Node HTTP module, using Express with Socket.IO is still very similar.

We created a new Express server with `var app = express()`. We passed this to the `http.Server()` method. By passing our Express app as the first argument to the HTTP server, we told Node that we wanted to use Express as our handler for HTTP requests.

Next, we passed the HTTP server directly to the `SocketIO` method exactly as we would have if we were using a nonExpress HTTP server. Socket.IO took the server instance to listen for new socket connections on it. The new connections came from the client side when we navigated to the page in our browser.

**See also**

Creating a Node HTTP server with Socket.IO.

**Using Socket.IO as a cross-browser WebSocket**

The native WebSocket implementation in browsers is much less robust than what Socket.IO offers. Sending a WebSocket message from the client only requires the data as a single argument. This means that you have to format your WebSocket data in such a way that you can easily determine what it is for.

If you want to emulate the ease of sending a message without a topic, you can use the `socket.send()` method to send messages as needed.
The benefits of using the Socket.IO syntax for this type of interaction over plain WebSockets are numerous. They include the built-in fallbacks for browsers that don’t support WebSockets. The benefits also include a single unified syntax that is easier to read and maintain.

**Getting ready**

To get started with Socket.IO as a cross-browser WebSocket, you will need to have Node, Express, and Socket.IO installed. If you have not installed them yet, refer to the previous recipe: *Creating an Express server with Socket.IO*.

**How to do it…**

Follow these instructions to use Socket.IO as a cross-browser WebSocket:

1. First, you'll need to set up your server-side `server.js` file as follows:

   ```javascript
   var io = require('socket.io')(5000),
   sockets = [];

   io.on('connection', function (socket) {
     sockets.push(socket);
     socket.on('message', function (message) {
       for (var i = 0; i < sockets.length; i++) {
         sockets[i].send(message);
       }
     });
   socket.on('disconnect', function () {
     console.log('The socket disconnected');
   });
   });
   ```

2. Next, you'll have to create a client-side `index.html` file with the following code:

   ```html
   <!doctype html>
   <html>
   <head></head>
   <body>
   <form id="my-form">
   <textarea id="message" placeholder="Message"></textarea>
   <p>
   <button type="submit">Send</button>
   ```
3. In our example, we have a simple form that allows the user to post a message that will be sent to all the connected sockets.
4. If you start your server with `node server` and open your `index.html` file by navigating to `http://5000/index.html` in your browser, you should see the form on the index page:

![Image of form on index page]

5. If you post a message to the form, it should send it to the server, and the server should broadcast it to all the available clients, as shown in the following screenshot:

![Image of broadcast message]

**How it works...**

The `socket.send(...)` method is a shortcut for `socket.emit('message', ...)`. We will take a look at this topic in Chapter 3, *Having Two-Way Conversations*. This is the reason when the server listens for a message topic, it gets called when the client calls `socket.send()`.

---

12
Our server stores an array of all the topics that connect to it. We will loop through all the connected sockets to send the message when it comes. We will also explore better ways to manage the connected sockets in the next chapter.

The client side aids the duty of sending the data from the form to the server. It also listens for new messages from the server to add to the list of available messages in our UI underneath the form.

**There's more...**

We will keep an array of all the connected sockets so that we will be able to easily send data to all of them as needed. However, keeping an array of all the connected sockets can be a little more tedious than just pushing the sockets to the array when they connect. For example, if a user leaves the page, the socket will disconnect, but it will still be included in the static array.

Fortunately, we will be able to tap into the socket disconnect event by calling `socket.on('disconnect')`. Using this method, we can remove the socket from our array and avoid sending messages to an abandoned socket connection.

Here is an example of how the disconnect event can be used to manage dropped connections:

```javascript
var io = require('socket.io')(5000),
    sockets = [];

io.on('connection', function (socket) {
    sockets.push(socket);
    socket.on('disconnect', function () {
        for (var i = 0; i < sockets.length; i++) {
            if (sockets[i].id === socket.id) {
                sockets.splice(i, 1);
            }
        }
        console.log('The socket disconnected. There are ' + sockets.length + ' connected sockets');
    });
});
```

**See also**

- The *Handling connection timeouts* section in *Chapter 2, Creating Real-Time Dashboards*
- The *Creating a simple chat room* section in *Chapter 3, Having Two-Way Conversations*.  


Debugging on the client

In the earlier versions of Socket.IO, debugging was extremely simple. This was because verbose logging was pushed to the developer console by default. Although this feature was a great way to dig into issues when they occurred, it could also get in the way by logging too much when no debugging was needed.

Now, Socket.IO gives us the ability to toggle certain parts of our logging on and off as needed. In this recipe, we will enable client-side debugging to have a better view of what is happening in our Socket.IO communication.

Getting ready

Starting with version 1.0, Socket.IO doesn't show any logging by default. However, it can easily be turned on. Behind the scenes, it will use an NPM module called debug, which allows logging to navigate to various scopes that can be turned on or off as needed.

How to do it...

To enable debugging on the client side, follow these steps:

1. On the client side, log settings are persisted through HTML5 localStorage, so you can turn logging on by setting the value of localStorage.debug.
2. To see all the logging messages, just set the value of debug to an asterisk, as shown in the following code:
   ```javascript
   localStorage.debug = '*';
   ```
3. Now that robust logging is turned on, you can open your developer tools and see a rich array of messages that details what is happening under the hood:

   ```javascript
   socket.io-client: url: http://localhost:5000 +msg
   socket.io-client: new 1s instance for http://localhost:5000 +msg
   socket.io-client:manager: readyState=0 +msg
   engine.io-client:socket: creating transport "polling" +msg
   engine.io-client:polling: poll =msg
   engine.io-client:polling: xhr poll =msg
   ```

   ```javascript
   xhr=3&transport=polling&t=1430183528367
   ```

   ```javascript
   engine.io-client:polling: xhr data null +msg
   engine.io-client:socket: setting transport polling =msg
   socket.io-client:manager: connect attempt will timeout after 20000 +msg
   socket.io-client:manager: readyState=0 +msg
   ```

   ```javascript
   XHR finished loading: GET "http://localhost:5000/socket.io/?EIO=3&transport=polling&t=1430183528367&token=85278666f52f15cb", +msg
   ```

   ```javascript
   ```
How it works...

The `localStorage` object in the browser is an object with key/value pairs that is maintained when you refresh the page or leave it entirely. It is useful for persisting data on the client side in modern browsers.

Socket.IO uses the debug NPM module. This views the `localStorage` key to determine the logging level to be displayed in the browser. The fact that the debugging level is set in `localStorage` can be very useful because you can set a debugging type anywhere even in production, and it will only log on to your machine. Also, you will be able to refresh the page and see the Socket.IO logging from the initial page load, which can be really handy for debugging events that occur earlier on the page life cycle.

There's more...

Not only can you set the logging to show everything, you can also listen for only certain log types by setting them up in `localStorage`. For example, if you are only interested in XHR requests, you can ask to only see messages in the `engine.io-client:polling-xhr` namespace with the following code:

```javascript
localStorage.debug = 'engine.io-client:polling-xhr';
```

You can also set multiple log types by separating them with a comma, as shown in the following code:

```javascript
localStorage.debug = 'engine.io-client:polling, engine.io-client:socket';
```

See also

The following recipe, *Debugging on the server*.

Debugging on the server

The same debugging package that is available on the client is available on the server as well.

The debugging option can be turned on with a Node environmental variable.

Getting ready

To get started with debugging on the server side, you will need to have Node and Socket.IO installed and an existing app that uses Socket.IO. To test this out, you can easily use any of the apps we built in the previous recipes in this chapter.
How to do it...

To get server-side debugging turned on, follow these steps:

1. To enable debugging at the time when you start your server, simply include the `DEBUG` environmental variable as the first argument when you start your Node server, as shown in the following code:
   ```
   DEBUG=* node server
   ```

2. If you would like to persist the `DEBUG` environmental variable without the need to pass it every time you start your Node server, you can export it ahead of time using the following code:
   ```
   export DEBUG=* 
   ```

3. Now, when you start your server, verbose logging will be used with the following code:
   ```
   node server
   ```

4. You can always update the `DEBUG` variable or even remove it completely by setting it to `null`, which will suppress logging entirely, as shown in the following code:
   ```
   export DEBUG=null
   ```

How it works...

Node.js environmental variables are available in `process.env` in any running Node process. They are often used to set up server-specific configurations, such as database connections and third-party credentials.

The great thing about using environmental variables to define the logging verbosity is that most cloud-based hosting providers allow you to change environmental variables on the fly, so you can easily toggle logging on or off without having to redeploy your code.

There's more...

Similar to client-side logging, you can set the logging type to something other than the wildcard. This allows you to only get debugging messages on the topic you want to listen to.

For example, listening for XHR requests is as simple as passing it to the environmental variables when you start your Node server with the following code:

```
DEBUG=socket.io:server node server
```
Where to buy this book
You can buy Socket.IO Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.