1

Wiring it Up

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 WebScoket
* 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 such as 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.

At my previous jobs, I created several real-time JavaScript dashboards pre-dating 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 get new data into the user interface. One method was pounding 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 of the data into huge JSON object. It was up to the client-side JavaScript application to look through the data and check if 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 good again.

Socket.IO is an open-source library created by Guillermo Rauch. It is built on top of 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 client-side in a syntax that looks like 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 looks something like this on the client-side. Please note that we are not falling back for browsers that do not support WebSockets.

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');

}

Socket.IO goes a step beyond just providing an easier to use, 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 like a traditional HTTP request. When data becomes available, the long-polling request is resolved, closing the loop of the long request cycle. At that point, a new long-polling request will typically be made. This gives the illusion of the continuous connection that WebSockets provides. While long-polling is less than ideal in the landscape of modern technology, it is a perfect fallback if it is 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.

While there are Socket.IO implementations in many server-side languages, we will be using Node.js in this book. With Node, 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 get 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 using the built-in Node http module.

## Getting Ready...

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

Once Node is installed, you will need to go into the directory where your project is located and create a new NPM package by typing npminitinto 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 into your terminal.

## How To Do It...

To create a Node http Server with Socket.IO, follow these steps.

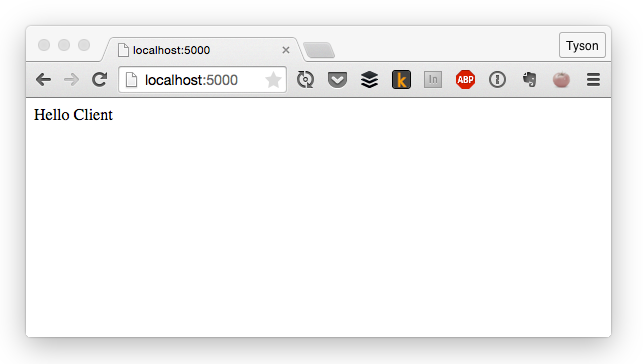
1. Create a new file called server.js. This will be your server-side code.

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);  
 });  
});

1. You might notice that server.js is reading a file called index.html. You'll need to create that as well.

<!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>

1. With your two files in place, you an start your server by typing node server in 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 are specifically telling it to listen on port 5000 in our server.js file. If you go to http://localhost:5000, you should see a message that says "Hello Client" your browser.



Insert Image B04893\_01\_01.png

1. You should also see a message in your terminal with an object containing a message that says "Hello Server".



Insert Image B04893\_01\_02.png

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 and the server displays a message that originated on the client. These messages were both relayed by Socket.io

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

The /socket.io/socket.io.js file is served up dynamically by Socket.IO, so you don't need to manually add that file anywhere. As long as your server and Socket.IO are set up correctly, the script will be there. 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 using 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 that message, it will render it to the DOM. It also emits a message of its own that the server listens for.

## There’s More

While all of 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 more. 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 on top of 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 on top of an Express server just as easily as it can run on a standard Node http server. In this section, we will get an Express server fired up and ensure that it can talk to the client-side via Socket.IO.

## Getting Ready...

Express is run on top of Node, so you will need to have Node installed on your machine. See the previous recipe for instructions on installing Node and Socket.IO.

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

## How To Do It...

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

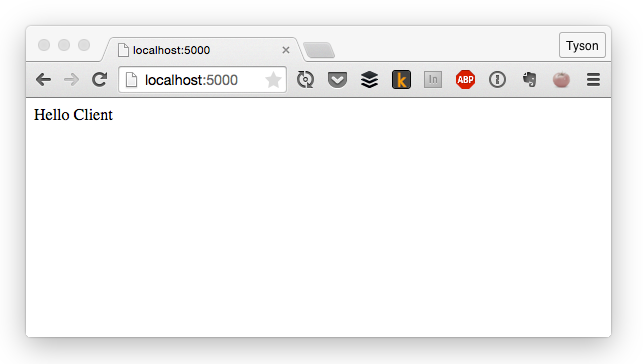
1. You will need to create a new server-side JavaScript file called "server.js". It will contain all of your server instantiation and handle your Socket.IO messaging. The server.js file will look like this:

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.Server(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);  
 });  
});

1. The server.js file is serving up a static html file called "index" when the user navigates to the root directory of the server. The html file will handle client-side Socket.IO messaging. It will look like this:

<!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>

1. Once both of your files are created, you can start up your server by typing node server in your terminal.
2. After the server starts up, you should be able to navigate to http://localhost:5000 and see a message that says "Hello Client":



Insert Image B04893\_01\_03.png

1. In your terminal, there should be a message that says "Hello Server".



Insert Image B04893\_01\_04.png

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. While 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 using var app = express() which we passed into the http.Server() method. By passing our Express app in 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 into the SocketIO method exactly like we would have if we were using a non-Express 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

* *CreatingCreating 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 itself as a single argument. That means that you have to format your WebSocket data in such a way that you can easily determine what it is for.

In cases where 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 using 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:

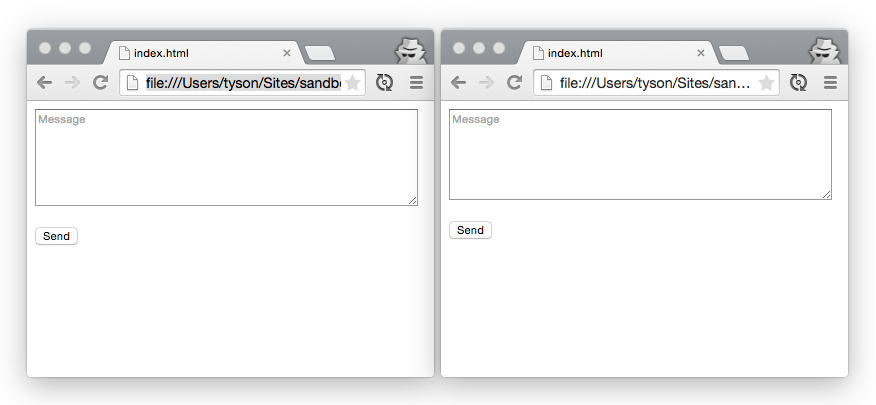
1. First, you'll need to set up your server-side "server.js" file:

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');  
 });  
});

1. Next you'll need to create a client-side "index.html" file:

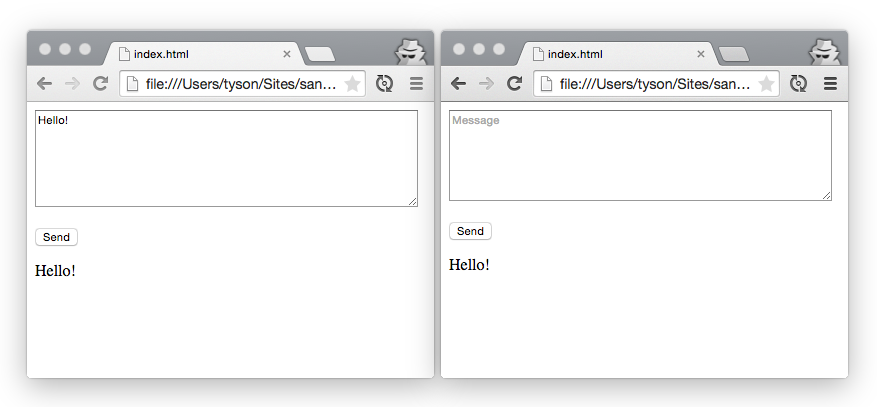
<!doctype html>  
<html>  
<head></head>  
<body>  
<form id="my-form">  
<textarea id="message" placeholder="Message"></textarea>  
<p>  
<button type="submit">Send</button>  
</p>  
</form>  
  
<div id="messages"></div>  
  
<script src="http://localhost:5000/socket.io/socket.io.js"></script>  
<script>  
 var socket = io('http://localhost:5000');  
 socket.on('connect', function () {  
  
 document  
 .getElementById('my-form')  
 .addEventListener('submit', function (e) {  
 e.preventDefault();  
 socket.send(document.getElementById('message').value);  
 });  
  
 socket.on('message', function (message) {  
 var messageNode = document.createTextNode(message),  
 messageElement = document.createElement('p');  
  
 messageElement.appendChild(messageNode);  
  
 document.getElementById('messages').appendChild(messageElement);  
 });  
 });  
</script>  
</body>  
</html>

1. In our example, we have a simple form that allows the user to post a message that will be sent to all of the connected sockets.
2. If you start your server with node server and open your index.html file by going to <http://5000/index.html> in your browser, you should see the form in the index page.



Insert Image B04893\_01\_05.png

1. If you post a message to the form, it should send it to the server and the server should broadcast it to all of the available clients.



Insert Image B04893\_01\_06.png

## How It Works...

The socket.send(...) method is a shortcut for socket.emit('message', ...) which we will be touching on in Chapter 3, Having Two-Way Conversations. That is why, when the server listens for a "message" topic, it gets called when the client calls socket.send().

Our server is storing an array of all of the topics that connect to it. We are looping over all of the connected sockets to send the message when it comes in. We will be exploring better ways to manage the connected sockets in the next chapter.

The client-side serves 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 are keeping an array of the connected sockets so that we will be able to easily send data to all of them as needed. However, keeping an array of connected sockets can be a little more tedious than just pushing sockets into the array when they connect. If a user leaves the page, for example, the socket will disconnect, but will still be included in the static array.

Fortunately, we are able to tap into the socket disconnect event by calling socket.on('disconnect'). Using that 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.

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

* Chapter 2: Creating Real-Time Dashboards, Handling Connection Timeouts
* Chapter 3: Having Two-Way Conversations, Creating A Simple Chat Room

# Debugging on the Client

In earlier versions of Socket.IO, debugging was extremely simple. That was because verbose logging was pushed into the developer console by default. While 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 get a better view into 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 is using an NPM module called "debug", which allows logging to go into various scopes that can be turned on or off as needed.

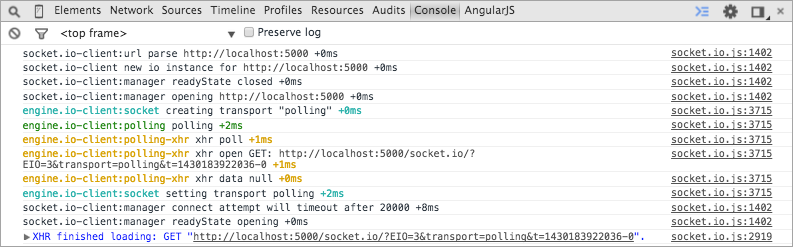
## How To Do It...

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 of the logging messages, just set the value of debug to an asterisk.

localStorage.debug = '\*';

1. Now that robust logging is turned on, you can open your developer tools and see a rich array of messages detailing what is happening under the hood.



Insert Image B04893\_01\_07.png

## 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, which looks at the localStorage key to determine the logging level to show 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 your machine. Also, you will be able to refresh the page and see Socket.IO logging from the initial page load, which can be really handy for debugging events that occur earlier in the page lifecycle.

## 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 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.

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

You can also set multiple log types by separating them with a comma.

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

## See Also

* *Debugging on the Server*
* Debugging on the Server

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

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

## Getting Ready...

To get started debugging on the server-side, you will need to have Node and Socket.IO installed and have an existing app that uses Socket.IO. To test this out, you can easily use any of the apps we built in 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 are starting your server, simply include the DEBUG environmental variable as the first argument when you start your Node server.

DEBUG=\* node server

1. If you would like to persist the DEBUG environmental variable without needing to pass it in every time you start your Node server, you can export it ahead of time.

export DEBUG=\*

1. Now, when you start up your server, verbose logging will be used.

node server

1. You can always update the DEBUG variable or even remove it completely by setting it to null, which will suppress logging entirely.

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 configuration 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 change environmental variables on the fly, so you can easily toggle logging on or off without needing to re-deploy your code.

## There's More...

Similarly 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 in to the environmental variables when you start your Node server.

DEBUG=socket.io:server node server