Previous Next

Towards a Real-Time Web

So far in this course you have seen how clients can make requests to a server, which then responds with information. This model allows you to produce sophisticated web applications controlled by input from the user. But what happens when you want the communication to flow in the other direction?

Let's take Slack as an example. Posting a message fits in well with our existing model. You type your message, press enter, and a request containing the message will be sent to the server. This is where it gets tricky. Because now you want the server to tell everyone else that there is a new message for them to read. But your server isn't able to make requests to a client - it can only respond to requests coming in from clients. So how could you create an application like Slack which allows bidirectional communication in real-time?

Polling

The simplest solution to this problem is by polling the server. Let's look at slack again. In a polling system, clients would continually send new requests to the server. If any new messages had been posted since the last time the client checked, then these would be sent back in the response. Otherwise the server would send back an empty response.

This pattern has some big downsides. Because each client is continually making requests to the server even when there is no new information, your server has to work very hard. You can reduce the workload by having clients poll the server less regularly, but this means that your app will have increase latency (i.e. messages will not appear until a while after they are sent).

Long Polling

In an attempt to improve this situation the long polling technique was invented. With long polling clients make an HTTP request which is held open by the server. In Node.js terms you can think of this as the server not sending the `end` event after receiving the request. When an incoming message comes in from a client the server would then send th message as a response to the clients which have open requests. These clients would then open up a new HTTP request to get more the next message.

In practice long-polling works relatively well. You still have some extra overhead in terms of the number of requests made, as each client has to make a new request whenever it receives a message. However long polling has a significantly lower overhead than a polling system, and pretty much solves the latency problems associated with polling.

Web Sockets and Socket.IO

The best current solution to this problem is Web Sockets. In a Web Socket system, each client is able to open up a long-running connection to the server. The connection is bidirectional, so as well as clients being able to push messages to the server, the server is able to push messages back to the client. This eliminates both the latency and the request overhead of the polling solutions.

Socket.IO is the most widely use system for implementing Web Sockets. It consists of a Node.js library to handle the server-side communication, and a JavaScript library to handle the client-side. In the next assignment you are going to take a first look at how to use Socket.IO by building a simple real-time chatroom. You can think of it as your own miniature version of Slack.

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Mark as completed

Previous Next

Previous Next

Create a Chatroom

In this assignment you are going to use Socket.IO to build a simple chat room application. Let's dive right in by setting up the project:

- Create a directory called *chatroom*
- Create a *public* directory inside *chatroom*
- Run `npm init` and fill in the details
- Run `npm install --save node-static` to install node-static
- Run `npm install --save socket.io` to install Socket.IO

Next, create a file called *server.js*. Initially this just wants to contain the code to serve the static files in the *public* directory:

```
var http = require('http');
var static = require('node-static');

var fileServer = new static.Server('./public');
var server = http.createServer(function(req, res) {
    fileServer.serve(req, res);
});

server.listen(8080);
```

Next in *public*, create an *index.html* file:

```
<html>
    <head>
        <title>Chatroom</title>
        <meta charset="utf-8">

        <!-- JS -->
        <script src="//code.jquery.com/jquery-2.1.0.js"></script>
        <script src="main.js"></script>
    </head>
    <body>
        <input type="text">
        <div id="messages"></div>
    </body>
</html>
```

And a `main.js` file:

```
$(document).ready(function() {
    var input = $('input');
    var messages = $('#messages');

    var addMessage = function(message) {
        messages.append('<div>' + message + '</div>');
    };

    input.on('keydown', function(event) {
        if (event.keyCode != 13) {
            return;
        }

        var message = input.val();
        addMessage(message);
        input.val('');
    });
});
```

Run `node server.js`, and visit http://localhost:8080. You should see the text input. Try entering a message and hitting enter. You should see the message posted below.

Let's take a quick look at the JavaScript to see how this works. First you use jQuery to select the `<input>` tag, and the messages `div`. You then create a function which appends a new `<div>` to the messages. Finally you add a `keydown` listener to the input, which calls the `addMessage` function with the contents of the input when the enter button is pressed, then clears the input.

Now that you have the basic structure of the app set up, let's start adding in the Socket.IO code to make our chatroom networked. First up you need to initialize Socket.IO in *server.js*:

```
var http = require('http');
var static = require('node-static');
var socket_io = require('socket.io');

var fileServer = new static.Server('./public');
var server = http.createServer(function(req, res) {
    fileServer.serve(req, res);
});

var io = socket_io(server);

io.on('connection', function (socket) {
    console.log('Client connected');
});

server.listen(8080);
```

First you `require` the Socket.IO module. Then you initialize an `io` object, by passing the http server to bind to into the `socket_io` function. This creates a `Server` object, which is an `EventEmitter`. Next you add a listener to the

`connection` event of the server. This will be called whenever a new client connects to the Socket.IO server.

Now let's look at how to connect to the server on the client side. Socket.IO serves the client library at `/socket.io/socket.io.js`, so add the following script tag to your *index.html* file:

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

Then edit your `main.js` file to connect to the Socket.IO server:

```
$(document).ready(function() {
    var socket = io();
    var input = $('input');
    var messages = $('#messages');

    ...
```

Here you create a `Manager` object by calling the `io` function. This object will automatically attempt to connect to the server, and will allow you to send and receive messages.

Try your code out by running the server, then visiting your site in a couple of tabs of your browser. You should see that the connect message is printed each time you visit the site.

Now let's try sending a message to the server when we send a message. Edit the `keydown` listener in `main.js`:

```
input.on('keydown', function(event) {
    if (event.keyCode != 13) {
        return;
    }
```

```
    var message = input.val();
    addMessage(message);
    socket.emit('message', message);
    input.val('');
});
```

Here we call the `socket.emit` function. This sends a message to the Socket.IO server. The first argument is a name for our message - in this case we simply call it `message`. The second argument is some data to attach to our message. In this case it's the contents of the text box.

Then let's add code to handle this message to *server.js*:

```
io.on('connection', function (socket) {
    console.log('Client connected');

    socket.on('message', function(message) {
        console.log('Received message:', message);
    });
});
```

Here you add a new listener to the socket which is used to communicate with the client. When a message with the name `message` is received on the socket you simply print out the message.

Restart the server and try entering a message. You should see it getting printed by the server.

Finally you need to broadcast the message to any other clients who are connected. Edit the listener in *server.js* again:

```
io.on('connection', function (socket) {
    console.log('Client connected');
```

```
        socket.on('message', function(message) {
            console.log('Received message:', message);
            socket.broadcast.emit('message', message);
        });
    });
```

There are three ways to communicate from the server to clients. To send a message to a single client you can use the `socket.emit` method (where `socket` is the object passed into your `connection` listener. To send a message to all connected clients you can use the `io.emit` method. And to send a message to all clients *except* one, you can use the `socket.broadcast.emit` method, which won't send the message to the client whose `socket` object you are using.

Here you call the `socket.broadcast.emit` function, to broadcast the message that you received from a client to all of the other clients.

Now let's add a listener for this event to the end of *main.js*:

```
socket.on('message', addMessage);
```

So when the server sends you a message with the name `message`, you add the attached data to the `messages` div using the `addMessage` function.

Restart your server and visit your app in a couple of tabs. You should see that when you post a message in one tab it is reflected in the other, and vice-versa.

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Mark as completed

Previous

Next