

Node.js Tutorial With Socket.io

Update: this tutorial has been update to run on node 5.0.0 and socket.io 1.3.7. If you have trouble with getting the code to execute, make sure you are running on these versions.

I was shocked to recently discover that there are no great quick tutorial on the basics of using socket.io with node js. While there isn't a whole lot to it, someone not familiar with node.js and the protocol for WebSockets is left to scrounge for random snippets of code. Trying to parse the protocol of Websockets while learning the socket.io library and the basics of Node.js all at the same time is no easy feat. And even more frustrating, someone new to Node.js is likely interested in trying it out for the features socket.io provide.

So I thought I would take a brief stab at filling this void. Lets get started:

What We Will Cover:

In this tutorial we will cover

- setting up a Node.js server and router (You can find this many places but it is a necessary first step)
- connecting socket.io to this server
- sending data to the client through the socket connection
- sending data to the server through the socket connection

Your Basic Server:

There are several modules you can use to initiate a server in Node.js, but they are all pretty much the same. We will use the http library.

```
var http = require('http');  
  
var server = http.createServer();  
server.listen(8001);
```

With these 3 lines we have a server. You can save this as `server.js` and run `node server.js` and you will see your command line 'stall' on the process waiting for something to happen on the server. But that's just it, a server that doesn't do anything isn't any fun. So lets have it do something:

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

var server = http.createServer(function(request, response){
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.write('hello world');
    response.end();
});

server.listen(8001);
```

With these changes we are now sending something to the client. Save `server.js` and run `node server.js`. Still nothing happens, but if we go to <http://localhost:8001> (<http://localhost:8001>) we see a page that says “hello world”!

The actions we put inside `createServer` will be executed each time <http://localhost:8001> (<http://localhost:8001>) is loaded. Lets slow down and go through what we did line by line.

```
var server = http.createServer(function(request, response){});
server.listen(8001);
```

Here we have created the server just as we had the first time, but now we are passing it an anonymous function that defines what happens with each request to the server and response from the server.

```
var server = http.createServer(function(request, response){
    //...
});
```

Here we simply say on every request response interaction with the server we will take some action.

```
response.writeHead(200, {'Content-Type': 'text/html'});
```

Here we define how we are sending data to the client. All elements declared by `writeHead` are written to the response header. Our `writeHead` describes the HTTP Code of the response and the content type. The content type can be any of the common MIME types. You can define several other elements found in the HTTP response header, but we won't worry about that for this tutorial.

```
response.write('hello world');
```

This defines the actual contents of our response. More precisely it defines the action of the response. Write says the the contents of the response is “hello world” and upon executing the response the server will append that content to the document being served.

Finally we have

```
response.end();
```

This simply says that we are done defining the response and action and executes them.

Creating The Router:

So now we have very basic server, but it can only server our root domain. Lets fix that. We add the following to `server.js`.

```
var http = require("http");
var url = require('url');
var fs = require('fs');

var server = http.createServer(function(request, response){
  var path = url.parse(request.url).pathname;

  switch(path){
    case '/':
      response.writeHead(200, {'Content-Type': 'text/html'});
      response.write('hello world');
      response.end();
      break;
    case '/socket.html':
      fs.readFile(__dirname + path, function(error, data){
        if (error){
          response.writeHead(404);
          response.write("opps this doesn't exist - 404");
          response.end();
        }
        else{
          response.writeHead(200, {"Content-Type": "text/html"});
          response.write(data, "utf8");
          response.end();
        }
      });
      break;
    default:
      response.writeHead(404);
      response.write("opps this doesn't exist - 404");
      response.end();
      break;
  }
});

server.listen(8001);
```

If you save this code as `server.js` you can rerun `node server.js` when you load <http://localhost:8001> (<http://localhost:8001>) into your browser you will see that nothing has changed. We still get “hello world” in the browser. However, we just build a router so now we should be able to go other urls. Looking at the code we just wrote it seems like we should be able to visit <http://localhost:8001/socket.html> (<http://localhost:8001/socket.html>) and something should happen. But when we try that we get our 404 message. Lets go through what we added and see why.

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

Here we have added two new modules, `url` and `fs`. `url` is used to to parse interpret and manipulate urls, you can learn more about it [here](http://nodejs.org/docs/v0.3.1/api/url.html) (<http://nodejs.org/docs/v0.3.1/api/url.html>). `fs` is used to handle files, you can read about it [here](http://nodejs.org/api/fs.html) (<http://nodejs.org/api/fs.html>).

```
var path = url.parse(request.url).pathname;
```

With this line we get the path that follows our root domain. So `localhost:8001` will assign `'/'` to path and `localhost:8001/some_other_path` will assign `'some_other_path'` to path. You can test this by adding `console.log(path)` immediately after this line.

Next we perform a switch case on the path to decide what we want the server to do. If the path is root then we perform our hello world action. If the path is `socket.html` then the server performs a different action.

```
fs.readFile(__dirname + path, function(error, data){...});
```

Here we use the `fs` module to open the file that is on the path the server receives. Like most methods in `node.js` `readFile` takes a callback function to define its action. This function defines the action we want taken once the file's contents have been gathered. In this case we are wanting to get the file `socket.html`. At this point you have probably noticed why we were getting that 404 message, `socket.html` doesn't exist!!!

Let's go through the rest of our changes and then we will solve that problem.

```
if (error){
  response.writeHead(404);
  response.write("opps this doesn't exist - 404");
}
```

The first argument we pass to the `readFile` callback contains the error code if we are unable to open the file (like say it doesn't exist). The second variable passed is the contents of the file if we are able to open it. So we say that if there is an error opening the file on the path

then we return a 404 header and write a message to the returned document.

```
else{
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.write(data, 'utf8');
}
```

Here is what we want to happen once we have a readable file. It's the same as what we do when we route to '/' but instead we write the contents of the document we read and encode it as utf8.

You will also want to note that we still call `response.end()` at the bottom of our `createServer` function so that whatever actions are routed to the response object are closed and executed.

Now that we see why we are getting that 404 lets fix it. Lets create a new page where we will use socket.io

Adding Socket.io:

To start with we want to get rid of the 404 error when we visit `http://localhost:8001/socket.html` so we are going to create a `socket.html` file.

```
<html>
  <head></head>
  <body>This is our socket.html file</body>
</html>
```

Now we save this to the same directory as our `server.js` file and run `node server.js`. Now when we go to <http://localhost:8001/socket.html> (<http://localhost:8001/socket.html>) we should see a page with "This is our socket.html file".

That's great but that's not really what we're here for. So let's get into the real bread and butter, Socket.io!

First we will instantiate socket.io and extend it from our server. We change our `server.js` file to look like this:

```

var http = require("http");
var url = require('url');
var fs = require('fs');
var io = require('socket.io');

var server = http.createServer(function(request, response){
    var path = url.parse(request.url).pathname;

    switch(path){
        case '/':
            response.writeHead(200, {'Content-Type': 'text/html'});
            response.write('hello world');
            response.end();
            break;
        case '/socket.html':
            fs.readFile(__dirname + path, function(error, data){
                if (error){
                    response.writeHead(404);
                    response.write("opps this doesn't exist - 404");
                    response.end();
                }
                else{
                    response.writeHead(200, {"Content-Type": "text/html"});
                    response.write(data, "utf8");
                    response.end();
                }
            });
            break;
        default:
            response.writeHead(404);
            response.write("opps this doesn't exist - 404");
            response.end();
            break;
    }
});

server.listen(8001);

io.listen(server);

```

All we have added here is a require for the socket.io module at the top and the line `io.listen(server);`. This simply says that when the server is instantiated (note: not connected to) we will open a listener for socket.io. This means that our server will ‘listen’ for pages loaded by the server that have a WebSocket connection instantiated on them.

So lets see what happens when we run `node server.js` and reload <http://localhost:8001/socket.html> (<http://localhost:8001/socket.html>). Nothing seems to change in the browser but our server is now listening for any websocket connections that may occur.

Now that the server is listening for socket.io connections lets give our socket.html page a WebSockets connection so that the server will actually have something to talk to. To do this we simple change socket.html to:

```
<html>
  <head>
    <script src="/socket.io/socket.io.js"></script>
  </head>
  <body>
    <script>
      var socket = io.connect();
    </script>
    <div>This is our socket.html file</div>
  </body>
</html>
```

Here we have added the socket.io.js script and a body script to create the client side socket connection. Now we have a WebSockets connection between the client and our server. Let's try sending some information from the server to the client through our socket.io connection

Sending Data To The Client:

Now we want to send some data from the server to socket.html. All data transactions in socket.io, like in most of node.js, can be handled with callbacks primarily we will utilize the `on` method. The `on` method in simplistic terms is used to map a method name to an anonymous function. Because we are using socket.io the `on` method simply uses the listener on the websocket connection for the method name and when it is found it executes the mapped anonymous function.

The flip side of the `on` method is the `emit` method. The `emit` method sends the mapped method name to the client or the server. It takes two arguments. The mapped method name and the data to be fed to the anonymous function.

So here is what we add to server.js:

```
var listener = io.listen(server);
listener.sockets.on('connection', function(socket){
  socket.emit('message', {'message': 'hello world'});
});
```

All we are saying here is when the listener gets a call for the “connection” action we will perform the function that follows it. When the “connection” action is called we trigger an emit action that will send a “message” action to the client. The “message” action will send the json object `{'message': 'hello world'}`.

The ‘connection’ action is triggered when `io.connection()`; is executed on socket.html. Socket.io has several build in mappable actions, but we can also create custom mappable actions as we will see shortly.

To retrieve the emitted “message” action on socket.html we just add an `on` method listener:

```
var socket = io.connect();

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

Now we have an emit on the server side that sends data to the client side. And an on method on the client side that receives that data and writes it to the browser's console. Run node server.js again and reload the page. If you open up the javascript console of your browser you will note that "hello world" has been written. Further more, if you check your command line you will note that you have more debug statements from socket.io explaining the transmission and contents of the message.

But we could just as easily produce this content through a normal page load or ajax call. We want the actions transmitted through our socket connection to be continuous. A good example of this is a clock. A clock updates the data it displays literally every second.

So let's do that. In javascript we can create repeatable actions with the setInterval function. setInterval takes an anonymous function and a number of milliseconds as arguments and executes the function every count of the milliseconds. So if you want to print hello world every one second you would simply say

```
setInterval(function(){
    console.log('hello world');
}, 1000);
```

So going back to our server.js file we want to emit data for our clock through our socket connection.

```
io.sockets.on('connection', function(socket){
    //send data to client
    setInterval(function(){
        socket.emit('date', {'date': new Date()});
    }, 1000);
});
```

We are doing a few things here. First we set up our setInterval method inside our websocket connection callback. We execute the function inside the setInterval every 1000 milliseconds. Then we say

```
socket.emit('date', {'date': new Date()});
```

which says that we send JSON with the date to a listener on the client side labeled 'date'.

So again, when we make our socket connection we repeatedly send JSON with the date to an on listener on the client every 1000 milliseconds.

Now we will update the client to receive our data.


```
<html>
  <head>
    <script src="/socket.io/socket.io.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.js"></script>
  <body>
    <script>
      var socket = io.connect();

      socket.on('date', function(data) {
        $('#date').text(data.date);
      });
    </script>
    <div id="date"></div>
  </body>
</html>
```

You will notice we added very little. We added a script to load jquery so that we can easily display the data we are receiving. We added an id to our div so that we can use it with jquery. And we added an on method to receive our data from the emit method on the server.

Our on method simply says when we receive an emit action labeled date, we take the data sent by the emit and put it inside our date div.

Try restarting your server and load the page. You should now see the date displayed and being updated every second. Go back to your server.js file and remove the 1000 from the setInterval. Now the date is being updated every millisecond.

Sending Data From the Client To the Server:

Now that we know how to send continuous data from the server to the client all we have left to do is learn how to send continuous data from the client back to the server. You will be happy to know that this process is completely the same as what we just did. So let's say we want to print every letter the user types into a textbox to the server console.

Again we will want to set up an emit function and an on listener. This time the emit will be on the client side and the on listener will be on the server side. We update our socket.html file to look like this:

```

<html>
<head>
  <script src="/socket.io/socket.io.js"></script>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.js"></script>
</head>
<body>
  <script>
    var socket = io.connect();
    socket.on('date', function(data) {
      $('#date').text(data.date);
    });

    $(document).ready(function() {
      $('#text').keypress(function(e) {
        socket.emit('client_data', {'letter': String.fromCharCode(e.charCode)});
      });
    });
  </script>
  <div id="date"></div>
  <textarea id="text"></textarea>
</body>
</html>

```

Here all we have added is a textarea and an emit event to be triggered every time a key is pressed inside that textarea. `String.fromCharCode(e.charCode)` takes the character code of the key that triggered the event and converts it back to a string of the character associated with that code number. So if the “a” key triggered the event then our JSON would be

```
{'letter': 'a'}.
```

Now we will add the on listener to our server.js file:

```

var listener = io.listen(server);

listener.sockets.on('connection', function(socket) {
  //send data to client
  setInterval(function() {
    socket.emit('date', {'date': new Date()});
  }, 1000);

  //recieve client data
  socket.on('client_data', function(data) {
    process.stdout.write(data.letter);
  });
});

```

You will see that here we have added the just a few lines again.

First we added

```
//recieve client data
socket.on('client_data', function(data){
  process.stdout.write(data.letter);
});
```

This is our listener for the `client_data` emit call. When an action triggers this listener we write the letter inside our JSON to the server console. We use `process.stdout.write()` because it writes to stdout without inserting new line characters like `console.log()` will.

So restart your server and try everything out. We should now have a clock running in our browser with data from the server and data being streamed from the browser to the command line when we type in the textarea.

Conclusion:

Hopefully, this has provided a more thorough introductory tutorial for beginners interested in using node.js and socket.io. If you have any questions drop me a line.

You can find the example project [here \(https://github.com/DanielNill/node-and-socket-example\)](https://github.com/DanielNill/node-and-socket-example)