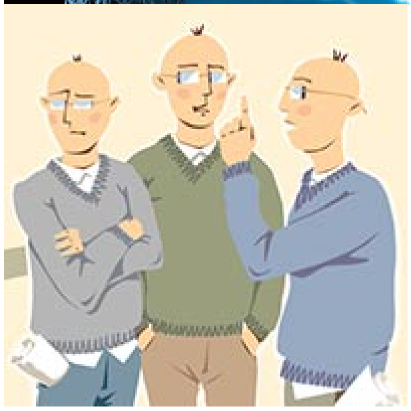
Advertise Here

JetBrains PhpStorm
Lightning-smart IDE for PHP Developers



Real Time Chat With NodeJS, Socket.io and ExpressJS

Krasimir Tsonev on May 14th 2013 with 47 Comments

Tutorial Details

-
- **Difficulty:** Advanced
- **Completion Time:** 1 Hour

NodeJS gives me the ability to write back-end code in one of my favorite languages: JavaScript. It's the perfect technology for building real time applications. In this tutorial, I'll show you how to build a web chat application, using ExpressJS and Socket.io.

Setup Environment

Of course, the first thing to do is get NodeJS installed on your system. If you are a Windows or Mac user, you can visit nodejs.org and download the installer. If you instead prefer Linux, I'd suggest that you refer to this link. Although I won't go into further details on this, if you encounter any installation issues, I'm happy to help; just leave a comment below this post.

Once you have NodeJS installed, you're ready to setup the needed instruments.

1. ExpressJS – this will manage the server and the response to the user
2. Jade – template engine
3. Socket.io – allows for real time communication between the front-end and back-end

Continuing on, within an empty directory, create a *package.json* file with the following content.

```
{
    "name": "RealTimeWebChat",
    "version": "0.0.0",
    "description": "Real time web chat",
    "dependencies": {
        "socket.io": "latest",
        "express": "latest",
        "jade": "latest"
    },
    "author": "developer"
}
```

By using the console (under Windows – command prompt), navigate to your folder and execute:

```
npm install
```

Within a few seconds, you'll have all the necessary dependencies downloaded to the *node_modules* directory.

Developing the Backend

Let's begin with a simple server, which will deliver the application's HTML page, and then continue with the more interesting bits: the real time communication. Create an *index.js* file with the following core expressjs code:

```
var express = require("express");
var app = express();
var port = 3700;

app.get("/", function(req, res){
    res.send("It works!");
});

app.listen(port);
console.log("Listening on port " + port);
```

Above, we've created an application and defined its port. Next, we registered a route, which, in this case, is a simple GET request without any parameters. For now, the route's handler simply sends some text to the client. Finally, of course, at the bottom, we run the server. To initialize the application, from the console, execute:

```
node index.js
```

The server is running, so you should be able to open *http://127.0.0.1:3700/* and see:

```
It works!
```

Now, instead of *"It works"* we should serve HTML. Instead of pure HTML, it can be beneficial to use a template engine. Jade is an excellent choice, which has good integration with ExpressJS. This is what I typically use in my own projects. Create a directory, called *tpl*, and put the following *page.jade* file within it:

```
!!!
html
    head
        title= "Real time web chat"
    body
        #content(style='width: 500px; height: 300px; margin: 0 0 20
        .controls
            input.field(style='width:350px;')
            input.send(type='button', value='send')
```

The Jade's syntax is not so complex, but, for a full guide, I suggest that you refer to jade-lang.com. In order to use Jade with ExpressJS, we require the following settings.

```
app.set('views', __dirname + '/tpl');
app.set('view engine', "jade");
app.engine('jade', require('jade').__express);
app.get("/", function(req, res){
    res.render("page");
});
```

This code informs Express where your template files are, and which template engine to use. It all specifies the function that will process the template's code. Once everything is setup, we can use the `.render` method of the `response` object, and simply send our Jade code to the user.

The output isn't special at this point; nothing more than a `div` element (the one with id `content`), which will be used as a holder for the chat messages and two controls (input field and button), which we will use to send the message.

Because we will use an external JavaScript file that will hold the front-end logic, we need to inform ExpressJS where to look for such resources. Create an empty directory, `public`, and add the following line before the call to the `.listen` method.

```
app.use(express.static(__dirname + '/public'));
```

So far so good; we have a server that successfully responds to GET requests. Now, it's time to add *Socket.io* integration. Change this line:

```
app.listen(port);
```

to:

```
var io = require('socket.io').listen(app.listen(port));
```

Above, we passed the ExpressJS server to Socket.io. In effect, our real time communication will still happen on the same port.

Moving forward, we need to write the code that will receive a message from the client, and send it to all the others. Every Socket.io application begins with a `connection` handler. We should have one:

```
io.sockets.on('connection', function (socket) {
    socket.emit('message', { message: 'welcome to the chat' });
    socket.on('send', function (data) {
        io.sockets.emit('message', data);
    });
});
```

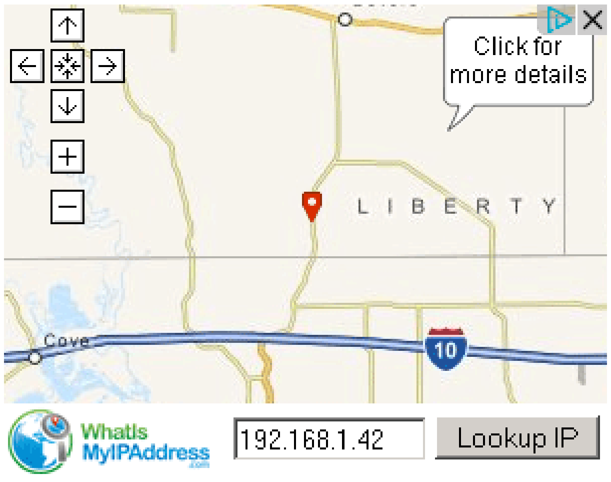
The object, `socket`, which is passed to your handler, is actually the socket of the client. Think about it as a junction between your server and the user's browser.

Upon a successful connection, we send a `welcome` type of message, and, of course, bind another handler that will be used as a receiver. As a result, the client should emit a message with the name, `send`, which we will catch. Following that, we simply forward the data sent by the user to all other sockets with `io.sockets.emit`.

With the code above, our back-end is ready to receive and send messages to the clients. Let's add some front-end code.

Developing the Front-end

Create `chat.js`, and place it within the `public` directory of your application. Paste the following code:



```
window.onload = function() {

    var messages = [];
    var socket = io.connect('http://localhost:3700');
    var field = document.getElementById("field");
    var sendButton = document.getElementById("send");
    var content = document.getElementById("content");

    socket.on('message', function (data) {
        if(data.message) {
            messages.push(data.message);
            var html = '';
            for(var i=0; i<messages.length; i++) {
                html += messages[i] + '<br />';
            }
            content.innerHTML = html;
        } else {
            console.log("There is a problem:", data);
        }
    });

    sendButton.onclick = function() {
        var text = field.value;
        socket.emit('send', { message: text });
    };

}
```

Our logic is wrapped in a `.onload` handler just to ensure that all the markup and external JavaScript is fully loaded. In the next few lines, we create an array, which will store all the messages, a `socket` object, and few shortcuts to our DOM elements. Again, similar to the back-end, we bind a function, which will react to the socket's activity. In our case, this is an event, named `message`. When such an event occurs, we expect to receive an object, *data*, with the property, `message`. Add that message to our storage and update the `content` `div`. We've also included the logic for sending the message. It's quite simple, simply emitting a message with the name, *send*.

If you open *http://localhost:3700*, you will encounter some errors popup. That's because we need to update `page.jade` to contain the necessary JavaScript files.

```
head
    title= "Real time web chat"
    script(src='/chat.js')
    script(src='/socket.io/socket.io.js')
```

Notice that Socket.io manages the delivery of *socket.io.js*. You don't have to worry about manually downloading this file.

We can again run our server with `node index.js` in the console and open *http://localhost:3700*. You should see the welcome message. Of course, if you send something, it should be shown in the content's `div`. If you want to be sure that it works, open a new tab (or, better, a new browser) and load the application. The great thing about Socket.io is that it works even if you stop the NodeJS server. The front-end will continue to work. Once the server is booted up again, your chat will be fine too.

In its current state, our chat is not perfect, and requires some improvements.

Improvements

The first change that we need to make is to the identity of the messages. Currently, it's not clear which messages is sent by whom. The good thing is that we don't have to update our NodeJS code to achieve this. That's because the server simply forwards the `data` object. So, we need to add a new property there, and read it later. Before making corrections to `chat.js`, let's add a new `input` field, where the user may add his/her name. Within `page.jade`, change the `controls div`:

```
.controls
    | Name:
    input#name(style='width:350px;')
    br
    input#field(style='width:350px;')
    input#send(type='button', value='send')
```

Next, in *code.js*:

```
window.onload = function() {

    var messages = [];
    var socket = io.connect('http://localhost:3700');
    var field = document.getElementById("field");
    var sendButton = document.getElementById("send");
    var content = document.getElementById("content");
    var name = document.getElementById("name");

    socket.on('message', function (data) {
        if(data.message) {
            messages.push(data);
            var html = '';
            for(var i=0; i<messages.length; i++) {
                html += '<b>' + (messages[i].username ? messages[i
                html += messages[i].message + '<br />';
            }
            content.innerHTML = html;
        } else {
            console.log("There is a problem:", data);
        }
    });

    sendButton.onclick = function() {
        if(name.value == "") {
            alert("Please type your name!");
        } else {
            var text = field.value;
            socket.emit('send', { message: text, username: name.va
        }
    };

}
```

To summarize the changes, we've:

1. Added a new shortcut for the username's `input` field
2. Updated the presentation of the messages a bit
3. Appended a new `username` property to the object, which is sent to the server

If the the number of messages becomes too high, the user will need to scroll the `div`:

```
content.innerHTML = html;
content.scrollTop = content.scrollHeight;
```

Keep in mind that the above solution will likely not work in IE7 and below, but that's okay: it's time for IE7 to fade away. However, if you want to ensure support, feel free to use jQuery:

```
$("#content").scrollTop($("#content")[0].scrollHeight);
```

It would also be nice if the input field is cleared after sending the message:

```
socket.emit('send', { message: text, username: name.value });
field.value = "";
```

The final boring problem is the clicking of the *send* button each time. With a touch of jQuery, we can listen for when the user presses the `Enter` key.

```
$(document).ready(function() {
    $("#field").keyup(function(e) {
        if(e.keyCode == 13) {
            sendMessage();
        }
    });
});
```

The function, `sendMessage`, could be registered, like so:

```
sendButton.onclick = sendMessage = function() {
    ...
};
```

Please note that this isn't a best practice, as it is registered as a global function.

But, for our little test here, it'll be fine.

Conclusion

NodeJS is an extremely useful technology, and provides us with a great deal of power and joy, especially when considering the fact that we can write pure JavaScript. As you can see, with only a few lines of code, we managed to write a fully functional real time chat application. Pretty neat!

> Want to learn more about building web apps with ExpressJS? We've got you covered!

Like 223 people like this. Be the first of your friends.

Save 100% on Your Data Plan
Download Apps, Games, Ringtone, Photos

Tags: expressjsnodejssocket.io

By Krasimir Tsonev

Krasimir Tsonev is a coder with over ten years of experience in web development. With a strong focus on quality and usability, he is interested in delivering cutting edge applications. Currently, with the rise of the mobile development, Krasimir is enthusiastic to work on responsive applications targeted to various devices. Living and working in Bulgaria, he graduated at the Technical University of Varna with a bachelor and master degree in computer science. If you'd like to stay up to date on his activities, refer to his blog or follow him on Twitter.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more