Using Node.js and Websockets to Build a Chat Service

Guillaume Besson on Sep 4th 2013 with 39 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Estimated Completion Time:** 1 hour

Node.js and Websockets are the perfect combination to write very fast, lag free applications which can send data to a huge number of clients. So why don't we start learning about these two topics by building a chat service! We will see how to install Node.js packages, serve a static page to the client with a basic web-server, and configure Socket.io to communicate with the client.

Why Choose Node.js and Socket.io?

So why use this combo?

There are a lot of platforms which can run a chat application, but by choosing Node.js we don't have to learn a completely different language, it's just JavaScript, but server-side.

Node.js is a platform built on Chrome's JavaScript runtime to make building applications in JavaScript that run on the server, easy. Node.js uses an event-driven, non-blocking I/O model, which makes it perfect for building real time apps.

More and more Node.js applications are being written with real-time communication in mind. A famous example is BrowserQuest from Mozilla, an MMORPG written entirely in Node.js whose source code has been released on Github.

Node.js comes with a built-in package manager : npm. We will use it to install packages that will help speed up our application development process.

We'll be using three packages for this tutorial: Jade, Express, and Socket.io.

Socket.io: the Node.js Websockets Plugin

The main feature of our application is the real-time communication between the client and the server.

HTML5 introduces Websockets, but it is far away from being supported by all users, so we need a backup solution.

Socket.io is our backup solution : it will test Websocket compatibility and if it's not supported it will use Adobe Flash, AJAX, or an iFrame.

Finally, it supports a very large set of browsers:

- Internet Explorer 5.5+
- Safari 3+
- Google Chrome 4+
- Firefox 3+
- Opera 10.61+
- iPhone Safari
- iPad Safari
- Android WebKit
- WebOs WebKit

It also offers very easy functions to communicate between the server and the client, on both sides.

Let's start by installing the three packages we will need.

Installing Our Dependencies

Npm allows us to install packages very fast, using one line, so first go to your directory and have npm download the needed packages:

```
1 npm install express jade socket.io
```

Now we can start building our server–side controller to serve the main page.

We are going to save all the server–side code into a `"server.js"` file which will be executed by Node.js.

Serving a Single Static Page

To serve our static page we will use Express, a package which simplifies the whole server–side page send process.

So let's include this package into our project and start the server:

```
1 var express = require('express'), app = express.createServer();
```

Next, we need to configure Express to serve the page from the repertory views with the Jade templating engine that we installed earlier.

Express uses a layout file by default, but we don't need it because we will only serve one page, so instead, we will disable it.

Express can also serve a static repertory to the client like a classic web server, so we will send a `"public"` folder which will contain all our JavaScript, CSS and image files.

```
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.set("view options", { layout: false });
app.configure(function() {
   app.use(express.static(__dirname + '/public'));
});
```

Next, let's create two folders inside our project folder named `"public"` and `"views"`.

Now we just need to configure Express to serve a `"home.jade"` file, which we will create in a moment, and then set Express to listen for a specific port.

I will use port `3000` but you can use whatever you'd prefer.

```
app.get('/', function(req, res){
  res.render('home.jade');
});
app.listen(3000);
```

Creating the Jade Template Page

Node.js uses templating engines to serve webpages. It's useful to send dynamic pages and to build them faster.

In this tutorial, we will use Jade. Its syntax is very clear and it supports everything we need.

> "Jade is a high performance templating engine heavily influenced by Haml and implemented with JavaScript for Node."

Now, I'm not going to go over Jade in detail, if you need more help, you can find

very well written documentation on its Github repo.

Jade Configuration

We installed Jade earlier, but we need to include it in our `server.js` file like we did for Express.

By convention, we include our libraries at the top of our file to use them later, without having to check if they are already included. So place the following code at the top of your `"server.js"` file :

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

And that completes our Jade configuration. Express is already setup to use Jade with our view files, to send an HTML response, we just need to create that file.

Creating Our Home Page

If we start our server now it will crash because we're requesting our app to send a page which doesn't exist yet.

We're not going to create a full featured page, just something basic that has a title, a container for the messages, a text area, a send button, and a user counter.

Go ahead and create a `"home.jade"` page inside the `"views"` folder with the following code:

```
doctype 5
html
   head
      title Chat
      script(src='https://ajax.googleapis.com/ajax/libs/jquery/1.7
      script(src="/socket.io/socket.io.js")
      script(src="script.js")
   body
      div.container
         header
            h1 A Chat application with Node.js and Socket.io
         input(type='text')#pseudoInput
         button#pseudoSet Set Pseudo
         div#chatEntries
```

```
15          div#chatControls
16             input(type='text')#messageInput
17             button#submit Send
```

> “Jade is all about indentation”

The Jade language is all about indentation. As you can see, we don’t need to close our containers, just indenting the children of the parent container is enough.

We also use a period `"."` and a pound sign `"#"` to indicate the class or ID of the element, just like in a CSS file.

We also link in three scripts at the top of the file. The first is jQuery from Google CDN, next we have the Socket.io script which is served automatically by the package, and finally a `"script.js"` file which will keep all of our custom JS functions.

The Socket.io Server-Side Configuration

Socket.io is event based, just like Node. It aims to make real–time apps possible in every browser and mobile device, blurring the lines between these different transport mechanisms. It’s care–free, real–time, and 100% JavaScript.

Like the other modules, we need to include it in our `server.js` file. We will also chain on our express server to listen for connections from the same address and port.

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

The first event we will use is the connection event. It is fired when a client tries to connect to the server; Socket.io creates a new socket that we will use to receive or send messages to the client.

Let's start by initializing the connection:

```
1 io.sockets.on('connection', function (socket) {
2    //our other events...
3 });
```

This function takes two arguments, the first one is the event and the second is the callback function, with the socket object.

Using code like this, we can create new events on the client and on the server with Socket.io. We will set the `"pseudo"` event and the `"message"` event next.

To do this, it's really simple, we just use the same syntax, but this time with our `socket` object and not with the `"io.sockets"` (with an "s") object. This allows us to communicate specifically with one client.

So inside our connection function, let's add in the `"pseudo"` event code.

```
1 socket.on('setPseudo', function (data) {
2    socket.set('pseudo', data);
```

```
});
```

The callback function takes one argument, this is the data from the client and in our case it contains the `pseudo`. With the `"set"` function, we assign a variable to the socket. The first argument is the name of this variable and the second is the value.

Next, we need to add in the code for the `"message"` event. It will get the user's pseudo, broadcast an array to all clients which contains the message we received as well as the user's pseudo and log it into our console.

```
socket.on('message', function (message) {
   socket.get('pseudo', function (error, name) {
      var data = { 'message' : message, pseudo : name };
      socket.broadcast.emit('message', data);
      console.log("user " + name + " send this : " + message);
   })
});
```

This completes our server–side configuration. If you'd like, you can go ahead and use other events to add new features to the chat.

The nice thing about Socket.io is that we don't have to worry about handling client disconnections. When it disconnects, Socket.io will no longer receive responses to "heartbeat" messages and will deactivate the session associated with the client. If it was just a temporary disconnection, the client will reconnect and continue with the session.

The Socket.io Client-Side Configuration

Now that our server is configured to manage messages, we need a client to send them.

The client–side of Socket.io is almost the same as the server–side. It also works with custom events and we will create the same ones as on the server.

So first, create a `"script.js"` file inside the `public` folder. We will store all of our functions inside of it.

We first need to start the socket.io connection between the client and the server. It will be stored in a variable, which we will use later to send or receive data. When the connection is not passed any arguments, it will automatically connect to the server which will serve the page.

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

Next, let's create some helper functions that we will need later. The first is a simple function to add a message to the screen with the user's pseudo.

```
1 function addMessage(msg, pseudo) {
2    $("#chatEntries").append('<div class="message"><p>' + pseudo + '
3 }
```

This helper uses the append function from jQuery to add a `div` at the end of the `#chatEntries div`.

Now we are going to write a function that we can call when we want to send a new message.

```
1 function sentMessage() {
2    if ($('#messageInput').val() != "")
3    {
4       socket.emit('message', $('#messageInput').val());
5       addMessage($('#messageInput').val(), "Me", new Date().toISOSt
6       $('#messageInput').val('');
7    }
8 }
```

First, we verify that our textarea is not empty, then we send a packet named `"message"` to the server which contains the message text, we print it on the screen with our `"addMessage"` function, and finally we remove all the text from the textarea.

Now, when the client opens the page, we need to set the user's pseudo first. This function will send the pseudo to the server and show the textarea and the submit button.

```
1 function setPseudo() {
2    if ($("#pseudoInput").val() != "")
3    {
4       socket.emit('setPseudo', $("#pseudoInput").val());
5       $('#chatControls').show();
```

```
        $('#pseudoInput').hide();
        $('#pseudoSet').hide();
    }
}
```

Additionally, we hide the pseudo setting controls when it's sent to the server.

Now just like we did on the server-side, we need to make sure we can receive incoming messages and this time we'll print them on the screen. We'll use the same syntax but this time we call the `"addMessage"` function.

```
socket.on('message', function(data) {
    addMessage(data['message'], data['pseudo']);
});
```

Just like with our server configuration, the packet that is sent to the client is an array containing the message and the pseudo. So we just call our `"addMessage"` function passing in the message and the pseudo, which we extract from the received data packet.

Now we just need to add the initialization function which is fired once the page is fully loaded.

```
$(function() {
    $("#chatControls").hide();
    $("#pseudoSet").click(function() {setPseudo()});
    $("#submit").click(function() {sentMessage();});
});
```

First, we hide the chat controls before the pseudo is set and then we set two click listeners which listen for clicks on our two submit buttons. The first is for the pseudo and the second is for the messages.

And that wraps up our client-side script.

Conclusion

We now have a working chat service. To start it, just run the following command :

```
node server.js
```

In your terminal you should get a message from Socket.io saying that the server is started. To see your page go to `127.0.0.1:3000` (or whichever port you chose previously).

The design is very basic, but you could easily add in a stylesheet with CSS3 transitions for incoming messages, HTML5 sounds, or Bootstrap from Twitter.

As you can see, the server and client scripts are fairly similar: this is the power of Node.js. You can build an application without having to write the code twice.

Finally, you may have noticed that it only took 25 lines of code inside our `server.js` file to create a functional chat app, with amazing performance. It is very short, but it also works very well.

Now if you're interested, I have created a better chat service application, with a good looking design, along with some additional features. It is hosted on Nodester and the source code is on Github.

Here is a preview of it.

Thanks for reading.

Tags: expressjadenode jswebsockets

By Guillaume Besson
This author has yet to write their bio.

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