# Node.js & WebSocket - Simple chat tutorial

5 years ago
Node.js is a brilliant product. It gives you so much freedom (... and also responsibility) and I think it's ideal for single purpose web servers.

Another great thing is WebSocket. ~~Although~~ Nowadays it's ~~not~~ widely supported (Chrome 14+, Firefox 7, Chrome for iOS, Chrome for Android(?) and maybe IE 10) and it's usage is probably in very specific applications like games or Google Docs. So I wanted to try to make some very simple real world application.

WebSocket requires it's own backend application to communicate with (server side). Therefore you have to write single purpose server and, in my opinion, in this situation node.js is much better than writing your server in Java, C++ or whatever

BTW, if you're looking for some more in-depth information on how WebSockets work I recommend this article Websockets 101.

In this tutorial I'm going to write very simple chat application based on WebSocket and node.js.

## Chat features

At the beginning every user can select their name and the server will assign them some random color and will post some system message to the console that a new user just connected. Then the user can post messages. When a user closes the browser window, server will post another system massage to the console that a user has disconnected.

Also, every new user will recieve entire message history.

## Live demo

Here~~'s~~ was live demo~~, feel free to play with in or examine the source code. Just one thing, it might be sometimes broken because I had to restart the server or just something unexpected happened. If so, leave me a comment please .~~, I'll try to put it back online asap.

## HTML + CSS

Frontend is very simple HTML and CSS for now. We'll add some JavaScripts later.

```
<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>WebSockets - Simple chat</title>


<style>

* { font-family:tahoma; font-size:12px; padding:0px; margin:0px; }

p { line-height:18px; }

div { width:500px; margin-left:auto; margin-right:auto;}

#content { padding:5px; background:#ddd; border-radius:5px; overflow-y: scroll;

border:1px solid #CCC; margin-top:10px; height: 160px; }

#input { border-radius:2px; border:1px solid #ccc;

margin-top:10px; padding:5px; width:400px; }

#status { width:88px; display:block; float:left; margin-top:15px; }

</style>

</head>

<body>

<div id="content"></div>

<div>

<span id="status">Connecting...</span>

<input type="text" id="input" disabled="disabled" />
```

```
</div>


<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>

<script src="./frontend.js"></script>

</body>

</html>
```

## Communication client -> server, server -> client

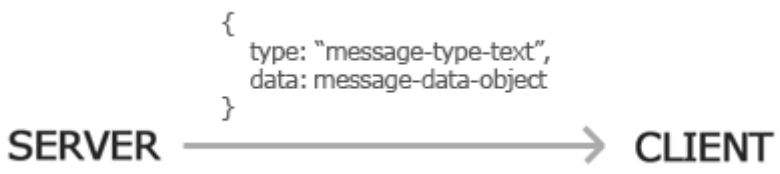
Great advantage of WebSocket is two way communication. In this tutorial it means situation when some user sends a message (**client -> server**) and then server sends that message to all conected users (**server -> client**) - broadcast.

For **client -> server** communication I choosed simple text because it's not necessary to wrap it in more complicated structure.


But for **server -> client** it's a bit more complex. We have to distinguish between 3 different types of message:

- server assigns a color to user
- server sends entire message history
- server broadcasts a message to all users

Therefore every message is a simple JavaScript object encoded into JSON.




## Node.js server

Node.js itself doesn't have support for WebSocket but there are already some plugins that implement WebSocket protocols. I've tried two of them:

- node-websocket-server - very easy to use, but it doesn't support draft-10. That's a big problem because Chrome 14+ supports only draft-10 which is not compatible with older drafts. According to issues on GitHub the autor is working on version 2.0 that should support also draft-10.
- WebSocket-Node - very easy to and well documented. Supports draft-10 and also older drafts.

In this tutotial I'm going to use the second one, so let's install it with `npm` (Node Package Manager) which comes together with node.js. There might be a little tricky if you're using Windows because it needs to compile some small part of the module from C++ (read more on github.com).

```
npm install websocket
```

### WebSocket server template

WebSocket server code template looks like this:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
 22
 23
 24
 25
 26
 27
 28
 29
 30
```

```
var WebSocketServer = require('websocket').server;
var http = require('http');

var server = http.createServer(function(request, response) {
    // process HTTP request. Since we're writing just WebSockets server
    // we don't have to implement anything.
});
server.listen(1337, function() { });

// create the server
wsServer = new WebSocketServer({
    httpServer: server
});

// WebSocket server
wsServer.on('request', function(request) {
    var connection = request.accept(null, request.origin);

    // This is the most important callback for us, we'll handle
    // all messages from users here.
    connection.on('message', function(message) {
        if (message.type === 'utf8') {
            // process WebSocket message
        }
    });

    connection.on('close', function(connection) {
        // close user connection
    });
});
```

So, this is just the most basic skeleton that we'll extend now with more logic.

That's all for the server part. I added comments where it was appropriate but I think it's very simple to understand.

## Frontend JavaScript

### Frontend template

Frontend template is basically just three callback methods:

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
```

```
20
21
22
23
24
25
```

```
$(function () {
    // if user is running mozilla then use it's built-in WebSocket
    window.WebSocket = window.WebSocket || window.MozWebSocket;

    var connection = new WebSocket('ws://127.0.0.1:1337');

    connection.onopen = function () {
        // connection is opened and ready to use
    };

    connection.onerror = function (error) {
        // an error occurred when sending/receiving data
    };

    connection.onmessage = function (message) {
        // try to decode json (I assume that each message from server is json)
        try {
            var json = JSON.parse(message.data);
        } catch (e) {
            console.log('This doesn\'t look like a valid JSON: ', message.data);
            return;
        }
        // handle incoming message
    };
});
```

### Frontend full source code

Frontend is quiet simple as well, I just added some logging and some enabling/disabling of the input field so it's more user friendly.

```
$(function () {

"use strict";


// for better performance - to avoid searching in DOM

var content = $('#content');

var input = $('#input');

var status = $('#status');


// my color assigned by the server

var myColor = false;

// my name sent to the server

var myName = false;


// if user is running mozilla then use it's built-in WebSocket

window.WebSocket = window.WebSocket || window.MozWebSocket;


// if browser doesn't support WebSocket, just show some notification and exit

if (!window.WebSocket) {

content.html($('<p>', { text: 'Sorry, but your browser doesn\'t '

+ 'support WebSockets.'} ));

input.hide();

$('span').hide();
```

```
return;

}

// open connection

var connection = new WebSocket('ws://127.0.0.1:1337');

connection.onopen = function () {

// first we want users to enter their names

input.removeAttr('disabled');

status.text('Choose name:');

};

connection.onerror = function (error) {

// just in there were some problems with conenction...

content.html($('<p>', { text: 'Sorry, but there\'s some problem with your '

+ 'connection or the server is down.' } ));

};

// most important part - incoming messages

connection.onmessage = function (message) {

// try to parse JSON message. Because we know that the server always returns

// JSON this should work without any problem but we should make sure that

// the massage is not chunked or otherwise damaged.

try {

var json = JSON.parse(message.data);

} catch (e) {

console.log('This doesn\'t look like a valid JSON: ', message.data);

return;

}

// NOTE: if you're not sure about the JSON structure

// check the server source code above

if (json.type === 'color') { // first response from the server with user's color

myColor = json.data;

status.text(myName + ': ').css('color', myColor);

input.removeAttr('disabled').focus();

// from now user can start sending messages

} else if (json.type === 'history') { // entire message history

// insert every single message to the chat window

for (var i=0; i < json.data.length; i++) {

addMessage(json.data[i].author, json.data[i].text,

json.data[i].color, new Date(json.data[i].time));

}

} else if (json.type === 'message') { // it's a single message

input.removeAttr('disabled'); // let the user write another message
```

```
addMessage(json.data.author, json.data.text,

json.data.color, new Date(json.data.time));

} else {

console.log('Hmm..., I\'ve never seen JSON like this: ', json);

}

};


/**

* Send mesage when user presses Enter key

*/

input.keydown(function(e) {

if (e.keyCode === 13) {

var msg = $(this).val();

if (!msg) {

return;

}

// send the message as an ordinary text

connection.send(msg);

$(this).val('');

// disable the input field to make the user wait until server

// sends back response

input.attr('disabled', 'disabled');


// we know that the first message sent from a user their name

if (myName === false) {

myName = msg;

}

}

});


/**

* This method is optional. If the server wasn't able to respond to the

* in 3 seconds then show some error message to notify the user that

* something is wrong.

*/

setInterval(function() {

if (connection.readyState !== 1) {

status.text('Error');

input.attr('disabled', 'disabled').val('Unable to comminucate '

+ 'with the WebSocket server.');

}

}, 3000);


/**
```

```
* Add message to the chat window
*/
function addMessage(author, message, color, dt) {
content.prepend('<p><span style="color:' + color + '">' + author + '</span> @ ' +
+ (dt.getHours() < 10 ? '0' + dt.getHours() : dt.getHours()) + ':'
+ (dt.getMinutes() < 10 ? '0' + dt.getMinutes() : dt.getMinutes())
+ ': ' + message + '</p>');
}
});
```

Again I tried to put comments where it's appropriate, but I think it's still very simple.

## Running the server

I wrote and tested the server on node.js **v0.4.10** and **v0.5.9** but I think I'm not using any special functions so it should run on older and newer version without any problem. If you're using Windows node.js > 0.5.x comes with Windows executable. I tested it on Windows as well.

So under Unix, Windows or whatever:

```
node chat-server.js
```

and you should see something like this

```
Thu Oct 20 2011 09:15:44 GMT+0200 (CEST) Server is listening on port 1337
```

Now you can open `chat.html` and if everything's fine it should ask you for your name and the server should write to the console:

```
Thu Oct 20 2011 09:27:21 GMT+0200 (CEST) Connection from origin null.
Thu Oct 20 2011 09:27:21 GMT+0200 (CEST) Connection accepted. Currently 1 client.
```

## So what's next?

That's actually all. I'm adding a few notes at the end just to make some things a little bit more obvious.

### Pros and cons of node.js

For this purpose is node.js ideal tool because:

- It's event driven. Together with closures it's very simple to imagine the client lifecycle.
- You don't have to care about threads, locks and all the parallel stuff.
- It's very fast (built on V8 JavaScript Engine)
- For games you can write the game logic just once and then use it for both server and frontend.
- It's just like any other JavaScript.

but on the other hand there are some gotchas:

- It's under rapid development and your server might not be compatible with newer versions of node.js.
- It's all quiet new. I haven't seen any real application of node.js. Just a bunch of games and some nice demos.
- Node.js unlike Apache doesn't use processes for each connection.

The last point has some important consequences. Look at the server source code where I'm declaring `colors` variable. I hard coded 7 colors but what if there were 7 active connections and 8th client tried to connect? Well, this would probably threw an exception and the server would broke down immediately.

The problem is that Apache runs separate process for each request and if you have a bug in your code it brakes just one process and doesn't influence the rest. Of course, creating new processes for each client is much better in terms of stability but it generates some overhead.

I like talk HTML5 Games with Rob Hawkes of Mozilla where Rob Hawkes mentions that he used monit to observe if the server is running and eventually start it again in the case it broke.

### What about socket.io?

Socket.io is an interesting project (it's a module for node.js) implementing WebSocket backend and frontend with many fallbacks for probably every possible browser you can imagine. So I should probably explain why I'm not using it.

The first reason is that I think it's always better to try to write it by yourself first because then you have much better insight what's going on and when it doesn't work you can easily figure out why.

The second reason is that I want to try to write some simple games using WebSocket. Therefore all the fallbacks provided by socket.io are useless because I need as low latency as it's possible. Also, there is some magic `/socket.io/socket.io.js` which is probably generated according to your browser's capabilities but I'm rather trying to avoid debuging javascripts for IE6 :).

I thing socket.io is great, but it would be very unpleasant if I spent month of writing a game and then realised that socket.io generates so much overhead so it's useless and I had to rewrite it.

Anyway, for other applications (like this chat for instance) socket.io would be ideal, because ±100ms is absolutely irrelevant and this chat would work on all browser and not just Chrome 14+ and Firefox 7+.

### What if I want some usage statistics?

I think good questions is how can I dig some information from the WebSocket server like number of active connections or number sent messages?

As I mentioned at the beginning is tied to HTTP server. At this moment it comes handy because we can run on the same port WebSocket server and also HTTP server.

So let's say, we want to be able to ask the server for number of current connections and number of all sent messages. For this purpose we have to create simple HTTP server that knows just one URL (`/status`) and sends JSON with these two numbers as a response.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

```
/** ... */

/**
 * HTTP server
 */
var server = http.createServer(function(request, response) {
    console.log((new Date()) + ' HTTP server. URL' + request.url + ' requested.');

    if (request.url === '/status') {
        response.writeHead(200, {'Content-Type': 'application/json'});
        var responseObject = {
            currentClients: clients.length,
            totalHistory: history.length
        }
        response.end(JSON.stringify(responseObject));
    } else {
        response.writeHead(404, {'Content-Type': 'text/plain'});
        response.end('Sorry, unknown url');
    }
});
server.listen(webSocketsServerPort, function() {
    console.log((new Date()) + " Server is listening on port " + webSocketsServerPort);
```

```
});

/** ... */
```

Now when you access `http://localhost:1337/status` you should see something similar to:

```
{"url":"/status","response":{"currentClients":2,"totalClients":4,"totalHistory":6}}
```

`Clients` and `history` are two global variables (I know, bad practise) that I'm using for different purpose but in this situation I can reuse them easily without modifying rest of the server.

### Running under Windows

As I menioned above you can run this chat on Windows too.

There have been a couple of changes in the websocket module and now for Windows you have to follow some extra steps.

~~Probably the only problem might be with `npm` (Node Package Manager). To be honest I'm usually avoiding `npm` because for me it's easier to just download particular module and place it in `node_modules` directory (that's where node.js searches for modules by default).~~

~~In other words to run this chat tutorial without `npm` download `websocket` module from github, unpack it, and rename the directory to `websocket`. Then place it in `node_modules` directory for example like this:~~

```
node/
node/node.exe
node/examples/
node/examples/chat.html
node/examples/chat.js
node/examples/server.js
node/examples/frontend.js
node/node_modules/
node/node_modules/websocket
node/node_modules/websocket/...
node/node_modules/websocket/...
...
```

## Download

~~download full source code (html, frontend, server) for this tutorial.~~
I put all source codes on GitHub, so feel free to modify whatever you want.

## Conclusion

That's all for this tutorial. I would be really glad if you could post to comment some examples of WebSocket usage that you liked (it doesn't have to be built on node.js).