

Get Started: Chat application

In this guide we'll create a basic chat application. It requires almost no basic prior knowledge of Node.js or Socket.IO, so it's ideal for users of all knowledge levels.

Introduction

Writing a chat application with popular web applications stacks like LAMP (PHP) has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it's a lot slower than it should be.

Sockets have traditionally been the solution around which most realtime chat systems are architected, providing a bi-directional communication channel between a client and a server.

This means that the server can *push* messages to clients. Whenever you write a chat message, the idea is that the server will get it and push it to all other connected clients.

The web framework

The first goal is to setup a simple HTML webpage that serves out a form and a list of messages. We're going to use the Node.js web framework `express` to this end. Make sure [Node.js is installed](#).

First let's create a `package.json` manifest file that describes our project. I recommend you place it in a dedicated empty directory (I'll call mine `chat-example`).

```
{
  "name": "socket-chat-example",
  "version": "0.0.1",
  "description": "my first socket.io app",
  "dependencies": {}
}
```

Now, in order to easily populate the `dependencies` with the things we need, we'll use `npm install --save`:

```
npm install --save express@4.10.2
```

Now that express is installed we can create an `index.js` file that will setup our application.

```
var app = require('express')();
var http = require('http').Server(app);

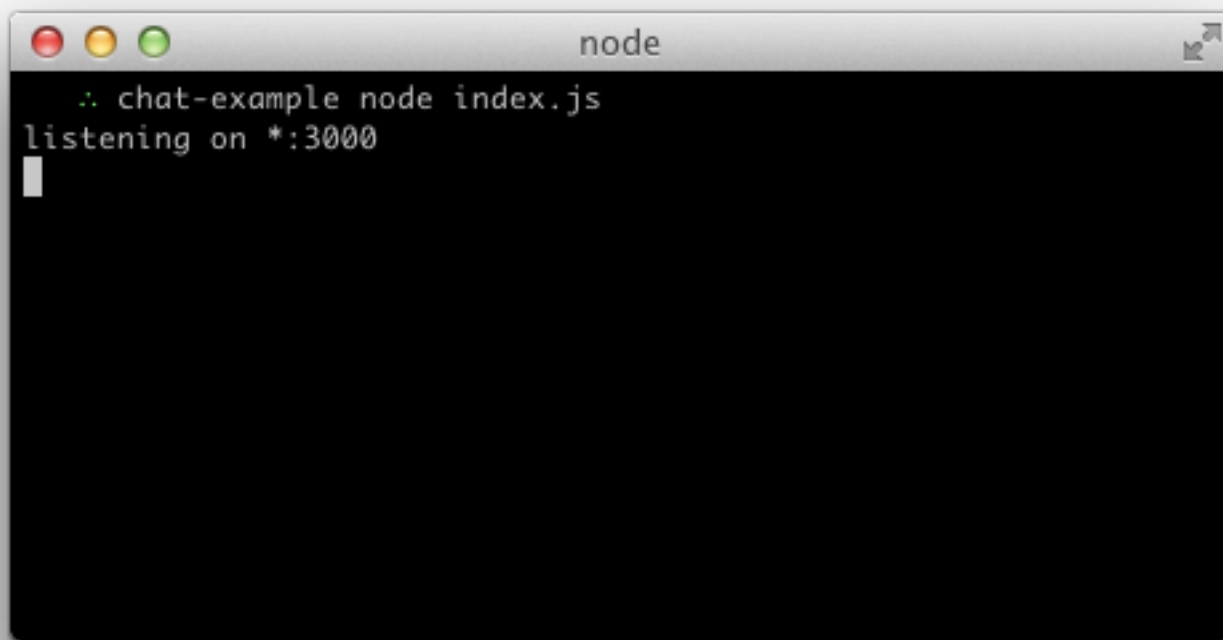
app.get('/', function(req, res){
  res.send('<h1>Hello world</h1>');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

This translates into the following:

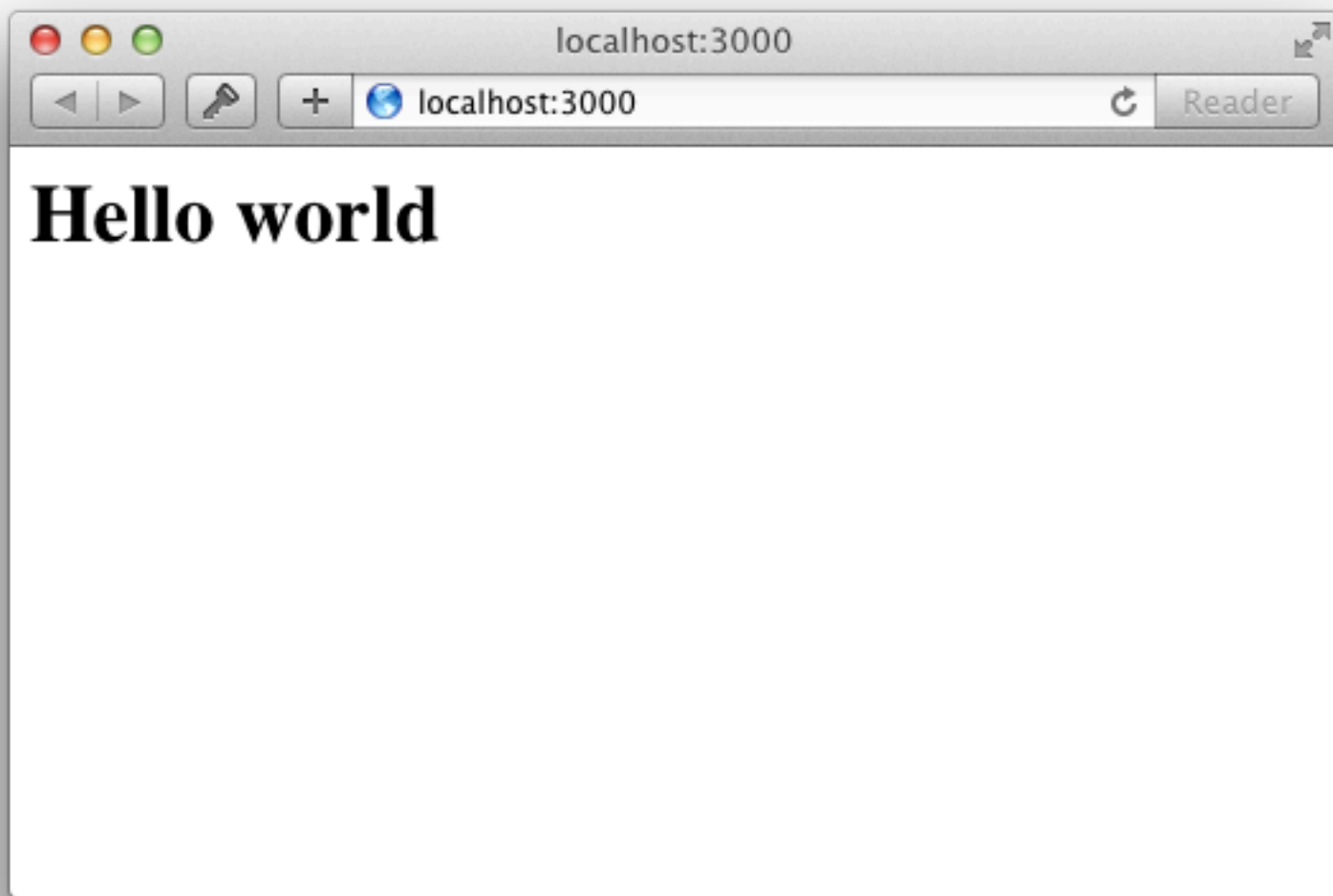
1. Express initializes `app` to be a function handler that you can supply to an HTTP server (as seen in line 2).
2. We define a route handler `/` that gets called when we hit our website home.
3. We make the http server listen on port 3000.

If you run `node index.js` you should see the following:

A screenshot of a terminal window titled 'node'. The window has a dark background and a light gray title bar with three colored window control buttons (red, yellow, green) on the left. The terminal shows the command `./ chat-example node index.js` being executed, followed by the output `listening on *:3000`. A white cursor is visible on the line following the output.

```
node
./ chat-example node index.js
listening on *:3000
```

And if you point your browser to `http://localhost:3000`:



Serving HTML

So far in `index.js` we're calling `res.send` and pass it a HTML string. Our code would look very confusing if we just placed our entire application's HTML there. Instead, we're going to create a `index.html` file and serve it.

Let's refactor our route handler to use `sendFile` instead:

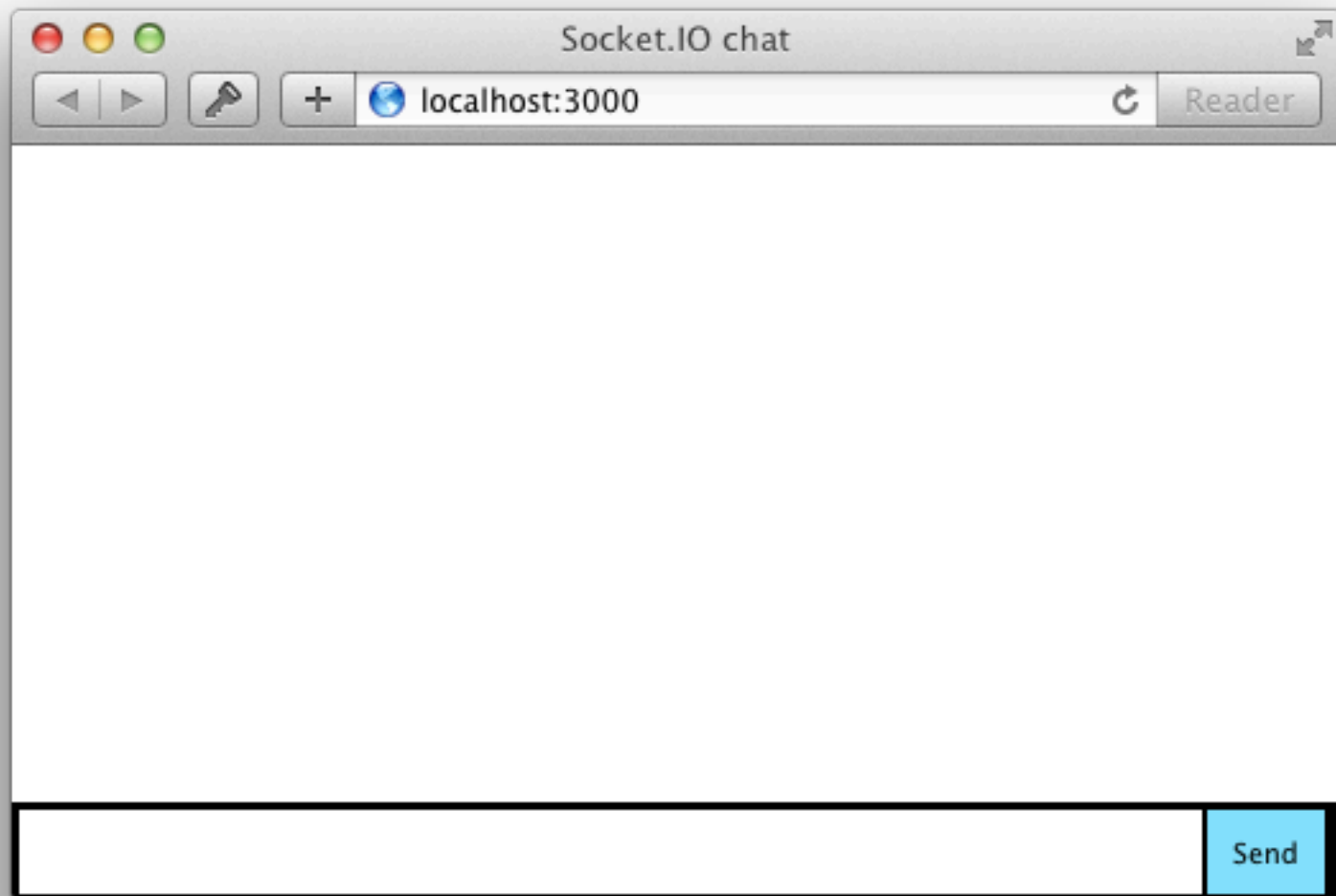
```
app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
```

And populate `index.html` with the following:

```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <style>
      * { margin: 0; padding: 0; box-sizing: border-box; }
      body { font: 13px Helvetica, Arial; }
      form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }
      form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }
      form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }
      #messages { list-style-type: none; margin: 0; padding: 0; }
      #messages li { padding: 5px 10px; }
      #messages li:nth-child(odd) { background: #eee; }
```

```
</style>
</head>
<body>
  <ul id="messages"></ul>
  <form action="">
    <input id="m" autocomplete="off" /><button>Send</button>
  </form>
</body>
</html>
```

If you restart the process (by hitting Control+C and running `node index` again) and refresh the page it should look like this:



Integrating Socket.IO

Socket.IO is composed of two parts:

- A server that integrates with (or mounts on) the Node.JS HTTP Server: `socket.io`
- A client library that loads on the browser side: `socket.io-client`

During development, `socket.io` serves the client automatically for us, as we'll see, so for now we only have to install one module:

```
npm install --save socket.io
```

That will install the module and add the dependency to `package.json`. Now let's edit `index.js` to add it:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile('index.html');
});

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

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Notice that I initialize a new instance of `socket.io` by passing the `http` (the HTTP server) object. Then I listen on the `connection` event for incoming sockets, and I log it to the console.

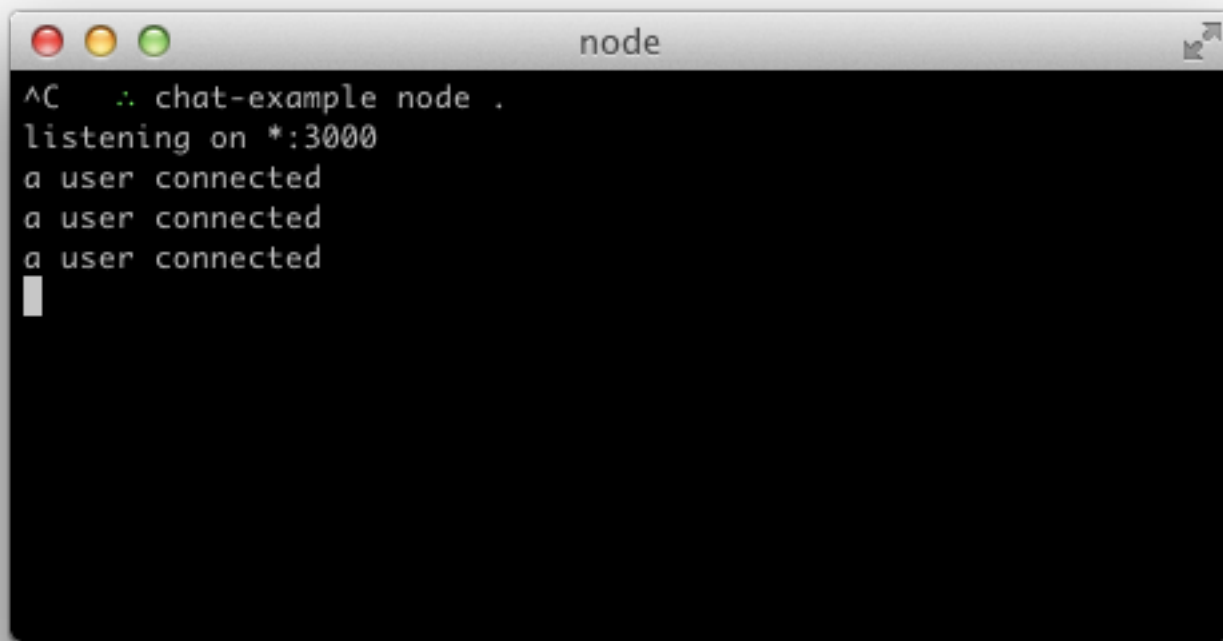
Now in `index.html` I add the following snippet before the `</body>` :

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();
</script>
```

That's all it takes to load the `socket.io-client` , which exposes a `io` global, and then connect.

Notice that I'm not specifying any URL when I call `io()` , since it defaults to trying to connect to the host that serves the page.

If you now reload the server and the website you should see the console print “a user connected”. Try opening several tabs, and you'll see several messages:



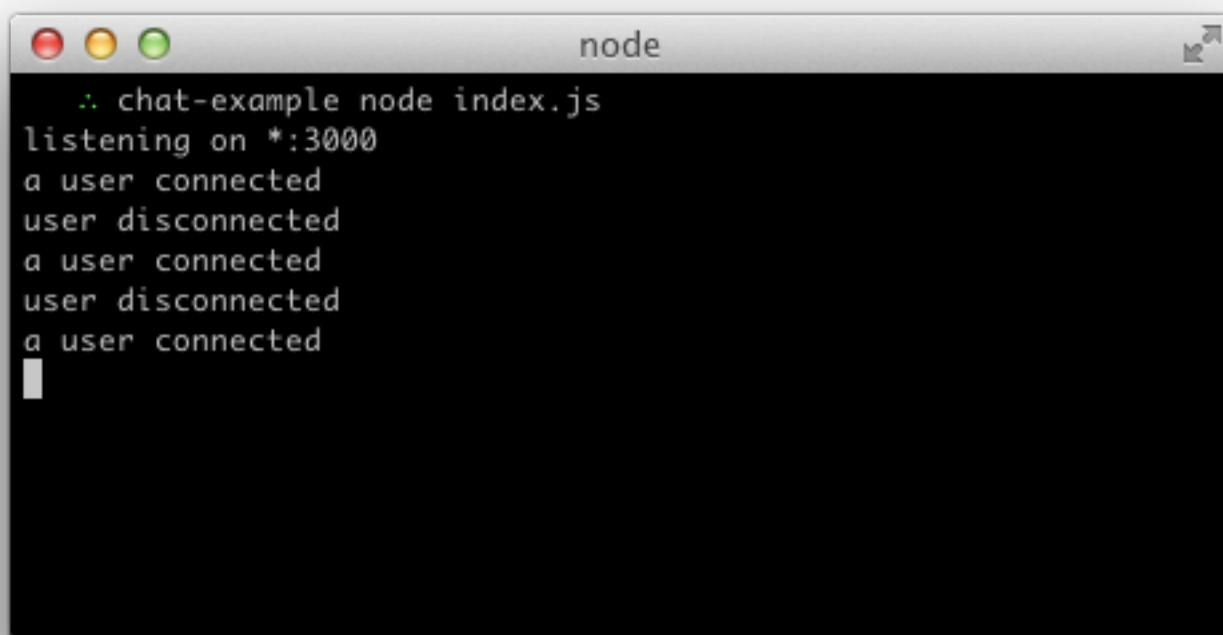
```
node
^C  ./chat-example node .
listening on *:3000
a user connected
a user connected
a user connected

```

Each socket also fires a special `disconnect` event:

```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

Then if you refresh a tab several times you can see it in action:



```
node
  ./chat-example node index.js
listening on *:3000
a user connected
user disconnected
a user connected
user disconnected
a user connected

```

Emitting events

The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too.

Let's make it so that when the user types in a message, the server gets it as a `chat message` event. The `script s` section in `index.html` should now look as follows:

```
<script src="/socket.io/socket.io.js"></script>
<script src="http://code.jquery.com/jquery-1.11.1.js"></script>
<script>
  var socket = io();
  $('form').submit(function(){
    socket.emit('chat message', $('#m').val());
    $('#m').val('');
    return false;
  });
</script>
```

And in `index.js` we print out the `chat message` event:

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    console.log('message: ' + msg);
  });
});
```

The result should be like the following video:

Broadcasting

The next goal is for us to emit the event from the server to the rest of the users.

In order to send an event to everyone, Socket.IO gives us the `io.emit` :

```
io.emit('some event', { for: 'everyone' });
```

If you want to send a message to everyone except for a certain socket, we have the `broadcast` flag:

```
io.on('connection', function(socket){
  socket.broadcast.emit('hi');
});
```

In this case, for the sake of simplicity we'll send the message to everyone, including the sender.

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

And on the client side when we capture a `chat message` event we'll include it in the page. The total client-side JavaScript code now amounts to:

```
<script>
  var socket = io();
  $('form').submit(function(){
    socket.emit('chat message', $('#m').val());
    $('#m').val('');
    return false;
  });
  socket.on('chat message', function(msg){
    $('#messages').append($('- ').text(msg));
  });
</script>

```

And that completes our chat application, in about 20 lines of code! This is what it looks like:

Homework

Here are some ideas to improve the application:

- Broadcast a message to connected users when someone connects or disconnects
- Add support for nicknames
- Don't send the same message to the user that sent it himself. Instead, append the message directly as soon as he presses enter.
- Add "{user} is typing" functionality
- Show who's online
- Add private messaging
- Share your improvements!

Getting this example

You can find it on GitHub [here](#).

```
$ git clone https://github.com/guille/chat-example.git
```

SOCKET.IO IS OPEN-SOURCE (MIT). RUN BY [CONTRIBUTORS](#).

 Follow @socketio

7,688 followers