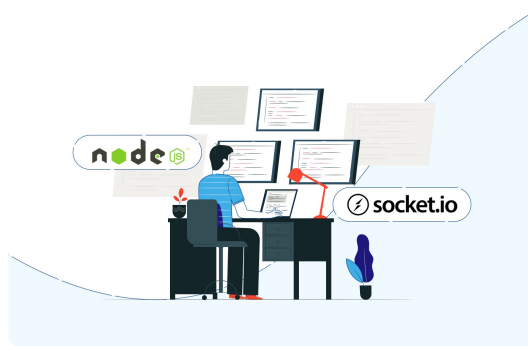


30 April 2020

Socket.io tutorial: Real-time communication in web development



Mateusz Pigula
Frontend Developer



There are a few ways of building a tool for real-time communication. Two of the most popular approaches presuppose the use of HTTP Long Polling or WebSockets. In the article below, you'll find a brief comparison of both solutions. I decided to focus on the latter solution a little bit more. That's why in the article, you'll find a straightforward Socket.io tutorial for building a real-time chat.

What are WebSockets?

WebSockets are quite an interesting technology in and of itself. Let's take a closer look at what it is.

WebSocket definition

WebSockets API is a technology providing a bidirectional communication channel between a client and a server. That means that the client no longer needs to be an initiator of a transaction while requesting data from both the server and database.

How do WebSockets work?

Typically, the client requests data starting a new connection and then the client loses connection. The server sending data also only has connection during the exchange. With WebSockets, during the first request to the backend server, except receiving incoming data, the socket.io client also establishes an initial connection. This process is known as the *WebSocket handshake*. 🤝 The Upgrade header is included in the request. That's how the client connected informs the server that it needs to create a connection.

See also: [A few simple tips on how to create WebSocket API in Node.js](#)

WebSockets vs HTTP

So, does it mean it's impossible to create a chat application without WebSockets? 😞

Well, there is a technique called HTTP Long Polling. Using this, the client sends a request and the server holds that opened until some new data is available. As soon as data appears and the client receives it, a new request is immediately sent and operation is repeated over and over again. However, a pretty big disadvantage of using HTTP Long Polling is that it consumes a lot of server resources. 😞

Are you ready to talk to your clients?

👤 Work with software development professionals who build scalable apps for big companies such as eSky, StageClip, or Reservix. Ranked as Poland's top software house by Clutch.

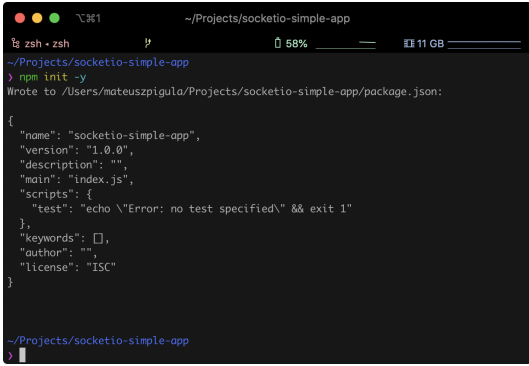
[Get a free consultation](#)

Socket.io chat room app tutorial – building a chat app



Below, I'll present to you a brief Socket.io tutorial on how to create a simple chat application with Vanilla JS frontend part and Node.js server. The Socket io library is a JavaScript library that enables real-time, bidirectional, event-based communication between the connected clients (browser) and the server side.

You need to create a project with the default `package.json` file.

A terminal window titled '~ / Projects/socketio-simple-app' showing the execution of 'npm init -y'. It displays the command, the file path where package.json was written, and the resulting JSON content for package.json.

```
zsh - zsh
~/Projects/socketio-simple-app
> npm init -y
Wrote to /Users/mateuszpigula/Projects/socketio-simple-app/package.json:

{
  "name": "socketio-simple-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

~/Projects/socketio-simple-app
```

Having this created, you need to set up a server and install all the needed dependencies. In order to do it, you need to create an `index.js` file and install `socket.io` and `express`. You can use the following command: `touch index.js && npm install express socket.io && npm install --save-dev nodemon .`

In `index.js`, you need to set up a local server and basic socket connection.

And basically, that's all you have to do on the backend side at the moment. Once you make a proper request, the connection should be established. It will be visible through

the logged message (as shown below).

```
1  const express = require("express");
2  const socket = require("socket.io");
3
4  // App setup
5  const PORT = 5000;
6  const app = express();
7  const server = app.listen(PORT, function () {
8    console.log(`Listening on port ${PORT}`);
9    console.log(`http://localhost:${PORT}`);
10 });
11
12 // Static files
13 app.use(express.static("public"));
14
15 // Socket setup
16 const io = socket(server);
17
18 io.on("connection", function (socket) {
19   console.log("Made socket connection");
20 });
```

socketio-server-setup.js hosted with ❤️ [view raw](#)
by [GitHub](#)

Now, you need to prepare the frontend part.

Let's start by creating the `public` folder and the following files:

`index.html` , `main.js` . Optionally, you can add `style.css` .

Below, you can see the

`index.html` scaffold. Basically, all you need are HTML tags you can refer to and some scripts from Socket.io included in the project. That's how your `index.html` file should look with the WebSockets scripts included.

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
```

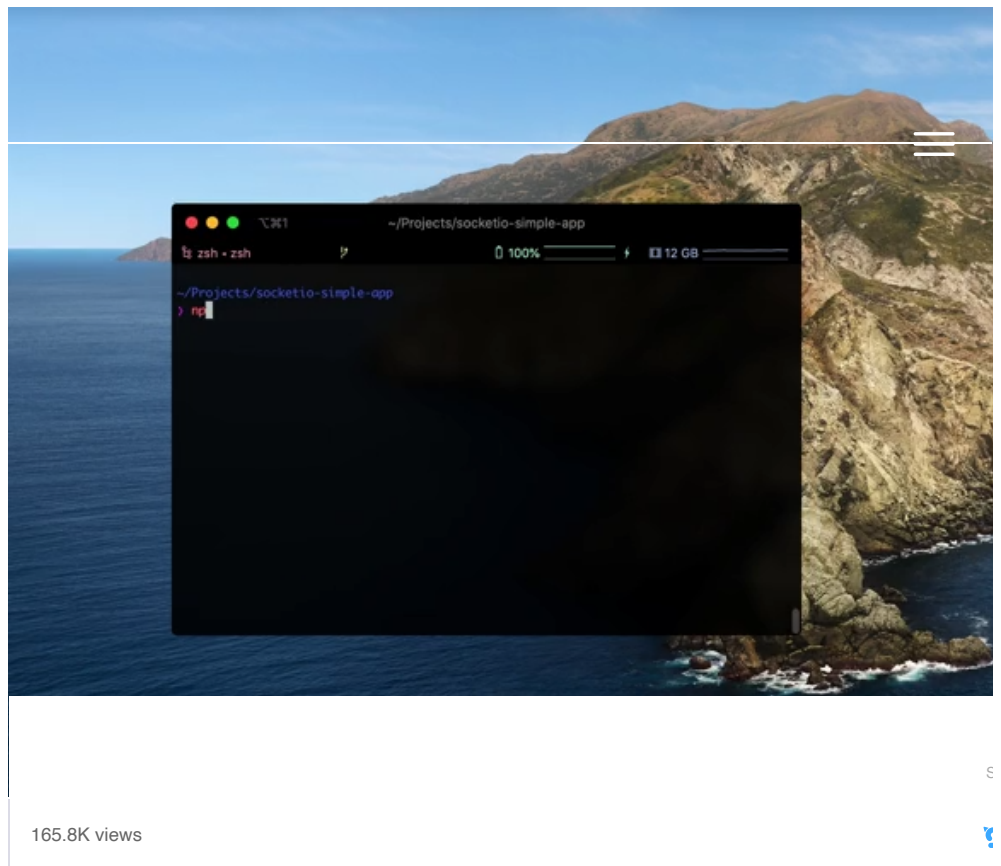
```

4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1"/>
6     <title>Socket.io simple chat</title>
7     <link rel="stylesheet" href="/static/css/main.css"/>
8 </head>
9 <body>
10    <div class="container">
11      <div class="inbox">
12        <div class="inbox__people">
13          <h4>Active users</h4>
14        </div>
15        <div class="inbox__messages">
16          <div class="messages__history">
17            <div class="fallback"></div>
18          </div>
19        </div>
20
21        <form class="message_form">
22          <input type="text" class="message_form__input" />
23          <button class="message_form__button">
24            Enter
25          </button>
26        </form>
27      </div>
28
29      <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"></script>
30      <script src="main.js"></script>
31    </body>
32  </html>

```

socketio-html.html hosted with ❤️ by [view raw](#)
[GitHub](#)

Then, you need to set up the connection on the frontend side. The only line of code you need in `main.js` is `const socket = io();`. It's shown in the video below.



As you can see – once the page is opened and the script is loaded, the connection is established. The next thing you need to do is to handle a new user connection, a new message and, last but not least – a currently typing user.

See also: [Strapi: Getting started with the Node.js headless CMS](#)

In socket, there are two ways to emit data such as events. One way is from the user to everyone (including user) and the other which emits an event to all the other instances. The idea is to display the list of all the active users. So, when a user connects – they need to inform about it and get the list of users who are already active.

```
1  const activeUsers = new Set();
```

```

2
3   io.on("connection", function (socket)
4     console.log("Made socket connection
5
6     socket.on("new user", function (data
7       socket.userId = data;
8       activeUsers.add(data);
9       io.emit("new user", [...activeUse
10    });
11
12    socket.on("disconnect", () => {
13      activeUsers.delete(socket.userId)
14      io.emit("user disconnected", sock
15    });
16  });

```

socketio-new-connection.js hosted with view raw
 by GitHub

When a user connects – they emit the event which includes the information about their username. You should set the property of `userId` on the socket. It will be needed when the user disconnects. To do that – you need to add a username to the Set of active users and emit an event with a list of all active users.

In the `main.js` I created two functions and two socket listeners.

```

1   const socket = io();
2
3   const inboxPeople = document.querySe
4
5   let userName = "";
6
7   const newUserConnected = (user) => {
8     userName = user || `User${Math.floa
9     socket.emit("new user", userName);
10    addToUsersBox(userName);
11  };
12
13  const addToUsersBox = (userName) => {
14    if (!!document.querySelector(`.${us
15      return;
16  }
17

```



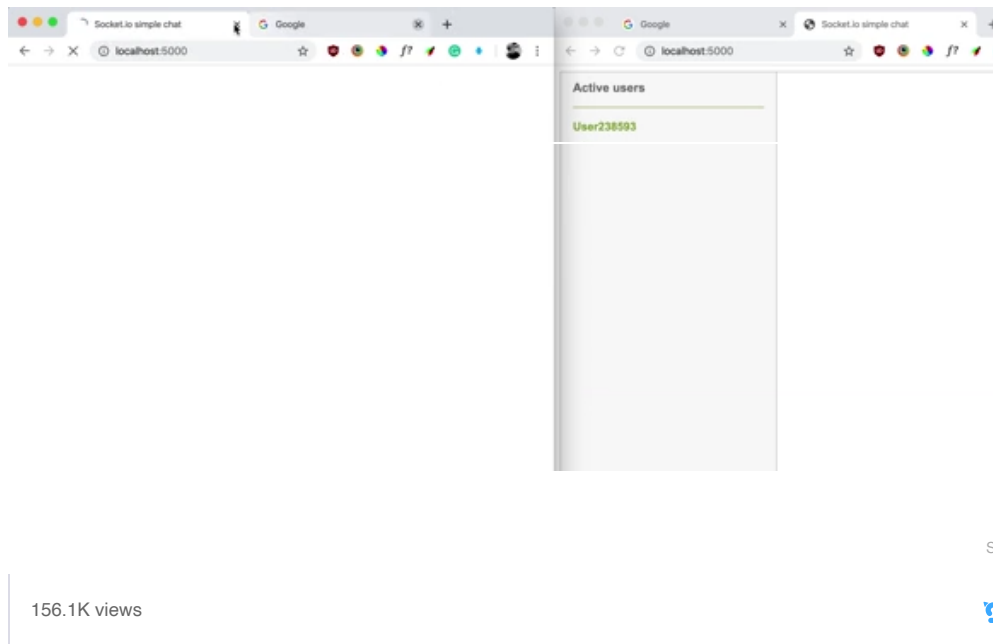
```

18   const userBox = `
19     <div class="chat_ib ${userName}-u
20       <h5>${userName}</h5>
21     </div>
22   `;
23   inboxPeople.innerHTML += userBox;
24 };
25
26 // new user is created so we generate
27 newUserConnected();
28
29 socket.on("new user", function (data) {
30   data.map((user) => addToUsersBox(user));
31 });
32
33 socket.on("user disconnected", function (data) {
34   document.querySelector(`.${userName}`)
35   });

```

socketio-new-connection-main.js [view raw](#)
 hosted with ❤ by GitHub

Firstly, I created a new user (you can easily extend functionality by adding some prompt with a real user name) and emit an event with that username. Also, I added them to the sidebar with all active users. When a user disconnects on the server side – they are removed from the Set of active users and a disconnection event with their username is emitted. After that – they are removed from the sidebar on the client side (as you can notice on the short video below).



Since we handled the way users can connect and disconnect, now it's time we handle the new messages.

On the server side – you do nothing more than adding the listener shown below.

```
1 socket.on("chat message", function (data) {
2   io.emit("chat message", data);
3 });
```

socketio-new-message-server.js hostedview raw
with ❤️ by GitHub

On the client side – you need to do a few more things, as per the example below.

```
1 const inputField = document.querySelector('input');
2 const messageForm = document.querySelector('form');
3 const messageBox = document.querySelector('div');
4
5 const addNewMessage = ({ user, message }) => {
6   const time = new Date();
7   const formattedTime = time.toLocaleTimeString();
8
9   const receivedMsg = `
10   <div class="incoming__message">
11     <div class="received__message">
12       <p>${message}</p>
```

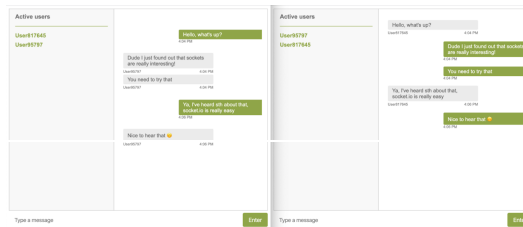
```

13     <div class="message__info">
14         <span class="message__author">
15             <span class="time_date">${fo
16         </div>
17     </div>
18 </div>`;
19
20 const myMsg = `
21 <div class="outgoing__message">
22     <div class="sent__message">
23         <p>${message}</p>
24         <div class="message__info">
25             <span class="time_date">${fo
26         </div>
27     </div>
28 </div>`;
29
30 messageBox.innerHTML += user === us
31 };
32
33 messageForm.addEventListener("submit"
34     e.preventDefault();
35     if (!inputField.value) {
36         return;
37     }
38
39     socket.emit("chat message", {
40         message: inputField.value,
41         nick: userName,
42     });
43
44     inputField.value = "";
45 });
46
47 socket.on("chat message", function (c
48     addNewMessage({ user: data.nick, me
49 });

```

socketio-new-message-client.js hosted view raw
with ❤️ by GitHub

The function responsible for displaying new messages is triggered right after receiving information from a socket. The message is pushed to the server in the listener function of the form submit. You just need to check if the client is a message author or not.



Now, I will show you how to emit the event responsible for informing all other connections except yours. We want to display info that a user is typing something at the moment.

On the server side, you need to add `socket.broadcast.emit` . Take a look below (it is the final version of the file).

```

1  const express = require("express");
2  const socket = require("socket.io");
3
4  // App setup
5  const PORT = 5000;
6  const app = express();
7  const server = app.listen(PORT, function () {
8    console.log(`Listening on port ${PORT}`);
9    console.log(`http://localhost:${PORT}`);
10 });
11
12 // Static files
13 app.use(express.static("public"));
14
15 // Socket setup
16 const io = socket(server);
17
18 const activeUsers = new Set();
19
20 io.on("connection", function (socket) {
21   console.log("Made socket connection");
22
23   socket.on("new user", function (data) {
24     socket.userId = data;
25     activeUsers.add(data);
26     io.emit("new user", [...activeUsers]);
27   });
28
29   socket.on("disconnect", () => {

```

```
30     activeUsers.delete(socket.userId);
31     io.emit("user disconnected", socket);
32 });
33
34 socket.on("chat message", function (data) {
35     io.emit("chat message", data);
36 });
37
38 socket.on("typing", function (data) {
39     socket.broadcast.emit("typing", data);
40 });
41 });
```

socketio-server-final.js hosted with ❤️ by GitHub

On the client side, you need to take a look at `inputField` `keyup` event listener and socket listener on `typing` . Below, you can see both things and the final version of `main.js` .

```

1  const socket = io();
2
3  const inboxPeople = document.querySelector('#inboxPeople');
4  const inputField = document.querySelector('#inputField');
5  const messageForm = document.querySelector('#messageForm');
6  const messageBox = document.querySelector('#messageBox');
7  const fallback = document.querySelector('#fallback');
8
9  let userName = '';
10
11  const newUserConnected = (user) => {
12    userName = user || `User${Math.floor(Math.random() * 1000)}';
13    socket.emit("new user", userName);
14    addToUsersBox(userName);
15  };
16
17  const addToUsersBox = (userName) => {
18    if (!!document.querySelector(`.user-${userName}`)) {
19      return;
20    }
21
22    const userBox = `
23      <div class="chat_ib ${userName}">
24        <h5>${userName}</h5>
25      </div>
26    `;
27    inboxPeople.innerHTML += userBox;
28  };
29
30  const addNewMessage = ({ user, message }) => {

```

```

31   const time = new Date();
32   const formattedTime = time.toLocaleString('en-US', {
33
34   const receivedMsg = `
35   <div class="incoming__message">
36     <div class="received__message">
37       <p>${message}</p>
38       <div class="message__info">
39         <span class="message__author">${user}</span>
40         <span class="time_date">${formattedTime}</span>
41       </div>
42     </div>
43   </div>`;
44
45   const myMsg = `
46   <div class="outgoing__message">
47     <div class="sent__message">
48       <p>${message}</p>
49       <div class="message__info">
50         <span class="time_date">${formattedTime}</span>
51       </div>
52     </div>
53   </div>`;
54
55   messageBox.innerHTML += user === user ? '' : `
56   `;
57
58   // new user is created so we generate a new user
59   newUserConnected();
60
61   messageForm.addEventListener("submit", (e) => {
62     e.preventDefault();
63     if (!inputField.value) {
64       return;
65     }
66
67     socket.emit("chat message", {
68       message: inputField.value,
69       nick: userName,
70     });
71
72     inputField.value = "";
73   });
74
75   inputField.addEventListener("keyup", (e) => {
76     socket.emit("typing", {
77       isTyping: inputField.value.length > 0,
78       nick: userName,
79     });
80   });
81
82   socket.on("new user", function (data) {
83     data.map((user) => addToUsersBox(user));
84   });
85

```

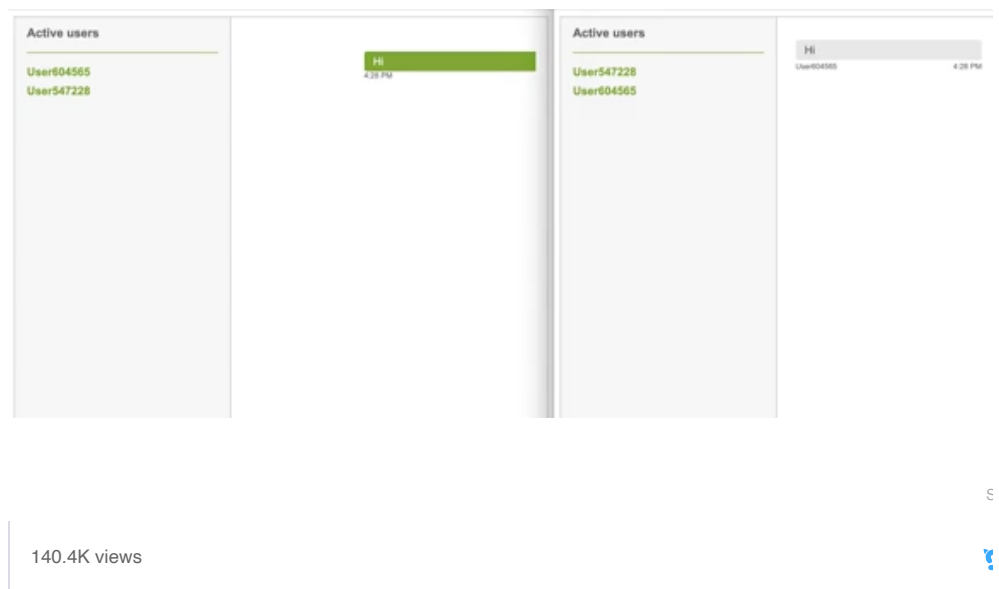
```

86 socket.on("user disconnected", function (data) {
87   document.querySelector(`.${userNick}`).innerHTML = "";
88 });
89
90 socket.on("chat message", function (data) {
91   addNewMessage({ user: data.nick, message: data.message });
92 });
93
94
95 socket.on("typing", function (data) {
96   const { isTyping, nick } = data;
97
98   if (!isTyping) {
99     fallback.innerHTML = "";
100    return;
101   }
102
103   fallback.innerHTML = `<p>${nick} is typing...`;
104 });

```

socketio-client-final.js hosted with ❤️ by [view raw](#) [GitHub](#)

And the final effect is presented in the video below. 🙌



See also: [Horizontal scaling with WebSocket – tutorial](#)

Socket.io chat app summary

As you can see, building a chat app (or any other web apps for that matter) equipped with basic chat messages features using Socket.io is not difficult. I hope that this brief Socket.io tutorial helped you understand just how simple using this approach is. And how many interesting things you can do with JavaScript and basic Node.js knowledge.

There are plenty of app ideas based on WebSockets. You can create:

an app that streams matches of your local sports team. You can stream games in the text version or video if possible (or maybe both? 🤔)

multiplayer game for you and your friends 🎮

secret chat available only for your group 🐼

app based on GPS, maybe a game that uses location? 🌐

I always strongly encourage you to learn more about other related concepts related to Socket.io such as socket namespace, socket object, state variable socket, Socket.io client, io server export default app, the message bus or broadcast flag. With that, you can create much more complex apps with multiple data points.

Your imagination is the limit!

**Mateusz Pięła**

Frontend Developer

Frontend Developer with a real fondness for CSS and data visualization. Mostly works with Vue and loves it for its simplicity. At the same time, Mateusz is respectful for React. He spends a lot of time learning new things, not limiting himself to technology stuff. Listens to music almost all the time.