LAST UPDATED 10TH MAY 2019 BY VALENTINO GAGLIARDI

Socket.IO, React and Node.js: Going Real-Time with WebSockets

Learn how to create a simple real-time application with Socket.IO, React and Node.js!

Looks like everybody is building chat apps with Socket.IO these days yet messaging applications are only the tip of the iceberg. Think a moment about it: there are a million of other things you can build within the real-time domain.

In the following post I would like to explore an interesting use case and hopefully give you s about what to build next with Socket.IO. We will start with some basic concepts all vay through exploring what Socket.IO and React can do for us when paired together. ıe end of the article you will build a super simple real-time application.

That will be quite a long post! Grab a cup of tea and take a seat before getting started!

Table of Contents

What you will learn

- what a WebSocket is
- how to Use Socket.IO and Node.js alongside with React

Requirements

To follow along with this tutorial you should have a basic understanding of JavaScript, Node.js, and ExpressJS. Also, make sure to get the latest version of Node.js. Last but not least, if you haven't got an API key from DarkSky yet, go grab one: we will use the DarkSky API later on inside our ExpressJS application.

The WebSocket protocol, Node.js and Socket.IO

WebSocket is an internet communication protocol with a relevant interesting feature: it provides a full-duplex channel over a single TCP connection.

With WebSockets, a client and a server will be able to talk to each other in real time, like they were involved in a telephone call: once connected, a client will be able to receive data from the server, without any need to continuously refresh the web page. On the other hand the server will also be able to receive data in real time from the client inside the same connection.

What's more interesting is the WebSockets ability to work with an event-driven model: ı the server and the client can react to events and messages.

WebSockets opened up an entire world of opportunities for web developers. If you're wondering how to implement this fantastic technology into your Node.js applications, well, the answer is **Socket.IO**, one of the most popular real-time engines for Node.js.

Socket.IO is one of the most popular real-time engines for Node.js

Socket.IO works by the means of **Node.js events**: you can listen for a connection event, fire up a function when a new user connects to the server, emit a message (basically an event) over a socket, and much more.

Socket.IO is used by countless companies and developers. It found its way through instant messaging applications, real-time analytics and monitoring, and it is used also for streaming and document collaboration.

However **one thing to keep in mind is that Socket.IO is not an WebSocket implementation**. The authors state that "Socket.IO indeed uses WebSocket as a transport when possible but a WebSocket client will not be able to connect to a Socket.IO server, and a Socket.IO client will not be able to connect to a WebSocket server".

Besides that, the framework behaves exactly like WebSockets and here lies its power. With this in place and with a basic understanding of the Websocket protocol it's time to get our hands dirty.

Socket.IO, React and Node.js: preparing the oject

To start off create an empty directory named socket-io-server and move into it:

```
mkdir socket-io-server && cd $_
```

Then initialize the package.json by running:

```
npm init -y
```

We won't publish any module to NPM so you can safely accept the default choices and just move on.

We also need to install Socket.io, which is the main dependency of our project, ExpressJS, and Axios. Express will help us building the server and Axios will be used to make HTTP requests to the DarkSky API:

```
npm i axios express socket.io
```

Socket.IO, React and Node.js: hands-on

So, the idea behind our little project is simple: Caty wants to know the **current temperature** in Florence and maybe also have the possibility to get an update every 10 seconds.

You might be tempted to put a call to DarkSky inside the **componentDidMount** method of a React component. Maybe you should poll the API every 10 seconds with a call to setInterval directly inside componentDidMount? Luckily, there are better ways: in our case, a simple real-time server will get the job done.

The server will use Socket.IO to emit a message every 10 seconds and the client will listen for the same message over a real-time socket. Sounds neat? It is. Am I going to put React into the mix when I could have simply render the HTML with Pug or Jade? Yes, because it is very interesting to see how React can work alongside with Socket.IO.

ll see soon. Create a file named **app.js** inside your project's directory. This will hold the al server:

```
const express = require("express");
const http = require("http");
const socketIo = require("socket.io");
const axios = require("axios");

const port = process.env.PORT || 4001;
const index = require("./routes/index");

const app = express();
app.use(index);

const server = http.createServer(app);

const io = socketIo(server); // < Interesting!

const getApiAndEmit = "TODO"
```

The code above should be no mystery for you: it is a bunch of requires followed by the call to a new ExpressJS application. What's rather interesting there is the call to socketIo() to initialize a new instance by passing in the server object. By doing so we have wired up the ExpressJS server to Socket.IO.

You should also have noticed a empty function:

```
const getApiAndEmit = "TODO"
```

we will fill it with some meaningful code next up.

Notice also how the application calls the index route: even if the server won't serve any HTML content we will need a very simple route in order to listen for any incoming connection.

Create a file named **index.js** inside a **routes** directory:

```
const express = require("express");
const router = express.Router();

router.get("/", (req, res) => {
  res.send({ response: "I am alive" }).status(200);
});

module.exports = router;
```

and you're done.

Socket.IO, React and Node.js: designing the server

The first and most important method you will encounter while working with Socket.IO is on(). The on() method takes two arguments: the name of the event, in this case "connection" and a callback which will be executed after every connection event. on() is nothing more than a core Node.js method tied to the EventEmitter class.

The connection event returns a **socket object** which will be passed to the callback function. By using said socket you will be able to send data back to a client in real time.

If you remember, Caty wants to know the temperature every 10 seconds: we can use setInterval inside the callback, and inside setInterval we can use another arrow function which will call the getApiAndEmit function we saw earlier. The code should be really straightforward:

```
io.on("connection", socket => {
  console.log("New client connected"), setInterval(
    () => getApiAndEmit(socket),
    10000
  );
  socket.on("disconnect", () => console.log("Client
disconnected"));
});
```

Notice also how we can listen for the **disconnect event**. It will be fired as soon as a client disconnects itself from the server. For now we will just print a simple message to the console.

IMPORTANT: As pointed out by some fellow readers, the code above has a flaw: it creates a new interval for every connected client. While Socket.IO was born to handle many concurrent connections our example assumes that only one user will visit the page: you. If you were to put that code in production, just don't. A more serious version of the above snippet would clear the interval upon subsequent connections:

```
let interval;

io.on("connection", socket => {
  console.log("New client connected");
  if (interval) {
    clearInterval(interval);
  }
  interval = setInterval(() => getApiAndEmit(socket), 10000);
  socket.on("disconnect", () => {
    console.log("Client disconnected");
  });
});
```

Anyway, for the scope of this post it's completely fine to go without clearing the interval id at all.

Socket.IO, React and Node.js: implementing the server

Now we can make the application listen for incoming connections:

```
server.listen(port, () => console.log(`Listening on port ${port}`));
```

Do you remember our **getApiAndEmit** function? It takes the socket as an argument. The **socket** is nothing more than the communication channel between client and server. We can write whatever we want inside it by emitting a message:

```
const getApiAndEmit = async socket => {
  try {
    const res = await axios.get(
      "https://api.darksky.net/forecast/PUT_YOUR_API_KEY_HERE/43.7695,11.2558"
    ); // Getting the data from DarkSky
    socket.emit("FromAPI", res.data.currently.temperature); // Emitting a new message. It will be consumed by the client
  } catch (error) {
    console.error(`Error: ${error.code}`);
  }
};
```

function takes the socket as an argument, makes an HTTP request to the DarkSky (don't forget to fill the url with your actual API key), and finally emits the message

"FromAPI" which will contain the current temperature value for the given coordinates.

The emitted message can be intercepted by the Socket.IO client (React in our case).

Server wise we are done and the complete code for **app.js** should look like this:

We can test our server by starting the application with:

```
node app.js
```

As soon as the server starts you'll see the following output: "Listening on port 4001" which confirms that everything is working fine!

Socket.IO, React and Node.js: implementing the React client

Now that we have our tiny real-time server in place it's time to make sense of our data. We want to display the current temperature in Florence and we will use React to get the job done.

Why? Because React does exactly what its name implies: it reacts to state changes. Our server will emit a message containing the current temperature which will be updated every 10 seconds.

React can store the temperature value inside a component's state and render only the piece of text subject to changes. If it's your first time with React don't waste time with build tools and use create-react-app:

```
npx create-react-app socket-io-client
```

(Note that you should create the project outside the folder where the server lives). Then move inside the project and install the Socket.IO client:

```
npm i socket.io-client
```

To keep things simple we will just use the **App.js** component which lies inside the src directory. Open up App.js. You can safely remove all the content inside the file and replace the code with the following:

```
import React, { Component } from "react";
import socketIOClient from "socket.io-client";

class App extends Component {
  constructor() {
    super();
    this.state = {
      response: false,
      endpoint: "http://127.0.0.1:4001"
    };
  }

  componentDidMount() {
    const { endpoint } = this.state;
    const socket = socketIOClient(endpoint);
    socket.on("FromAPI", data => this.setState({ response: data }));
  }

  render() {
    const { response } = this.state;
    return (
        <div style={{ textAlign: "center" }}>
          {response
              ? <p>
                The temperature in Florence is: {response} °F
              </p>
              : <p>Loading...</p>}
        </div>
    );
  }
}

export default App;
```

Now save and close the file and go straight to the next section!

Socket.IO, React and Node.js: pairing up back-end and front-end

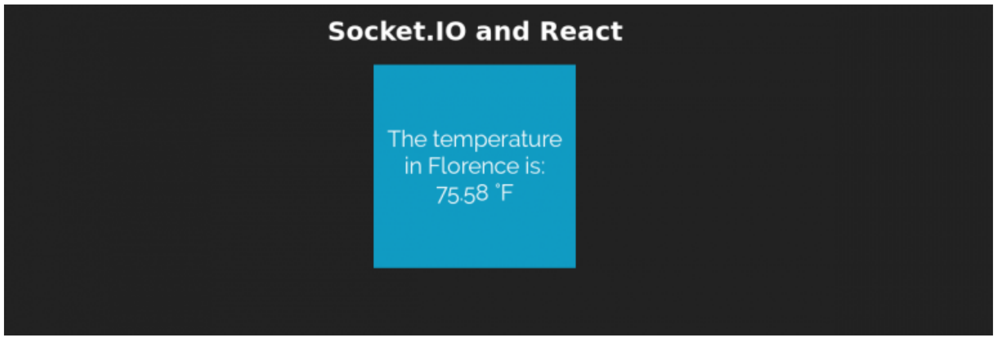
Now **open a terminal**, go into the server folder and start the Socket.io server:

```
cd socket-io-server

node app.js
```

In **another terminal** go into the client folder and start the React project:

```
cd socket-io-client

npm start
```

Wait 10 seconds and you should see the following output (I've added some styling to my component, feel free to add some CSS to your App.js too):



If you keep an eye on the page you'll notice the temperature changing over time, every 10 seconds. It's the magic of Socket.IO: as soon as the React component gets mounted, the **componentDidMount creates a new connection to our Socket.IO server** by instantiating a new socket.

Remember, the socket is a communication channel and we're able to listen for any event happening inside it. If you take a look at the server-side code the "FromAPI" message/event gets fired as soon as a new client connects to the server.

The client can listen for the event with the on() method and do something with the data contained inside the message/event. In our case we simply want to store the temperature inside our component's state. That's it! Once established the connection will receive the updates from the server without any need to refresh the page!

ting from 2019 there is no need to use ES6 classes for storing a component's state.
ı React Hooks even a function will do.

Socket.IO, React and Node.js: Where to go from here

Even if our example was quite basic it should be clear how Socket.IO is not only suitable for instant messaging but also for a vast range of purposes: the limit is only in our creativity.

I suggest exploring the Socket.IO's documentation to learn more about Rooms, Namespaces and other API methods: Socket.IO Docs. Also, a good understanding of the Node.js event-driven architecture will be useful for mastering Socket.IO: start with the official docs.

Thanks for reading and stay tuned on this blog!

Valentino Gagliardi

Hi! I'm Valentino! Educator and consultant, I help people learning to code with on-site and remote workshops. Looking for JavaScript and Python training? Let's get in touch!

NODE.JS