

## Real-time web applications

Welcome to the final assignment! In this class, you will learn about writing real-time web applications. As you probably (hopefully) know, a standard HTTP connection works in a pretty straightforward fashion – request, response.

That's it...

In order to achieve a sense of real-time communication we would have to periodically ask the server in short intervals (the shorter the intervals, the more real-timey the app) if it's got anything new for us. This is known as polling (you may also see it being called short polling), and as you can probably guess it is usually a pretty inefficient way of going about this problem. Sure, it may be a good solution if your requests are not very frequent (e.g., you update sth every minute or so), however, here we are talking real-time, so it's basically a no go.

Believe it or not, there are actually several alternative solutions to this problem, but each does it in a different way and each one is suited for a slightly different problem. First (in contrast to short polling) we have...

...drumroll please...

long polling! This too is an HTTP-based method and similarly to regular polling, it requires the client to issue a request first in order to get any information. However, unlike in regular polling, the server doesn't respond immediately, but only once it has something new to offer (e.g., the requested data are available or the desired event happens). After receiving the response from the server, the client immediately sends another request and the process starts over.

The third option is [Server-Side Events](#), which is carried out through the use of the [EventSource](#) interface. This is achieved by opening a persistent HTTP connection, however, this connection is unidirectional and the messages can only be sent from the server to the client (like the name suggests...). So this would be perfect for receiving some live updates from the server, like maintaining a fresh news feed.

The fourth and final option which we'll talk about is the [WebSocket](#) protocol, which establishes a bidirectional communication channel between the client and the server. As you can probably sense, it is by far the most capable one, however, just remember that you'll not always need it and you always have the three other options mentioned above. It is very easy to forget about them when you have such a convenient tool as WebSockets ("If all you have is a hammer, everything looks like a nail").

You can get a general sense of the differences between these 4 techniques by reading [this](#) answer from stackoverflow.

In this class, we'll rely on WebSockets and possibly long pooling.

Wait. What do you mean “possibly”? How can you not know!?

Well, let me explain. We’ll use the [socket.io](https://socket.io) engine, which provides a higher level access to the real-time goodness offered by WebSockets and (if WebSockets are unavailable) long pooling. The selection of an appropriate technology is done seamlessly by this library, so we won’t have to worry about it too much. We’ll just use the fun stuff:-)

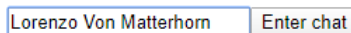
Now, after this brief introduction, we’re ready to go. In this assignment, you’ll build the prototypical WebSocket application – a chat app. What online stores are to traditional web applications and todo lists are to single-page web applications – chat apps are to WebSockets. In the first part of the exercise I’ll try to lead you by hand so that you’ll get your bearings. Then, I’ll let you loose on some tasks to do.

Let us begin!

In this assignment, I would like you to use all the knowledge that you’ve gathered over the duration of this course so far. So, our server will be written in Node.js while the client side will be written in React. If you didn’t complete the React assignment, you can write the client in plain HTML + JavaScript. You should be able to infer how to do it from the React examples. If you didn’t complete the Node assignment – you’ll catch up – it’s actually pretty easy:-)

First, let me lay out the functionality of the application, so that if you want to do it on your own, you won’t have to follow the tutorial step by step.

The client starts off by allowing the users to enter their nick, like so:



Lorenzo Von Matterhorn Enter chat

After that, the user immediately joins the main chat room and can start broadcasting messages to all those who are in. The user also sees a list of currently active users. Something like this:

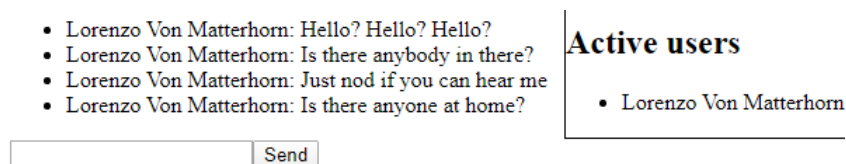


Send

**Active users**

- Lorenzo Von Matterhorn

When a user sends a message, it is broadcasted to all users and displayed with his nickname next to it, like so:



- Lorenzo Von Matterhorn: Hello? Hello? Hello?
- Lorenzo Von Matterhorn: Is there anybody in there?
- Lorenzo Von Matterhorn: Just nod if you can hear me
- Lorenzo Von Matterhorn: Is there anyone at home?

Send

**Active users**

- Lorenzo Von Matterhorn

When a user joins or leaves the chat all other users are notified about this fact like so:

- Lorenzo Von Matterhorn: Hello? Hello? Hello?
- Lorenzo Von Matterhorn: Is there anybody in there?
- Lorenzo Von Matterhorn: Just nod if you can hear me
- Lorenzo Von Matterhorn: Is there anyone at home?
- ladiesman217 has joined the chat!

### Active users

- Lorenzo Von Matterhorn
- ladiesman217

When either of the users sends a message, all other users see it instantly.

- Lorenzo Von Matterhorn: Hello? Hello? Hello?
- Lorenzo Von Matterhorn: Is there anybody in there?
- Lorenzo Von Matterhorn: Just nod if you can hear me
- Lorenzo Von Matterhorn: Is there anyone at home?
- ladiesman217 has joined the chat!
- ladiesman217: Yo!
- ladiesman217: Have you seen my car?
- ladiesman217: old...
- ladiesman217: yellow...
- ladiesman217: transforms into a giant robot...

### Active users

- Lorenzo Von Matterhorn
- ladiesman217

And that's pretty much it.

We'll do the basic messaging part together and then you'll add the rest on your own. We'll write the server first and then add the client.

1. To keep things in order, create a **chat** folder and a **server** subfolder inside it, then go to the **server** folder and initialize a new project using command `npm init` (you can leave all entries that you'll be asked for as default or complete them as you'll see fit). After that, install the socket.io module using `npm install socket.io`. Now create an empty file called `server.js` and put the following code inside.

```
const io = require("socket.io");
const server = io.listen(3000);

server.on('connection', (socket) => {
  console.log('connected');
  server.emit('message', 'Hello, World!');
});
```

Believe it or not, you have almost all the tools you'll need on the server side! Let's explain them a bit, shall we? In the first two lines, we load the socket.io module and start the server to listen on port 3000. Then, we define an event listener on the `connection` event (or `connect` – they are synonyms). This is one of a few predefined server-side events which gets fired whenever somebody connects to our server. In the callback to this event, we get the socket that has been established for this connection. It will be useful later, however, for now we'll just log the fact that somebody has connected to the server in the console and send a `'Hello, World!'` message to everyone who is connected (including the newly connected user). The `emit` function works by emitting an event named like the first parameter (in our case it's `'message'`, but you could use any name you want)

and sending whatever data we provide in the next parameters with it (in our case it's just a simple 'Hello, World!' text, but we could send as many values as we want).

Now, it would be nice to be able to test our server. To do it, let's write a simple client.

2. First, we'll write the most basic client that I can think about, just so that you see what's the absolute minimum required to get this thing going. Then, we'll move on to write a proper React client. In your **chat** folder create a new html file called **test.html** and place the following code inside.

```
<html>
<head>
  <script src="http://localhost:3000/socket.io/socket.io.js"></script>
  <script>
    var socket = io('http://localhost:3000');
    socket.on('message', (message) => {
      document.getElementById('root').append(message);
    })
  </script>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

As you can see all we need to do is to establish the connection and we can start listening (and emitting if we wanted to). In this case, we are waiting for the 'message' event to happen, and when it does, we take the message and append it to our div.

3. Now it's probably a good time to test our solution! First, start our server either by using **node server.js** command or **nodemon server.js** (if you have nodemon installed... if not – I highly recommend you do it: **npm install nodemon -g**). After that, just open our test file in the browser, preferably in several tabs, and see how it works.
4. Ok. Now we'll start working on our real client (you can remove the **test.html** file). Go to the **chat** folder and create a new app called **client** using the **create-react-app** module: **npm create-react-app client**. Now go to the **client** directory and install the **socket.io-client** module: **npm install socket.io-client**. Now we're ready to go.

5. Open the **App** component and replace whatever you'll find in there with the following code. I'm using functional components, but if you prefer using class components – be my guest! If you're not too familiar with functional components just know that the `useEffect` function as we're using it here works basically the same as the `componentDidMount` event in class components while `useState` signifies... well... using state:-) However, I highly recommend you familiarize yourself with the basics of using functional components as they are much cleaner and easier to use than class components in most cases.

```
import React, { useState, useEffect } from "react";
import io from "socket.io-client";
const socket = io("http://localhost:3000");

export default function App() {
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    socket.on('message', data => {
      setMessages(m => [...m, data]);
    });
  }, []);

  return (
    <div>
      <ul>
        {messages.map(m => <li>{m}</li>)}
      </ul>
    </div>
  );
}
```

As you can see, it works exactly the same as in our test client (except for all the React stuff). We load the **socket.io-client** module, establish a connection with our server, and set up an event listener. Now go ahead and test it. To start the client type **npm start** in the **client** folder. Again, preferably open the client in several tabs.

6. Ok. Now that we've got the basics out of the way, let's implement a basic chat. We're actually not that far off! We'll start with the client as here we've got a bit more to do. To keep things clean, we'll create a separate Input component, which will be responsible for entering and sending messages. That's right – a textbox and a button. Just remember to implement it as a [controlled component](#). When the form is submitted, we'll call the `send` function (passed to our component as a parameter) with our message to send and clear the textbox. To sum up, we want a component with a form containing a text input and a submit input. I highly encourage you to try doing it on your own, since after the React assignment it should be easy and there's nothing WebSocket'y about it! However, if you'll experience some difficulties, you have the solution below.

```
import React, { useState } from "react";

export default function Input({ send }) {
  const [text, setText] = useState('');

  const handleChange = (e) => {
    setText(e.target.value);
  };

  const handleSubmit = (e) => {
    send(text);
    setText('');
    e.preventDefault();
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={text} onChange={handleChange} />
      <input type="submit" value='Send' />
    </form>
  );
}
```

7. Now, we just have to use the `Input` component in our `App` component and implement the `send` function to pass on to it. Write the `send` function first. It should accept a message and send it to the server using `socket.emit('message', message)`. Then, import the `Input` component, call it below the list of messages and pass the `send` function to it. After that, the `App` component should look like this.

```
import React, { useState, useEffect } from "react";
import io from "socket.io-client";
import Input from "../Components/Input"
const socket = io("http://localhost:3000");

export default function App() {
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    socket.on('message', data => {
      setMessages(m => [...m, data]);
    });
  }, []);

  const send = (message) => {
    socket.emit('message', message);
  }

  return (
    <div>
      <ul>
        {messages.map(m => <li>{m}</li>)}
      </ul>
      <Input send={send} buttonText='Send' />
    </div>
  );
}
```

8. We're almost there. We just have to make a small tweak on the server side. Here, after a new user is connected, instead of sending 'Hello, World!', we'll just wait for new messages to arrive and send them to all users. The server code should now look like this.

```
const io = require("socket.io");
const server = io.listen(3000);

server.on('connection', (socket) => {
  console.log('connected');
  socket.on('message', (message) => {
    server.emit('message', message);
  });
});
```

Pay attention to the fact that the 'connection' event listener is bound to the server while the 'message' event listener is bound to the specific socket, so each newly connected user gets a new socket. You can identify each socket (and as a consequence each user) using `socket.id`.

9. And that's it! You can test the app and you should see a functioning chat.
10. I think you're almost ready to do the rest of the app on your own. Just a few more useful pieces of code before I let you go.

On the server side:

- `socket.emit('some event', 'some data');` – to send data only to a specific user. You can use it for example to notify a user that he successfully entered the chat, so that the UI can change from entering a nickname to sending messages.
- `server.emit('some event', 'some data');` – to send data to everybody. Apart from sending messages, you can also use it to send the updated list of users every time a user connects or disconnects.
- `socket.broadcast.emit('some event', 'some data');` – to send data to everyone except the user who sent it. This one you can use to inform other users that a new user entered or left the chat.
- `socket.on('disconnect', () => { ... });` – a specific event (just like 'connect') which occurs when a given connection ends. You can use it to detect when to notify other users that some user has left the chat and to update the users list.
- `socket.id` – ... You can use it to associate a nickname (user) with a given socket.

On the client side you're pretty much covered with `socket.on('some event', data => { ... });` and `socket.emit('some event', data);`, both of which you already know.

If I were you, I would start doing the rest of the assignment as follows.

- In the client, add `isLoggedIn` flag to the state of the App component and create a `login` function, which will emit a 'login' event sending a user's nickname passed as a parameter. Then, using conditional rendering, if the user is logged in, display the chat, and if the user is not logged in, display a div with h1 asking for a nickname and our previously defined `Input` component with `login` function passed as the `send` parameter.



- On the server, add a `'login'` event listener, which emits back to the user an event which will confirm that he/she successfully logged in (e.g., `'loggedIn'`) and broadcasts to all other users a message that a given username has joined the chat.
- Back in the client, in the `useEffect` hook (componentDidMount when using class components) add a `'loggedIn'` event listener and simply change the `isLoggedIn` value to true.

And you're one step closer to the final solution!

After that, it should be fairly straightforward. For example, to add user nicknames appearing before each message, on the server, you can create a global user array and add a new entry to it every time a new user logs in. The entry could look like this: `{ nick: nick, socketId: socket.id }`, where the `nick` is from the `'login'` event. After that, whenever a user sends a new message, you simply lookup his/her `nick` in the array based on the `socket id` and prepend it to the message before you emit it to everyone.

I think you'll figure out the rest on your own.

Good luck!