Set Up a Connection Over WebSocket: Video Call with WebRTC Step 2

Dornhoth
Mar 19 · 5 min read ★

Photo by Cytonn Photography on Unsplash

We want to create a video chat using the WebRTC protocol. We saw in the previous article how to access the webcam and microphone streams from the browser. We now want to establish a P2P connection between two users.

The WebRTC connection between your local browser and a remote user (peers) will be represented by the JavaScript interface RTCPeerConnection. But you have to coordinate this connection first, exchanging messages between peers to find each other, control the communication and then terminate it. This is the signaling process. The signaling process is not part of the WebRTC specifications, you are free to use whatever messaging protocol you want to establish and control the connection. You could technically deliver those messages per post, but a common solution is to use the WebSocket protocol.

In this article, we are going to allow two users to communicate over WebSocket. We are going to build an example with a Node server, in which peers can create a connection, say hello to each other, and close the connection.

WebSocket

WebSocket is a communication protocol, kind of a concurrent to HTTP. Like HTTP, WebSocket enables the communication between a client (a browser) and a server.

When communicating over HTTP, the server can only react to the client's requests. It doesn't remember anything about previous requests, HTTP is stateless. WebSocket, on the other hand, lets you open a channel between the client and the server. Once this channel is open and until it gets closed, the communication can happen both ways: the server can send data when it likes, and so can the client.

WebSocket is actually designed to work over HTTP, so we can use a normal web server to build it on. We are going to implement a WebSocket communication between our client and a Node server.

Set up the Node project

We are going to use a Node server for this with a library implementing WebSocket: WebSocket-Node.

If you don't already have it on your machine, you should install Node.js. Create the folder in which you want to have your WebSocket project (I named mine *web-socket*). Then into that folder initialize your project:

```
npm init
```

Create a file index.js and install the WebSocket-Node library using npm:

```
npm install websocket --save
```

Following the documentation of the library, we first need to define an HTTP server. Ours will listen to the port 1337. We can then create the WebSocket server on top of it.

We want to do something really simple to understand how WebSocket works. Everyone can connect and send messages through our server. When a user sends a message, every other connected user will receive it.

We need to keep track of the connected users, we do so in the *clients* array. When a user requests a connection, we first accept it and we notify every other connected user. We generate a random id and new connection with this id is then added to the array.

When the user sends a message, the event *message* is emitted. Every connected user, except the one sending the message, is sent some data containing the text of the message and the index of the user sending it.

Finally, when a user disconnects, he is removed from the *clients* array and every other user gets notified about it.

Let's now check if this is working by building a simple user interface.

Client

We are not going to build anything complicated in the client, we just want to make sure the communication is working. We will display three buttons:

- one button to connect
- one button to send a "Hello!" message
- one button to disconnect

When a user isn't connected, only the first button is enabled, otherwise only the two last buttons are.

Under the buttons, we display some area (called *console* in the code) where the messages are going to be printed.

You can create a new folder for the client code (I named mine *client*). Once again, we are keeping things simple, so you just need *index.html*, *index.js* and *styles.css* files. The

HTML and CSS are pretty straight forward:

You can find the whole *index.js* file below, but to make it more understandable we are first going to focus on extracts.

We need to define what happens on click on the buttons. When clicking on the first, we want to initialise the connection. When the second is clicked, we want to send a "Hello!" message through the connection. A click on the last button should close the connection. *connection* is a global variable, initialised in the *setupWebSocketConnection* function.

The most interesting part is of course how the connection is being set up.

We simply create an instance of the *WebSocket* class, passing the url as an argument. Then we define the event handlers. When the connection is open, we add a message to the console and set the disabled attributes of the buttons correctly. When a message is received, we display it in the console.

Once the connection is open, sending a message through it is as simple as:

```
connection.send('Hello!');
```

which you could see in the event listener function. Closing the connection is also simple, as you can see in the *closeConnection* function:

As promised, here the full *index.js* file:

Testing

Let's now start the server and try our messaging system between two clients. To run the server, go to the *web-socket* folder and run:

```
node index.js
```

You should see "Server listening at port 1337" in your terminal.

To test if the two clients can communicate through the server, open your *index.html* in two tabs or two browser windows.

Now click the buttons in whatever order you want. You can see that messages are being exchanged between the clients:

The communication over WebSocket is working.

. . .

We did setup a WebSocket server to allow two clients to exchange messages. Back to

the bigger picture, we can now use this channel to exchange the messages necessary to create and control the WebRTC connection. This is what we are going to do in the next article.

Thanks to p.nut.

JavaScript Web Development Programming Software Development