# Establish a WebRTC Connection: Video Call with WebRTC Step 3

Dornhoth
Mar 28 · 8 min read ★

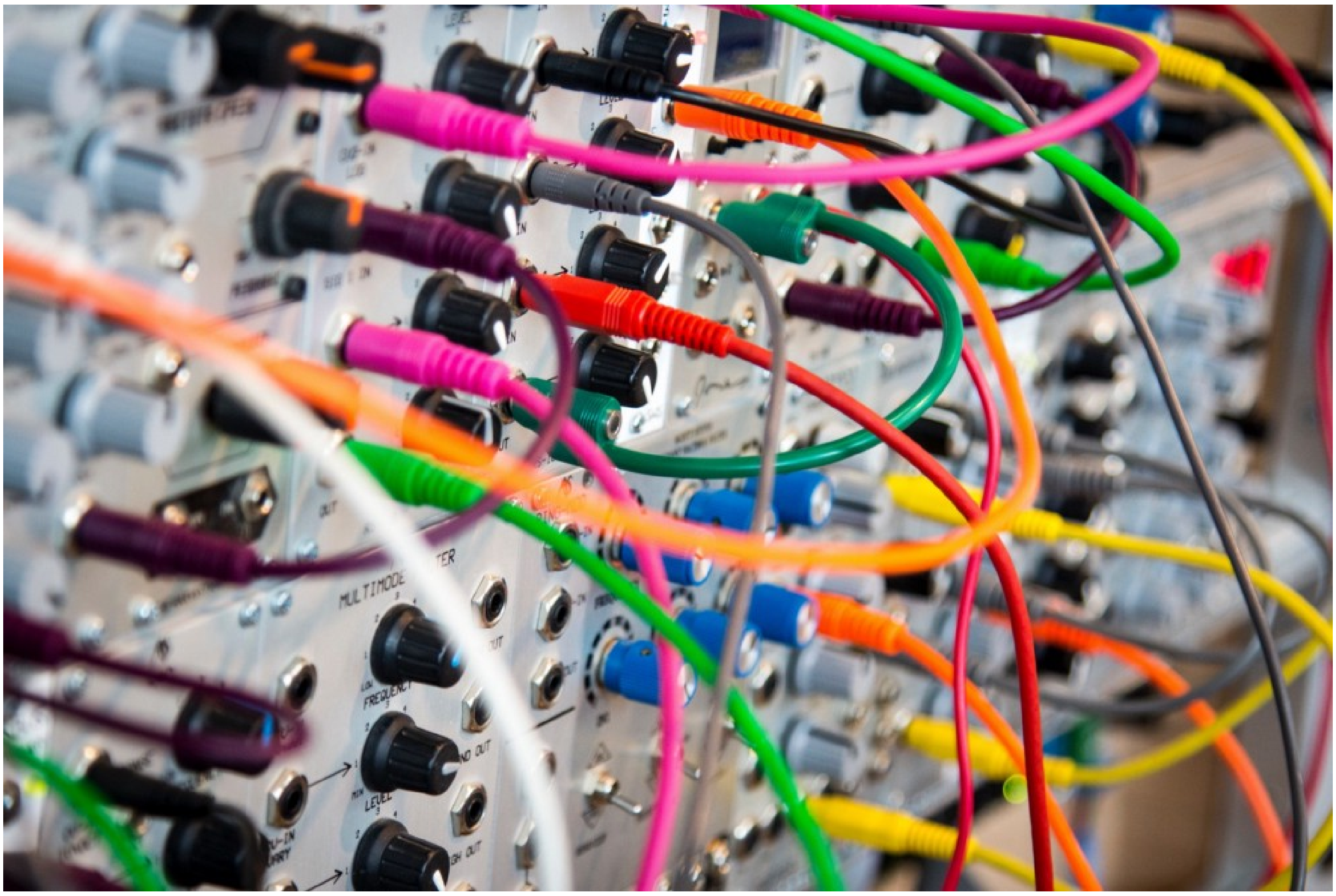


Photo by John Barkiple on Unsplash

WebRTC (Web Real Time Communication) is an open source project allowing you to create peer-to-peer connections between browsers. This connection can be used for different purposes, a main one being high quality and high performance video calls.

This article is the third of a series in which we are creating such a video chat using WebRTC. You can find the first two articles there:

- Step 1: Data Stream from your Webcam and Microphone
- Step 2: Set up a Connection over WebSocket

In the first article, we accessed the video and audio streams from the user's webcam and microphone in the browser. In the second, we enabled the communication of two clients through WebSocket. We will adapt this example for the signaling process.

In this article we will actually start the video chat.

## The Signaling Server

In order for a peer-to-peer connection to be established, peers first have to exchange about the media types they want to share, to tell each other when they want to start or stop the communication, and they have to find each other in the network. This is the signaling process.

The signaling isn't part of the WebRTC specifications. That means that you have to take care yourself of exchanging the messages needed to establish and control the connection. That also means that you are free to use whatever communication mechanism you want. You could theoretically use emails for this, but a reasonable solution is to use WebSocket. This is why we built a WebSocket server in the previous article, that we will now adapt slightly for the signaling.

The signaling mechanism doesn't need to know anything about the messages being exchanged. We simplify the WebSocket server that we created previously. For help to make this run on Node, please read this article.

```
const http = require('http');
const server = require('websocket').server;

const httpServer = http.createServer(() => { });
httpServer.listen(1337, () => {
  console.log('Server listening at port 1337');
});

const wsServer = new server({
  httpServer,
});
```

```
let clients = [];

wsServer.on('request', request => {
  const connection = request.accept();
  const id = (Math.random() * 10000);
  clients.push({ connection, id });

  connection.on('message', message => {
    console.log(message);
    clients
      .filter(client => client.id !== id)
      .forEach(client => client.connection.send(message.utf8Data));
  });

  connection.on('close', () => {
    clients = clients.filter(client => client.id !== id);
  });
});
```

index.js hosted with ❤ by GitHub view raw

We keep track of all connected clients. When a client sends a message, the message is broadcasted to everyone. This is not the end version of this but that will suffice to establish the WebRTC connection. In the next article, we will improve it to allow users to find the person they want to chat with and only communicate with her.

## Connection Offers and Answers

Three types of messages have to be exchanged over the signaling mechanism:

- Media data: which type of media do you want to share (audio only or video), with which constraints (quality for example).
- Session control data: to open and close the communication.
- Network data: users need to get each other's IP addresses and ports and check if they can establish a peer to peer connection.

Let's say our users are called Alice and Bob. Alice must first create and send a connection offer to Bob:

### Offer

1. If not already using some communication channel with Bob, Alice should join one (we use our WebSocket server running on port 1337).

```
const signaling = new WebSocket('ws://127.0.0.1:1337');
```

2. Alice creates a RTCPeerConnection object in her browser. It is a JavaScript interface, part of the WebRTC API, that represents the connection between the local browser and the remote peer.

```
const peerConnection = new RTCPeerConnection({
  iceServers: [{ urls: 'stun:stun.test.com:19000' }],
});
```

The parameter passed to the constructor contains the server urls needed by the ICE agent. More about this later or here.

3. Alice adds the tracks (audio and video) that she wants to share over the connection to her RTCPeerConnection object.

```
const stream = await navigator.mediaDevices.getUserMedia({
  audio: true,
  video: true,
});

stream.getTracks().forEach(track => peerConnection.addTrack(
  track,
  stream,
));
```

4. Alice creates a SDP offer. SDP stands for Session Description Protocol.

```
const offer = await peerConnection.createOffer();
```

It is the format used to describe the communication parameters. It contains the media description and network informations, and looks like this.

```
   v=0
   o=alice 123456789 123456789 IN IP4 some-host.com
   s=-
   c=IN IP4 some-host.com
```

```
    t=0 0
    m=audio 49170 RTP/AVP 0
    a=rtpmap:0 PCMU/8000
    m=audio 49170 RTP/AVP 31
    a=rtpmap:31 H261/90000
    m=audio 49170 RTP/AVP 32
    a=rtpmap:32 MPV/90000
```

5. Alice sets the local description of the connection to be this SDP by calling *setLocalDescription()*.

```
await peerConnection.setLocalDescription(offer);
```

6. Alice sends this offer to Bob over the signaling server.

```
signaling.send(JSON.stringify({
  message_type: MESSAGE_TYPE.SDP,
  content: offer,
}));
```

## Answer

Bob also has to be connected to the signaling server and has to have created a RTCPeerConnection object. After Alice sends him an offer, Bob has to do following:

1. Bob receives Alice's offer and sets it as the remote description in his RTCPeerConnection object calling *setRemoteDescription()*.

```
await peerConnection.setRemoteDescription(offerFromAlice);
```

2. Bob creates a SDP answer, containing the same kind of information as the SDP offer Alice sent.

```
const answer = await peerConnection.createAnswer();
```

3. Bob sets the local description of the connection to be this SDP by calling *setLocalDescription()*.

```
await peerConnection.setLocalDescription(answerFromBob);
```

4. Bob sends this answer to Alice over the signaling mechanism.

```
signaling.send(JSON.stringify({
  message_type: MESSAGE_TYPE.SDP,
  content: answerFromBob,
}));
```

We are now back at Alice. She receives Bob's answer and sets it as the remote description in her RTCPeerConnection object calling *setRemoteDescription()*.

```
await peerConnection.setRemoteDescription(answerFromBob);
```

Alice and Bob have now exchanged the media data, and notified each other that they want to start a video chat. They now have to share network information to establish a direct connection if possible. This is not as easy as it sounds, but luckily the ICE framework is doing it for us.

## ICE Candidates

Because of the historic lack of IP addresses (only around 4 billions addresses were available with IPv4), users are usually hidden behind NAT (Network Address Translations) gateways. The ICE (Interactive Connectivity Establishment) framework allows a peer to discover and communicate its public IP address. That works thanks to the STUN server which URL we gave as a parameter in the RTCPeerConnection object. It might be that a direct connection isn't possible due to the network configuration of the peers, in which case the connection will have to happen over a relay server, or TURN server. The server has to be given as a parameter to the RTCPeerConnection as well.

```
const peerConnection = new RTCPeerConnection({
  iceServers: [
    { urls: 'stun:stun.test.com:19000' },
    { urls: 'turn:turn:19001' },
  ],
});
```

The ICE agent takes care of this exploration and decision making for us, checks the possibility of a direct connection, and if it can't be done, establishes the connection over a TURN server (if it has been provided).

Alice and Bob only have to listen to the event *icecandidate* of the RTCPeerConnection. It is triggered every time a ICE candidate is found. They should then send their candidates to each other.

```
peerConnection.onicecandidate = (iceEvent) => {
  signaling.send(JSON.stringify({
    message_type: MESSAGE_TYPE.CANDIDATE,
    content: iceEvent.candidate,
  }));
};
```

When receiving the candidate of the other, Alice and Bob should pass it to the ICE agent of their RTCPeerConnection object.

```
await peerConnection.addIceCandidate(content);
```

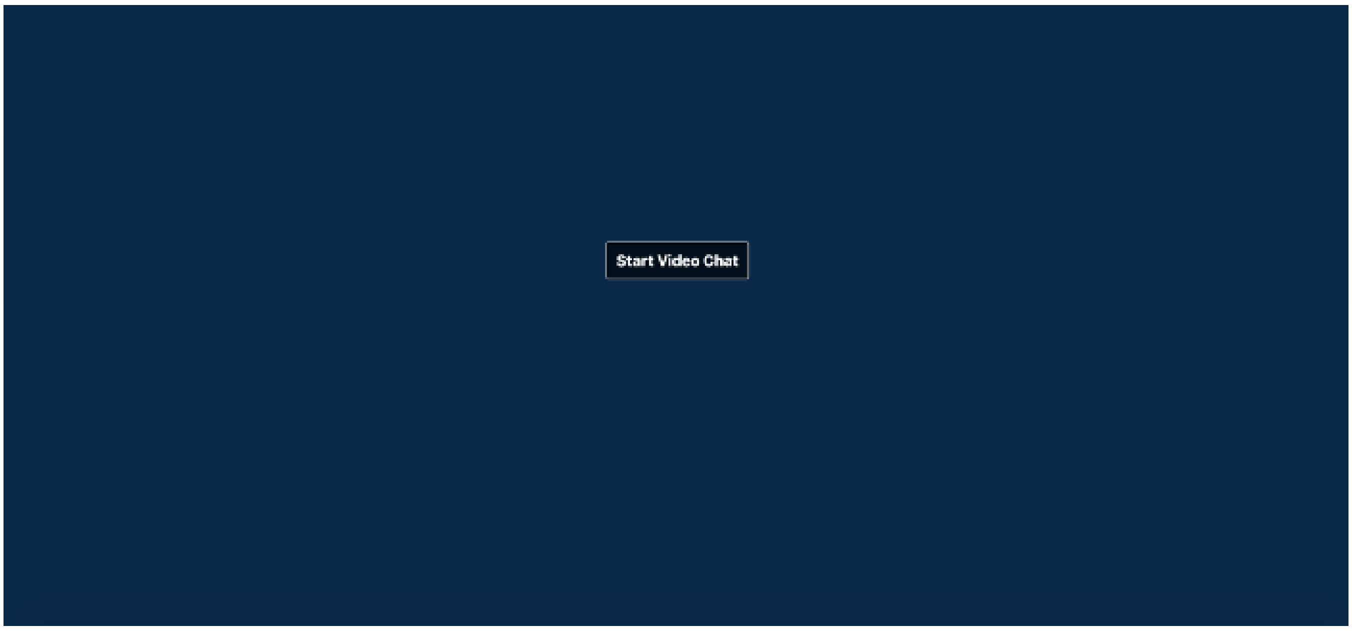
The ICE agent will take care of the negotiation and will finalize the connection. If you want more details about NATs and ICE you can have a look at this article.

When the connection is established, the tracks data start being exchanged over the connection. You can implement the *ontrack* event handler to display them:

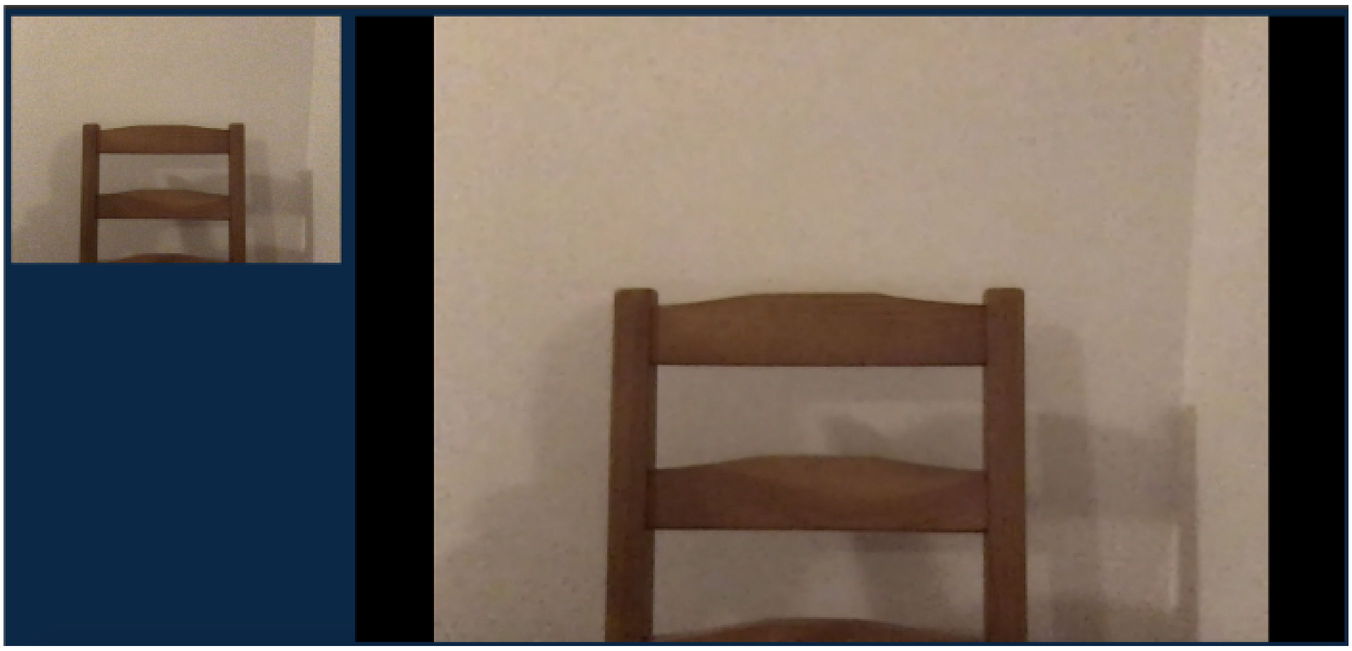
```
peerConnection.ontrack = (event) => {
  const video = document.getElementById('remote-view');
  if (!video.srcObject) {
    video.srcObject = event.streams[0];
  }
};
```

## Client Code

Our page first looks like this:



After clicking on the start button, you'll see yourself on the top left corner but won't see your contact until the connection gets established. Once it is, you'll be able to chat.



In a folder for your client application, create an *index.html* and a *styles.css* file and copy this code.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <title>VideoChat</title>
  <script src="index.js"></script>
  <link rel="stylesheet" href="styles.css">
```

```html
</head>

<body>
  <div id="chat-room" style="display: none;">
    <div id="videos">
      <video id="self-view" autoplay ></video>
      <video id="remote-view" autoplay></video>
    </div>
  </div>
  <button id="start">Start Video Chat</button>
</body>

</html>
```

```css

html, body {
    min-height: 100% !important;
    height: 100%;
    margin: 0;
}

body {
  background-color:#01284a;
  padding: 8px;
}

button {
  display: block;
  margin: auto;
  position: relative;
  top: 40%;
  padding: 10px;
  font-size: 16px;
  background-color: #000f1c;
  border-radius: 3px;
  color: white;
  font-weight: bold;
  cursor: pointer;
}

video {
  width: 100%;
}

#videos {
  display: grid;
  grid-gap: 16px;
  width: 100%;
  grid-template-columns: 1fr 3fr;
}

#self-view {
  grid-column: 1;
  background-color: black;
}

#remote-view {
  grid-column: 2;
  background-color: black;
  height: 95vh;
}
```

styles.css hosted with ❤ by GitHub view raw

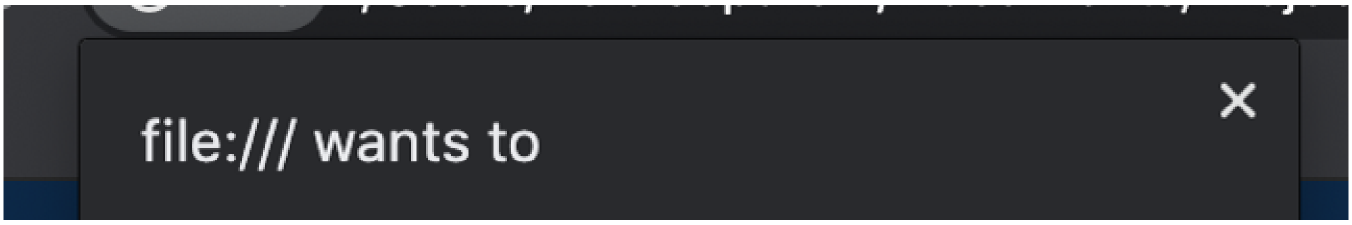
You will also need an *index.js* for the JavaScript. This is the *index.js* file at the end:

```
(function () {
  "use strict";

  const MESSAGE_TYPE = {
    SDP: 'SDP',
    CANDIDATE: 'CANDIDATE',
  }

  document.addEventListener('click', async (event) => {
    if (event.target.id === 'start') {
      startChat();
    }
  });

  const startChat = async () => {
    try {
      const stream = await navigator.mediaDevices.getUserMedia({ audio: true, video: t
      showChatRoom();

      const signaling = new WebSocket('ws://127.0.0.1:1337');
      const peerConnection = createPeerConnection(signaling);

      addMessageHandler(signaling, peerConnection);

      stream.getTracks().forEach(track => peerConnection.addTrack(track, stream));
      document.getElementById('self-view').srcObject = stream;

    } catch (err) {
      console.error(err);
    console.error(err);
  }
};

const createPeerConnection = (signaling) => {
  const peerConnection = new RTCPeerConnection({
    iceServers: [{ urls: 'stun:stun.l.test.com:19000' }],
  });

  peerConnection.onnegotiationneeded = async () => {
   await createAndSendOffer();
  };

  peerConnection.onicecandidate = (iceEvent) => {
    if (iceEvent && iceEvent.candidate) {
      signaling.send(JSON.stringify({
        message_type: MESSAGE_TYPE.CANDIDATE,
        content: iceEvent.candidate,
      }));
    }
  };

  peerConnection.ontrack = (event) => {
    const video = document.getElementById('remote-view');
    if (!video.srcObject) {
      video.srcObject = event.streams[0];
    }
  };

  return peerConnection;
};

const addMessageHandler = (signaling, peerConnection) => {
  signaling.onmessage = async (message) => {
    const data = JSON.parse(message.data);

    if (!data) {
      return;
    }

    const { message_type, content } = data;
    try {
      if (message_type === MESSAGE_TYPE.CANDIDATE && content) {
        await peerConnection.addIceCandidate(content);
      } else if (message_type === MESSAGE_TYPE.SDP) {
        if (content.type === 'offer') {
          await peerConnection.setRemoteDescription(content);
          const answer = await peerConnection.createAnswer();
77              await peerConnection.setLocalDescription(answer);
78              signaling.send(JSON.stringify({
79                message_type: MESSAGE_TYPE.SDP,
80                content: answer,
81              }));
82            } else if (content.type === 'answer') {
83              await peerConnection.setRemoteDescription(content);
84            } else {
85              console.log('Unsupported SDP type.');
86            }
87          }
88        } catch (err) {
89          console.error(err);
90        }
91      };
92    };
93
94    const createAndSendOffer = async (signaling, peerConnection) => {
95      const offer = await peerConnection.createOffer();
96      await peerConnection.setLocalDescription(offer);
97
98      signaling.send(JSON.stringify({ message_type: MESSAGE_TYPE.SDP, content: offer }))
99    };
100
101   const showChatRoom = () => {
102     document.getElementById('start').style.display = 'none';
103     document.getElementById('chat-room').style.display = 'block';
104   };
105 })();
```

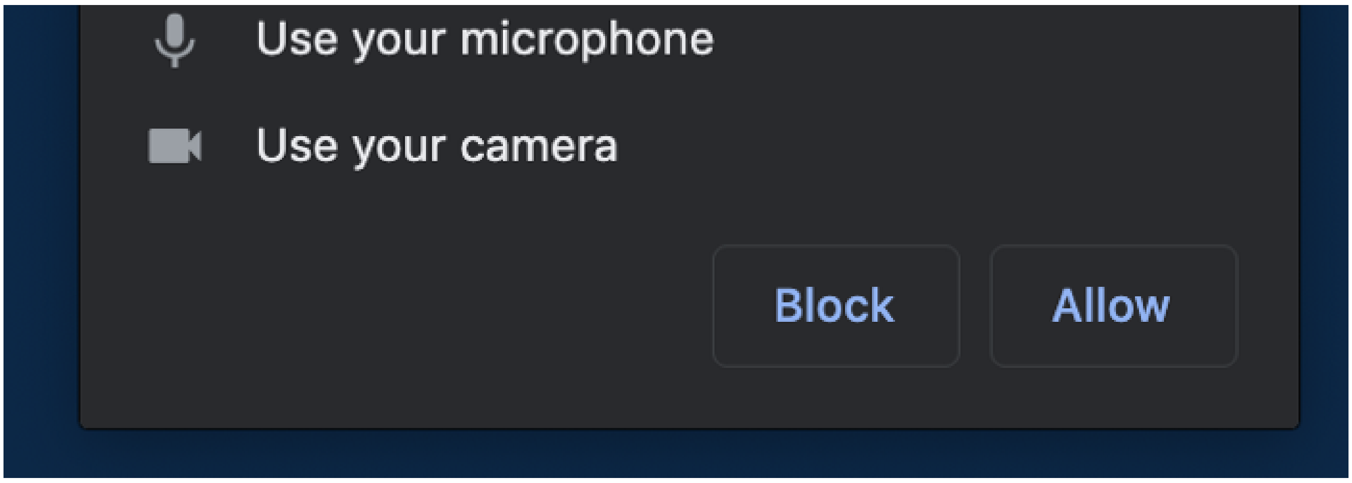
First we define the message types that the user can expect to receive. The first one, ‘SDP’, is for offers and answers. The second one, ‘CANDIDATE’, is for ICE candidates.

On click on the start page button, we want to start the chat, this is what we are doing line 9 to 13.

Let’s now look at the *startChat* function. We first request the data from the camera and microphone. This should trigger an access request from your browser that you have to accept before it can go further:



file:/// wants to



Once you have accepted, we show the chat room, displaying the video elements and hiding the start button. We establish the connection to the WebSocket server (line 20) and call this connection *signaling*. We create the RTCPeerConnection object in the *createPeerConnection* function. It gives a STUN server as a parameter (it is a fake one, you can replace it by a public STUN server) and defines the two event handlers we talked about : *onicecandidate* that will send the ICE candidates to the peers and *ontrack* that will set the received tracks to our video HTML element. It has an additional event handler, and it is actually a pretty important one: *onnegationneeded*. This event is fired when we add tracks to the connection, and later when something happens requiring a renegotiation. It is here that the signaling exchange will actually get started.

Back to the *startChat* function, after having created the RTCPeerConnection object, we define what to do when receiving a message in the *addMessageHandler* function. If we receive a candidate, we give it to the ICE agent as we described earlier. If we receive an offer, we set the remote description, create an answer, save the answer as local description and send it to the peer. When we receive an answer, we just set it as the local offer.

We then set the our local tracks to the RTCPeerConnection object and display them in the video element meant for this. Setting the tracks on the peer connection object will trigger the *negogationneeded* event and the event listener will call the *createAndSendOffer* function.

Start the WebSocket server and open the client in two different tabs. After clicking on “Start” on both pages, you should be able to communicate with yourself.

. . .

We have now established a connection thanks to WebRTC. Our solution is still not optimal. We handled the signaling process, but for now there is no way to allow two given users, and only them, to communicate. The connection is established by the two first users clicking on the button. In the next article, we are going to solve this problem by adapting our signaling server.

Thanks to p.nut.

JavaScript WebRTC Web Development Programming Tech