**RV Meetup - Decentralized Video Chat**

Decentralized video chat platform powered by WebRTC using Twilio STUN/TURN infrastructure. RV Meetup provides video quality and latency simply not available with traditional technology.

**Features**

* Screen sharing
* Picture in picture
* Live captions
* Text chat
* Auto-scaling video quality
* No download required, entirely browser based
* Direct peer to peer connection ensures lowest latency
* Single use disposable chat rooms

**Quick start**

* You will need to have Node.js installed, this project has been tested with Node version 10.X and 12.X

**Set up credentials**

* Rename .env.template to .env
* Sign up for free twilio account <https://www.twilio.com/login>
* Get your Account SID and Auth Token from the Twillio console
* Fill in your credentials in the .env file

**Install dependencies**

npm install

**Start the server**

npm start

* Open localhost:3000 in browser
* If you want to use a client on another computer/network, make sure you publish your server on an HTTPS connection. You can use a service like [ngrok](https://ngrok.com/) for that.

Implementation Details

WebRTC (Web Real Time Connection) is an open source project with a JavaScript API allowing you to create peer-to-peer connections between browsers. Even though it can be used for different types of data transfer, its main application is video calls. In the previous articles, we accessed the microphone and camera data streams from the browser, established a connection over WebSocket between users that we then used for the signaling process, and we allowed users to establish the connection with a given contact using a shared code

A first step to create a video chat is to have access to the user’s devices, like webcam and microphone, and their stream of data. In this first article, we are going to access our devices and display the video (with audio) in our browser.

**Access User Audio and Video Stream**

We first need to have access to the user’s webcam and microphone. The navigator has access to the media devices through the [MediaDevices](https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices) interface.

window.navigator.mediaDevices

We can get a list of all the devices the browser has access to by calling in the console:

const devices = await window.navigator

.mediaDevices

.enumerateDevices();

In my browser, I get an array of five devices, one video input and five audio input. But we are not interested in the devices themselves, we want to get the data stream from them.

Media content, like audio and video, are represented by the [MediaStream](https://developer.mozilla.org/en-US/docs/Web/API/MediaStream) interface. To get access to the data stream, you need to call the *getUserMedia* function on the *mediaDevices* object. We need to give the types of media we want as a parameter, here audio and video. We can give a bunch of constraints as parameter, not only the media type, but also the size of the video we want, the orientation, … We keep it simple here:

const stream = await window.navigator.mediaDevices.getUserMedia(

{

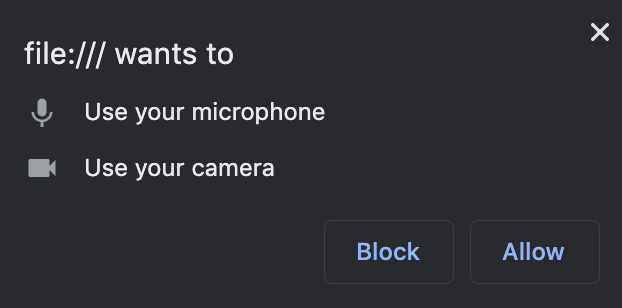
video: true,

audio: true,

},

);

This returns a Promise, as you have to allow access to your devices. On Google Chrome, you should see such a dialog:



If you say yes, you’ll receive the MediaStream object. If your webcam has this feature, you should see a little light next to it, telling you that it is now on.

Our MediaStream object consists of two tracks, one track corresponding to one media type. You can access them by calling

stream.getAudioTracks();

stream.getVideoTracks();

These methods return arrays of tracks, each track containing some info like the kind (audio or video for example), the label, the idea, but also if the track is enabled, muted, etc.

**HTML5 Video Tag**

We now have access to the data stream from the microphone and webcam, we just want to display them in our browser to make sure it works. This used to be a pain in the neck, but the HTML5 video tag allows us to build a video in a page pretty easily.

|  |  |
| --- | --- |
|  | <!DOCTYPE html> |
|  | <html> |
|  |  |
|  | <head> |
|  | <meta charset="UTF-8"> |
|  | <title>VideoChat</title> |
|  | <script src="index.js"></script> |
|  | <link rel="stylesheet" href="styles.css"> |
|  | </head> |
|  |  |
|  | <body> |
|  | <div> |
|  | <video id="video" controls autoplay></video> |
|  | </div> |
|  | <div> |
|  | <button id="button">Start Video</button> |
|  | </div> |
|  | </body> |
|  |  |
|  | </html> |

|  |  |
| --- | --- |
|  | body { |
|  | text-align: center; |
|  | } |
|  |  |
|  | video { |
|  | width: 500px; |
|  | height: 500px; |
|  | margin: 20px; |
|  | } |
|  |  |
|  | button { |
|  | padding: 5px; |
|  | background-color: seagreen; |
|  | color: white; |
|  | border-radius: 3px; |
|  | font-weight: bold; |
|  | } |

We have a video tag with controls (the play, mute, and stop buttons) and a button to start the video.

Let’s look at the JavaScript code:

|  |  |
| --- | --- |
| (function () { |  |
|  | "use strict"; |
|  |  |
|  | document.addEventListener('click', async event => { |
|  | if (event.target.id === 'button') { |
|  | const stream = await window.navigator.mediaDevices.getUserMedia({ video: true, audio: true }); |
|  | const video = document.getElementById('video'); |
|  | video.srcObject = stream; |
|  | video.play(); |
|  | } |
|  | }); |
|  |  |
|  | })(); |

When you click on the button, we create the stream as we did in the previous section. Your browser should ask for your permission to start the video (we don’t handle the case where the user refuses here). Once the stream is available, it is set as the source object for the video tag and the video is started. You should now see and hear yourself.

Second Step

The WebRTC connection between your local browser and a remote user (peers) will be represented by the JavaScript interface [RTCPeerConnection](https://developer.mozilla.org/en-US/docs/Web/API/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](https://github.com/Worlize/WebSocket-Node).

If you don’t already have it on your machine, you should install [Node.js](https://nodejs.org/en/). 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](https://github.com/Worlize/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.

|  |  |
| --- | --- |
|  | 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, |
|  | }); |

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.

|  |  |
| --- | --- |
|  | let clients = []; |
|  |  |
|  | wsServer.on('request', request => { |
|  | const connection = request.accept(); |
|  | const id = Math.floor(Math.random() \* 100); |
|  |  |
|  | clients.forEach(client => client.connection.send(JSON.stringify({ |
|  | client: id, |
|  | text: 'I am now connected', |
|  | }))); |
|  |  |
|  | clients.push({ connection, id }); |
|  |  |
|  | connection.on('message', message => { |
|  | clients |
|  | .filter(client => client.id !== id) |
|  | .forEach(client => client.connection.send(JSON.stringify({ |
|  | client: id, |
|  | text: message.utf8Data, |
|  | }))); |
|  | }); |
|  |  |
|  | connection.on('close', () => { |
|  | clients = clients.filter(client => client.id !== id); |
|  | clients.forEach(client => client.connection.send(JSON.stringify({ |
|  | client: id, |
|  | text: 'I disconnected', |
|  | }))); |
|  | }); |
|  | }); |

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

Third Step

**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

|  |  |
| --- | --- |
|  | 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); |
|  | }); |
|  | }); |

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](https://developer.mozilla.org/en-US/docs/Web/API/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](https://levelup.gitconnected.com/webrtc-the-ice-framework-stun-and-turn-servers-10b2972483bb).

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](https://en.wikipedia.org/wiki/Session_Description_Protocol) 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](https://levelup.gitconnected.com/webrtc-the-ice-framework-stun-and-turn-servers-10b2972483bb).

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];

}

};

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:



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.