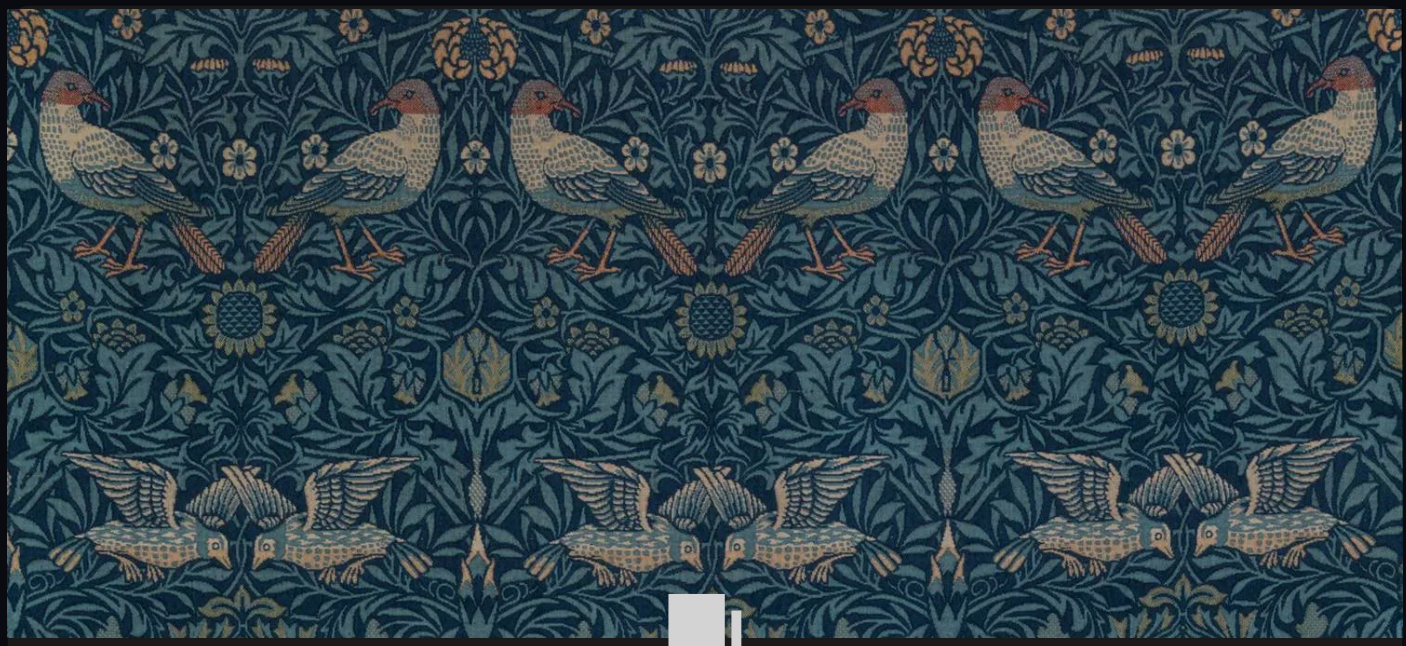

Week 23.2

Implementing WebRTC

Up until now, our discussions have primarily revolved around theoretical concepts. In this lecture, Harkirat takes a `practical approach` by guiding us through the hands-on process of `implementing the frontend and backend of webrtc`

We'll be applying the knowledge we've gained so far, specifically focusing on implementing the frontend using `React` and the backend using `NodeJS` — thus exploring WebRTC in detail.

> While there are `no specific notes` provided for this section, a mini guide is outlined below to assist you in navigating through the process of building the application. Therefore, it is strongly `advised to actively follow along` during the lecture for a hands-on learning experience.

Backend

To implement the signaling server for our WebRTC application, we'll be using Node.js with the `ws` library for creating a WebSocket server. Here's a step-by-step guide to setting up the backend:

1. **Create a new TypeScript project:**
    - Run `npm init -y` to initialize a new Node.js project.
    - Run `npx tsc --init` to create a `tsconfig.json` file for TypeScript configuration.

- In the `tsconfig.json` file, update the `rootDir` and `outDir` properties to specify the source and output directories, respectively:

```
"rootDir": "./src",
"outDir": "./dist",
```

3. Create a simple WebSocket server:

- Create a new file `src/index.ts` and add the following code:

```
import { WebSocketServer } from 'ws';

const wss = new WebSocketServer({ port: 8080 });

let senderSocket: null | WebSocket = null;
let receiverSocket: null | WebSocket = null;

wss.on('connection', function connection(ws) {
  ws.on('error', console.error);

  ws.on('message', function message(data: any) {
    const message = JSON.parse(data);
    // Handle incoming messages here
  });

  ws.send('something');
})
```

- This code sets up a WebSocket server listening on port 8080 and initializes variables to store the sender and receiver sockets.

4. Run the server:

- Compile the TypeScript code by running `tsc -b`.
- Start the server by running `node dist/index.js`.

5. Add message handlers:

- Update the `message` event handler in `src/index.ts` to handle different types of messages:

```
ws.on('message', function message(data: any) {
  const message = JSON.parse(data);
  if (message.type === 'sender') {
    senderSocket = ws;
  } else if (message.type === 'receiver') {
    receiverSocket = ws;
  } else if (message.type === 'createOffer') {
    if (ws !== senderSocket) {
      return;
    }
    receiverSocket?.send(JSON.stringify({ type: 'createOffer', sdp: message.sdp }));
  } else if (message.type === 'createAnswer') {
    if (ws !== receiverSocket) {
      return;
    }
    senderSocket?.send(JSON.stringify({ type: 'createAnswer', sdp: message.sdp }));
  } else if (message.type === 'iceCandidate') {
    if (ws === senderSocket) {
      receiverSocket?.send(JSON.stringify({ type: 'iceCandidate', candidate: message.candidate }));
    } else if (ws === receiverSocket) {
      senderSocket?.send(JSON.stringify({ type: 'iceCandidate', candidate: message.candidate }));
```

- This code handles different message types (`sender` , `receiver` , `createOffer` , `createAnswer` , `iceCandidate`) and forwards the messages to the appropriate sockets.

This implementation provides a basic signaling server that can facilitate one-way communication between two tabs. To support two-way communication and multiple rooms, you can refer to the provided example: https://github.com/hkirat/omegle/

The provided code sets up a WebSocket server using the `ws` library in TypeScript. It handles different types of messages (`sender` , `receiver` , `createOffer` , `createAnswer` , `iceCandidate`) and forwards them to the appropriate sockets (sender or receiver).

Here's a breakdown of the code:

1. Import the required modules:

```
import { WebSocket, WebSocketServer } from 'ws';
```

2. Create a new WebSocket server:

```
const wss = new WebSocketServer({ port: 8080 });
```

This creates a new WebSocket server listening on port 8080.

3. Initialize variables to store sender and receiver sockets:

```
let senderSocket: null | WebSocket = null;
let receiverSocket: null | WebSocket = null;
```

4. Handle new WebSocket connections:

```
wss.on('connection', function connection(ws) {
  ws.on('error', console.error);

  ws.on('message', function message(data: any) {
    const message = JSON.parse(data);
    // Handle incoming messages here
  });

  ws.send('something');
});
```

- When a new WebSocket connection is established, the server listens for the `message` event and handles incoming messages.
- The `ws.send('something');` line is just an example of sending a message back to the client.

5. Handle incoming messages:

```
ws.on('message', function message(data: any) {
  const message = JSON.parse(data);
  if (message.type === 'sender') {
    senderSocket = ws;
  } else if (message.type === 'receiver') {
    receiverSocket = ws;
  } else if (message.type === 'createOffer') {
    if (ws !== senderSocket) {
      return;
    }
    receiverSocket?.send(JSON.stringify({ type: 'createOffer', sdp: message.sdp }));
```

```
    }
    senderSocket?.send(JSON.stringify({ type: 'createAnswer', sdp: message.sdp }));
  } else if (message.type === 'iceCandidate') {
    if (ws === senderSocket) {
      receiverSocket?.send(JSON.stringify({ type: 'iceCandidate', candidate: message.candidate }));
    } else if (ws === receiverSocket) {
      senderSocket?.send(JSON.stringify({ type: 'iceCandidate', candidate: message.candidate }));
    }
  }
});
```

- The server handles different types of messages:
  - `sender` : Stores the sender socket.
  - `receiver` : Stores the receiver socket.
  - `createOffer` : Forwards the offer SDP from the sender to the receiver.
  - `createAnswer` : Forwards the answer SDP from the receiver to the sender.
  - `iceCandidate` : Forwards the ICE candidate from the sender to the receiver, or vice versa.

> This implementation provides a basic signaling server that can facilitate one-way communication between two tabs. To support two-way communication and multiple rooms, you can refer to the provided example: https://github.com/hkirat/omegle/

Frontend

To create the frontend for our WebRTC application, we'll be using React with Vite as the build tool. Here's a step-by-step guide to setting up the frontend:

1. **Create a new React project with Vite:**
   - Run `npm create vite@latest` and follow the prompts to create a new React project.
2. **Add routing:**
   - Install the `react-router-dom` package: `npm install react-router-dom`.
   - In `src/App.tsx`, import the necessary components and set up the routes:

```
import { useState } from 'react'
import './App.css'
import { Route, BrowserRouter, Routes } from 'react-router-dom'
import { Sender } from './components/Sender'
import { Receiver } from './components/Receiver'

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/sender" element={<Sender />} />
        <Route path="/receiver" element={<Receiver />} />
      </Routes>
    </BrowserRouter>
  )
}

export default App
```

WebRTC connections locally.

4. **Create the Sender component**:

- Create a new file `src/components/Sender.tsx` and add the following code:

```
import { useEffect, useState } from "react"

export const Sender = () => {
  const [socket, setSocket] = useState<WebSocket | null>(null);
  const [pc, setPC] = useState<RTCPeerConnection | null>(null);

  useEffect(() => {
    const socket = new WebSocket('ws://localhost:8080');
    setSocket(socket);
    socket.onopen = () => {
      socket.send(JSON.stringify({
        type: 'sender'
      }));
    }
  }, []);

  const initiateConn = async () => {
    // ... (implementation omitted for brevity)
  }

  const getCameraStreamAndSend = (pc: RTCPeerConnection) => {
    // ... (implementation omitted for brevity)
  }

  return (
    <div>
      Sender
      <button onClick={initiateConn}> Send data </button>
    </div>
  )
}
```

- This component sets up a WebSocket connection with the signaling server, creates an RTCPeerConnection instance, and handles the necessary events for establishing the WebRTC connection and sending media data.

5. **Create the Receiver component**:

- Create a new file `src/components/Receiver.tsx` and add the following code:

```
import { useEffect } from "react"

export const Receiver = () => {
  useEffect(() => {
    const socket = new WebSocket('ws://localhost:8080');
    socket.onopen = () => {
      socket.send(JSON.stringify({
        type: 'receiver'
      }));
    }
    startReceiving(socket);
  }, []);

  function startReceiving(socket: WebSocket) {
    // ... (implementation omitted for brevity)
  }
```

- This component sets up a WebSocket connection with the signaling server, creates an RTCPeerConnection instance, and handles the necessary events for receiving media data from the sender.

The provided code sets up a basic WebRTC application with a sender and receiver component. The sender component initiates the WebRTC connection, requests camera access, and sends the media data to the receiver. The receiver component listens for incoming media data and renders it in a video element.

To extend this implementation, you can consider the following enhancements:

1. **Multiple Producers**:
    - Modify the signaling server to support multiple senders (producers) and multiple receivers.
    - Update the frontend components to handle multiple connections and media streams.
2. **Room Logic**:
    - Implement room functionality on the signaling server to allow users to join specific rooms.
    - Update the frontend components to allow users to create or join rooms.
3. **Two-Way Communication**:
    - Modify the signaling server to handle bi-directional communication between peers.
    - Update the frontend components to enable both sending and receiving media data.
4. **SFU (Selective Forwarding Unit) Integration**:
    - Instead of using a pure peer-to-peer approach, integrate an SFU server like Mediasoup.
    - Update the signaling server to communicate with the SFU server.
    - Modify the frontend components to connect to the SFU server and handle media data accordingly.

> By implementing these enhancements, you can create a more robust and feature-rich WebRTC application that supports multiple producers, room functionality, two-way communication, and scalable media handling with an SFU.

0 Comments

Most upvotes ⌄ All comments ⌄

Add a comment...

Comment