**Bài Chạy ứng dụng voice call với WebRTC và Socket.IO**

**Bước 1: Thiết lập cấu trúc dự án**

Tạo một cấu trúc dự án như sau:

webrtc-voice-call/

├── index.html

├── server.js

├── package.json

**Bước 2: Cài đặt các phụ thuộc**

Trong thư mục dự án, chạy lệnh sau để khởi tạo một dự án Node.js và cài đặt **Socket.IO**:

**npm init -y**

**npm install express socket.io**

**Bước 3: Cấu hình máy chủ server.js**

Tạo tệp server.js với nội dung như đã được cung cấp ở phần trước:

const express = require('express');

const http = require('http');

const socketIo = require('socket.io');

const app = express();

const server = http.createServer(app);

const io = socketIo(server);

app.use(express.static(\_\_dirname));

io.on('connection', (socket) => {

console.log('A user connected');

socket.on('offer', (offer) => {

socket.broadcast.emit('offer', offer);

});

socket.on('answer', (answer) => {

socket.broadcast.emit('answer', answer);

});

socket.on('candidate', (candidate) => {

socket.broadcast.emit('candidate', candidate);

});

socket.on('disconnect', () => {

console.log('A user disconnected');

});

});

server.listen(3000, () => {

console.log('Server is running on port 3000');

});

**Bước 4: Tạo tệp HTML index.html**

Tạo tệp index.html trong cùng thư mục với nội dung WebRTC:

<!DOCTYPE html>

<html>

<head>

<title>WebRTC Voice Call</title>

</head>

<body>

<h2>Voice Call Demo</h2>

<button id="startCall">Start Call</button>

<audio id="remoteAudio" autoplay></audio>

<script src="/socket.io/socket.io.js"></script>

<script>

const socket = io.connect();

let localStream;

let peerConnection;

const servers = { iceServers: [{ urls: "stun:stun.l.google.com:19302" }] };

const remoteAudio = document.getElementById('remoteAudio');

const startCallButton = document.getElementById('startCall');

startCallButton.onclick = async () => {

localStream = await navigator.mediaDevices.getUserMedia({ audio: true });

peerConnection = new RTCPeerConnection(servers);

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

peerConnection.onicecandidate = ({ candidate }) => {

if (candidate) socket.emit('candidate', candidate);

};

peerConnection.ontrack = ({ streams: [stream] }) => {

remoteAudio.srcObject = stream;

};

const offer = await peerConnection.createOffer();

await peerConnection.setLocalDescription(offer);

socket.emit('offer', offer);

};

socket.on('offer', async (offer) => {

peerConnection = new RTCPeerConnection(servers);

peerConnection.onicecandidate = ({ candidate }) => {

if (candidate) socket.emit('candidate', candidate);

};

peerConnection.ontrack = ({ streams: [stream] }) => {

remoteAudio.srcObject = stream;

};

await peerConnection.setRemoteDescription(offer);

const answer = await peerConnection.createAnswer();

await peerConnection.setLocalDescription(answer);

socket.emit('answer', answer);

});

socket.on('answer', (answer) => {

peerConnection.setRemoteDescription(answer);

});

socket.on('candidate', (candidate) => {

peerConnection.addIceCandidate(new RTCIceCandidate(candidate));

});

</script>

</body>

</html>

**Bước 5: Chạy máy chủ Node.js**

* Mở terminal và điều hướng vào thư mục dự án.
* Chạy lệnh:

node server.js

Máy chủ Node.js của bạn sẽ chạy trên cổng 3000.

**Bước 6: Mở trình duyệt và kiểm tra ứng dụng**

* Mở trình duyệt và truy cập http://localhost:3000/.
* Nhấn nút "Start Call" để bắt đầu cuộc gọi.
* Khi có hai trình duyệt kết nối, âm thanh từ microphone của một thiết bị sẽ truyền tới thiết bị khác qua WebRTC.

**Kết luận**

* Đối với **Twilio**, bạn có thể chạy ứng dụng đơn giản chỉ với Node.js và thực hiện cuộc gọi tới bất kỳ số điện thoại nào.
* Đối với **WebRTC**, bạn sẽ cần chạy máy chủ Node.js kết hợp với Socket.IO để thiết lập cuộc gọi P2P giữa các trình duyệt.

Đúng, để sử dụng WebRTC cho voice call, cả hai thiết bị (client) cần có **microphone** để thu âm và gửi âm thanh tới thiết bị khác. Đây là quy trình cơ bản của WebRTC khi xử lý cuộc gọi âm thanh hoặc video:

* **Microphone** sẽ thu âm thanh từ người dùng và truyền dữ liệu âm thanh đó qua kết nối **peer-to-peer** (P2P) đến thiết bị đối tác.
* Trên thiết bị nhận, âm thanh sẽ được phát qua **loa** hoặc **tai nghe**.

**Một số lưu ý:**

* **Trình duyệt cần quyền truy cập microphone**: Khi bắt đầu cuộc gọi, trình duyệt sẽ yêu cầu người dùng cấp quyền sử dụng microphone (và camera nếu đó là cuộc gọi video).
* Nếu không có microphone trên thiết bị hoặc người dùng không cấp quyền, ứng dụng sẽ không thể thu âm thanh để gửi đến thiết bị khác.

**Cách kiểm tra microphone trong WebRTC**

Trong ví dụ WebRTC trước, có dòng mã sau dùng để yêu cầu quyền truy cập microphone:

localStream = await navigator.mediaDevices.getUserMedia({ audio: true });

Nếu người dùng không có microphone hoặc từ chối quyền truy cập, hàm này sẽ trả về lỗi, và bạn có thể bắt lỗi để hiển thị thông báo hoặc xử lý tiếp:

navigator.mediaDevices.getUserMedia({ audio: true })

.then(stream => {

// Microphone được bật và có âm thanh thu từ stream

})

.catch(err => {

console.error('Không thể truy cập microphone:', err);

alert('Microphone cần thiết để thực hiện cuộc gọi.');

});

Nếu bạn chỉ muốn sử dụng tính năng **nghe** (nhận âm thanh), bạn vẫn có thể chạy ứng dụng trên thiết bị chỉ có loa, nhưng cuộc gọi sẽ bị giới hạn ở việc phát âm thanh mà không thể gửi dữ liệu âm thanh từ phía không có microphone.

### II. Tại Sao Cần Chứng Chỉ SSL cho Voice Call?

1. **Yêu Cầu Bảo Mật**: WebRTC yêu cầu một kết nối bảo mật (HTTPS) để truyền tải âm thanh và video. Nếu không có HTTPS, trình duyệt sẽ từ chối kết nối WebRTC. Điều này áp dụng cho cả voice call và video call.
2. **Ngăn Chặn Nghe Lén**: Việc sử dụng HTTPS giúp mã hóa dữ liệu truyền tải, từ đó bảo vệ cuộc gọi khỏi các cuộc tấn công nghe lén.
3. **Cảnh Báo từ Trình Duyệt**: Nếu bạn không sử dụng HTTPS, trình duyệt sẽ hiển thị cảnh báo bảo mật cho người dùng, điều này có thể làm giảm độ tin cậy của ứng dụng của bạn.

const express = require('express');

const https = require('https'); // Thay đổi từ http sang https

const fs = require('fs'); // Thư viện để đọc chứng chỉ SSL

const socketIo = require('socket.io');

const app = express();

// Đọc chứng chỉ SSL từ tệp

const options = {

key: fs.readFileSync('F:/BT\_NODEJS/VOICE-VIDEO/192.168.1.96-key.pem'), // Update this path

cert: fs.readFileSync('F:/BT\_NODEJS/VOICE-VIDEO/192.168.1.96.pem') // Update this path

};

// Tạo máy chủ HTTPS

const server = https.createServer(options, app);

const io = socketIo(server);

app.use(express.static(\_\_dirname));

io.on('connection', (socket) => {

console.log('A user connected');

socket.on('offer', (offer) => {

socket.broadcast.emit('offer', offer);

});

socket.on('answer', (answer) => {

socket.broadcast.emit('answer', answer);

});

socket.on('candidate', (candidate) => {

socket.broadcast.emit('candidate', candidate);

});

socket.on('disconnect', () => {

console.log('A user disconnected');

});

});

// Khởi động máy chủ trên cổng 443 (hoặc cổng mà bạn muốn)

const PORT = process.env.PORT || 443;

server.listen(PORT, () => {

console.log(`Server is running on https://localhost:${PORT}`);

});

**Chạy node server.js**

 **Express**: Thư viện chính để xây dựng ứng dụng web.

 **HTTPS**: Thư viện để tạo máy chủ HTTPS.

 **fs**: Thư viện để làm việc với hệ thống tệp, bao gồm việc đọc các tệp chứng chỉ.

 **Socket.IO**: Thư viện cho giao tiếp thời gian thực giữa client và server.

 **options**: Một đối tượng chứa thông tin về chứng chỉ SSL.

 **fs.readFileSync**: Đọc tệp một cách đồng bộ. Bạn sẽ cung cấp đường dẫn đến tệp chứa khóa riêng (.pem) và chứng chỉ (.pem).

const server = https.createServer(options, app); //Tạo một máy chủ HTTPS sử dụng các tùy chọn đã định nghĩa (chứng chỉ SSL) và ứng dụng Express.

const io = socketIo(server); //Kết nối Socket.IO với máy chủ HTTPS. Điều này cho phép bạn sử dụng các tính năng của Socket.IO cho giao tiếp thời gian thực.

app.use(express.static(\_\_dirname)); //Sử dụng middleware của Express để phục vụ các tệp tĩnh (như HTML, CSS, JS) từ thư mục hiện tại (\_\_dirname). Điều này có nghĩa là bất kỳ tệp nào trong thư mục đó đều có thể được truy cập qua URL.

 **io.on('connection', (socket) => {...})**: Lắng nghe sự kiện khi một client kết nối.

 **socket.on('offer', (offer) => {...})**: Lắng nghe sự kiện 'offer' từ client và phát (broadcast) nó đến tất cả các client khác.

 **socket.on('answer', (answer) => {...})**: Tương tự như trên, lắng nghe sự kiện 'answer'.

 **socket.on('candidate', (candidate) => {...})**: Lắng nghe sự kiện 'candidate', dùng để gửi thông tin ICE candidate cho việc thiết lập kết nối WebRTC.

 **socket.on('disconnect', () => {...})**: Lắng nghe sự kiện khi một client ngắt kết nối.

 **const PORT**: Lấy cổng từ biến môi trường hoặc sử dụng cổng 443 (mặc định cho HTTPS).

 **server.listen(PORT, () => {...})**: Bắt đầu máy chủ và lắng nghe trên cổng đã chỉ định. In ra thông báo cho biết máy chủ đang chạy.

**Ghi chú:** Việc tạo chứng chỉ trong mục VideoCall.

**File index.html**

<!DOCTYPE html>

<html lang="vi">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>WebRTC Cuộc gọi thoại - Hỗ trợ Di động</title>

<style>

/\* Định dạng CSS thân thiện với thiết bị di động \*/

body {

font-family: Arial, sans-serif;

margin: 0;

padding: 0;

text-align: center;

}

h2 {

color: #333;

}

#startCall {

background-color: #4CAF50;

color: white;

padding: 15px 30px;

margin: 20px;

border: none;

font-size: 18px;

border-radius: 8px;

}

#startCall:hover {

background-color: #45a049;

}

audio {

width: 80%;

margin: 20px auto;

}

/\* Responsive cho thiết bị di động \*/

@media (max-width: 600px) {

#startCall {

width: 100%;

padding: 10px 0;

font-size: 16px;

}

audio {

width: 100%;

}

}

</style>

</head>

<body>

<h2>Demo Cuộc gọi thoại với WebRTC - Di động</h2>

<button id="startCall">Bắt đầu cuộc gọi</button>

<audio id="remoteAudio" autoplay></audio>

<!-- Thêm thư viện socket.io client -->

<script src="/socket.io/socket.io.js"></script>

<script>

// Kết nối WebSocket tự động tới máy chủ hiện tại

const socket = io();

let localStream;

let peerConnection;

const servers = { iceServers: [{ urls: "stun:stun.l.google.com:19302" }] };

const remoteAudio = document.getElementById('remoteAudio');

const startCallButton = document.getElementById('startCall');

// Khi người dùng nhấn nút bắt đầu cuộc gọi

startCallButton.onclick = async () => {

// Truy cập microphone để lấy luồng âm thanh

localStream = await navigator.mediaDevices.getUserMedia({ audio: true });

peerConnection = new RTCPeerConnection(servers);

// Thêm các track âm thanh vào peer connection

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

// Xử lý khi nhận được ICE candidate

peerConnection.onicecandidate = ({ candidate }) => {

if (candidate) socket.emit('candidate', candidate);

};

// Xử lý khi nhận luồng âm thanh từ người dùng khác

peerConnection.ontrack = ({ streams: [stream] }) => {

remoteAudio.srcObject = stream;

};

// Tạo và gửi offer đến máy ngang hàng

const offer = await peerConnection.createOffer();

await peerConnection.setLocalDescription(offer);

socket.emit('offer', offer);

};

// Khi nhận được offer từ máy ngang hàng khác

socket.on('offer', async (offer) => {

peerConnection = new RTCPeerConnection(servers);

peerConnection.onicecandidate = ({ candidate }) => {

if (candidate) socket.emit('candidate', candidate);

};

peerConnection.ontrack = ({ streams: [stream] }) => {

remoteAudio.srcObject = stream;

};

// Thiết lập remote offer và tạo answer

await peerConnection.setRemoteDescription(offer);

const answer = await peerConnection.createAnswer();

await peerConnection.setLocalDescription(answer);

socket.emit('answer', answer);

});

// Khi nhận được answer từ máy ngang hàng

socket.on('answer', (answer) => {

peerConnection.setRemoteDescription(answer);

});

// Khi nhận được ICE candidate từ máy ngang hàng

socket.on('candidate', (candidate) => {

peerConnection.addIceCandidate(new RTCIceCandidate(candidate));

});

</script>

</body>

</html>

**Giải thích code:**

<body>

<h2>Demo Cuộc gọi thoại với WebRTC - Di động</h2>

<button id="startCall">Bắt đầu cuộc gọi</button>

<audio id="remoteAudio" autoplay></audio>

 **<h2>**: Tiêu đề của trang, hiển thị "Demo Cuộc gọi thoại với WebRTC - Di động".

 **<button id="startCall">**: Nút này sẽ kích hoạt quá trình bắt đầu cuộc gọi khi người dùng nhấn vào. Thuộc tính id="startCall" giúp định danh phần tử này để xử lý sự kiện trong JavaScript.

 **<audio id="remoteAudio" autoplay>**: Phần tử <audio> được sử dụng để phát âm thanh từ người dùng khác. Thuộc tính autoplay giúp âm thanh được phát tự động khi luồng âm thanh đến.

<script src="/socket.io/socket.io.js"></script>

<script>

// Kết nối WebSocket tự động tới máy chủ hiện tại

const socket = io();

 **<script src="/socket.io/socket.io.js">**: Nhúng thư viện **Socket.io** vào trang để sử dụng WebSocket. File này có thể được tải từ máy chủ nơi trang đang chạy, đảm bảo tính tương thích giữa frontend và backend.

 **const socket = io();**: Kết nối WebSocket tự động đến máy chủ nơi trang web đang được phục vụ. Vì không chỉ định URL, nó sẽ kết nối tới cùng máy chủ cung cấp trang HTML này.

let localStream;

let peerConnection;

const servers = { iceServers: [{ urls: "stun:stun.l.google.com:19302" }] };

const remoteAudio = document.getElementById('remoteAudio');

const startCallButton = document.getElementById('startCall');

 **let localStream; let peerConnection;**: Khai báo các biến để lưu trữ luồng âm thanh của người dùng (localStream) và kết nối peer-to-peer (peerConnection).

 **const servers**: Đây là cấu hình của các ICE server, dùng để định vị máy ngang hàng qua NAT hoặc tường lửa. Ở đây sử dụng máy chủ STUN miễn phí của Google (stun:stun.l.google.com:19302).

 **const remoteAudio**: Lấy phần tử HTML <audio> để phát âm thanh từ người dùng khác.

 **const startCallButton**: Lấy phần tử nút Bắt đầu cuộc gọi để xử lý sự kiện khi người dùng nhấn vào nút này.

startCallButton.onclick = async () => {

// Truy cập microphone để lấy luồng âm thanh

localStream = await navigator.mediaDevices.getUserMedia({ audio: true });

peerConnection = new RTCPeerConnection(servers);

// Thêm các track âm thanh vào peer connection

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

// Xử lý khi nhận được ICE candidate

peerConnection.onicecandidate = ({ candidate }) => {

if (candidate) socket.emit('candidate', candidate);

};

// Xử lý khi nhận luồng âm thanh từ người dùng khác

peerConnection.ontrack = ({ streams: [stream] }) => {

remoteAudio.srcObject = stream;

};

// Tạo và gửi offer đến máy ngang hàng

const offer = await peerConnection.createOffer();

await peerConnection.setLocalDescription(offer);

socket.emit('offer', offer);

};

 **navigator.mediaDevices.getUserMedia({ audio: true })**: Trình duyệt sẽ yêu cầu quyền truy cập microphone của người dùng để thu âm thanh (luồng âm thanh được lưu vào localStream).

 **RTCPeerConnection(servers)**: Tạo kết nối WebRTC peer-to-peer giữa hai người dùng với cấu hình các ICE servers.

 **addTrack()**: Thêm luồng âm thanh (track) của người dùng vào kết nối WebRTC.

 **onicecandidate**: Khi kết nối tìm thấy một ICE candidate (địa chỉ mạng để kết nối), nó sẽ gửi thông tin qua WebSocket tới máy ngang hàng khác.

 **ontrack**: Khi nhận được âm thanh từ máy ngang hàng khác, luồng âm thanh sẽ được phát qua phần tử <audio>.

 **createOffer()**: Tạo một "offer" (lời mời kết nối) để gửi tới máy ngang hàng. Sau khi offer được tạo, nó sẽ được thiết lập làm mô tả địa phương (local description) cho kết nối và gửi qua WebSocket.

socket.on('offer', async (offer) => {

peerConnection = new RTCPeerConnection(servers);

peerConnection.onicecandidate = ({ candidate }) => {

if (candidate) socket.emit('candidate', candidate);

};

peerConnection.ontrack = ({ streams: [stream] }) => {

remoteAudio.srcObject = stream;

};

// Thiết lập remote offer và tạo answer

await peerConnection.setRemoteDescription(offer);

const answer = await peerConnection.createAnswer();

await peerConnection.setLocalDescription(answer);

socket.emit('answer', answer);

});

socket.on('answer', (answer) => {

peerConnection.setRemoteDescription(answer);

});

socket.on('candidate', (candidate) => {

peerConnection.addIceCandidate(new RTCIceCandidate(candidate));

});

 **socket.on('offer')**: Khi nhận được offer từ máy ngang hàng khác, trình duyệt thiết lập nó làm mô tả từ xa (remote description), sau đó tạo và gửi một answer để chấp nhận kết nối.

 **socket.on('answer')**: Khi nhận được answer từ máy ngang hàng khác, answer này được thiết lập làm mô tả từ xa (remote description), hoàn thành quá trình bắt tay (handshake) của WebRTC.

 **socket.on('candidate')**: Khi nhận được ICE candidate từ máy ngang hàng khác, nó được thêm vào kết nối WebRTC để giúp thiết lập đường dẫn mạng giữa hai người dùng.

**II. TẠO CHỨNG CHỈ VỚI OPENSSL**

**1. Vào cmd quyền admin:**

**1. 1.Tạo Khóa Riêng (Private Key)**

Mở terminal hoặc command prompt và chạy lệnh sau để tạo khóa riêng:

Di chuyển vào cmd với F:\BT\_NODEJS\VoiceCAll\key

F:\BT\_NODEJS\VoiceCAll\key >openssl genrsa -out 192.168.1.96-key.pem 2048

**1.2. Tạo Yêu Cầu Chứng Chỉ (CSR)**

Tiếp theo, tạo yêu cầu chứng chỉ (CSR):

F:\BT\_NODEJS\VoiceCAll\key >openssl req -new – key 192.168.1.96-key.pem –out 192.168.1.96.csr

F:\BT\_NODEJS\VoiceCAll\key>openssl x509 -req -days 365 -in 192.168.1.96.csr -signkey 192.168.1.96-key.pem -out 192.168.1.96.pem

**1.3. Tạo file openssl.cnf**

# OpenSSL configuration file.

# Establish a home for this configuration file

HOME = .

# The default configuration section

[req]

default\_bits = 2048

distinguished\_name = req\_distinguished\_name

prompt = no

[req\_distinguished\_name]

C = VN

ST = HN

L = Hanoi

O = My Organization

OU = My Organizational Unit

CN = 192.168.1.96

emailAddress =nguyenquocviet2018ca@gmail.com

**1.4. Tạo chứng chỉ tự ký:**

openssl x509 -req -days 365 -in 192.168.1.96.csr –signkey 192.168.1.96-key.pem -out 192.168.1.96.pem

Việc tạo ra ba thành phần khác nhau trong quy trình cấp chứng chỉ SSL là để đảm bảo tính bảo mật và chức năng của chứng chỉ. Dưới đây là mô tả chi tiết về ba thành phần đó:

**1. Private Key (Khóa Riêng)**

* **Mục đích**: Khóa riêng là một phần quan trọng trong cơ chế mã hóa bất đối xứng. Nó được sử dụng để mã hóa thông tin và tạo chữ ký số.
* **Bảo mật**: Khóa riêng phải được giữ bí mật và không bao giờ được chia sẻ. Nếu người khác có được khóa riêng, họ có thể giả mạo chứng chỉ của bạn.
* **Tạo ra**: Khóa riêng được tạo ra đầu tiên và thường được lưu trữ trong tệp .pem (ví dụ: 192.168.1.96-key.pem).

**2. Certificate Signing Request (CSR)**

* **Mục đích**: CSR là một yêu cầu gửi đến một cơ quan cấp chứng chỉ (CA) để yêu cầu cấp chứng chỉ SSL. Nó chứa thông tin về tổ chức và khóa công khai.
* **Chứa thông tin**: CSR bao gồm khóa công khai và thông tin nhận dạng, chẳng hạn như tên miền, tên tổ chức, địa chỉ email, v.v.
* **Tạo ra**: CSR được tạo ra từ khóa riêng. Lệnh OpenSSL sẽ lấy khóa riêng để tạo ra CSR (ví dụ: 192.168.1.96.csr).

**3. Public Certificate (Chứng Chỉ Công Khai)**

* **Mục đích**: Chứng chỉ công khai (thường có định dạng .pem hoặc .crt) là chứng chỉ được cấp bởi CA hoặc tự ký. Nó chứa khóa công khai và thông tin xác thực.
* **Chức năng**: Chứng chỉ này cho phép người dùng xác minh rằng họ đang kết nối với máy chủ mà họ dự định. Nó cũng giúp mã hóa thông tin trao đổi giữa máy chủ và khách hàng.
* **Tạo ra**: Chứng chỉ công khai được tạo ra từ CSR và khóa riêng. Lệnh OpenSSL sẽ ký CSR bằng khóa riêng để tạo ra chứng chỉ (ví dụ: 192.168.1.96.pem).

**Tóm lại**

* **Khóa riêng** (Private Key) là bí mật và phải được bảo vệ.
* **CSR** (Certificate Signing Request) được sử dụng để yêu cầu cấp chứng chỉ và chứa thông tin cần thiết.
* **Chứng chỉ công khai** (Public Certificate) được cấp phát từ CSR và có thể được chia sẻ với bất kỳ ai muốn xác thực máy chủ của bạn.

Việc tạo ra ba thành phần này là một phần quan trọng trong quy trình thiết lập SSL/TLS, giúp đảm bảo an toàn và bảo mật cho các kết nối mạng. Nếu bạn có bất kỳ câu hỏi nào thêm hoặc cần giải thích sâu hơn về bất kỳ phần nào, hãy cho tôi biết!