**EXPERIMENT 8**

Enable real-time communication via WebSockets

## **Title: Enabling Real-Time Communication via WebSockets**

### **Aim**

To implement **real-time, bidirectional communication** between the client and server for the Interview Simulator Website using the WebSocket protocol, enabling instant message exchange (e.g., live chat, instant feedback, coding interview collaboration).

### **Theory**

**1. Introduction to WebSockets** The **WebSocket protocol** enables persistent, full-duplex communication between a client and a server over a single TCP connection. Unlike traditional HTTP requests (which are one-way and stateless), WebSockets allow **continuous, two-way data exchange** without repeatedly opening new connections.

This makes them ideal for **real-time web applications** such as:

* Live chat systems
* Online interviews and collaborative editors
* Multiplayer games
* Live dashboards or notifications

**2. How WebSockets Work**

* A WebSocket starts as a normal HTTP connection.
* The client sends an **HTTP Upgrade request** to switch protocols.
* Once upgraded, both client and server can send messages to each other independently.
* The connection remains open until explicitly closed by either party.

**3. Advantages of WebSockets**

* **Low latency:** Messages are pushed instantly without polling.
* **Bidirectional:** Both client and server can initiate communication.
* **Efficient:** Reduced overhead compared to repetitive HTTP requests.
* **Persistent Connection:** Maintains continuous communication.

**4. Use in the Interview Simulator Website** In your project, WebSockets can power:

* **Live interviewer–candidate chat**
* **Real-time code collaboration** (e.g., sharing text or code in an editor)
* **Instant feedback or status updates** (e.g., “Candidate submitted solution”)

### **Procedure**

#### **Step 1: Set Up a Basic Node.js WebSocket Server**

Initialize a Node.js project:  
  
 mkdir websocket-server

cd websocket-server

npm init -y

npm install ws express

Create a file server.js:  
  
 const express = require('express');

const http = require('http');

const WebSocket = require('ws');

const app = express();

const server = http.createServer(app);

const wss = new WebSocket.Server({ server });

wss.on('connection', (ws) => {

console.log('Client connected');

ws.on('message', (message) => {

console.log(`Received: ${message}`);

ws.send(`Server echo: ${message}`);

});

ws.on('close', () => console.log('Client disconnected'));

});

app.use(express.static('public'));

server.listen(8080, () => console.log('WebSocket server running on port 8080'));

#### **Step 2: Create the WebSocket Client (HTML + JS)**

Inside a public/ folder, create index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>WebSocket Demo</title>

</head>

<body>

<h2>WebSocket Chat</h2>

<input id="msgInput" type="text" placeholder="Type a message" />

<button onclick="sendMessage()">Send</button>

<pre id="output"></pre>

<script>

const socket = new WebSocket("ws://localhost:8080");

socket.onopen = () => appendMessage("Connected to server");

socket.onmessage = (event) => appendMessage("Server: " + event.data);

socket.onclose = () => appendMessage("Disconnected");

function sendMessage() {

const msg = document.getElementById('msgInput').value;

socket.send(msg);

appendMessage("You: " + msg);

}

function appendMessage(msg) {

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

output.textContent += msg + "\\n";

}

</script>

</body>

</html>

#### **Step 3: Serve the HTML File**

The Express server in server.js already serves static files from the /public folder, so you can directly open the app at:  
 👉 http://localhost:8080

#### **Step 4: Run the WebSocket Server**

Start your server:

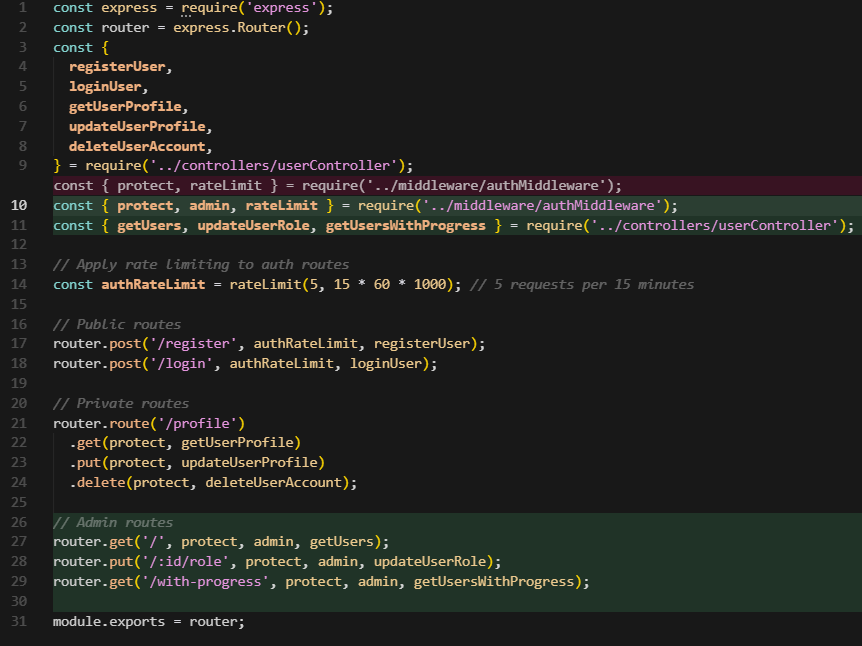
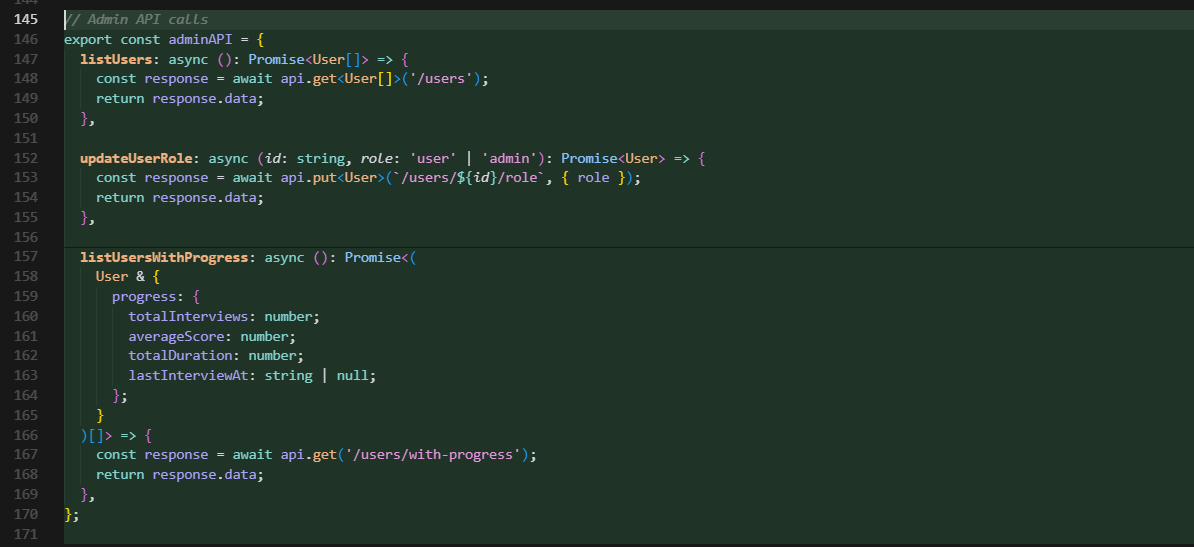
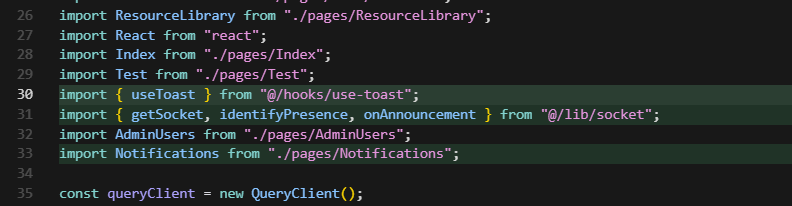
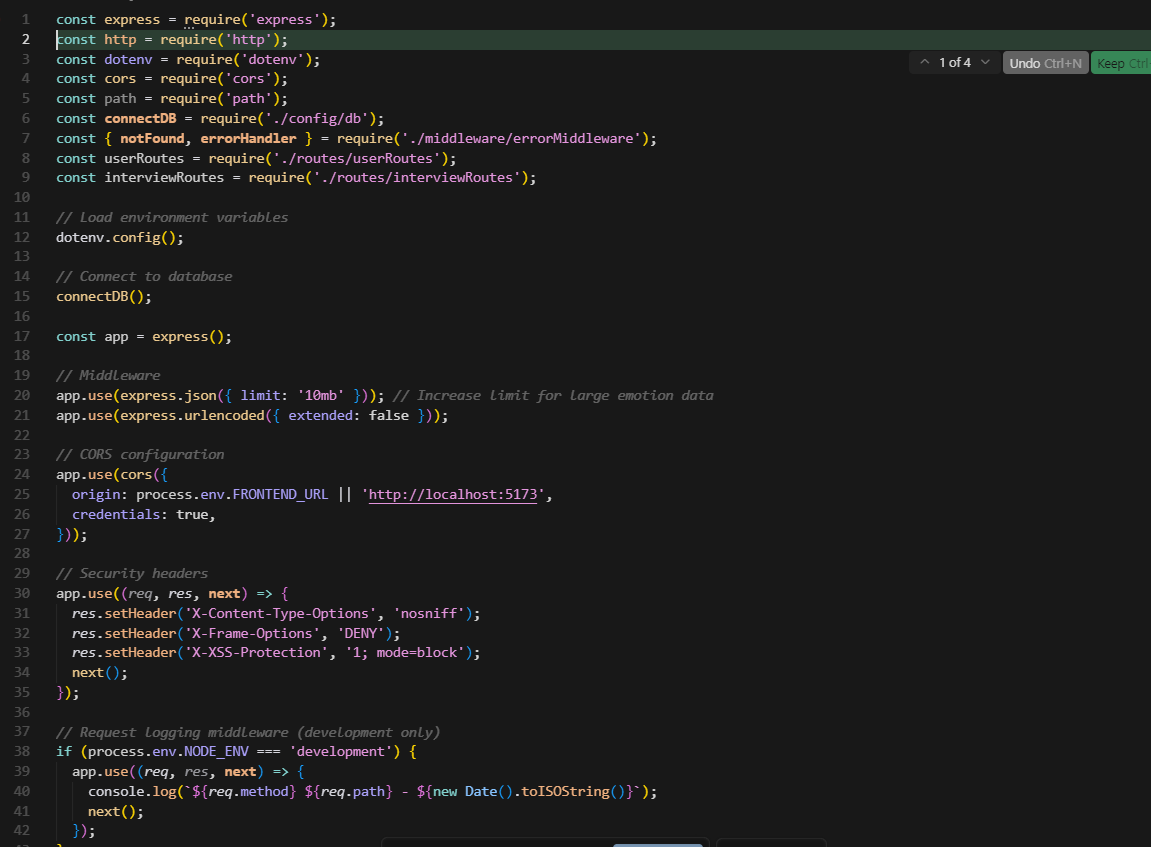
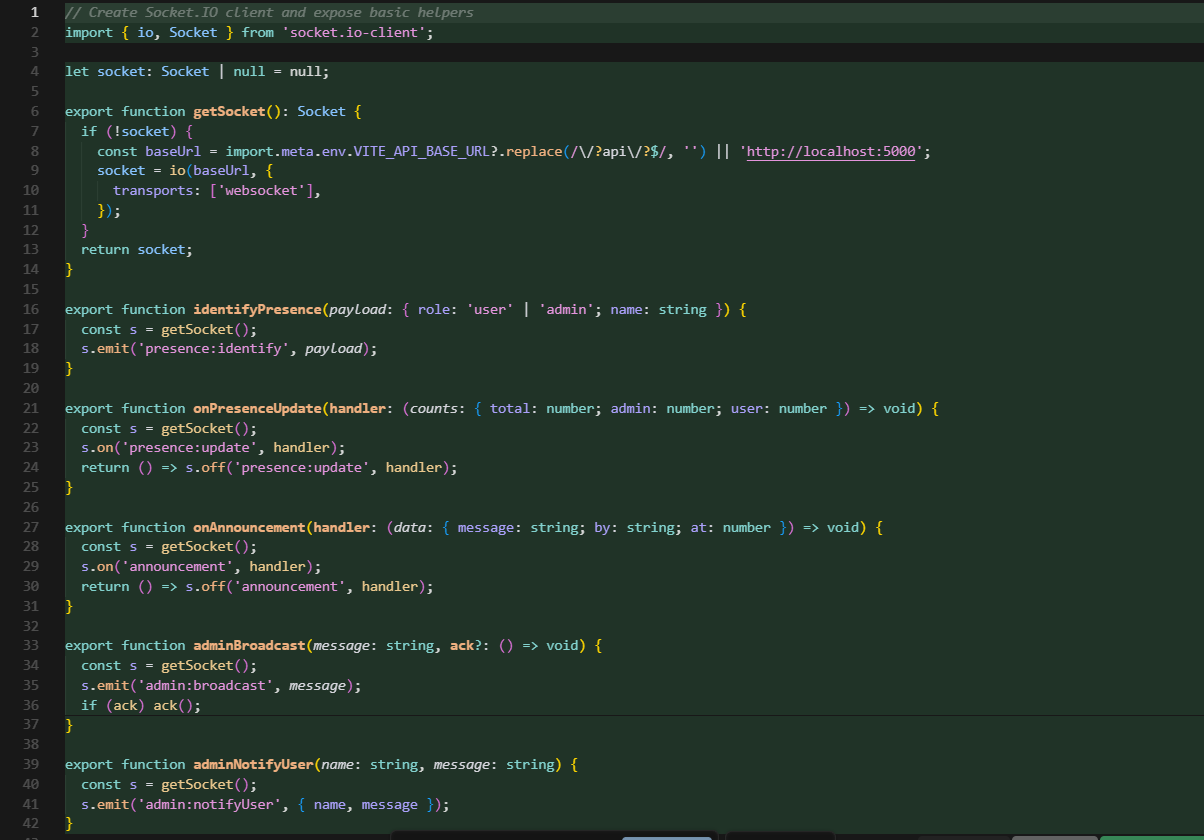
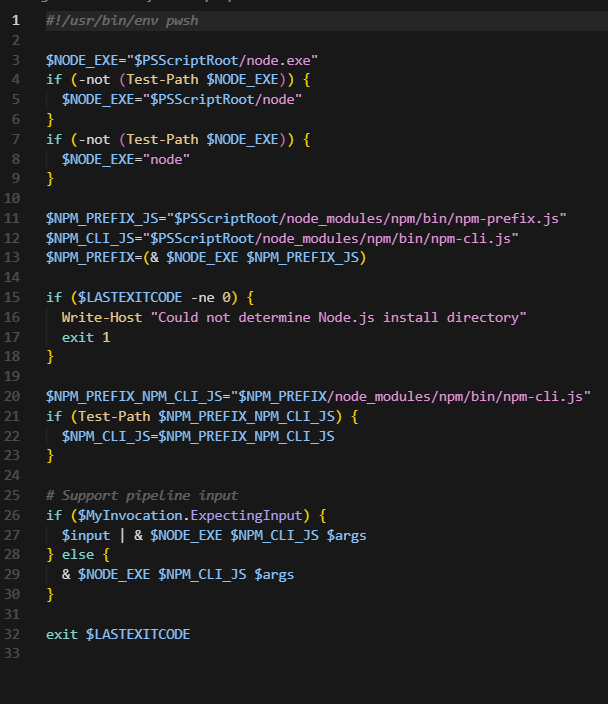
node server.js

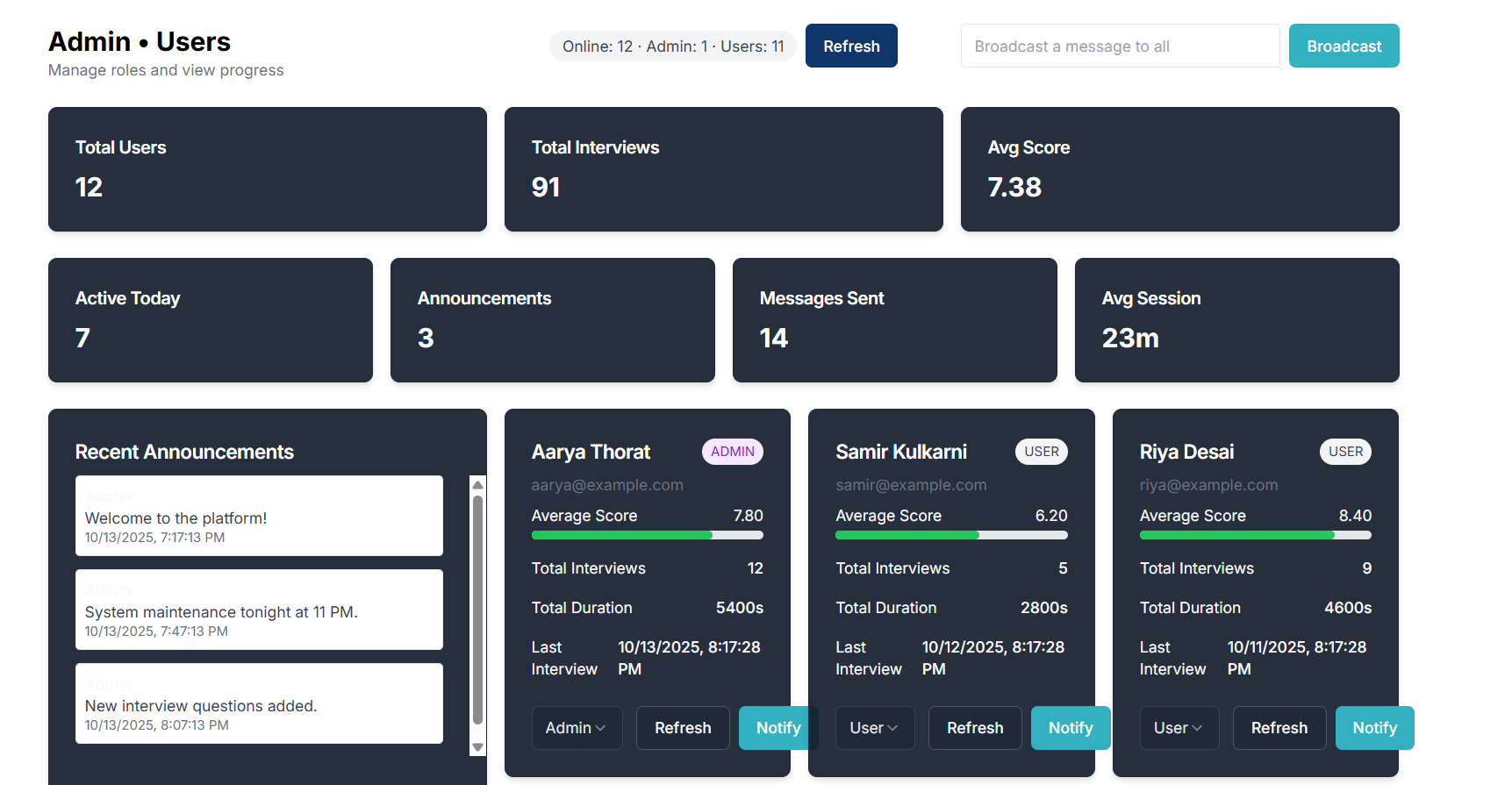
Open multiple browser tabs pointing to http://localhost:8080.  
 Send messages from one tab — they’ll be echoed by the server to the sender (and can later be broadcast to others).

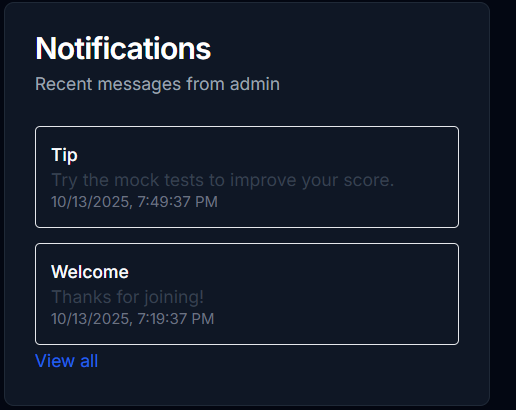
### **Example Communication Flow**

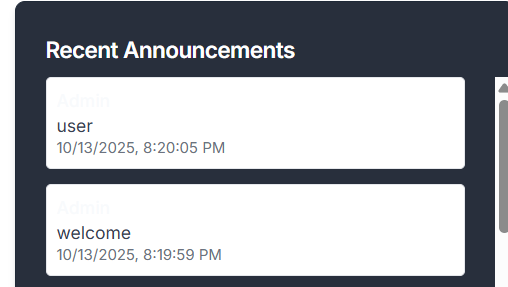
1. The browser connects to the WebSocket server.
2. The user types a message → The message is sent to the server via WebSocket.
3. The server receives and echoes it back.
4. The client displays the server’s response in real time.

In your **Interview Simulator Website**, this can be extended for real-time interviewer–candidate communication, typing indicators, or instant evaluation results.

**Code-  
**

****

****

****

### **Conclusion**

By integrating WebSockets, the Interview Simulator Website achieves **real-time communication**, a key feature for interactive interviews. Unlike traditional polling or REST APIs, WebSockets maintain an open connection, allowing **instant bidirectional data flow** between client and server.

This setup enhances user experience with features like **live chat, collaborative editing, and instant feedback**. In production, WebSockets can be secured using **wss://** and scaled using **load balancers** or frameworks like **Socket.IO** for reliability and cross-browser support.