Gatherly - Real-Time WebRTC Video Conferencing

A PROJECT REPORT

Submitted by

Yeshika(23BDA70132)

in partial fulfillment for the award of the degree of

Bachelor of Engineering

IN

Big Data Analyst

Chandigarh University

November 2025

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

BONAFIDE CERTIFICATE

Certified that this project report **"Gatherly - Real-Time WebRTC Video Conferencing"** is a work of "Yeshika" (23BDA70132) who carried out the project under my supervision.

SIGNATURE	SIGNATURE
Of the Supervisor	of the AGM-Technical

Submitted for the project viva-voce

Examination held on

INTERNAL EXAMINER	EXTERNAL EXAMINER

1. PROJECT DESCRIPTION

1.1 Overview

Gatherly is a fully functional, real-time video conferencing application built on a modern, full-stack architecture. The system demonstrates a practical approach to building a scalable and user-friendly communication platform using Node.js for the backend and React for the frontend. It leverages the power of WebRTC (Web Real-Time Communication) to establish direct, peer-to-peer video and audio streams between clients, ensuring low-latency and high-performance communication. A central signaling server built with Socket.IO orchestrates the connection lifecycle, managing user sessions within distinct rooms and facilitating the initial WebRTC handshake..

1.2 Problem Statement

The primary challenge addressed by this project is the implementation of a real-time, multi-user video communication system from the ground up. While many commercial solutions exist, they often obscure the underlying complexities. This project aims to solve the following key problems:

- Establish direct, low-latency video and audio streams between multiple browsers over the internet.
- Create a mechanism to allow users in the same "room" to discover and connect to one another.
- Design a responsive, intuitive, and feature-rich user interface that provides users with essential meeting controls.
- Implement a clean, scalable project structure with a clear separation between frontend and backend concerns.
- Provide a seamless user flow, from creating a meeting to inviting others via a simple, shareable link

1.3 Main Features

- **Real-Time Video/Audio Communication:** High-quality, low-latency video and audio streaming directly between participants' browsers using WebRTC.
- **Dynamic Room Management:** Users can create new meeting rooms with a unique, randomly generated 6-character ID, or join existing rooms.
- **Shareable Invitation Links:** A core feature is the ability to invite others via a URL. When a user clicks an invite link (e.g., .../?roomId=ABCDEF), they are taken directly to the lobby for that specific meeting.
- **Interactive Lobby:** Before joining a call, users enter a lobby where they can preview their camera, set their display name, and confirm their media devices are working correctly.
- **Responsive Video Gallery:** The main meeting UI features a dynamic grid that automatically arranges participant video tiles in a balanced and aesthetically pleasing layout that adapts to the number of people in the call.
- **Core In-Meeting Controls:** A modern, floating control bar provides users with all essential functions: Mute/Unmute, Start/Stop Camera, and Screen Sharing.
- **Participant List & Hand Raising:** A toggleable side panel displays a real-time list of all attendees. Users can use the "Raise Hand" feature to signal they wish to speak, which displays a visual indicator next to their name.
- **Simple Text Chat:** The application includes a basic, non-persistent text chat in a side panel for in-meeting communication.

1.4 System Architecture

The application is built on a hybrid architecture that is standard for WebRTC applications: a centralized server for signaling and a decentralized, peer-to-peer network for media streaming.

1. **Backend (Signaling Server):** A lightweight Node.js server using Express and Socket.IO. Its only responsibility is to act as a "switchboard." When a user wants to join a room, they connect to this server. The server then informs other users in the room of the new arrival and helps them exchange the necessary information (SDP offers/answers and ICE candidates) to establish a direct connection.
2. **Frontend (Client):** A React Single Page Application (SPA) that runs in the user's browser. It captures the user's camera and microphone, connects to the signaling server, and, most importantly, establishes direct WebRTC RTCPeerConnections with all other participants in the room.
3. **Communication Flow:**
    - All users connect to the signaling server via a persistent WebSocket (Socket.IO) connection.
    - The server manages room logic and relays handshake messages between clients.
    - Once the handshake is complete, all video and audio data is streamed directly between the clients' browsers over an encrypted P2P channel. This minimizes latency and server costs.

1.5 Technology Stack

- **Backend Services**
    - **Node.js:** JavaScript runtime for server-side execution.
    - **Express.js:** Web framework used to create the basic server structure.
    - **Socket.IO:** A library for enabling real-time, bidirectional, event-based communication for signaling.
- **Frontend Application**
    - **React:** A JavaScript library for building component-based, interactive user interfaces.
    - **TypeScript:** Adds static typing to the project for improved code quality, readability, and error detection.
    - **Vite:** A modern, high-performance development server and build tool that provides instant transpilation and hot-reloading.
    - **Tailwind CSS:** A utility-first CSS framework for rapidly styling the user interface.
- **Real-Time Protocol**
    - **WebRTC (Web Real-Time Communication):** The core browser API that enables peer-to-peer streaming of video and audio without the need for plugins.

1.6 Future Enhancements

- **Cloud Deployment:** The most critical next step is to deploy the backend to a Platform-as-a-Service (PaaS) like **Render** and the frontend to a static hosting service like **Netlify** or **Vercel**. This would make the application publicly accessible to anyone on the internet.
- **User Authentication:** Implement a full user registration and login system, likely using JSON Web Tokens (JWTs). This would secure meetings and enable user-specific features like contact lists or saved meeting history.
- **Persistent Chat:** Integrate a NoSQL database like MongoDB or a real-time database like Firebase to store chat messages, allowing users to see the history if they join late or reconnect to a meeting.
- **Configure a TURN Server:** To improve connection reliability, a TURN server could be implemented (e.g., using an open-source project like Coturn or a paid service). This would help users connect even when they are behind highly restrictive corporate or university firewalls.
- **Externalize Configuration:** Remove all hardcoded values like the server URL (http://localhost:3001) and manage them using environment variables. This is a standard practice for building portable, "12-factor" applications that can easily be deployed to different environments (like staging or production).

2. HARDWARE/SOFTWARE REQUIREMENTS

2.1 Hardware Requirements

Component	Minimum Specification	Recommended Specification
Processor	Intel i5 / Ryzen 5	Intel Core i5 / AMD Ryzen 5
RAM	8 GB	16 GB
Storage	128 GB SSD	256 GB SSD
Network	5 Mbps	25 Mbps

2.2 Software Requirements

- **Node.js**: Version 18.x (LTS) or higher is recommended for performance and security.
- **npm**: Version 8.x or higher (typically bundled with Node.js).
- **Git**: Version 2.30.0 or higher for source code management.
- **Web Framework**: Express.js
- **Real-time Engine**: Socket.IO
- UI **Library**: React (Version 18 or higher).
- **Language**: TypeScript
- **Project Scaffolding**: Vite
- **Styling**: Tailwind CSS
- **Real-time Client**: Socket.IO Client
- **Real-Time Protocol**: WebRTC (natively supported in modern browsers).
- **Signaling Transport**: WebSockets (managed by Socket.IO).
- **Windows**: Windows 10 or 11.
- **macOS**: macOS 11 (Big Sur) or later.
- **Linux**: Ubuntu 20.04 LTS, Debian 11, Fedora 36 or later distributions.

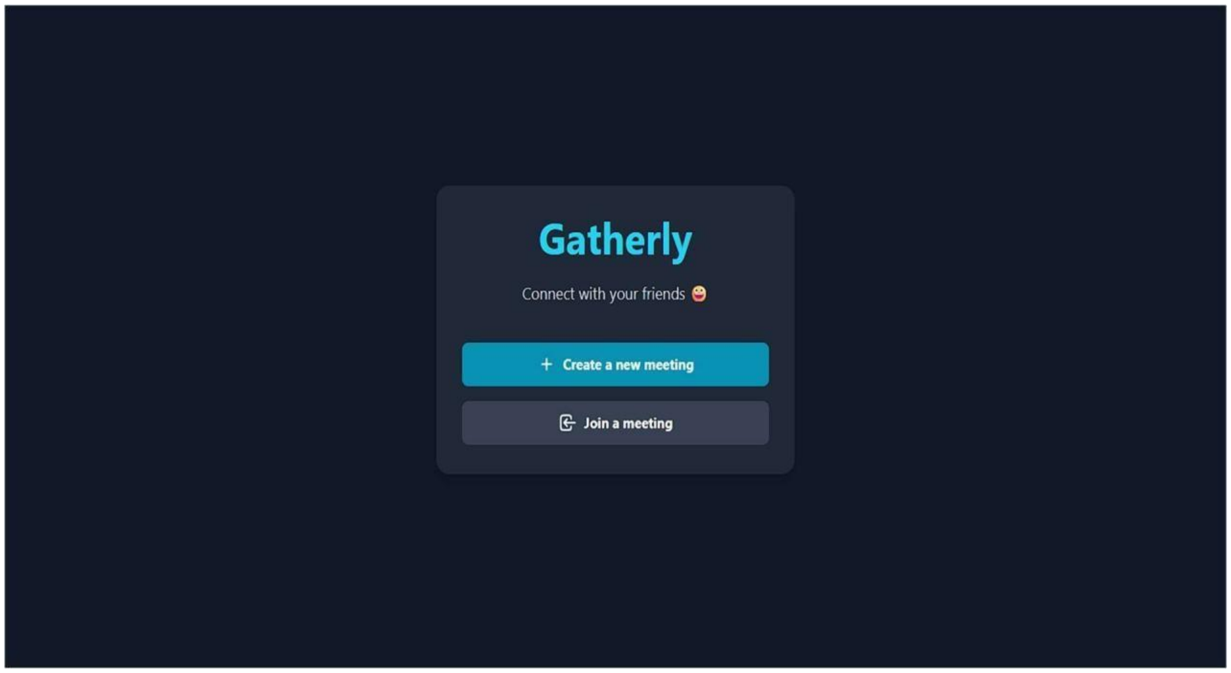
2.3 Deployment Requirements

- **Application Hosting (Backend):** A platform that supports Node.js applications, such as Heroku, Render, or Railway.
- **Static Hosting (Frontend):** A platform for hosting static files, such as Netlify, Vercel, or GitHub Pages.
- **HTTPS:** A valid SSL/TLS certificate is mandatory for WebRTC to function securely in production environments. Services like Let's Encrypt provide free certificates, and most hosting platforms handle this automatically.

3. FRONT-END SCREENS

5.2 Login Page

The Gatherly application does not feature a traditional login system with user accounts. Instead, its entry point is a clean and functional Home Page. This page serves as the main gateway, presenting the user with two clear choices: creating a brand-new meeting or joining a pre-existing one, ensuring a frictionless start to their video conferencing experience. The application deliberately forgoes a traditional user account and login system to maximize accessibility and speed. The Home Page serves as the primary entry point, immediately presenting the user with the application's core purpose. The minimalist design reduces cognitive load.



Key Elements of the User Interface

1. Header and Title

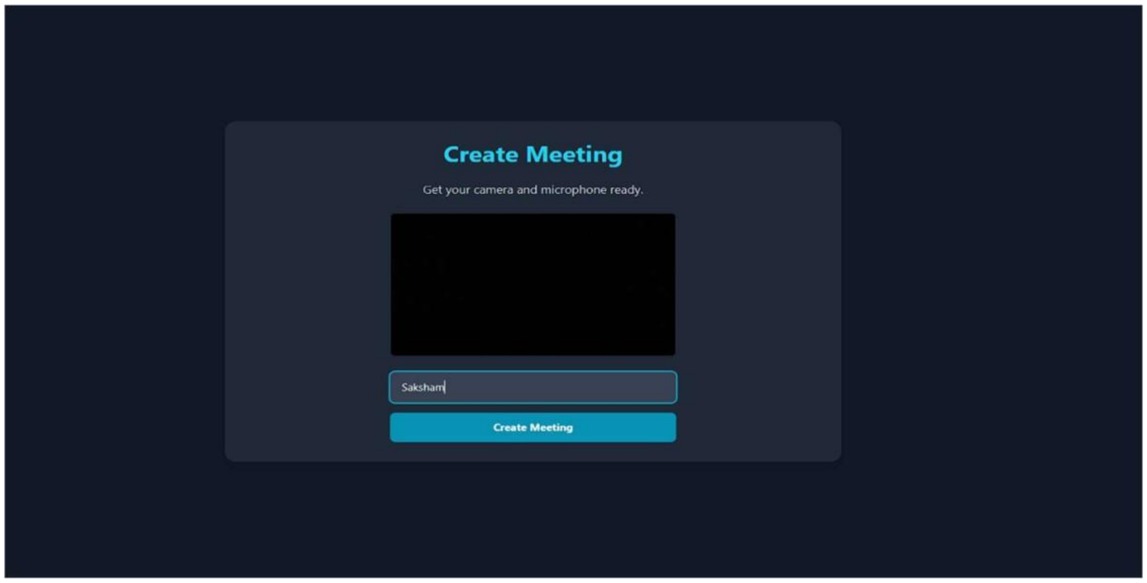
The top of the page features a prominent <h1> title, "Gatherly," to establish the application's brand identity. Below it, a friendly and concise subtitle, "Connect with your friends 😛," communicates the tool's purpose in a welcoming tone. This combination immediately informs the user what the application is and what it does, setting a positive context for their interaction.

2. Action Buttons :

The primary interactive components are two large, clearly labeled buttons. The "Create a new meeting" button is styled as the primary call-to-action with a vibrant color, encouraging users to start a new session. The "Join a meeting" button is styled as a secondary action, providing an equally clear but visually distinct path for users who have been invited to a call.

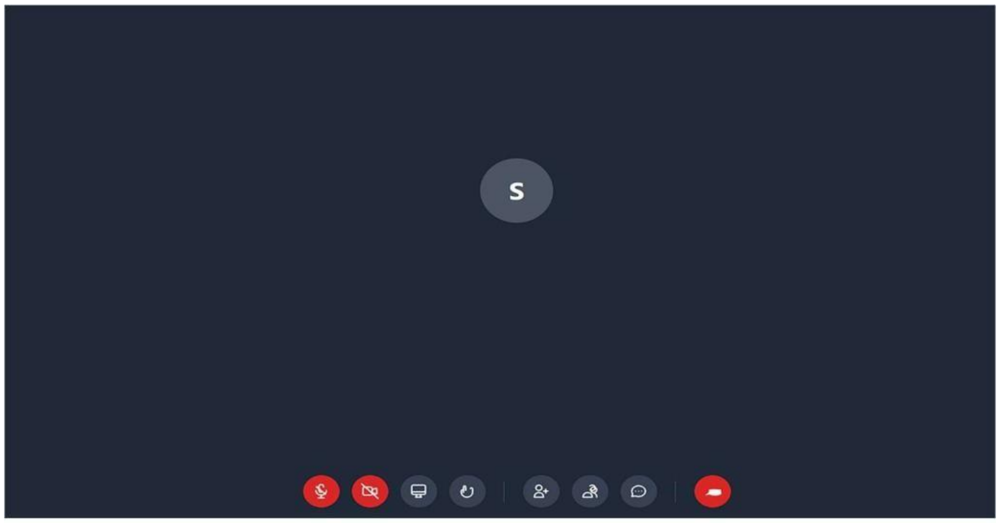
3. Lobby Screen :

This screen acts as an essential intermediate step before a user enters a live call. It contains a real-time video preview for the user to check their appearance and background, a mandatory text input field to set their display name, and a context-aware action button ("Create Meeting" or "Join Meeting") that is only enabled once the user is ready. This ensures a smooth and prepared entry into the meeting.



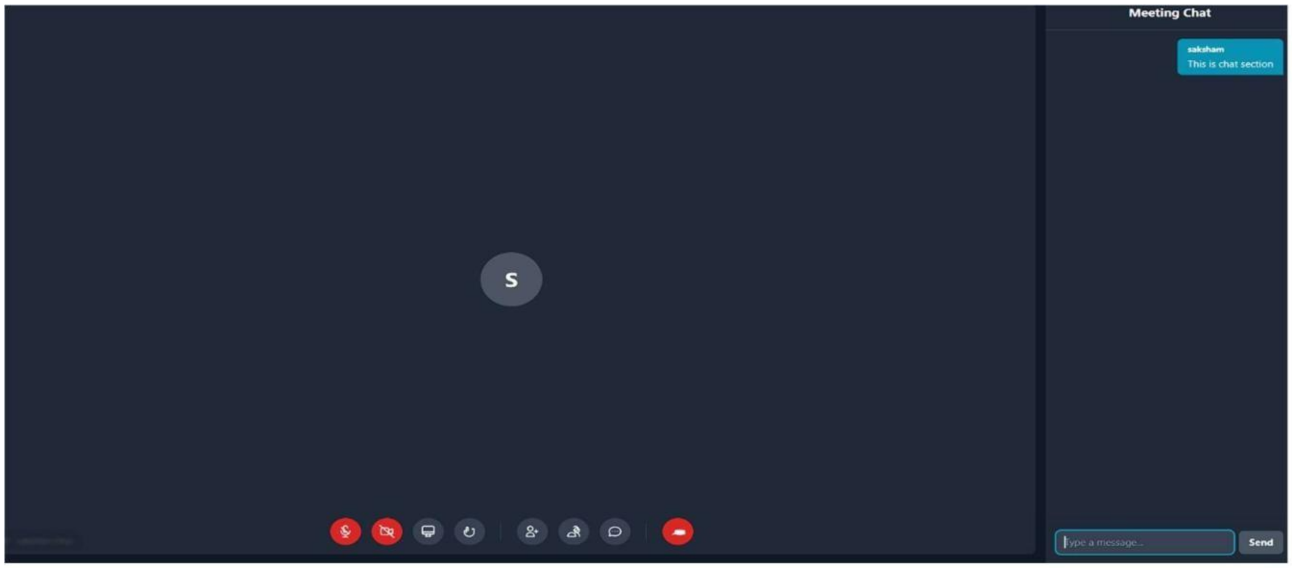
4. Main Meeting Interface :

The core meeting UI is dominated by a dynamic video gallery that responsively arranges participant tiles. A modern, floating Control Bar is overlaid at the bottom of the screen, providing persistent access to critical in-call functions like mute, camera control, screen sharing, and ending the call. This layered design ensures that user controls are always accessible and never obscured by the video content.



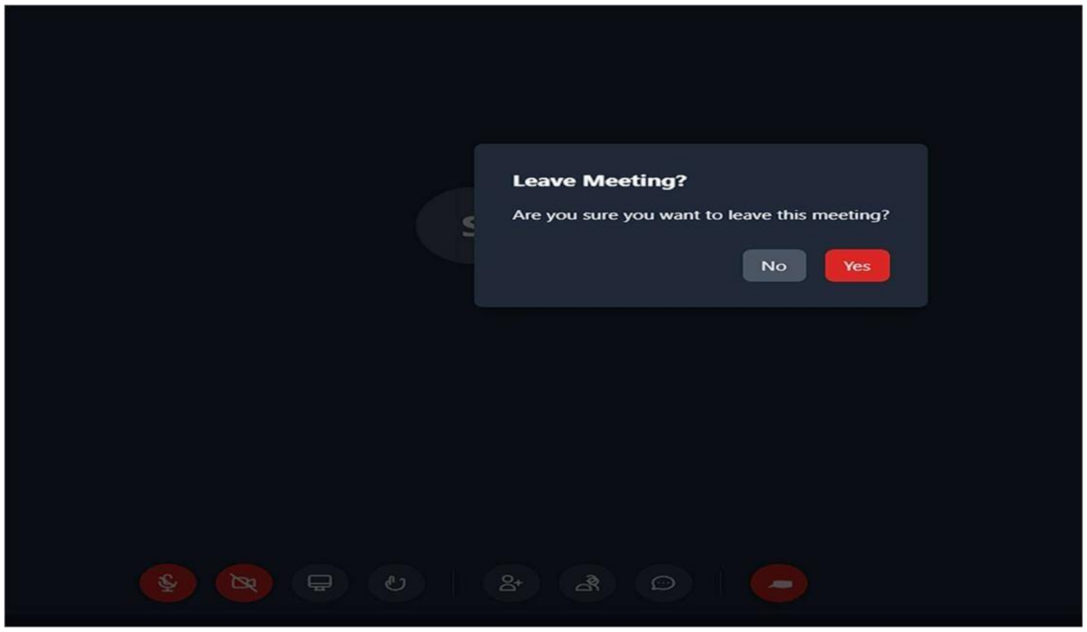
5. Chat Interface:

The application includes a real-time text chat feature, accessible via a button on the control bar. The chat opens in a side panel, allowing users to communicate without interrupting the speaker. It is ideal for sharing links, asking questions, or troubleshooting. The interface clearly distinguishes between messages sent by the user and those from other participants.



6. Leave Confirmation Dialog:

To improve the user experience and prevent accidental departures, the application features a confirmation dialog. When a user clicks the "End Call" button, a modal window appears asking them to confirm their decision. This simple step ensures that leaving a meeting is always an intentional act.



4. LIMITATION & FUTURE SCOPE

The Gatherly project successfully implements a core Minimum Viable Product (MVP) that demonstrates a sophisticated, real-time communication architecture. However, to be considered a fully production-ready system, it has several important limitations that provide a clear roadmap for future development.

4.1 Current Limitations:

1. **No Authentication or Security**: The application is completely open, lacking a user login or authorization system. This means meeting rooms are not private, and any user with a room ID can join. This poses a significant security risk in a production environment as there is no way to control access or verify user identity.

2. **Ephemeral Data and State**: All application data, including chat messages and the server-side room list, is stored in memory. If the server restarts, all active meetings are terminated. Similarly, chat history is lost for any user who refreshes their browser, creating a disjointed user experience.

3. **Limited Network Traversal**: The system relies exclusively on public STUN servers to facilitate peer-to-peer connections. It does not have a TURN server configured, which means users behind certain restrictive corporate or university firewalls (symmetric NATs) may be unable to establish a connection with other peers, leading to call failures.

4. **Basic State Synchronization**: While the core video and audio streams are synchronized, other important user states are not. For instance, when a user mutes their microphone or turns off their camera, this status is not broadcast, so other participants do not see a visual indicator of their status.

5. **Hardcoded Configuration**: Critical configuration values, most notably the backend signaling server URL, are hardcoded directly into the frontend source code. This makes the application inflexible and difficult to deploy to different environments (e.g., from a local machine to a live production server) without manually changing the code.

4.2 Future Scope and Enhancements

1. Implement User Authentication:

The highest priority enhancement is to secure the application by adding a full user registration and login system, likely using JSON Web Tokens (JWTs). This would enable private, user-specific meeting rooms, prevent unauthorized access, and provide a foundation for features like contact lists and meeting history.

2. Deploy to the Cloud:

To make Gatherly publicly accessible, the backend server should be deployed to a Platform-as-a-Service (PaaS) like Render or Heroku. The frontend application should be built and deployed to a static hosting service like Netlify or Vercel. This would allow anyone on the internet to use the application.

3. **Configure a TURN Serve:**

   To dramatically improve connection reliability, a TURN server should be implemented. This could be done by setting up an open-source project like Coturn on a cloud server or by using a paid third-party service. This would ensure users behind restrictive firewalls can still connect to meetings successfully.

4. **Add Persistent Chat:**

   The chat feature can be greatly improved by integrating a real-time, NoSQL database like MongoDB or Firebase. This would allow chat messages to be saved and retrieved, meaning users who join late or reconnect to a meeting could see the entire conversation history, making the feature far more useful.

5. **Advanced State Synchronization:**

   The signaling server's logic should be expanded to broadcast additional user states. This would include events for when a participant mutes/unmutes their microphone, turns their camera on/off, or raises/lowers their hand. The frontend would then display real-time visual indicators for these actions on every participant's tile.

6. **Implement Meeting Recording:**

   A powerful feature would be the ability to record meetings. This could be implemented either client-side, using the browser's MediaRecorder API, or server-side, using a more complex media server. The resulting video files could then be saved to a cloud storage service like AWS S3 for later viewing and sharing.

7. **Externalize Configuration:**

   All hardcoded values, especially the backend URL, should be removed from the source code and managed using environment variables (e.g., via a .env file). This is a standard "12-factor app" practice that makes the application portable, secure, and easy to configure for different environments like development, staging, and production.

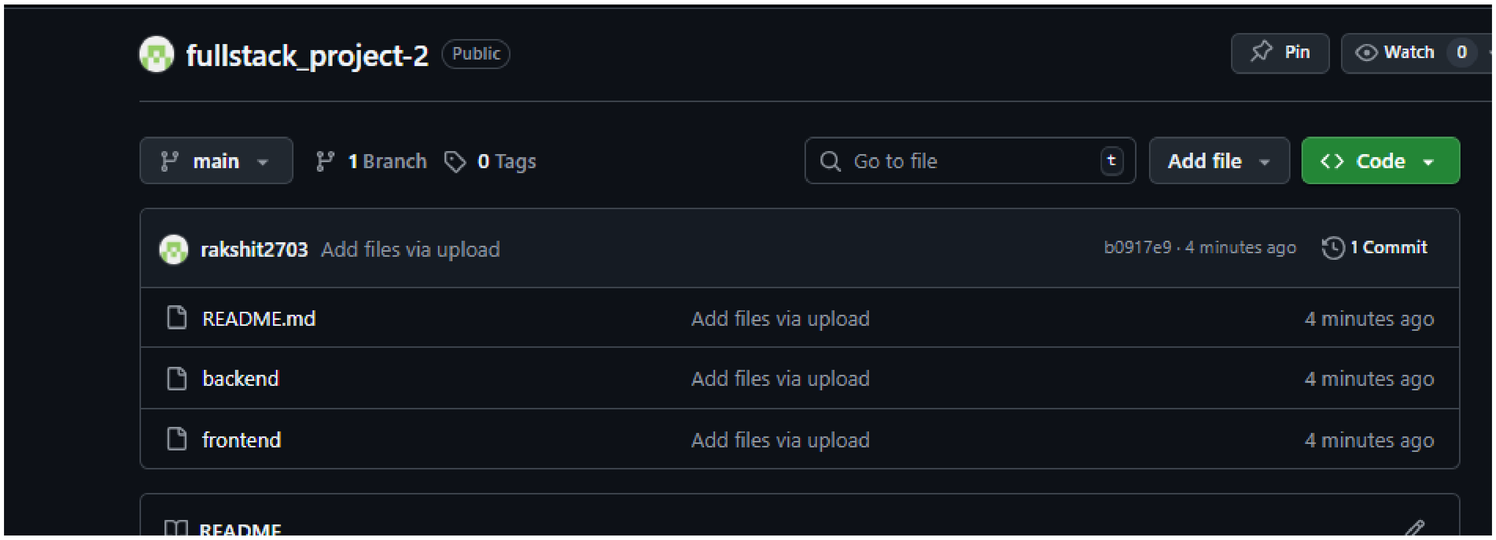
8. **Speaker Detection Highlighting:**

   To improve the user experience in calls with many participants, a speaker detection feature could be implemented. By monitoring audio levels from each remote stream, the UI could automatically place a colored border or highlight around the video tile of the person who is currently speaking, making it easier to follow the conversation.

5. GITHUB URL

Project Repository: https://github.com/rakshit2703/fullstack_project-2.git

Repository Structure:



fullstack_project-2 Public

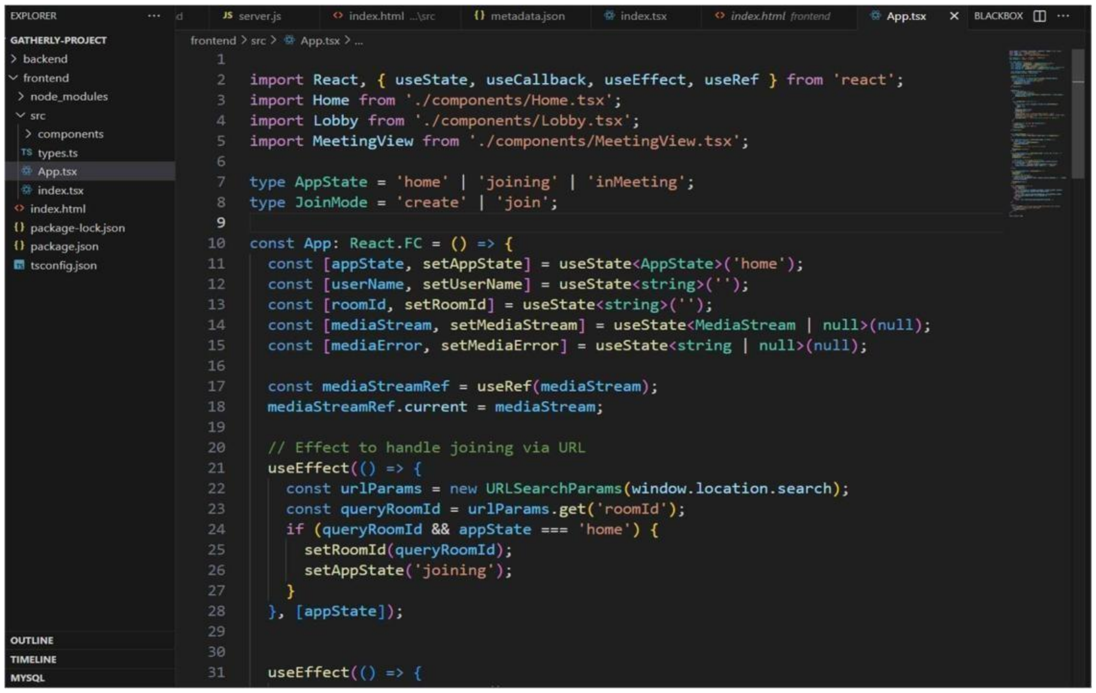
rakshit2703 Add files via upload b0917e9 · 4 minutes ago 1 Commit

README.md	Add files via upload	4 minutes ago
backend	Add files via upload	4 minutes ago
frontend	Add files via upload	4 minutes ago

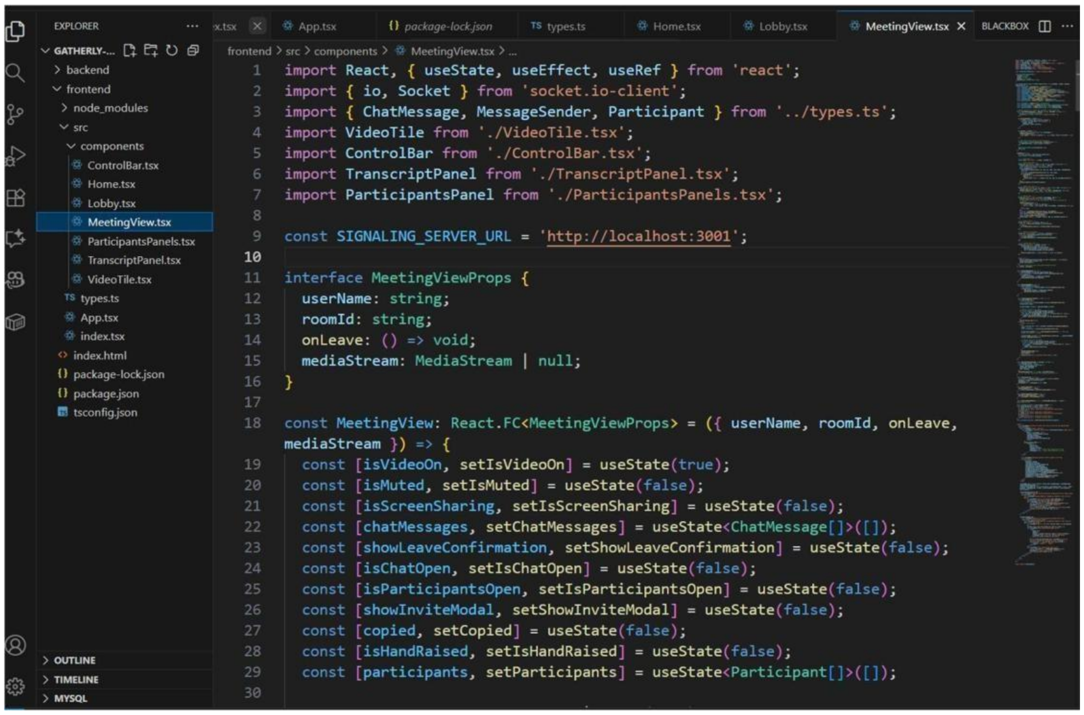
README

6. Project Code:

6.1 Frontend:

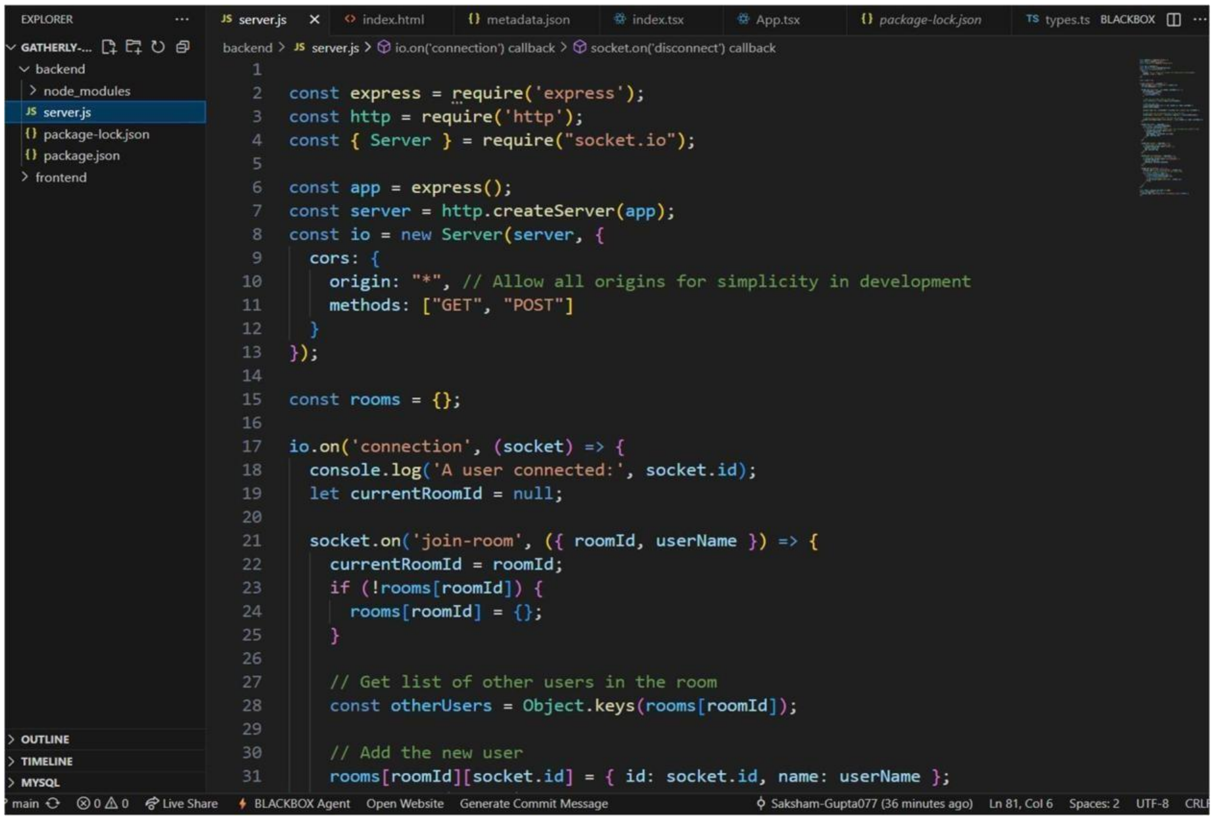


```tsx
import React, { useState, useCallback, useEffect, useRef } from 'react';
import Home from './components/Home.tsx';
import Lobby from './components/Lobby.tsx';
import MeetingView from './components/MeetingView.tsx';

type AppState = 'home' | 'joining' | 'inMeeting';
type JoinMode = 'create' | 'join';

const App: React.FC = () => {
  const [appState, setAppState] = useState<AppState>('home');
  const [userName, setUserName] = useState<string>('');
  const [roomId, setRoomId] = useState<string>('');
  const [mediaStream, setMediaStream] = useState<MediaStream | null>(null);
  const [mediaError, setMediaError] = useState<string | null>(null);

  const mediaStreamRef = useRef(mediaStream);
  mediaStreamRef.current = mediaStream;

  // Effect to handle joining via URL
  useEffect(() => {
    const urlParams = new URLSearchParams(window.location.search);
    const queryRoomId = urlParams.get('roomId');
    if (queryRoomId && appState === 'home') {
      setRoomId(queryRoomId);
      setAppState('joining');
    }
  }, [appState]);


  useEffect(() => {
```

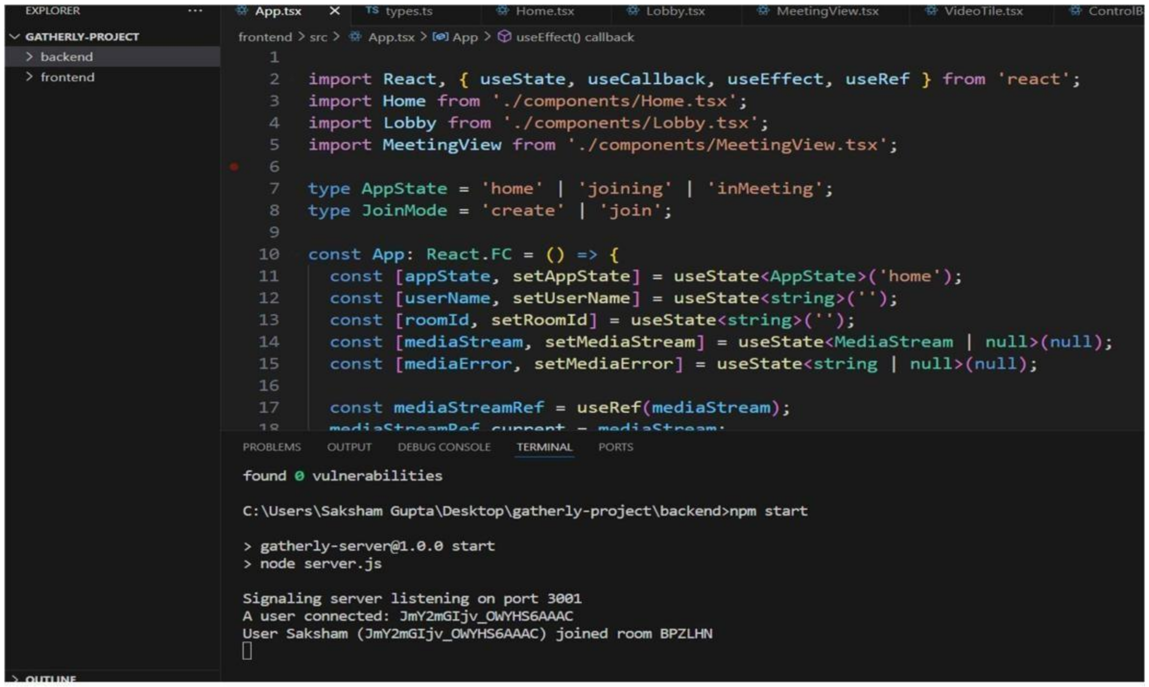


```tsx
import React, { useState, useEffect, useRef } from 'react';
import { io, Socket } from 'socket.io-client';
import { ChatMessage, MessageSender, Participant } from '../types.ts';
import VideoTile from './VideoTile.tsx';
import ControlBar from './ControlBar.tsx';
import TranscriptPanel from './TranscriptPanel.tsx';
import ParticipantsPanel from './ParticipantsPanels.tsx';

const SIGNALING_SERVER_URL = 'http://localhost:3001';

interface MeetingViewProps {
  userName: string;
  roomId: string;
  onLeave: () => void;
  mediaStream: MediaStream | null;
}

const MeetingView: React.FC<MeetingViewProps> = ({ userName, roomId, onLeave, mediaStream }) => {
  const [isVideoOn, setIsVideoOn] = useState(true);
  const [isMuted, setIsMuted] = useState(false);
  const [isScreenSharing, setIsScreenSharing] = useState(false);
  const [chatMessages, setChatMessages] = useState<ChatMessage[]>([]);
  const [showLeaveConfirmation, setShowLeaveConfirmation] = useState(false);
  const [isChatOpen, setIsChatOpen] = useState(false);
  const [isParticipantsOpen, setIsParticipantsOpen] = useState(false);
  const [showInviteModal, setShowInviteModal] = useState(false);
  const [copied, setCopied] = useState(false);
  const [isHandRaised, setIsHandRaised] = useState(false);
  const [participants, setParticipants] = useState<Participant[]>([]);
```

6.2 Backend:



```
const express = require('express');
const http = require('http');
const { Server } = require("socket.io");

const app = express();
const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: "*", // Allow all origins for simplicity in development
    methods: ["GET", "POST"]
  }
});

const rooms = {};

io.on('connection', (socket) => {
  console.log('A user connected:', socket.id);
  let currentRoomId = null;

  socket.on('join-room', ({ roomId, userName }) => {
    currentRoomId = roomId;
    if (!rooms[roomId]) {
      rooms[roomId] = {};
    }

    // Get list of other users in the room
    const otherUsers = Object.keys(rooms[roomId]);

    // Add the new user
    rooms[roomId][socket.id] = { id: socket.id, name: userName };
```



```
import React, { useState, useCallback, useEffect, useRef } from 'react';
import Home from './components/Home.tsx';
import Lobby from './components/Lobby.tsx';
import MeetingView from './components/MeetingView.tsx';

type AppState = 'home' | 'joining' | 'inMeeting';
type JoinMode = 'create' | 'join';

const App: React.FC = () => {
  const [appState, setAppState] = useState<AppState>('home');
  const [userName, setUserName] = useState<string>('');
  const [roomId, setRoomId] = useState<string>('');
  const [mediaStream, setMediaStream] = useState<MediaStream | null>(null);
  const [mediaError, setMediaError] = useState<string | null>(null);

  const mediaStreamRef = useRef(mediaStream);
```

```
found 0 vulnerabilities

C:\Users\Saksham Gupta\Desktop\gatherly-project\backend>npm start

> gatherly-server@1.0.0 start
> node server.js

Signaling server listening on port 3001
A user connected: JmY2mGIjv_OWYHS6AAAC
User Saksham (JmY2mGIjv_OWYHS6AAAC) joined room BPZLHN
```

REFERENCES

1. **React Documentation.** (2023). *Meta Platforms, Inc. Retrieved from https://react.dev/*
(Primary reference for building all user interface components, managing state with Hooks, and structuring the single-page application.)

2. **Node.js Documentation.** (2023). *Node.js Foundation. Retrieved from https://nodejs.org/en/docs/*
(Consulted for the JavaScript runtime environment, core modules like http, and for setting up the backend signaling server.)

3. **WebRTC API Documentation**. (2023). MDN Web Docs. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API *(The foundational resource for understanding the core peer-to-peer communication APIs , including RTCPeerConnection, MediaStream, and the signaling lifecycle.)*

4. **Socket.IO Documentation.** (2023). *Socket.IO Team. Retrieved from https://socket.io/docs/*
(The definitive guide for implementing the real-time, event-based signaling logic on both the Node.js server and the React client.)

5. **Vite Documentation.** (2023). *Evan You & Vite Contributors. Retrieved from https://vitejs.dev/*
(Used extensively for setting up the modern frontend development environment, configuring the dev server, and bundling the application.

6. **Tailwind CSS Documentation.** (2023). *Tailwind Labs Inc. Retrieved from https://tailwindcss.com/docs/ (The primary resource for all styling, utilizing its utility-first classes to rapidly build the responsive and modern user interface for the application.)*

BIBLIOGRAPHY

1. ***MDN Web Docs. (n.d.). WebRTC API. Mozilla.***
   Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (An essential resource for the low-level technical details of the WebRTC protocol. This was the primary reference for understanding the RTCPeerConnection API, the signaling state machine, and the methods for capturing and attaching media streams.)

2. ***Fowler, M. (2014, March 25). Microservices. martinfowler.com.***
   Retrieved from https://martinfowler.com/articles/microservices.html
   (A seminal article defining the principles of microservices architecture. Although Gatherly's backend is a single service, this resource provided the foundational concept of separating the signaling logic into a distinct, single-responsibility service, decoupled from the frontend clients.)

3. ***Socket.IO Documentation. (n.d.). Socket.IO.***
   Retrieved from https://socket.io/docs/
   (The official documentation for the WebSocket library used for signaling. This was the definitive guide for implementing the event-based communication between the server and clients, including room management, broadcasting events, and handling connections.)

4. ***Google Developers. (n.d.). Real-time communication with WebRTC. web.dev.***
   Retrieved from https://web.dev/articles/webrtc-basics
   (Consulted for practical examples and simplified explanations of the core WebRTC concepts, particularly the signaling process and the roles of STUN/TURN servers in establishing a peer-to-peer connection.)

5. ***Stack Overflow. (Community resource).***
   Various threads related to "webrtc react hooks", "socket.io connection", "vite failed to resolve import", "webrtc multiple peers".
   (An indispensable resource for troubleshooting specific technical errors, React Hook implementation patterns for managing media streams, Vite configuration issues, and common hurdles encountered when scaling from a two-peer to a multi-peer connection.)

6. ***Grigorik, I. (2013). High Performance Browser Networking. O'Reilly Media.***
   Retrieved from https://hpbn.co/webrtc/
   (An in-depth resource that provides a deep understanding of the networking principles behind WebRTC, including the roles of SDP, ICE, STUN, and TURN in navigating network address translators (NATs) to establish reliable peer connections.)