Convoflow

Connect, Communicate, Collaborate

Acknowledgment

I would like to extend my heartfelt gratitude to "**I'm Beside You**" for providing me with the opportunity to undertake this assignment. The task of implementing the messaging service prototype allowed me to leverage my skills and knowledge in web development while also challenging me to think critically about system design and architecture.

Despite the limited time frame and the concurrent mid-semester examinations, I strived to meet all the requirements of the assignment to the best of my ability. This experience not only enhanced my technical competencies but also reinforced my commitment to delivering quality work under pressure.

Thank you once again for this opportunity.

Contents:

Introduction

Convoflow is a messaging service prototype designed to deliver a seamless real-time communication experience. It utilizes **Socket.io** for instant messaging and incorporates **WebRTC** for video calls, with user data secured through encryption in **MongoDB**.

Tech Stack:

- Client: React JS
- Server: Node JS, Express JS
- Database: MongoDB

Key Features:

- Sleek and user-friendly interface
- Comprehensive user registration and authentication
- Real-time messaging with typing indicators
- Direct text communication between users
- One-on-one messaging
- User search functionality
- Group chat creation and management
- Real-time message updates
- In-app notifications
- Group user management (add/remove)
- User profile viewing
- Secure login with password encryption
- Persistent chat history
- Fully responsive design
- AI-powered chatbot
- Emoji support
- video calling feature
- Timestamp for message delivery

Project planning:

Task	Start Date	End Date	Duration
Project Planning	Sep 13	Sep 13	1 day
Define Requirements & Scope	Sep 13	Sep 13	1 day
Design Phase	Sep 14	Sep 15	2 days
UI/UX Design	Sep 14	Sep 15	2 days
System Design Document	Sep 14	Sep 15	2 days
Backend Development	Sep 16	Sep 17	2 days
Set up Server Environment	Sep 16	Sep 16	1 day
Develop REST APIs	Sep 16	Sep 17	2 days
Database Setup and Integration	Sep 16	Sep 17	2 days
Frontend Development	Sep 18	Sep 19	2 days
Implement Core Features	Sep 18	Sep 19	2 days
Implement Optional Features	Sep 18	Sep 19	2 days
Testing & Documentation	Sep 20	Sep 20	1 day
Test Functionality	Sep 20	Sep 20	1 day
Finalize Documentation	Sep 20	Sep 20	1 day

System Requirements Specification (SRS)

1. Introduction

1.1 Purpose

The purpose of this document is to provide a comprehensive specification for the development of the Convoflow messaging service prototype. This prototype aims to demonstrate core messaging functionalities including real-time communication, user authentication, and group chat features.

1.2 Scope

The Convoflow prototype will provide a robust messaging platform with the following features:

- User registration and authentication
- Real-time text messaging
- Group chat functionality
- Optional AI-powered chatbot and video/audio calling

2. Functional Requirements

2.1 User Registration and Authentication

- **Registration:** Users must be able to register with a unique username, email, and password.
- **Login:** Users must be able to log in with their credentials.
- **Password Security:** Passwords must be hashed using bcrypt before storage.
- **Session Management:** Implement secure session management for user authentication.

2.2 Messaging

- **Text Messaging:** Users should be able to send and receive text messages.
- **Real-Time Updates:** Messages should be delivered in real-time using Socket.io.
- **Group Chats:** Users should be able to create, join, and manage group chats.
- **Message History:** Chat history should be stored and retrievable.

2.3 Group Chat Functionality

- **Create Groups:** Users should be able to create and name groups.

- **Add/Remove Members:** Group admins should be able to add or remove members.
- **Group Messaging:** Support sending and receiving messages in groups.
- **View Profiles:** Users should be able to view the profiles of group members.

2.4 Optional Features

- **AI Chatbot:** An AI-powered chatbot should assist users with basic queries and interactions.
- **Video/Audio Calling:** Provide video/audio calling feature.

3. Design Constraints

3.1 Design Practices

- **Atomic Design:** Implement Atomic Design principles for modular and scalable UI components.

3.2 Technology Stack

- **Frontend:** React JS
- **Backend:** Node JS with Express JS
- **Database:** MongoDB

4. Non-Functional Requirements

4.1 Performance Requirements

- **Latency:**
    - End-to-End Latency: The prototype aims for a maximum end-to-end latency of 200 milliseconds for user interactions, including message sending and receiving.
    - Real-Time Communication: Real-time updates should occur within 100 milliseconds of the event to ensure a smooth user experience.
- **Availability:**
    - Uptime: The system should achieve a high availability target of 99.9%, translating to less than 8.76 hours of downtime annually.
    - Failover: Although not yet deployed, the design should consider failover mechanisms to ensure continuity in the event of a server failure.
- **Scalability:**
    - Horizontal Scalability: The prototype should be designed with horizontal scaling in mind to handle increased user load efficiently.

- Load Balancing: Utilize load balancing strategies to distribute incoming requests evenly across servers.

4.2 Traffic Estimation

- **Request Handling:**
  - Express.js: Designed to handle approximately 15,000 requests per second.
  - Basic HTTP Module: Node.js's HTTP module can handle up to 70,000 requests per second.

4.3 Storage Estimation

- **Database:** MongoDB
  - Document Size: Supports individual document sizes up to 16MB.
  - Nested Depth: Avoid excessive nesting (up to 100 levels) for optimal performance and manageability.
  - Data Storage: Use MongoDB for storing user profiles and chat history.

4.4 Security Requirements

- **Authentication and Authorization:**
  - Secure Authentication: Implement JWT (JSON Web Tokens) for secure user authentication.
  - Password Hashing: Use bcrypt.js to securely hash and store user passwords.

4.5 Usability Requirements

- **User Interface:**
  - Responsiveness: Ensure the UI is responsive across various devices, including desktops, tablets, and smartphones.

4.6 Maintainability

- **Code Quality:**
  - Documentation: Maintain comprehensive code documentation to facilitate future maintenance.
  - atomic Design: Follow atomic design principles to enhance maintainability and scalability.

5. Why I Chose These Technologies

React JS

- **Familiarity and Speed:** My experience with React allows for faster UI development. Chakra UI provides pre-designed components that enhance design efficiency.

Node JS with Express JS

- **Efficiency for Real-Time Communication:** Node.js is well-suited for real-time features. Express.js simplifies API development, enabling quick implementation of core functionalities.

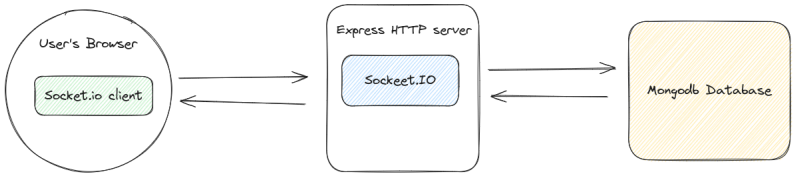
MongoDB

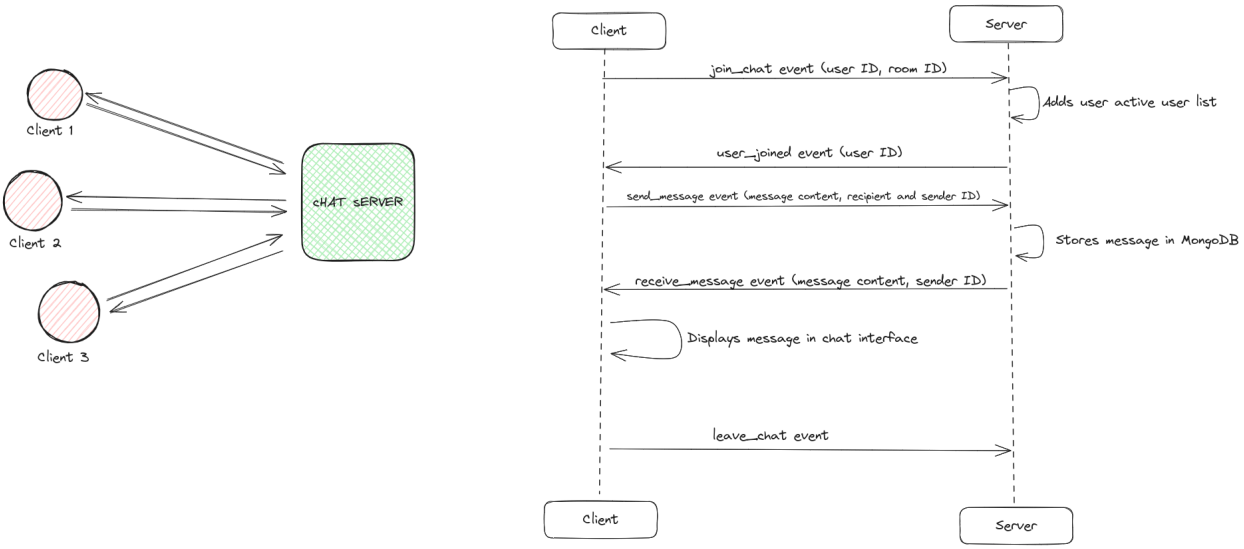
- **Flexibility and Scalability:** MongoDB's schema flexibility is ideal for a prototype where requirements may evolve. Its scalability supports potentially large volumes of chat data.

Time Constraints

- **Rapid Development:** With only a week to complete the prototype, I opted for familiar technologies to minimize learning curves and meet tight deadlines.

Architecture Overview



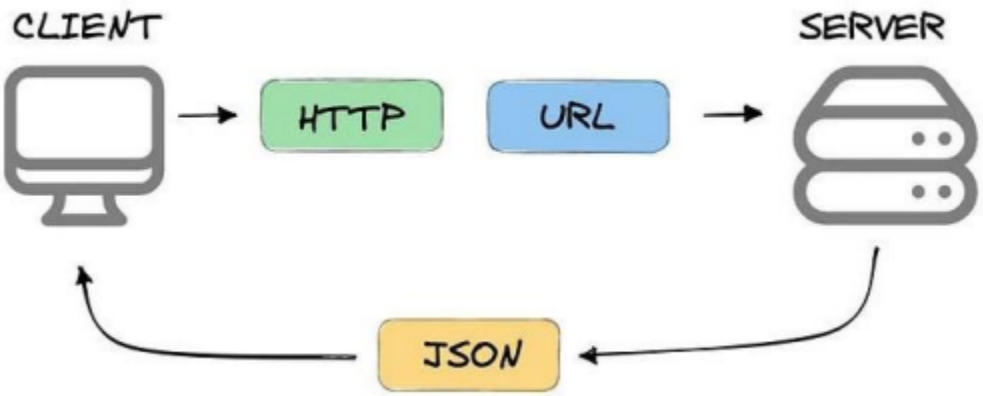


1. Client-Side (Frontend)

- **Framework: React JS** React JS is used for building dynamic and interactive user interfaces. It handles the efficient rendering of components based on state changes and updates the UI without reloading the page.
- **UI Libraries: Chakra UI and Material UI** Chakra UI and Material UI provide pre-built, customizable components for the user interface. They help in developing a responsive, modern UI with ease.
- **Styling: Emotion** Emotion is used for writing and managing CSS styles in JS, enabling scoped and dynamic styles that fit within the design system.
- **State Management: React Hooks** React's `useState` and `useEffect` hooks are used to manage local component state and lifecycle events. This keeps the UI reactive to changes like new messages or incoming calls.
- **Routing: React Router** React Router handles navigation between different views, such as chat interfaces, video calls, and user profiles.

- **Real-Time Communication: Socket.io** Socket.io enables real-time communication between the client and server for instant messaging and live updates, making interactions seamless without page reloads.
- **Video Calling: Simple-Peer** Simple-Peer is used to establish peer-to-peer WebRTC connections for video calls, allowing users to communicate directly without routing media through a central server.
- **User Interaction** The frontend includes responsive UI components such as chat interfaces, input fields, video call screens, and user search features, ensuring a smooth user experience.

2. Server-Side (Backend)



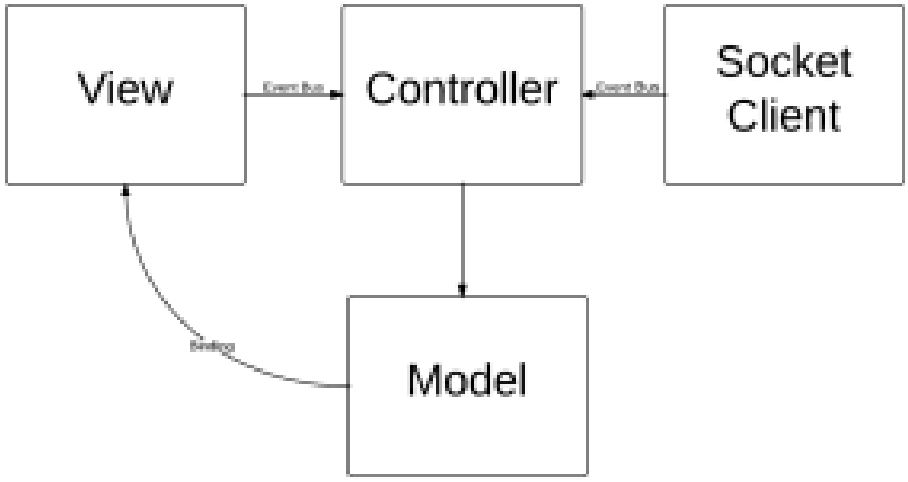
- **Framework: Node.js with Express.js** The server is built using Node.js and Express.js, providing a robust backend that handles API requests, routing, and middleware for efficient web service operations.
- **Database: MongoDB with Mongoose** MongoDB serves as the primary database, storing user profiles, chat histories, and message data. Mongoose is used as an ODM for schema-based modeling and managing relationships between data.
- **Authentication: JSON Web Tokens (JWT)** JWT is used for secure authentication, ensuring that only authorized users can access protected routes after logging in.
- **Middleware:** Express middleware handles authentication, error handling, and logging. Custom middleware is used to streamline processes like validating tokens and handling request errors.
- **Real-Time Communication: Socket.io** The backend also integrates Socket.io to manage real-time messaging and signaling for video calls.

Events like "new message" or "typing indicator" are handled through Socket.io channels.

3. **Database Layer**
    - **Structure:**
        - **Users**: Stores user credentials, profile data, and preferences.
        - **Messages**: Contains chat messages with sender/receiver information and timestamps.
        - **Chats**: Manages chat groups for both individual and group conversations.
    - **Models:** Mongoose schemas define data validation and relationships, ensuring that stored data is structured and consistent.
4. **Video Call Feature**
    - **Peer Connections:** Simple-Peer establishes peer-to-peer video call connections between users without routing the media through the server, ensuring low latency and direct communication.
    - **Socket Events:**
        - `callUser`: Initiates a video call by sending a signaling request to the target user.
        - `answerCall`: Responds to the signaling request, accepting the call.
        - `callAccepted`: Confirms that both users have connected successfully, establishing the peer-to-peer link.
5. **User Flow**
    - **User Registration/Login:** Users register or log in, receiving a JWT token to authorize subsequent requests.
    - **Chat Functionality:** Users can send and receive real-time messages, with the chat interface dynamically updating via Socket.io.
    - **Video Calling:** Users initiate video calls by entering the target user's ID. Simple-Peer handles the WebRTC connection, and Socket.io manages call events.
6. **Deployment Considerations**
    - **Hosting:** The backend can be hosted on services like Heroku or DigitalOcean, while the frontend can be deployed on platforms such as Vercel or Netlify for optimized performance.
    - **Environment Variables:** Sensitive data like JWT secrets and database credentials are managed through `.env` files using the dotenv library,

ensuring that credentials are secure and flexible for different environments.

Model-View-Controller (MVC) architecture



In developing Convoflow, I implemented the **Model-View-Controller (MVC) architecture**, which helped me maintain a clean separation between different layers of the application. This was crucial given the complexity of the messaging platform, where I needed to handle user interactions, real-time communication, and data persistence.

- **Model**: MongoDB models, defined using Mongoose, allowed me to structure the application's data effectively. By separating the data management layer, I could ensure that user information, chat histories, and messages were stored and retrieved consistently. This made it easier to handle complex data relationships like one-on-one and group chats.
- **View**: The frontend, built with React, served as the user interface, allowing users to interact with the system. By keeping the view layer distinct, I could focus on optimizing user experience, adding features like responsive design, real-time updates through Socket.io, and smooth navigation using React Router.
- **Controller**: The controllers in the backend, using Express.js, handled business logic and served as the bridge between the model and the view. This made it easy to manage API routes, authenticate users using JSON Web Tokens, and control real-time events like message sending or video call signals.

Database Design

Overview

The Convoflow messaging service utilizes MongoDB to manage users, chats, and messages. This section outlines the key read and write operations supported by the database, providing a clear understanding of how data flows within the application.

Read Operations

1. **Retrieve Messages Between User A and User B**
    - **Purpose**: Fetch all messages exchanged between two users in a one-on-one chat.
2. **Retrieve All Messages in Group G**
    - **Purpose**: Fetch all messages sent in a specific group chat.
3. **Find All Member User IDs in Group G**
    - **Purpose**: Retrieve all users that are members of a specific group chat.
4. **Retrieve User ID by Username or Email**
    - **Purpose**: Find a user ID based on their username or email.

Write Operations

1. **Save a Message Between User A and User B**
    - **Purpose**: Save a new message sent by User A to User B in their chat.
2. **Save a New Message by User A in Group G**
    - **Purpose**: Save a message sent by User A in a group chat.
3. **Add User A to Group G**
    - **Purpose**: Add User A to a specific group chat.
4. **Delete User A from Group G**
    - **Purpose**: Remove User A from a specific group chat.

Users Collection

Field	Type	Description
_id	ObjectId	Unique identifier for each user
name	String	The user's name
email	String	The user's email address (unique)
password	String	The encrypted password for authentication
pic	String	URL to the user's profile picture (default provided)
createdAt	Date	Timestamp for when the user was created
updatedAt	Date	Timestamp for when the user was last updated

Chats Collection

Field	Type	Description
_id	ObjectId	Unique identifier for each chat
chatName	String	Name of the chat (applicable for group chats)
isGroupChat	Boolean	Indicates if the chat is a group chat
users	Array	List of user IDs participating in the chat
latestMessage	ObjectId	Reference to the most recent message
groupAdmin	ObjectId	Reference to the user who is the admin of the group chat
createdAt	Date	Timestamp for when the chat was created
updatedAt	Date	Timestamp for when the chat was last updated

Messages Collection

Field	Type	Description
_id	ObjectId	Unique identifier for each message
sender	ObjectId	Reference to the user who sent the message
content	String	The text content of the message
chat	ObjectId	Reference to the chat this message belongs to
timestamp	Date	Date and time the message was sent

API Services

1. **User Services**:
    - **Registration**: Handles user registration by creating a new user in the database.
    - **Login**: Authenticates users and generates a JWT for session management.
    - **Profile Management**: Allows users to retrieve and update their profile information.
2. **Chat Services**:
    - **Create Chat**: Initiates a new chat session between users.
    - **Get Chats**: Retrieves a list of chats for a user.
    - **Delete Chat**: Removes a specific chat from the user's list.
3. **Message Services**:
    - **Send Message**: Sends a message to a specific chat.
    - **Get Messages**: Retrieves messages from a specific chat.
    - **Delete Message**: Deletes a specific message from a chat.

File References

- **routes/userRoutes.js**: Contains routes for user-related operations.
- **routes/chatRoutes.js**: Contains routes for chat-related operations.
- **routes/messageRoutes.js**: Contains routes for message-related operations.

Functionality Overview

- Each of these services is defined in the respective controller files:
    - **controllers/userControllers.js**: Implements user-related functionalities.
    - **controllers/chatControllers.js**: Implements chat-related functionalities.
    - **controllers/messageControllers.js**: Implements message-related functionalities.

Dependencies and Libraries Used

backend:

1. **Express**: I chose Express because it simplifies the creation of web applications and APIs in Node.js. Its routing capabilities and middleware integration enhance the server's functionality, allowing for clean and efficient code.
2. **Mongoose**: Mongoose is an ODM library for MongoDB and Node.js that provides a schema-based solution to model application data. I selected it for its ability to simplify interactions with MongoDB, making data management more intuitive and organized.
3. **jsonwebtoken (JWT)**: This library is crucial for user authentication. I opted for jsonwebtoken because it enables secure token generation and verification, ensuring that only authenticated users can access certain routes, thereby enhancing security.
4. **bcryptjs**: I use bcryptjs to hash user passwords securely. This library is essential for protecting sensitive user information by storing only hashed versions of passwords, making it significantly more difficult for attackers to retrieve them.
5. **dotenv**: Dotenv is a module that loads environment variables from a .env file into `process.env`. I chose this library to manage sensitive data like database connection strings and JWT secrets without hardcoding them into the codebase, which improves security and flexibility.
6. **socket.io**: I implemented socket.io for real-time, bidirectional communication between clients and servers. This library is key for enabling real-time messaging and typing indicators in Convoflow, significantly enhancing the user experience.
7. **express-async-handler**: This middleware simplifies error handling in asynchronous route handlers. I selected it to reduce boilerplate code and improve error management in my controllers, making the codebase cleaner and more maintainable.
8. **PeerJS**: I chose PeerJS for setting up WebRTC connections for video calling features. Although the video calls are not fully functional yet, it provides the necessary infrastructure for peer-to-peer connections, which is essential for future development

Frontend

1. **@chakra-ui/react:** I selected Chakra UI for its ease of creating responsive, accessible, and customizable components. It simplifies the UI development process by providing pre-designed components that align with modern design systems.
2. **@chakra-ui/icons:** This library offers a collection of icons that integrate seamlessly with Chakra UI. It enhances the visual elements of Convoflow, like buttons and navigation menus.
3. **@emotion/react & @emotion/styled:** Emotion provides powerful styling capabilities in JavaScript. I chose these to allow for custom, dynamic styles while maintaining the flexibility of Chakra UI's theme system.
4. **@mui/material:** I added Material UI to complement Chakra UI with additional components, like dialogs or grids, to provide a more robust UI experience.
5. **@mui/icons-material:** This library provides Material Design icons, which I use to enrich the visual design and ensure consistency with Material UI components.
6. **axios:** For handling HTTP requests, I chose Axios due to its simplicity and versatility. It allows me to make API calls to the backend easily, helping to manage data and user interactions in Convoflow.
7. **emoji-picker-react:** This was implemented to offer users the ability to add emojis to their messages, making conversations more expressive and enjoyable.
8. **framer-motion:** I used Framer Motion for adding smooth animations to Convoflow's interface. It enhances the user experience with visually appealing transitions and effects.
9. **peerjs:** PeerJS simplifies the implementation of WebRTC for video calls. I chose this to establish peer-to-peer video connections in Convoflow's video chat feature, setting the foundation for future improvements.
10. **react-copy-to-clipboard:** I incorporated this to streamline copying text, such as user IDs, making it easy for users to initiate video calls by sharing their IDs.
11. **react-lottie:** This library enables me to integrate fun Lottie animations into Convoflow, enhancing user engagement through animated feedback, such as loading indicators.
12. **react-notification-badge:** I added this to display notification badges for new messages or events in Convoflow, helping users stay updated with minimal distractions.
13. **react-redux:** I use Redux to manage global state across the app, ensuring that data like user information and messages are consistent throughout the interface.
14. **react-router-dom:** This is used for routing between different pages of Convoflow, such as the login, chat, and video call screens, allowing for smooth navigation and a better user flow.

15. **socket.io-client:** The Socket.io client enables real-time communication with the backend, providing instant messaging and live notifications that are essential for an interactive chat experience.
16. **simple-peer:** This is used alongside PeerJS to manage WebRTC connections for video calling, handling the peer-to-peer video streams in a straightforward way.
17. **Connect.Chat:** I integrated Connect.Chat to incorporate a chatbot into Convoflow, enhancing user interactions by providing automated responses and assistance.

Risk Analysis

1. Time Constraints

- **Risk**: Incomplete features due to tight schedule.
- **Mitigation**: Focused on core functionalities first.

2. Technical Challenges

- **Risk**: Unforeseen bugs or integration issues.
- **Mitigation**: Set aside time for testing and debugging.

3. Real-Time Communication

- **Risk**: Difficulties with Socket.io and PeerJS.
- **Mitigation**: Implemented a basic version of features before enhancements.

4. Security Vulnerabilities

- **Risk**: User data mishandling.
- **Mitigation**: Used secure libraries for authentication and data protection.

5. User Experience

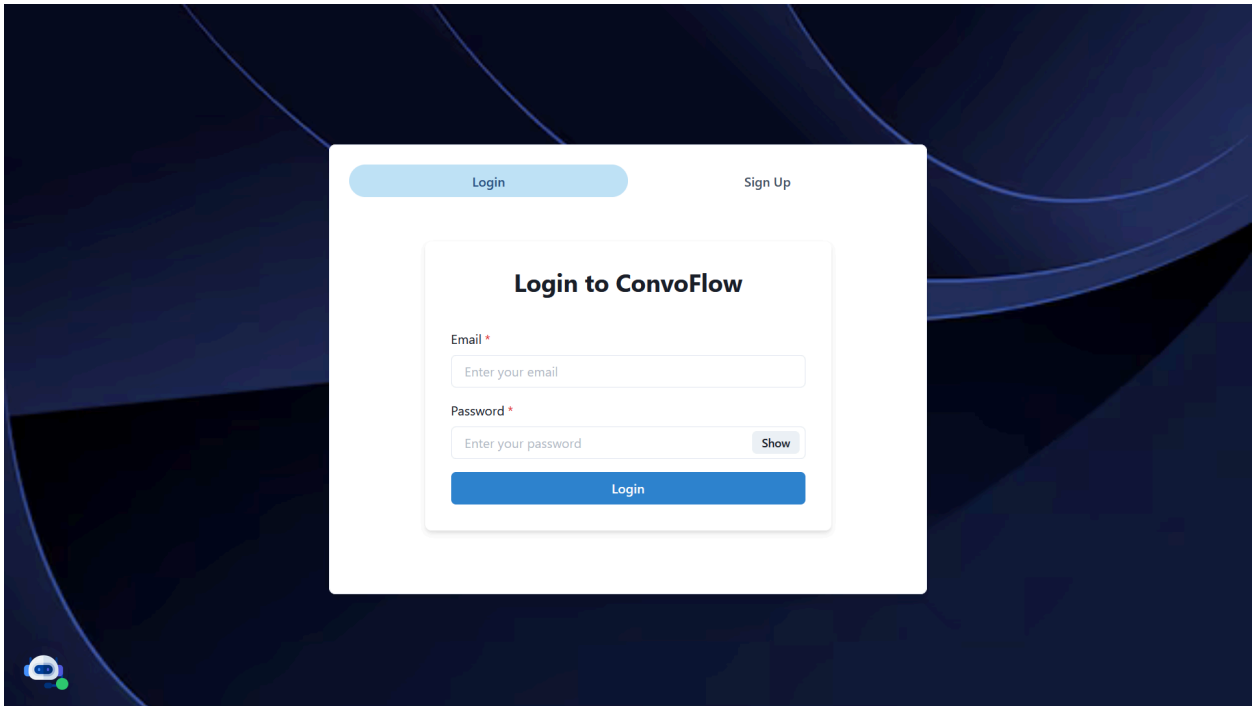
- **Risk**: Confusing interface.
- **Mitigation**: Designed a clear, intuitive UI with guidance.

6. Testing and Validation

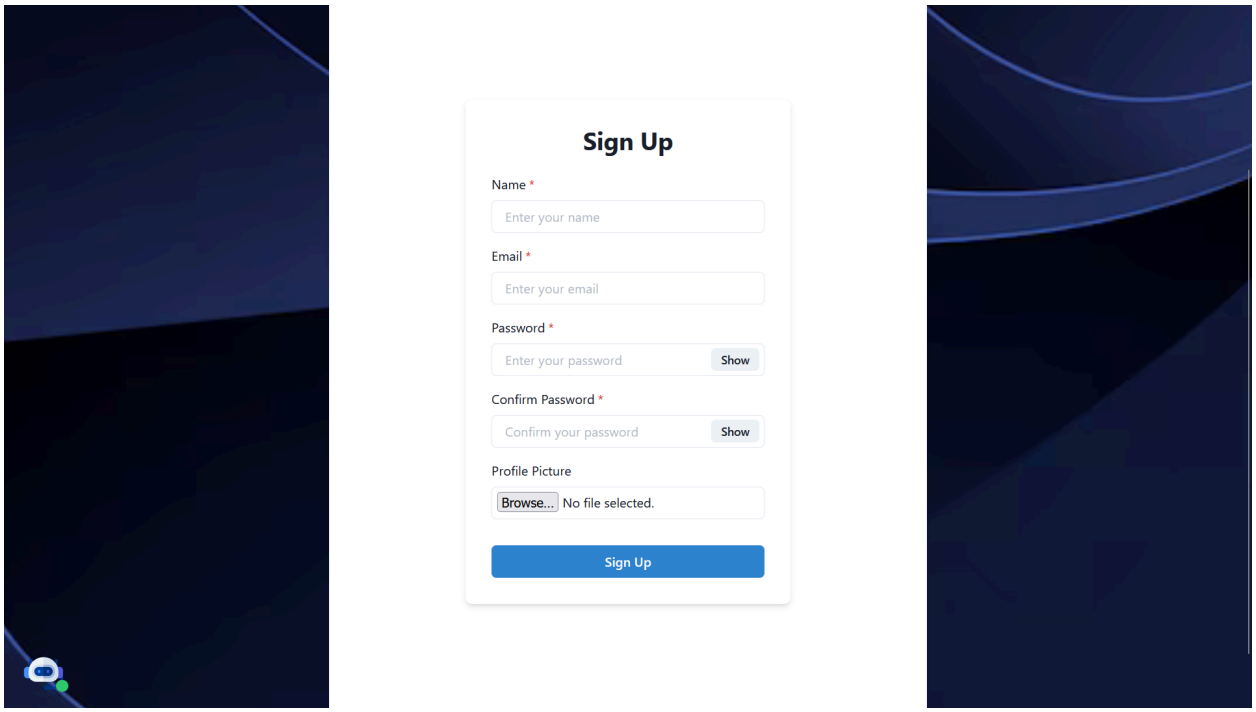
- **Risk**: Potential undiscovered bugs.
- **Mitigation**: Conducted thorough testing on essential features.

Testing and Verification

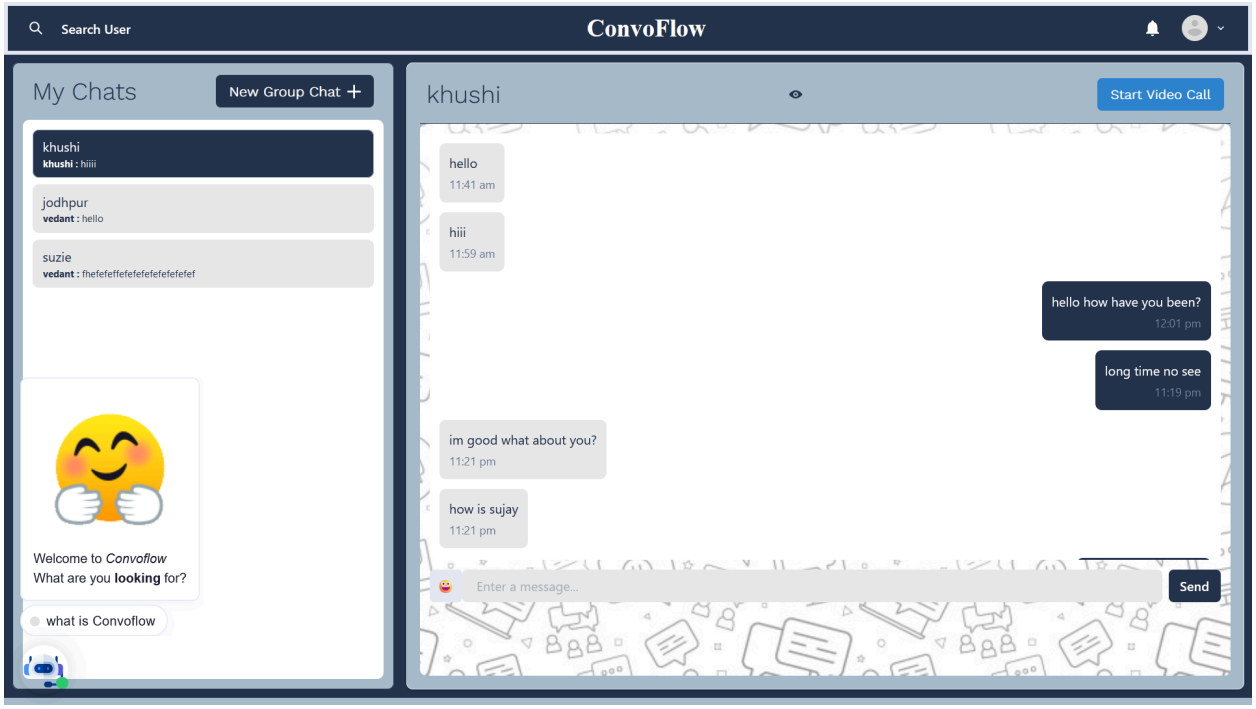
login:


Login
Sign Up
Login to ConvoFlow
Email *
Enter your email
Password *
Enter your password
Show
Login

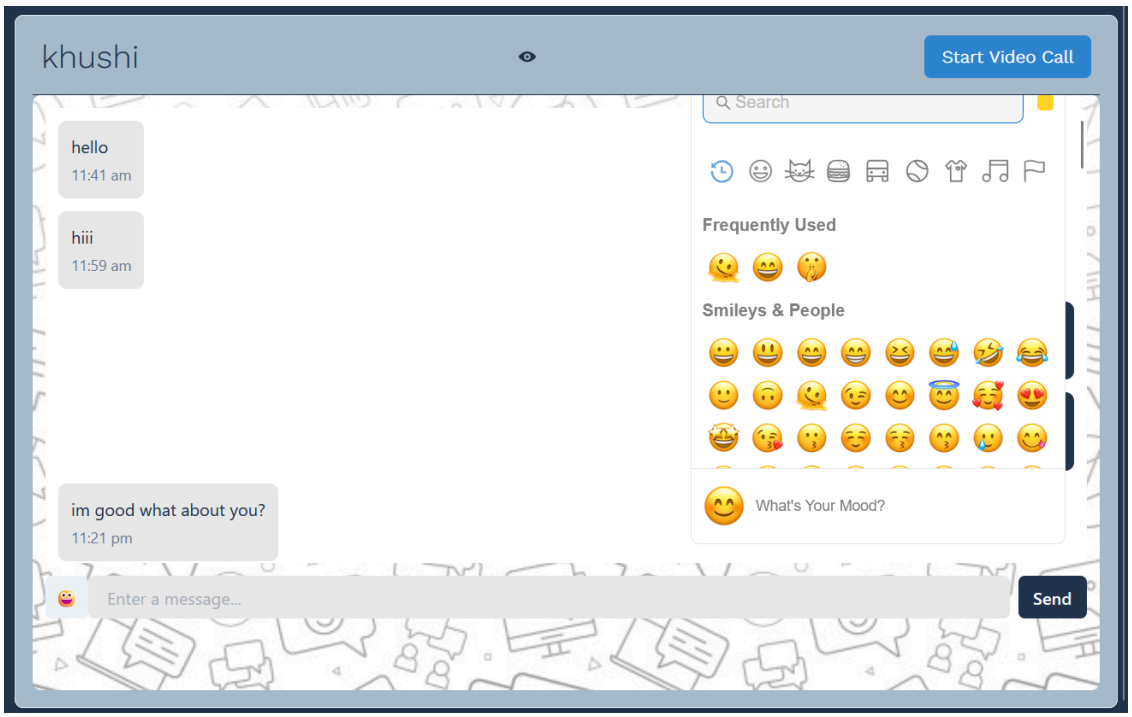
Signup:


Sign Up
Name *
Enter your name
Email *
Enter your email
Password *
Enter your password
Show
Confirm Password *
Confirm your password
Show
Profile Picture
Browse...
No file selected.
Sign Up

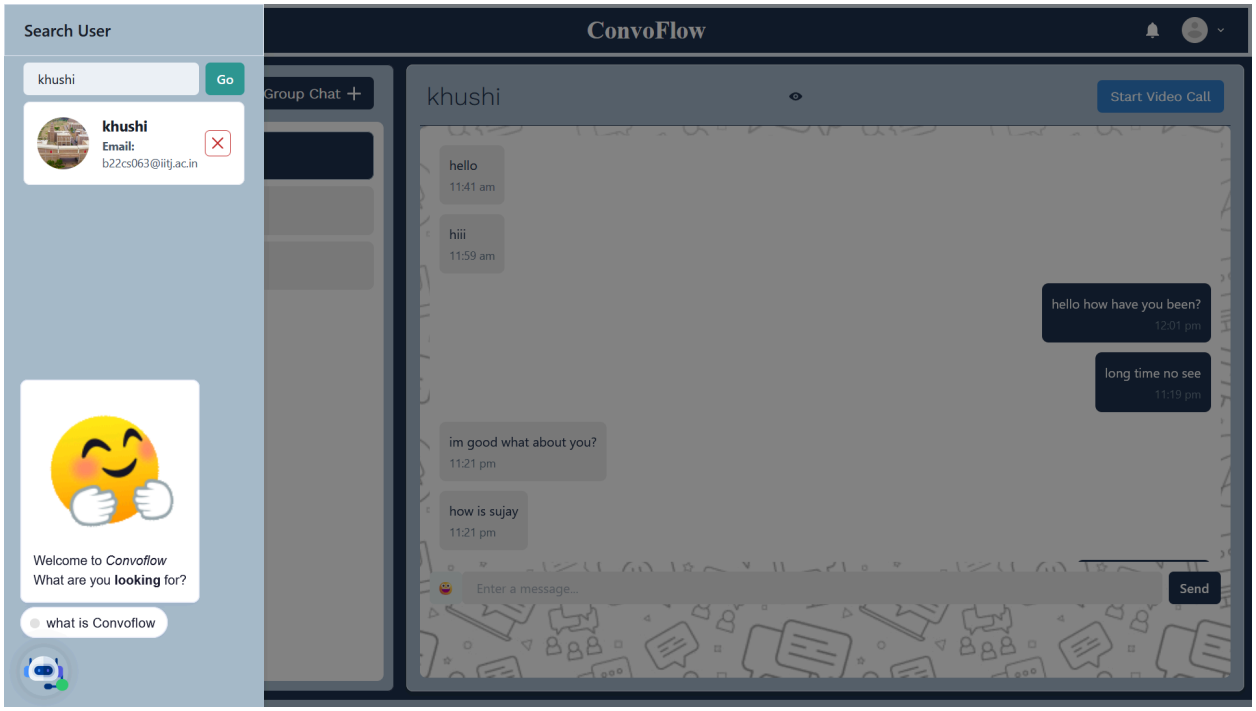
Chat:



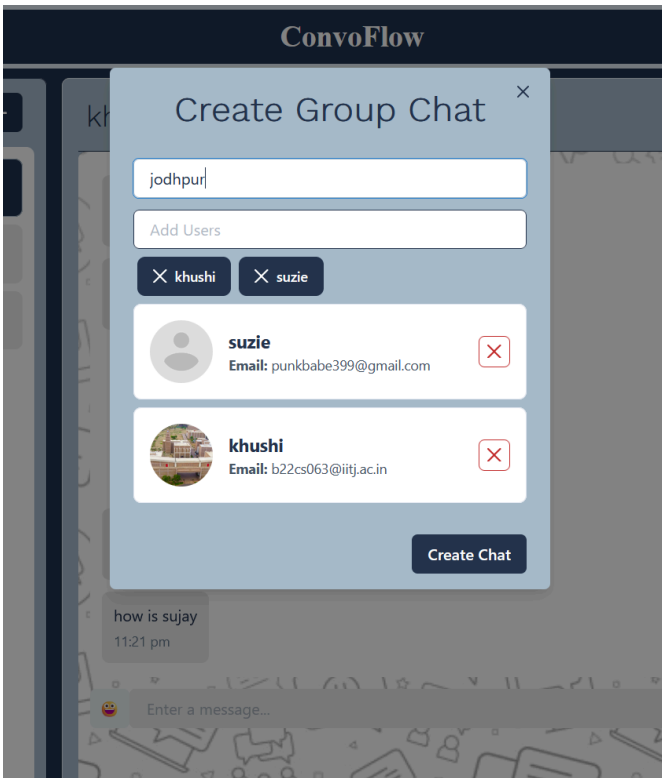
One to one chat:



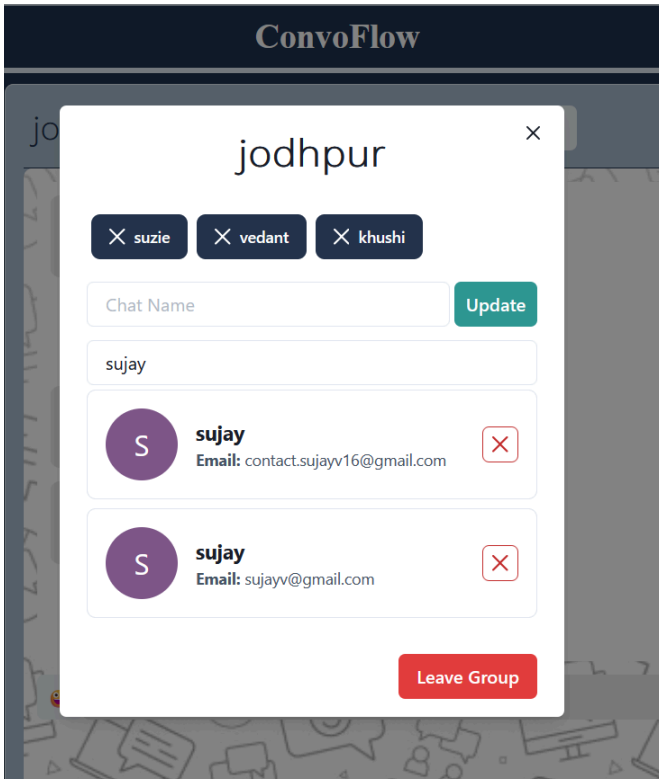
Search users:



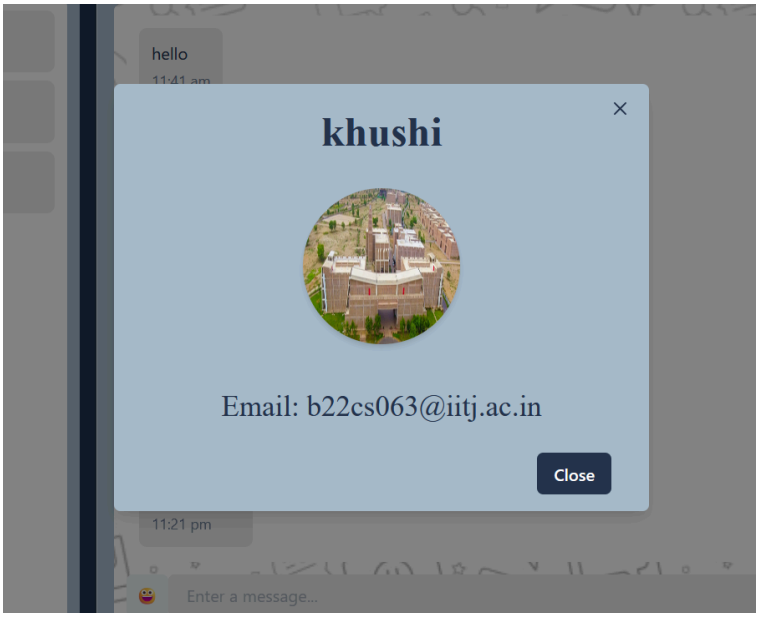
Create group:



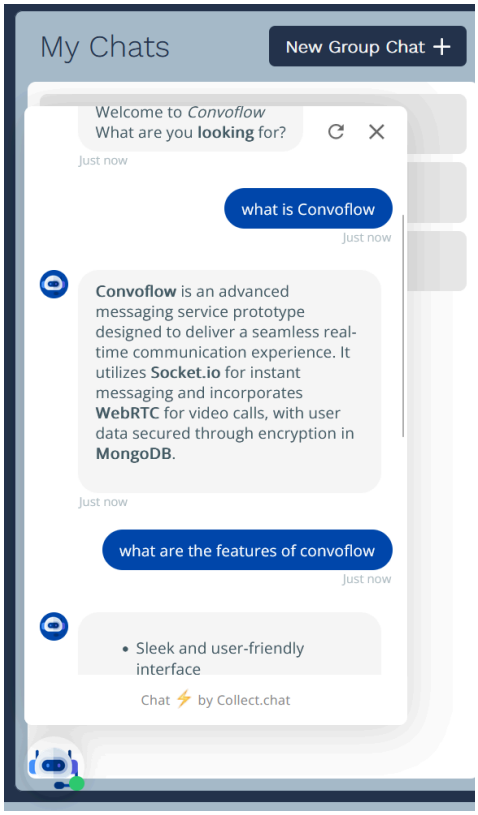
Modify group:



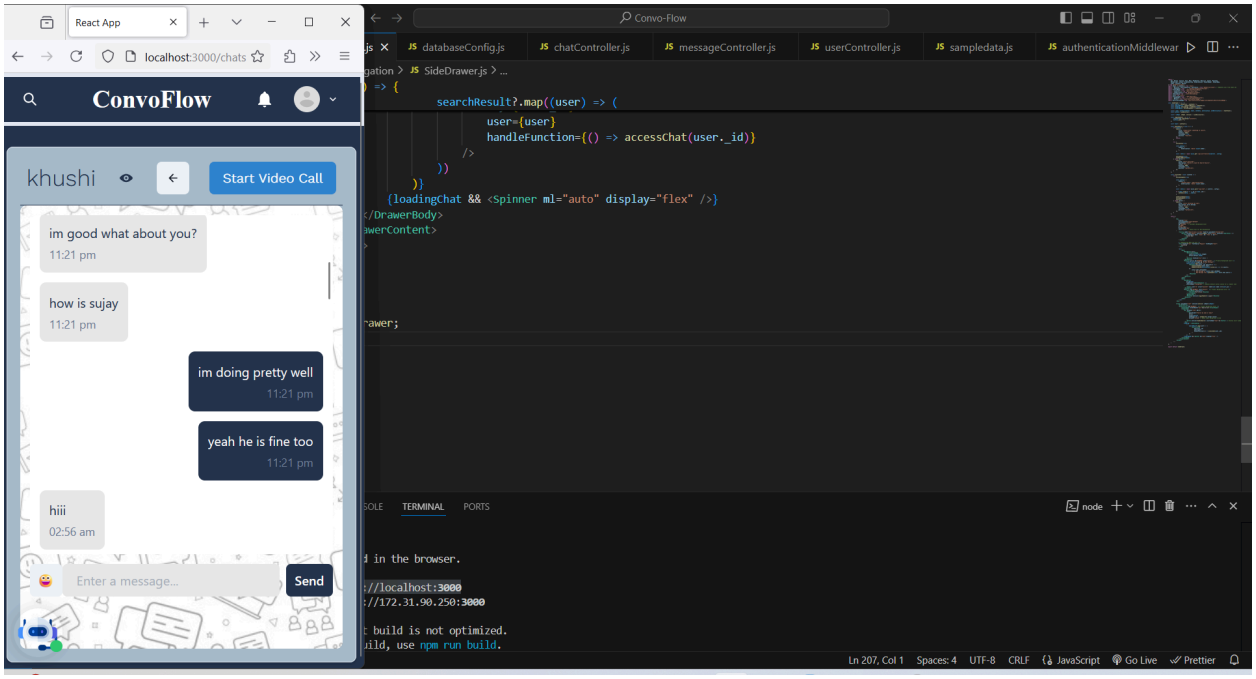
User profiles:

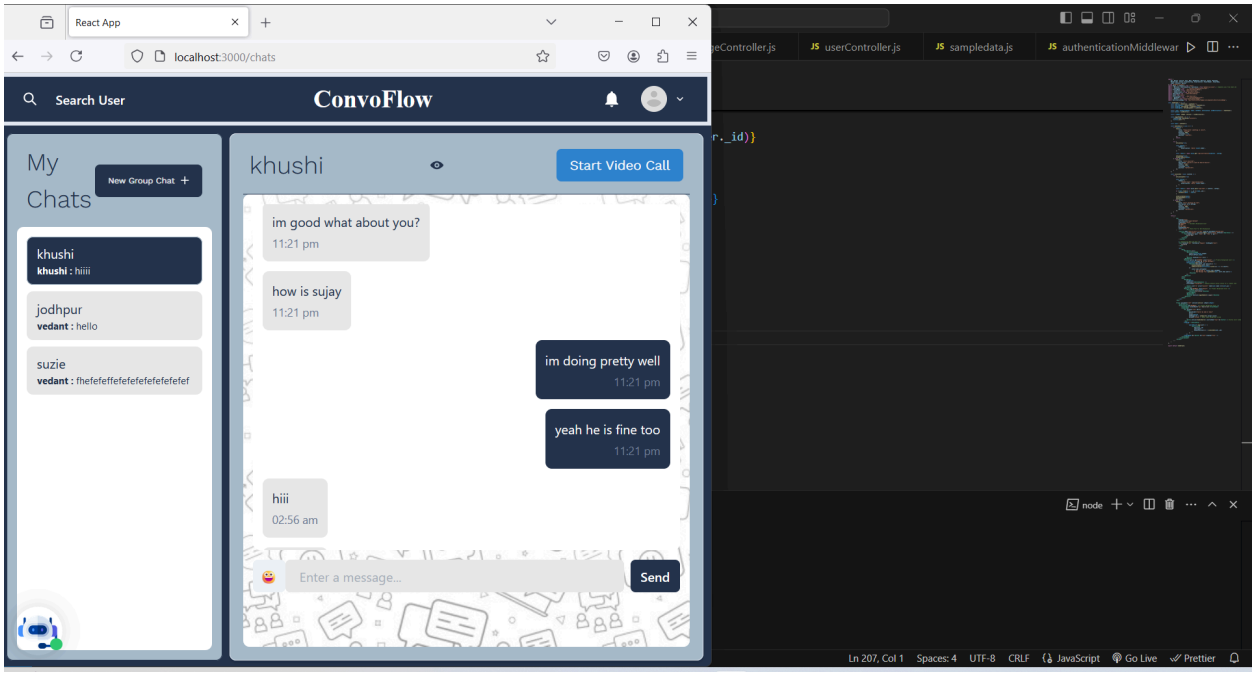


Chatbot:

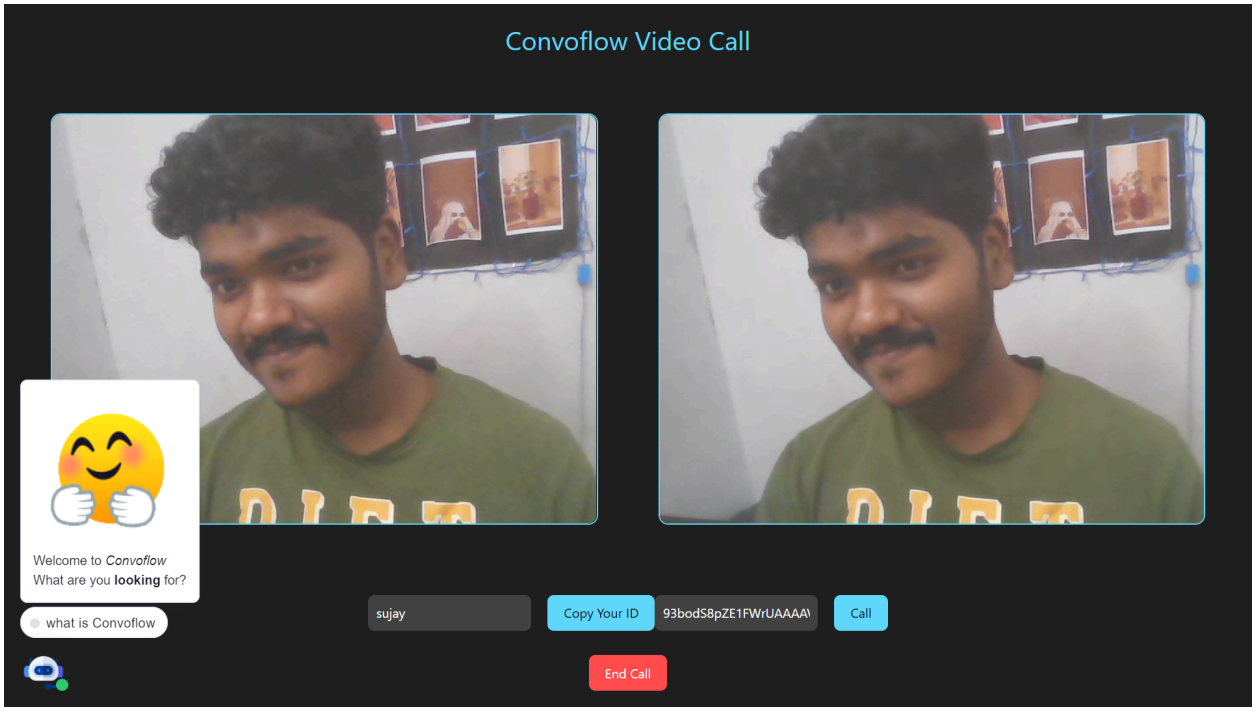


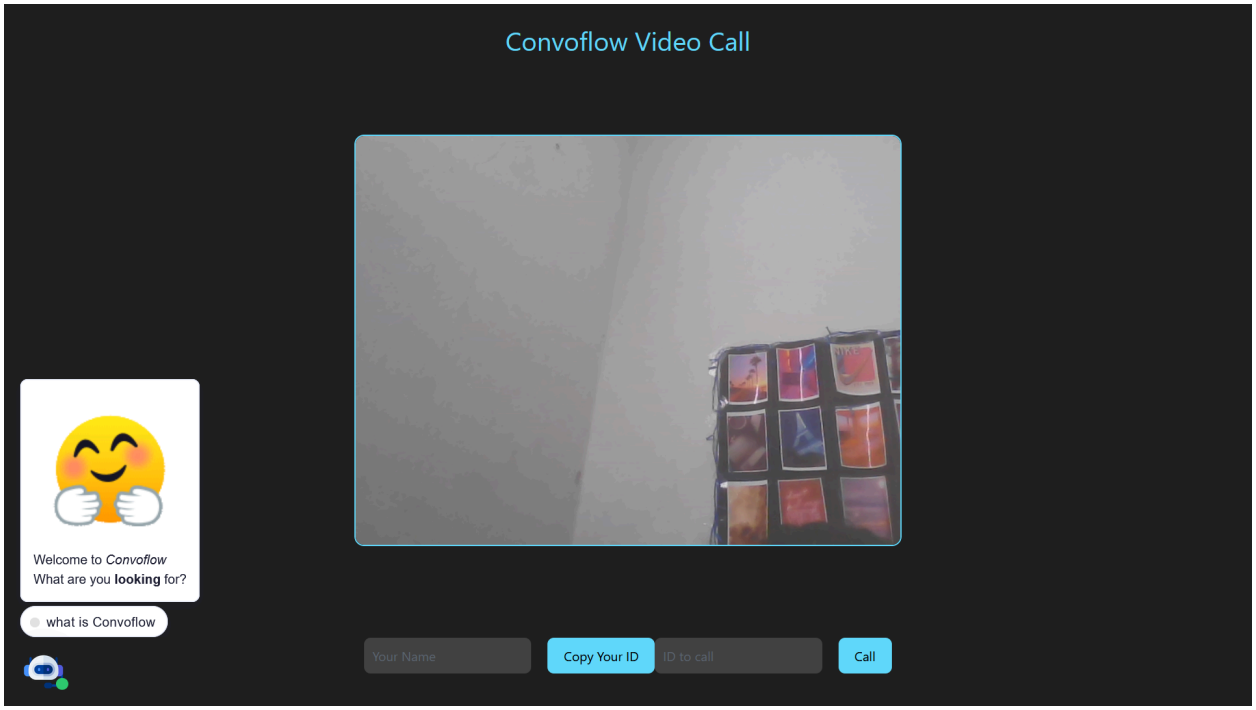
Responsive:




React App
localhost:3000/chats
Search User
ConvoFlow
My Chats
New Group Chat +
khushi
khushi : hiiii
jodhpur
vedant : hello
suzie
vedant : fhefeffefefefefefefefef
khushi
Start Video Call
im good what about you?
11:21 pm
how is sujay
11:21 pm
im doing pretty well
11:21 pm
yeah he is fine too
11:21 pm
hiii
02:56 am
Enter a message...
Send
Ln 207, Col 1
Spaces: 4
UTF-8
CRLF
JavaScript
Go Live
Prettier

Video call:

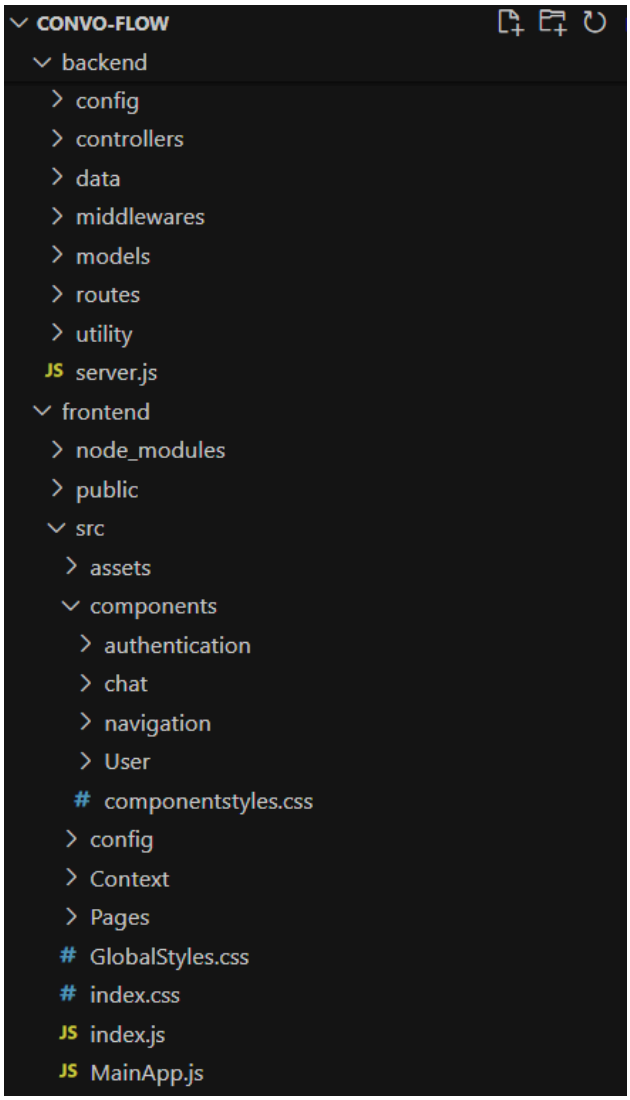

Convoflow Video Call
Welcome to Convoflow
What are you looking for?
what is Convoflow
sujay
Copy Your ID
93bodS8pZE1FWrUAAAAV
Call
End Call



Inspection documentation

Feature/Functionality	Yes/No	Comments
User registration implemented	Yes	Users can register with email and password.
User authentication (login/logout)	Yes	JWT-based authentication is functional.
Real-time messaging	Yes	Integrated with Socket.io for real-time updates.
One-on-one chat functionality	Yes	Users can initiate private chats.
Group chat functionality	Yes	Multiple users can participate in group chats.
Typing indicators for messages	Yes	Users see when others are typing.
Notifications for new messages	Yes	Users receive alerts for incoming messages.
User search feature	Yes	Users can search for other users.
Responsive design	Yes	UI adapts well to different screen sizes.
AI-powered chatbot integration	Yes	Chatbot feature is functional for FAQs.
Emoji support for messages	Yes	Users can send and receive emojis in chats.
Timestamps for messages	Yes	Each message shows the time it was sent.
Video calling functionality (ID entry required)	Yes	Functional, but requires user IDs for connection.

Atomic design:



Atoms: I started with the most basic elements, like buttons, input fields, and icons. Each atom is designed to be reusable throughout the application.
Molecules: I combined atoms to form simple components, such as form groups or chat bubbles. This allows me to build more complex UI elements while maintaining consistency.
Organisms: Next, I assembled molecules into larger components that represent distinct sections of the application, like the chat interface or user profile section. This level focuses on the functional aspects of the UI.
Templates: I created page layouts using organisms. Templates define the overall structure and arrangement of components without being tied to specific content.
Pages: Finally, I populated templates with actual content to create the final views. This ensures that the UI is not only functional but also visually cohesive.

Installation Guide

To set up and run the Convoflow messaging service prototype locally, please follow these steps:

Prerequisites

Ensure you have the following installed on your machine:

- **Node.js** (version 14 or higher)
- **npm** (Node package manager, comes with Node.js)
- **MongoDB** account (for database setup)

Environment Variables

Before running the project, you need to configure the following environment variables in a `.env` file located in the root directory of the project:

```
PORT=5000
MONGO_URI=<Your MongoDB Connection String>
JWT_SECRET=<Your JWT Secret Key>
NODE_ENV=production
```

- **PORT**: The port on which your server will run.
- **MONGO_URI**: This can be generated from your MongoDB project profile.
- **JWT_SECRET**: You can name this variable whatever you want; it's used for signing JWT tokens.
- **NODE_ENV**: Set this to `production`.

Steps to Run Locally

Clone the Project Open your terminal and run the following command to clone the Convoflow repository:

```
git clone https://github.com/sujayv16/ConvoFlow
```

1. **Navigate to the Project Directory** Change to the newly cloned project directory:

   ```
   cd ConvoFlow
   ```

2. **Install Backend Dependencies** Install the required dependencies for the backend:

```
npm install
```

3. **Install Frontend Dependencies** Move to the frontend directory and install its dependencies:

```
cd frontend/

npm install
```

4. **Start the Application** Return to the main project directory:

```
cd ..
```

Then, start both the client and server simultaneously:

```
npm run start
```

5. **Additional Information**

- After running the above command, your application should be up and running, accessible via `http://localhost:5000`.
- Make sure your MongoDB server is running for the application to function correctly.