Mini Project for ROSP Lab

“Real-time Chat App”

(SEMESTER VII)

submitted in
partial fulfilment of requirement for the award of degree of

Bachelor of Engineering
In
Information Technology

By

Name of Student	Roll No.
Vinit Salvi	**56**
Prathamesh Shirdhankar	**60**
Rushikesh Surve	**61**

Under guidance of

Prof. Onkar Dike

Department of Information Technology
Finolex Academy of Management and Technology, Ratnagiri

ACADEMIC YEAR 2024-25

Hope Foundation's
Finolex Academy of Management and Technology
P60-, P60-1, MIDC, Mirjole, Ratnagiri, Maharashtra, Pin 415639
Department of Information Technology

CERTIFICATE

The Mini Project titled **"Real-time Chat App"** is submitted by **"Mr. Prathamesh Shirdhankar, Mr. Rushikesh Surve, Mr. Vinit Salvi"** in the completion of BE (Sem VII) Bachelor in Information Technology, has been carried out under my supervision at the Department of Information Technology at Finolex Academy of Management and Technology, Ratnagiri. The work is comprehensive, complete and fit for evaluation.

Prof. Onkar Dike
Assistant Professor

Dr. Vinayak Bharadi
Head of Department

External Examiner

Abstract

This project report presents the development of a **Real-Time Web-Based Chat Application** using the MERN (MongoDB, Express, React, Node.js) stack and Socket.IO for real-time communication. The primary objective of this project was to explore and apply open-source technologies to build a functional, user-friendly, and scalable web application that allows real-time 1-to-1 chatting between users.

The chat app features essential functionality such as **user registration, login, and secure authentication using JWT**. Each user's email is utilized as a unique identifier for the chat, ensuring message delivery between the correct users. The application leverages **Socket.IO** for **instant message transmission**, ensuring smooth, real-time communication, and **MongoDB** for storing user credentials and message history, allowing persistent conversations.

While the current project is a simplified version with limited features, it sets the foundation for future enhancements such as **group chats, file sharing, message notifications,** and more. The project provided hands-on experience with the MERN stack, real-time communication protocols, and security measures in a modern web application, helping the team strengthen their understanding of full-stack development and open-source technologies.

Contents

Chapter 1 : Project Overview

1.1 Introduction and Motivation

The project we undertook was a **Real-Time Web-Based Chat Application**, aimed at providing us with hands-on experience with various open-source technologies. This chat app was developed using the **MERN stack** (MongoDB, Express, React, Node.js) along with **Socket.IO** to facilitate real-time communication. The system allows users to register, log in, and engage in one-on-one chat sessions, offering basic but essential functionality for a messaging platform.

The chat app is designed to let users send and receive messages in real-time, making it an interactive platform that simulates the fundamental features of popular messaging apps. Though simple in scope, it serves as a foundational project to grasp full-stack development using modern technologies.

Motivation

The primary motivation for this project was to gain practical experience with open-source technologies and full-stack development as part of our **ROSP (Real-Time Open Source Programming) Lab**. By working on a real-world application like a chat app, we were able to:

- **Familiarize ourselves with the MERN stack**: This is a widely used technology stack in modern web development, and it's essential for developing scalable, efficient, and maintainable web applications.
- **Understand real-time communication**: The use of **Socket.IO** helped us learn how real-time, bidirectional communication can be implemented over web sockets, which is critical for applications like chat apps, live collaboration tools, and gaming platforms.
- **Learn full-stack development**: By developing both the frontend (React) and backend (Node.js with Express) and connecting them through APIs and a NoSQL database

(MongoDB), we were able to grasp the full development lifecycle of a web application.

- **Experience building a basic user authentication system**: Managing user authentication and ensuring a secure login and registration system is a vital part of any web app.

Our project allowed us to focus on understanding the core technologies, which would serve as a foundation for building more complex applications in the future. Furthermore, the potential for adding additional features like group chats, media sharing, and notifications provides a clear roadmap for future expansion.

1.2 Problem Statement

In today's digital world, communication has become predominantly real-time and web-based, with various platforms offering messaging services. However, building such a platform from scratch involves solving a range of technical challenges, including managing user authentication, facilitating instant message delivery, and ensuring scalability. The goal of this project was to create a **Real-Time Web-Based Chat Application** that addresses these challenges using modern open-source technologies, specifically the **MERN stack** along with **Socket.IO**.

The primary problems we aimed to address were:

1. **Real-Time Communication**: Creating a system that allows users to engage in real-time, one-on-one conversations without delays. This required managing the complexities of real-time data transfer over the web.
2. **User Authentication and Security**: Ensuring that users can securely register and log in to the system using email as a unique identifier, which involved protecting user data and handling secure authentication mechanisms.
3. **Data Persistence**: Storing user information, chat history, and messages in a persistent and scalable way using a NoSQL database (MongoDB) while ensuring that the data could be accessed efficiently.
4. **Simple and Interactive User Interface**: Designing a user-friendly interface using React to allow seamless interaction between users without overwhelming complexity.

By tackling these problems, the goal was to build a foundation for a chat application that is functional and scalable, with the possibility of expanding to support additional features in the future, such as group chats, notifications, and media sharing.

The project allowed us to understand and implement solutions to these core problems while gaining a deeper insight into modern web development techniques.

1.3 Requirement Analysis

Functional Requirements

Functional requirements describe the specific behavior or functions of the system. For our **Real-Time Web-Based Chat Application**, the primary functional requirements include:

1. **User Registration**:
    - The system must allow new users to register with a valid email address and a unique username.
    - Passwords should be securely stored using hashing techniques to ensure security.
2. **User Login**:
    - Users should be able to log in using their registered email and password.
    - The system should verify credentials and authenticate users.
3. **Real-Time Messaging**:
    - The system must enable real-time one-on-one chat between two users.
    - Messages should be sent and received instantly using **Socket.IO** to handle real-time communication.
    - Once sent, messages should be displayed immediately on both users' screens without requiring a page refresh.
4. **Persistent Message Storage**:
    - All chat messages should be stored in a **MongoDB** database to allow users to view their chat history even after logging out and logging back in.
5. **Unique Email-Based Communication**:
    - Each user should have a unique email ID, and users can only chat with others who are also registered in the system.
6. **User Interface for Chatting**:
    - A simple and intuitive chat interface should allow users to view active conversations, type and send messages, and see incoming messages in real time.

7. **Logout Functionality**:
    - Users should be able to log out securely, and sessions should be properly terminated.

Non-Functional Requirements

Non-functional requirements describe the system's quality attributes, performance, and constraints. These requirements ensure that the application is usable, efficient, and secure.

1. **Performance**:
    - The chat application must handle real-time message delivery with minimal delay (low latency).
    - The application should be responsive, and the front-end interface should load quickly on different devices.
2. **Scalability**:
    - The system should be designed to support multiple users without significant performance degradation.
    - Future versions should be able to accommodate features like group chat or multimedia sharing without major architectural changes.
3. **Security**:
    - User credentials should be stored securely using password hashing and salting techniques.
    - The application should ensure secure transmission of data over the network (e.g., through HTTPS for production).
    - Users' private data and messages should be protected from unauthorised access.
4. **Usability**:
    - The user interface must be simple, intuitive, and easy to navigate, even for first-time users.
    - The app should work across different browsers and screen sizes (mobile responsiveness).

5. **Reliability and Availability**:
    - The system should be available and operational at all times. The database and server should be able to recover from crashes or unexpected shutdowns without data loss.
    - The app should be able to handle a reasonable number of concurrent users without interruptions or downtime.
6. **Maintainability**:
    - The system should be easy to update and extend in the future, whether to add new features or improve performance.
    - The codebase should follow clean coding standards, making it easier for developers to maintain and debug.
7. **Data Integrity**:
    - The system must ensure that no data is lost during transmission, and that all messages are accurately delivered and stored in the database.
8. **Compatibility**:
    - The chat application should be compatible with modern web browsers like Chrome, Firefox, and Edge, and should work well on both desktop and mobile platforms.

1.4 Project Design

The design of the **Real-Time Web-Based Chat Application** follows a modular and scalable architecture using the **MERN stack** (MongoDB, Express, React, Node.js) combined with **Socket.IO** for real-time communication. The design process involved both front-end and back-end considerations, focusing on user interaction, message handling, and database integration. Below, we describe the key components of the project design.

System Architecture

The system is structured into three primary layers:

1. **Client-Side (Frontend)**:
    - **React** was used to build the user interface (UI). The main features of the client-side design include:
        - **User Authentication Forms**: A login and registration page where users can sign up or log in using their email.
        - **Chat Interface**: A simple chat window where users can send and receive messages in real-time. The interface displays the conversation with another user in a scrollable view.
        - **State Management**: React's local state and hooks were used to manage user data and chat updates, ensuring real-time message rendering.
2. **Server-Side (Backend)**:
    - **Node.js** and **Express** were used to handle HTTP requests, manage user authentication, and route data between the front-end and database. Key server-side design elements include:
        - **User Authentication**: An authentication system that securely registers users and verifies their credentials during login using JSON Web Tokens (JWT). Passwords are hashed and stored securely in the database.
        - **Chat Functionality**: The server listens for messages from clients using **Socket.IO**, which enables bi-directional communication between the client and server.
        - **RESTful API**: The server provides APIs for user registration, login, and fetching user data.
3. **Database Layer**:
    - **MongoDB** serves as the NoSQL database for storing user information and chat messages. Each chat session and user data (email, password hash) are stored as documents in collections.
        - **User Collection**: Stores registered users with unique emails.
        - **Messages Collection**: Stores the chat messages exchanged between users, including timestamps and sender/receiver IDs.

Real-Time Communication Design

- **Socket.IO** is the key technology used to implement real-time communication. The client and server maintain an active WebSocket connection, allowing instant message transmission.
    - When a user sends a message, the message is sent via the WebSocket connection to the server, which then relays it to the intended recipient in real-time.
    - The chat messages are also saved to the database for persistence, ensuring the chat history is available even after a user logs out and back in.

User Authentication Design

- **JWT (JSON Web Token)**: The authentication system uses JWT to securely verify users. Upon successful login, a token is generated and sent to the client. This token is stored in the client's local storage and is used for authentication in future API requests. Passwords are hashed using a cryptographic algorithm before being stored in MongoDB, ensuring user data security.

User Interface Design

- **Login/Register Pages**: The app has a simple form-based UI for logging in and registering, ensuring ease of access for new and returning users.
- **Chat Window**: After logging in, users are redirected to a chat interface where they can search for other users by email and initiate one-on-one conversations.
- **Message Display**: Messages are displayed in real-time in a scrollable window. New messages appear instantly as the chat progresses, providing a seamless user experience.

Database Design

- **NoSQL Database (MongoDB)**: The database schema is flexible to support scalability. Key collections include:
    - **Users**: Each user document contains details like email, passwordHash, and authToken.
    - **Messages**: Each message document contains a sender, receiver, messageContent, and timestamp.

Data Flow

The data flow within the application can be broken down into the following steps:

1. **User Registration/Login**:
    - The user submits their credentials through the frontend.
    - The server authenticates the user, and upon success, a JWT token is generated and returned to the client.
2. **Real-Time Messaging**:
    - The user selects a recipient from their contacts or searches by email.
    - Messages are sent from the client through Socket.IO to the server.
    - The server processes the message and sends it to the intended recipient in real-time while also storing the message in the MongoDB database.

Future Enhancements

Though the current design supports basic one-on-one chatting, it has been structured in a way that makes it possible to add additional features with minimal changes. Future design considerations could include:

- **Group Chat**: Expanding the database schema and the Socket.IO logic to handle group conversations.
- **Media Sharing**: Allowing users to send images or files, which would require changes in the UI and backend for file storage.
- **Notifications**: Adding real-time notifications to alert users when they receive new messages.

Chapter 2 : Implementation

2.1 Implementation Details

The **Real-Time Web-Based Chat Application** was implemented using a combination of front-end, back-end, database, and real-time communication technologies. Below is a detailed breakdown of the implementation process, covering each component.

2.1.1 Frontend Implementation (React)

- **React Framework**: The frontend was built using **React**, a popular JavaScript library for building interactive user interfaces. The key focus was on creating a simple, intuitive UI to facilitate user registration, login, and chatting.
    - **Components**: The application was broken down into several reusable components:
        - **Login/Register Component**: Handles user authentication via forms. When a user registers, the data is sent to the backend API for validation and storage. Similarly, during login, credentials are verified, and upon success, a JWT token is received and stored in the browser's local storage.
        - **Chat Component**: This is the core interface for real-time messaging. It consists of:
            - **Message Input Box**: Allows the user to type and send messages.
            - **Message Display Area**: Shows a scrollable list of sent and received messages in real-time.
            - **User Selection**: Users can search for others by their email and initiate a chat session.
    - **State Management**: React's state hooks (e.g., useState, useEffect) were used to manage the application state, such as storing the user's login status and chat

data. React's component-based architecture helped to manage the dynamic nature of real-time chat updates.

- **WebSocket Integration with Socket.IO**: React components were integrated with **Socket.IO** to establish and manage a WebSocket connection between the client and server for real-time messaging.

2.1.2 Backend Implementation (Node.js and Express)

- **Node.js**: The backend was implemented using **Node.js**, a JavaScript runtime that allows us to run JavaScript on the server side. **Express** was used to build a RESTful API to handle user authentication, user management, and chat history retrieval.
  - **API Endpoints**:
    - **/register (POST)**: Accepts user registration details (email, password) and securely stores the hashed password in the MongoDB database.
    - **/login (POST)**: Accepts login credentials, verifies the user's password, and returns a JWT token upon successful authentication.
    - **/getUsers (GET)**: Retrieves the list of registered users to allow users to search for contacts.
    - **/chatHistory (GET)**: Retrieves the chat history between two users from the MongoDB database.
  - **User Authentication**:
    - **JWT Tokens**: Upon successful login, the server generates a JWT token that is sent back to the client. The client stores this token locally (in localStorage) and includes it in headers for all subsequent requests to authenticate the user.
  - **Password Hashing**: The **bcrypt** library was used to hash passwords before storing them in MongoDB, ensuring that sensitive user data is stored securely.
  - **Socket.IO Integration**: The backend uses **Socket.IO** to manage the WebSocket connections between the client and server. It listens for incoming chat messages and broadcasts them to the intended recipient in real time.

2.1.3 Database Implementation (MongoDB)

- **MongoDB**: A NoSQL database, MongoDB was chosen for its flexibility and scalability. The database was hosted using **MongoDB Atlas**, a cloud-based MongoDB service.
    - **Collections**:
        - **Users Collection**: Stores user details such as email and password hash. Each user is uniquely identified by their email address.
        - **Messages Collection**: Stores the chat messages between users. Each document in the collection contains the sender's email, recipient's email, message content, and a timestamp.
    - **Schema Design**: Since MongoDB is a document-based database, each user and message is stored as a document in its respective collection. This flexible schema design allows for easy scalability if new fields or features need to be added in the future (e.g., media messages, group chats).

2.1.4 Real-Time Messaging Implementation (Socket.IO)

- **WebSockets and Socket.IO**: The core of the real-time chat functionality was implemented using **Socket.IO**, which facilitates WebSocket-based communication between the client and server.
    - **Client-Side Socket.IO**: The client initiates a WebSocket connection with the server when the user logs in. Socket.IO events are used to listen for and send real-time messages.
        1. **Message Events**: When a user sends a message, it triggers a message event, sending the message to the server.
        2. **Receiving Messages**: The server listens for incoming messages and relays them to the intended recipient's client via the open WebSocket connection.

- **Server-Side Socket.IO**: The server handles the real-time broadcasting of messages to the recipient's socket. Upon receiving a message from a sender, the server:
    1. Emits the message to the recipient's socket in real-time.
    2. Saves the message to the MongoDB database for future reference (message persistence).
- **Socket Room Management**: Users are assigned to rooms based on their email ID when they log in, allowing the server to easily direct messages to the correct recipient.

2.1.5 Message Persistence

To ensure chat history is retained between sessions, all messages sent via Socket.IO are also saved in the MongoDB database. When users log in, the application fetches the chat history from the database so they can view past conversations.

- **Message Storage**: Each message is stored as a document in the Messages collection in MongoDB. The schema for each message includes the sender, receiver, content, and timestamp.
- **Fetching Chat History**: When a user opens a chat with another user, the server fetches the conversation history from the MongoDB collection and sends it back to the client for display.

2.1.6 Security Implementation

- **JWT Authentication**: JSON Web Tokens (JWT) were implemented to secure the application. After the user logs in, a JWT is generated and sent to the client, which is then used to authenticate subsequent API requests.
- **Password Hashing**: The **bcrypt** library was used to hash user passwords before storing them in the MongoDB database, ensuring that no sensitive data is stored in plain text.

- **HTTPS**: Although not fully implemented in the current version, future iterations of the project can incorporate HTTPS to ensure secure communication between clients and the server.

2.1.7 Error Handling and Testing

- **Error Handling**: Error-handling mechanisms were implemented to handle common issues like:
    - Invalid login credentials.
    - Connection issues with Socket.IO.
    - Missing or invalid JWT tokens.
- **Testing**: Manual testing was conducted throughout the project to ensure that:
    - Real-time communication works seamlessly between users.
    - User registration and login systems work as intended.
    - Chat history is accurately stored and retrieved from the database.

2.2 Technologies Used

The **Real-Time Web-Based Chat Application** was developed using a combination of modern open-source technologies that enable full-stack development, real-time communication, and efficient data management. Below is an overview of the key technologies used in the project, highlighting their roles and benefits.

2.2.1 Frontend Technologies

- **React (JavaScript Library)**:
    - **Role**: Used for building the user interface (UI) of the chat application. React's component-based architecture allowed us to build reusable, interactive components like the login form, chat window, and message display.
    - **Benefits**: React is efficient in rendering changes to the UI by using a virtual DOM, which optimizes updates and improves performance. It also promotes the development of scalable and maintainable code through component-based design.
- **CSS (Cascading Style Sheets)**:
- **Role**: Used for styling the chat application, making the UI visually appealing and user-friendly. Basic CSS was used to create layouts for the login page, registration page, and chat interface.
- **Benefits**: CSS allows for a responsive and attractive design, ensuring the application is easy to use across different devices and screen sizes.

2.2.2 Backend Technologies

- **Node.js (JavaScript Runtime)**:
    - **Role**: Used to build the backend server. Node.js provides a runtime environment for running JavaScript on the server, enabling communication between the frontend and the database, managing user authentication, and handling chat logic.

- **Benefits**: Node.js is well-suited for handling asynchronous events, which is essential for real-time applications. Its non-blocking architecture allows for efficient handling of multiple user connections.

- **Express (Web Application Framework)**:
  - **Role**: Used as a framework for building the RESTful API on the server. Express simplifies the development of server-side logic, including routing, middleware integration, and handling HTTP requests/responses.
  - **Benefits**: Express is lightweight, flexible, and easy to use. It helps organize server-side code effectively and allows for rapid API development.

2.2.3 Real-Time Communication

- **Socket.IO (WebSocket Library)**:
  - **Role**: Used to enable real-time, bidirectional communication between the client and the server. Socket.IO creates WebSocket connections that allow messages to be sent instantly between users.
  - **Benefits**: Socket.IO abstracts the complexities of WebSocket communication, making it easy to implement real-time features like live messaging. It is also highly reliable, falling back to alternative transports (like HTTP long-polling) if WebSockets are unavailable.

2.2.4 Database

- **MongoDB (NoSQL Database)**:
  - **Role**: Used as the database to store user data (e.g., registration details) and chat messages. MongoDB's document-based storage was ideal for managing dynamic data structures such as chat messages.
  - **Benefits**: MongoDB's flexible schema allows for rapid development and easy scaling as the application grows. It handles large amounts of data and is well-suited for applications requiring high read/write performance, such as a chat app.
- **Mongoose (MongoDB Object Data Modeling)**:

- **Role**: Used to define schemas and interact with MongoDB in a structured manner. Mongoose allows for the creation of models for users and messages, ensuring data is stored consistently.
- **Benefits**: Mongoose simplifies database operations by providing built-in validation and query-building capabilities, making it easier to interact with MongoDB in a safe and efficient way.

2.2.5 Authentication and Security

- **JWT (JSON Web Tokens)**:
  - **Role**: Used for secure user authentication. JWT is generated upon successful login and stored on the client-side to be used for authenticating API requests.
  - **Benefits**: JWT allows stateless authentication, meaning the server does not need to maintain session information, making the system scalable and secure. It also ensures that sensitive information (like passwords) is never exposed in transit.
- **bcrypt (Password Hashing)**:
  - **Role**: Used to securely hash user passwords before storing them in the database. bcrypt ensures that even if the database is compromised, passwords remain protected.
  - **Benefits**: bcrypt is a reliable and secure cryptographic hashing algorithm, adding an extra layer of security to the user authentication process by making it computationally expensive for attackers to crack passwords.

2.2.6 Development Tools and Platforms

- **Visual Studio Code (Code Editor)**:
  - **Role**: Used as the primary code editor for the development of the chat application. Its wide array of extensions, including support for JavaScript, React, Node.js, and MongoDB, made development smoother and more efficient.

- **Benefits**: VS Code offers an integrated development environment with debugging tools, a terminal, and support for version control, making it ideal for full-stack development.

- **Git (Version Control)**:
  - **Role**: Used for version control, allowing us to track changes, collaborate, and manage code efficiently during the project.
  - **Benefits**: Git's distributed nature and ability to create branches helped manage development, test features, and roll back changes when necessary.
- **GitHub (Code Repository)**:
  - **Role**: GitHub was used to host the project code, enabling team collaboration and version control. It also provided a platform to showcase and review the codebase.
  - **Benefits**: GitHub offers collaboration tools like pull requests, issue tracking, and project boards, making it easy for team members to contribute and manage code remotely.

These technologies formed the backbone of our **Real-Time Web-Based Chat Application**, each playing a critical role in ensuring that the app functions efficiently and securely. By using this tech stack, we were able to implement the key features of the project while maintaining flexibility for future enhancements.

2.3 Test Cases

Testing the **Real-Time Web-Based Chat Application** was a crucial step to ensure that the system functions as intended and that the key features—such as user authentication, real-time messaging, and data persistence—work seamlessly. Below are the main test cases that were created to verify the functionality of the system. Each test case outlines the scenario, the expected result, and the outcome after execution.

Test Case 1: User Registration

- **Test Scenario**: A new user attempts to register with valid email and password credentials.
- **Test Steps**:
    1. Navigate to the registration page.
    2. Enter a valid email and password.
    3. Click on the "Register" button.
- **Expected Result**: The user is successfully registered, and the system stores their details in the MongoDB database. The user should see a success message or be redirected to the login page.
- **Actual Result**: ✅ Passed.

Test Case 2: Duplicate User Registration

- **Test Scenario**: A user attempts to register using an email that already exists in the system.
- **Test Steps**:
    1. Navigate to the registration page.
    2. Enter an email address that is already registered.
    3. Click on the "Register" button.
- **Expected Result**: The system rejects the registration attempt, and the user is informed that the email is already in use.

- **Actual Result**: ✅ Passed.

Test Case 3: User Login

- **Test Scenario**: A registered user attempts to log in with correct credentials.
- **Test Steps**:
    1. Navigate to the login page.
    2. Enter valid email and password.
    3. Click the "Login" button.
- **Expected Result**: The user is authenticated, a JWT token is generated, and the user is redirected to the chat interface.
- **Actual Result**: ✅ Passed.

Test Case 4: Incorrect Login Credentials

- **Test Scenario**: A user attempts to log in with incorrect email or password.
- **Test Steps**:
    1. Navigate to the login page.
    2. Enter an incorrect email or password.
    3. Click the "Login" button.
- **Expected Result**: The system rejects the login attempt and displays an error message indicating that the credentials are invalid.
- **Actual Result**: ✅ Passed.

Test Case 5: Real-Time Messaging

- **Test Scenario**: Two users exchange messages in real time.
- **Test Steps**:

1. User A logs in and selects User B to chat with.
2. User B logs in separately and opens a chat with User A.
3. User A sends a message, and User B should see the message instantly.
4. User B responds, and User A should see the reply instantly.

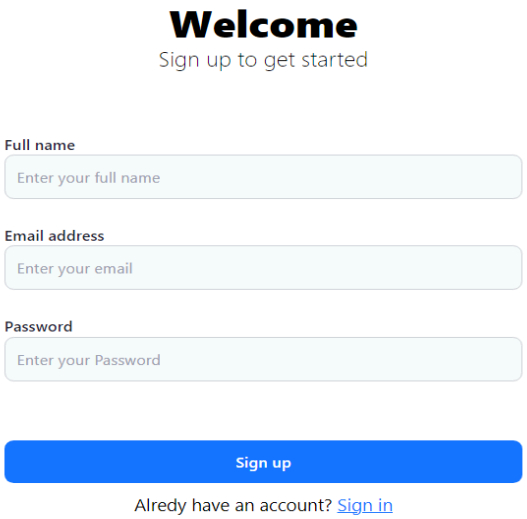
- **Expected Result**: Messages should be transmitted instantly between User A and User B without any noticeable delay, and both users should be able to view the full chat history in real time.
- **Actual Result**: ✅ Passed.

Test Case 6: Message Persistence

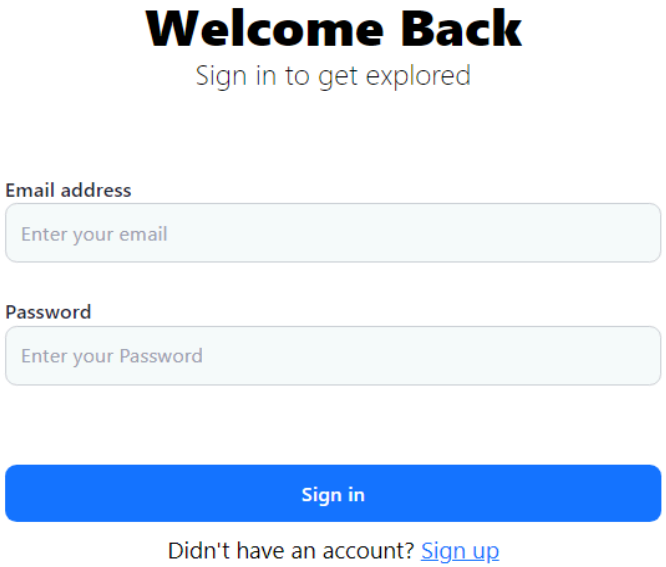
- **Test Scenario**: Messages between users are stored in the database and can be retrieved when the user logs in again.
- **Test Steps**:
  1. User A logs in, sends a message to User B, and logs out.
  2. User A logs back in after some time.
  3. The previous conversation with User B should still be visible in the chat window.
- **Expected Result**: The full conversation should be retrieved from the database and displayed correctly in the chat interface.
- **Actual Result**: ✅ Passed.

Results

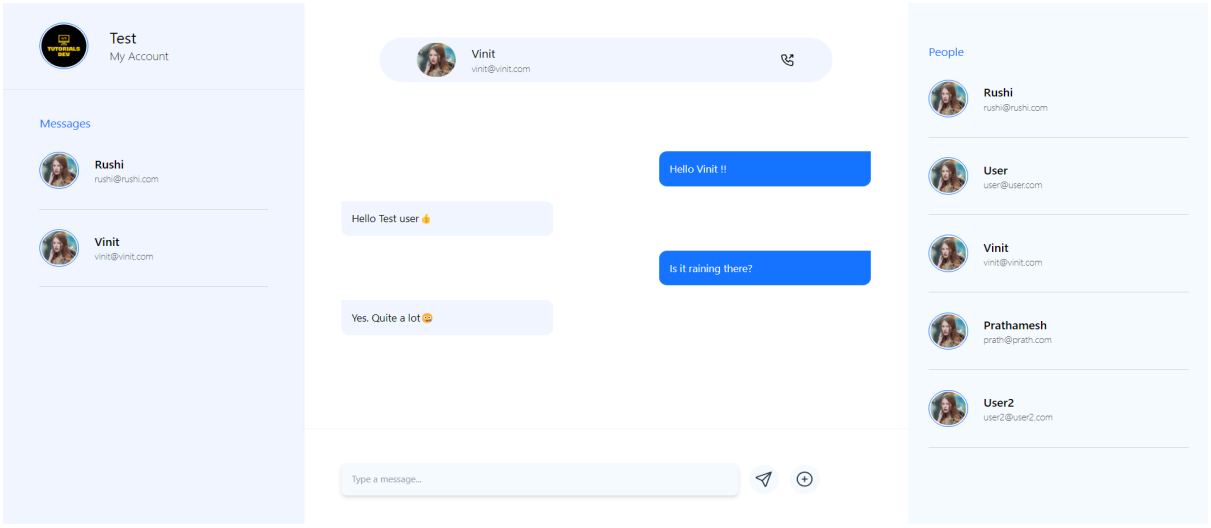
Signup Page



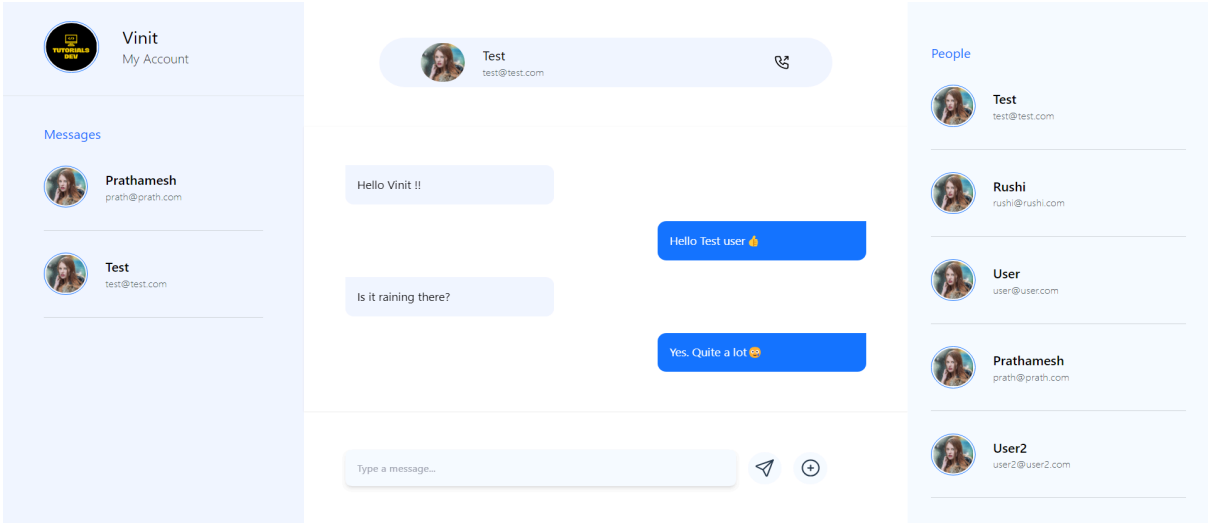
Sign in page



User 1 chat screen



User 2 chat screen



References

React Documentation

- Official documentation for React:
  Available at: https://react.dev

Node.js Documentation

- Official documentation for Node.js:
  Available at: https://nodejs.org

Express.js Documentation

- Official documentation for Express.js:
  Available at: https://expressjs.com

MongoDB Documentation

- Official documentation for MongoDB:
  Available at: https://mongodb.com/docs

Socket.IO Documentation

- Official documentation for Socket.IO:
  Available at: https://socket.io/docs

Online Tutorials and Blogs

- Various online resources including YouTube tutorials and blog posts on chat application development using MERN stack and Socket.IO.

Generative AI Tools

- AI-assisted tools, particularly useful for debugging and optimising the code during the project development.