System Design Documentation and Prototype setup

Frontend (Client)

- **Technology Used**: React, Socket.io-client, TailwindCSS
- **Purpose**: Provides a responsive user interface where users can send and receive real-time messages and notifications.

Key Components:

- **Chat Window**: Displays messages exchanged in real-time between users.
- **Message Input**: Allows users to type and send messages.
- **User List**: Displays all the online users in the chat room.
- **Notifications**: Alerts users of incoming messages.

Libraries Used:

- **React**: Manages UI components dynamically based on real-time data updates.
- **React-router-dom**: Manages navigation between chat rooms and user profile pages.
- **Socket.io-client**: Enables real-time, bi-directional communication between the client and server.
- **TailwindCSS**: Provides utility-first styling for fast and responsive UI design.

Working:

- The frontend uses Socket.io-client to establish a WebSocket connection with the backend server.
- When a user sends a message, it is transmitted through the WebSocket connection to the server.
- The server then broadcasts the message to all connected clients, and the frontend updates the chat window in real time.

Backend (WebSocket Server)

- **Technology Used**: Node.js, Express, Socket.io, MongoDB
- **Purpose**: Facilitates real-time communication by acting as a Socket.io server, storing messages and handling user connections.

Key Components:

- **Socket.io Server**: Manages WebSocket connections for real-time messaging.
- **Express.js**: Manages HTTP requests, such as loading the client and providing REST APIs for user and message data.
- **MongoDB**: Stores user details, chat rooms, and message history.

Libraries Used:

- **Node.js**: The runtime environment for executing JavaScript on the server side.

- **Express.js**: Simplifies handling HTTP requests and serving the chat application's static files.
- **Socket.io**: Handles WebSocket connections for real-time features like messaging.
- **MongoDB**: Used for persisting chat room data, user data, and message histories.

Working:

- When a client sends a message, it reaches the server via a WebSocket connection.
- The server processes the message (e.g., appends a timestamp, user details) and broadcasts it to all other connected clients in the same chat room.
- The server also manages user connections, handling events such as when users join or leave a chat room.

Database (MongoDB)

- **Technology**: MongoDB
- **Purpose**: Stores persistent data, such as user profiles, chat rooms, and message histories.

Advantages:
Document-Oriented: Stores chat messages in a flexible, JSON-like format, making it easier to manage unstructured data like chat histories.
Scalability: MongoDB scales well, which is beneficial as the chat application grows in terms of users and stored messages.

WebSocket Communication:

WebSockets enable persistent and real-time communication between the client and the server. This is essential in chat applications, where low-latency, bi-directional communication is needed to deliver messages instantly.

Working:

1. **Connection Establishment**:
   - The client initiates a WebSocket connection with the server when the chat app is loaded.
   - This connection remains open as long as the session is active.
2. **Real-Time Data Exchange**:
   - The client sends messages to the server over the open WebSocket connection.
3. **Persistent Connection**:
   - Unlike HTTP, WebSocket keeps the connection open, allowing continuous communication without repeatedly establishing new connections.
4. **Disconnection**:
   - If a user disconnects, the server is notified, and any relevant updates (e.g., the user leaving the chat) can be sent to other connected clients

```
+----------------------------+            +-------------------------------+       +----------------------------+
|   Frontend (UI)       |              |  WebSocket Server  |        |       Database        |
|  (React, Socket.io)  |   --------> | (Node.js, Express)   |  ------> |    (MongoDB)          |
|                              |             |                              |         |                              |
|    Chat Window       <-------------  | Handles real-time     <---------|    Stores Chat          |
|    Message Input     <----------->    |   communication      |<--------|   History, Users        |
|  Online Users List    |  <--------  |    with clients          |          |                              |
+---------------------------+           +-----------------------------+       +-----------------------------+

     ^
     |
     |
     |
     |
+-----------------------------+
|      Client                   |
|   (Socket.io)               |
|                                |
| Receives messages,   |
| sends real-time           |
| updates to the  server  |
+-----------------------------+
```

Summary of the System Flow:

User Interaction (Frontend):

- Users interact with the chat interface, sending and receiving messages in real-time.

WebSocket Communication:

- A persistent WebSocket connection is established between the client and server using **Socket.io**.
- Messages are sent from the client to the server and broadcasted to all connected clients in the same chat room.

Backend Processing:

- The server receives messages, processes them (e.g., adds timestamps), and broadcasts them to other users.
- The server also manages user connections and disconnections.

Database Interaction (MongoDB):

- Messages and user data are stored in **MongoDB** to ensure persistence.
- When a user joins a chat room, they can retrieve chat history from the database.

Real-Time Updates:

- Clients instantly receive updates, such as new messages or changes in user status, ensuring a seamless real-time chat experience.

Chat Application Setup and Running Guide

Prerequisites

- Node.js
- MongoDB

Project Setup

- Clone the Repository

  Command used: **git clone https://github.com/yourusername/chat-app.git cd chat-app**

Install Dependencies

After cloning the project, install the necessary dependencies listed in the `package.json` file:

Command used:**npm install**

Running the MongoDB Database

Remote MongoDB: I have used the remote database MongoDB Atlas, so make sure the connection string in the backend code points to the correct MongoDB instance.

Running the Backend (WebSocket Server)

Start the WebSocket Server: Navigate to the backend directory (if separated) and start the Node.js server
Command used:**node server.js**
The server will establish WebSocket connections for real-time communication between clients. Make sure it's running before starting the frontend.

Running the Frontend (React Application)

Start the React Development Server:
Command used:**npm start**
This will start the frontend on `http://localhost:3000`. We can now access the chat app in our browser.

Dependencies and Libraries Used:

Frontend:

1.React:

Used to build the dynamic and interactive user interface. React efficiently manages UI updates when new messages are received or sent.

2.Socket.io-client :

Establishes WebSocket connections from the client to the server, enabling real-time bi-directional communication for sending and receiving messages instantly

3.TailwindCSS:

A utility-first CSS framework that allows rapid styling of the user interface, ensuring responsive and clean designs without writing extensive custom CSS.

4.React-router-dom

Provides routing capabilities for navigating between different views like chat rooms, user profiles, etc.

Backend

1.Node.js:

A non-blocking, event-driven runtime, perfect for handling real-time communication and multiple WebSocket connections.

2.Express.js:

Simplifies the server-side routing and serves static assets (like the frontend) while handling the REST API.

3.Socket.io:

Enables WebSocket connections on the server side to manage real-time communication between clients.

Database

1.MongoDB:

A NoSQL database suitable for storing real-time chat data (messages, users, chat rooms) as flexible, JSON-like documents. Its scalability and ability to handle dynamic data make it ideal for chat applications.

Verifying the Setup

- Open multiple browser windows or use different devices to simulate multiple users.
- Ensure real-time messaging works by sending and receiving messages instantly.
- Confirm the MongoDB database is storing messages and user data properly.

By following these instructions, we should be able to successfully install the necessary dependencies, run both the frontend and backend, and test the real-time messaging features.

Created by- Rajit Singh Purviya, email id-rajitsingh.purviya.cd.eee22@itbhu.ac.in,

roll no-22085131.