**System Design Documentation**

**MERN Chat App**

**I. Introduction**

The MERN Chat App is a web-based real-time messaging service developed using the MERN stack, comprising React for the frontend, Node.js for the backend, and MongoDB for database storage. This system design document outlines the core features, architecture, and technologies used in the development of the chat app.

**II. System Architecture**

The system architecture of the MERN Chat App can be divided into the following components:

1. **Frontend**: Developed using React, responsible for the user interface and communication with the backend through REST APIs and WebSockets.

2. **Backend:** Built with Node.js, handling user authentication, message handling, and WebSocket communication for real-time updates.

3. **Database**: MongoDB is used as the database for storing user information and chat messages.

**III. Messaging Service Features**

This prototype of the MERN Chat App includes the following core features:

- User registration and authentication.

- Sending and receiving text messages between users.

- Group chat functionality.

- Real-time message updates.

- setting profile pictures

**IV. Design**

The app features a clean and intuitive user interface that ensures a seamless messaging experience for users.

**V. Technologies Used**

**- Frontend Dependencies :**

- React for building the user interface.

- Redux for state management.

- Bootstrap and React Bootstrap for styling.

- React Router for client-side routing.

- Socket.io-client for real-time communication.

- Other dependencies as listed in package.json.

**- Backend Dependencies:**

- Express for creating RESTful APIs.

- Socket.io for WebSocket communication.

- MongoDB for data storage.

- Bcrypt for password hashing.

- Dotenv for environment variable management.

- Other dependencies as listed in package.json.

**VI) Data Models and Schema:**

**User Model Schema:**

The User model is used to represent user information in the MERN Chat App. It includes the following fields:

- name (String): The user's name, required, and cannot be blank.

- email (String): The user's email address, stored in lowercase, unique, and required. It is validated using the `isEmail` function from the 'validator' library.

- password (String): The user's password, required, and cannot be blank. It is hashed using bcrypt for security.

- picture (String): The user's profile picture, stored as a URL.

- newMessages (Object): An object that keeps track of new messages for the user.

- status (String): The user's status, which is set to 'online' by default.

**Message Model Schema:**

The Message model is used to store chat messages in the MERN Chat App. It includes the following fields:

- content (String): The content of the message.

- from (Object): Details of the sender.

- socketid (String): The unique socket ID associated with the message.

- time (String): The time at which the message was sent.

- date (String):The date on which the message was sent.

- to (String): The recipient's user ID or some identifier.

In your system design documentation, you should provide details about authentication and authorization in your MERN Chat App. Here's what you can write for the authentication and authorization section:

**VII. Authentication and Authorization**

**Authentication:**

1. User Registration: New users can register by providing their name, email, and password. The password is securely hashed before storage in the database.

2. User Login: Registered users can log in using their email and password. The system checks the credentials against the stored hashes to grant access.

3. Password Hashing: The app uses bcrypt to securely hash user passwords before storing them in the database. This enhances the security of user data.

4. JSON Web Tokens (JWT): Upon successful login, users are issued JWT tokens. These tokens are included in the request headers to authenticate and authorize users when making subsequent requests to protected endpoints.

**Authorization:**

1. Protected Routes: Certain routes and functionalities are protected and require a valid JWT token in the request headers to access. This ensures that only authenticated users can perform actions like sending messages, joining chat rooms, or changing profile details.

2. Role-Based Access Control (RBAC): The app may implement role-based access control in the future to grant different levels of access to users based on their roles or permissions. For example, administrators may have more privileges than regular users.

3. Ownership and Access Control: The app may include logic to control access to resources based on ownership. For instance, a user can only edit or delete their own messages, but not messages from other users.

In the real-time communication section of your system design documentation, you should describe how real-time communication is implemented in your MERN Chat App using WebSocket technology, particularly Socket.io. Here's what you can write:

**VIII. Real-Time Communication**

**Key Components:**

1. WebSocket (Socket.io): WebSocket is a communication protocol that enables full-duplex, bidirectional communication between a client (e.g., a web browser) and a server. Socket.io is a JavaScript library that simplifies WebSocket implementation and provides real-time features such as event-based messaging.

2. Server-Side Implementation: The Node.js backend of the application is responsible for handling WebSocket connections using Socket.io. It establishes WebSocket connections with clients and manages real-time events and messaging.

3. Client-Side Integration: The React-based frontend integrates with the WebSocket server using the Socket.io client library. This enables users to send and receive messages in real-time, and also receive updates about new messages

**Key Features:**

- Message Broadcasting: When a user sends a message, the backend broadcasts it to the appropriate recipients in real-time. This ensures that messages are delivered instantaneously to the intended recipients.

- Event-Based Communication: Socket.io facilitates event-based communication. Events, such as "new-message" or "user-joined," are emitted and listened to by both the server and clients, enabling real-time updates and interactions.

- User Presence: Users' presence status is updated in real-time, allowing others to see when a user is online or offline. This information is displayed within the chat app.

Use Cases:

- Real-Time Chat: Users can engage in real-time text conversations with other users or within group chat rooms. Messages are delivered instantly and appear in the chat interface as they are sent.

- Online/Offline Status: Users can see the real-time status of their contacts (e.g., "online" when active, "offline" when they log out).

**Performance Considerations:**

To ensure optimal performance, the application is designed to handle a large number of concurrent WebSocket connections efficiently. This includes optimizing server resources, minimizing latency, and load balancing if required.

**Scalability:**

The architecture is designed to be scalable, allowing for the addition of more WebSocket instances or load balancers as the user base grows. This ensures that real-time communication remains responsive and dependable, even under increased traffic.

**Security:**

To enhance security, the WebSocket connections are secured using encryption (e.g., TLS/SSL) to protect the confidentiality and integrity of messages during transit.