FlowChat

1. Overview

The chat web application provides a platform where users can send real-time messages to each other. The application includes authentication, private messaging, message history, and real-time communication through WebSockets. The system uses a client-server model, incorporating modern web technologies for scalability and efficiency.

2. System Architecture

2.1 Components

1. Client (Frontend):
   - Manages user interface (UI) and communication with the server.
   - Uses REST APIs for authentication and WebSockets for real-time messaging.

2. Server (Backend):
   - Handles business logic, authentication, and routing.
   - Facilitates WebSocket connections for real-time message exchange.
   - Utilises a database to store messages and user details.

3. Database:
   - Stores user information, chat messages, and authentication data.
   - NoSQL database (MongoDB) used for handling document-based data.

4. WebSocket Communication:
   - Provides full-duplex communication for real-time messaging.
   - Socket.IO is used for managing WebSocket connections.

2.2 Data Flow

Authentication Flow:

1. User signs up or logs in using a username and password.
2. JWT (JSON Web Token) is generated and sent to the client.
3. The client stores the JWT and attaches it to all subsequent requests for validation.

Messaging Flow:

1. After a successful login, the client opens a WebSocket connection.
2. Messages sent by users are broadcast to the respective recipient via the WebSocket connection.
3. The server stores all messages in the database and sends them to the intended recipient in real-time.

3. Technologies and Dependencies

3.1 Frontend

- React.js: For building the user interface.
- Socket.IO Client: To handle WebSocket communication for real-time message transfer.
- Axios: For handling HTTP requests.

3.2 Backend

- Node.js with Express.js: Manages API routes, authentication, and WebSocket handling.
- Socket.IO: Real-time event-based communication between client and server.
- JWT (JSON Web Token): Token-based authentication system.
- Bcrypt: For securely hashing passwords before storing them in the database.
- MongoDB (with Mongoose): NoSQL database for storing user profiles and chat messages.

3.3 Dependencies Summary

Library/Dependency	Version	Purpose
Express.js	^4.19.2	Backend web framework
Socket.IO	^4.7.1	Real-time communication
Mongoose	^7.4.0	MongoDB ORM
jsonwebtoken (JWT)	^9.0.0	User authentication
bcryptjs	^5.1.0	Password hashing
Cors	^2.8.5	Cross-Origin Resource Sharing
Axios	^1.6.0	HTTP client for frontend communication

4. Authentication Design with JWT

4.1 Signup Flow

1. User Submits Details: The user provides a username and password.
2. Password Encryption: The backend hashes the password using bcrypt before saving it.
3. User Creation: User data is saved in MongoDB.
4. Response: A success message is returned to the client.

4.2 Login Flow

1. User Authentication: The user sends their username and password.
2. Password Verification: The server compares the provided password with the stored hash using bcrypt.
3. JWT Issuance: Upon successful authentication, a JWT is generated and sent to the client for future API requests.

4.3 Token Verification Middleware

All protected routes verify the JWT before processing.

5. Real-Time Messaging System

5.1 WebSocket Design

The messaging system relies on Socket.IO for real-time communication. A WebSocket connection is initiated once the user is authenticated. When a user sends a message, the message is broadcast to the server, which relays it to the intended recipient.

6. Database Design

6.1 User Schema

```
const userSchema = new mongoose.Schema({
    username: { type: String, unique: true, required: true },
    password: { type: String, required: true }
});
```

6.2 Message Schema

```
const messageSchema = new mongoose.Schema({
    sender: { type: mongoose.Schema.Types.ObjectId, ref: 'User',
required: true },
    recipient: { type: mongoose.Schema.Types.ObjectId, ref:
'User', required: true },
    content: { type: String, required: true },
    timestamp: { type: Date, default: Date.now }
});
```

7. Setup and Run Instructions

7.1 Prerequisites

- Node.js: v14+ (for server-side development)
- MongoDB: v4.x+ (for database management)

7.2 Project Setup

1. Clone the Repository:

```
git clone https://github.com/username/chat-app.git
cd chat-app
```

2. Install Dependencies:
Run the following command to install all the required dependencies for both the client and server.

```
npm install
```

3. Environment Setup:

Create a `.env` file in the root directory and add the following:

```
PORT=5000
JWT_SECRET=your_jwt_secret
MONGO_URI=mongodb://localhost:27017/chat-app
```

4. Run MongoDB:
If MongoDB isn't running, start the service locally or use MongoDB Atlas:

```
mongodb
```

5. Run the Server:

Start the backend server

```
npm run server
```

6. Run the Client:

Navigate to the client directory and start the React app:

```
npm start
```

7.3 Additional Setup
- If using MongoDB Atlas, update the `MONGO_URI` in the `.env` file with your MongoDB Atlas connection string.
- Make sure to replace the `JWT_SECRET` with a secure key.

8. Libraries and Rationale

8.1 Node.js & Express.js
- Why: Node.js is ideal for real-time applications and handles concurrent connections efficiently. Express.js simplifies routing and server-side development.

8.2 Socket.IO
- Why: It abstracts the complexities of WebSocket communication and allows fallback to other protocols for real-time communication.

8.3 MongoDB with Mongoose
- Why: MongoDB offers high scalability and flexibility in storing data. Mongoose provides a schema-based solution for MongoDB, making it easier to model and validate application data.

8.4 JWT & Bcrypt
- Why: JWT allows for stateless and secure token-based authentication, while Bcrypt ensures password safety through hashing.