System Design Document for KumoTalk Chat Application

1. Overview

KumoTalk is a real-time chat application designed to provide both group and private messaging features with support for online user tracking, typing indicators, and theme toggling (light/dark modes). The chat app utilizes a full-stack solution with a frontend developed in Next.js and a backend built using Node.js, Express, and MongoDB, with Socket.IO enabling real-time communication.

2. Requirements

Functional Requirements

- User Authentication: Users can register and log in using JWT-based authentication.
- Group Chat: Users can participate in a global chat room.
- Private Chat: Users can send direct messages to specific users.
- Online User List: Shows users currently connected to the system.
- Typing Indicator: Displays when a user is typing in a conversation.
- Theme Toggle: Users can switch between light and dark mode.
- Logout: Users can log out, removing their session.

Non-Functional Requirements

- Scalability: The application should handle multiple concurrent users without performance degradation.
- Real-time Communication: Latency between sending and receiving messages should be minimal.
- Security: Secure JWT-based authentication.
- Responsive Design: The frontend should work well across various screen sizes.

3. System Architecture

3.1. High-Level Architecture

- Frontend: Built with Next.js (React framework), it manages user interactions, authentication, chat UI, and communication with the backend via Socket.IO and REST API.

- Backend: Built with Node.js and Express, it provides REST APIs for user authentication and Socket.IO for real-time messaging.
- Database: MongoDB is used for persistent data storage, including user data and messages.
- Socket.IO: Handles real-time WebSocket communication for messaging, online status, and typing indicators.

4. Components and Flow

4.1. Frontend

Technologies: Next.js, Tailwind CSS, Socket.IO Client
Purpose: Handle user interactions such as login, messaging, and theme toggling. Establish a WebSocket connection for real-time updates.

4.2. Backend

Technologies: Node.js, Express, MongoDB, Socket.IO
Purpose: Manage user authentication (JWT), facilitate WebSocket communication for real-time messaging, and interact with MongoDB for persistent storage.

4.3. MongoDB

Database: MongoDB Atlas (cloud-hosted)
Collections: Users and Messages collections store user data and chat history.

5. Detailed Flow

5.1. User Registration and Login Flow

1. Register: User submits email, username, and password. The backend hashes the password using bcrypt and stores the details.
2. Login: User submits login credentials, and upon success, receives a JWT token.

5.2. Chat Flow (Real-Time Messaging)

1. Connecting to WebSocket: User establishes a WebSocket connection using Socket.IO.
2. Group Chat: Messages are broadcast to all connected clients in real-time.
3. Private Chat: User sends direct messages to a specific recipient.
4. Typing Indicator: Shows when another user is typing.

5.3. Online User Tracking

When a user connects or disconnects, the backend updates the online user list in real-time.

5.4. Logout

User logs out, removing the JWT token and disconnecting the WebSocket.

6. Database Schema Design

6.1. User Schema

{ '_id': ObjectId, 'username': String, 'email': String, 'password': String (hashed), 'status': String (online/offline) }

6.2. Message Schema

{ '_id': ObjectId, 'sender': ObjectId, 'recipient': ObjectId, 'message': String, 'timestamp': Date }

7. APIs

7.1. User Registration

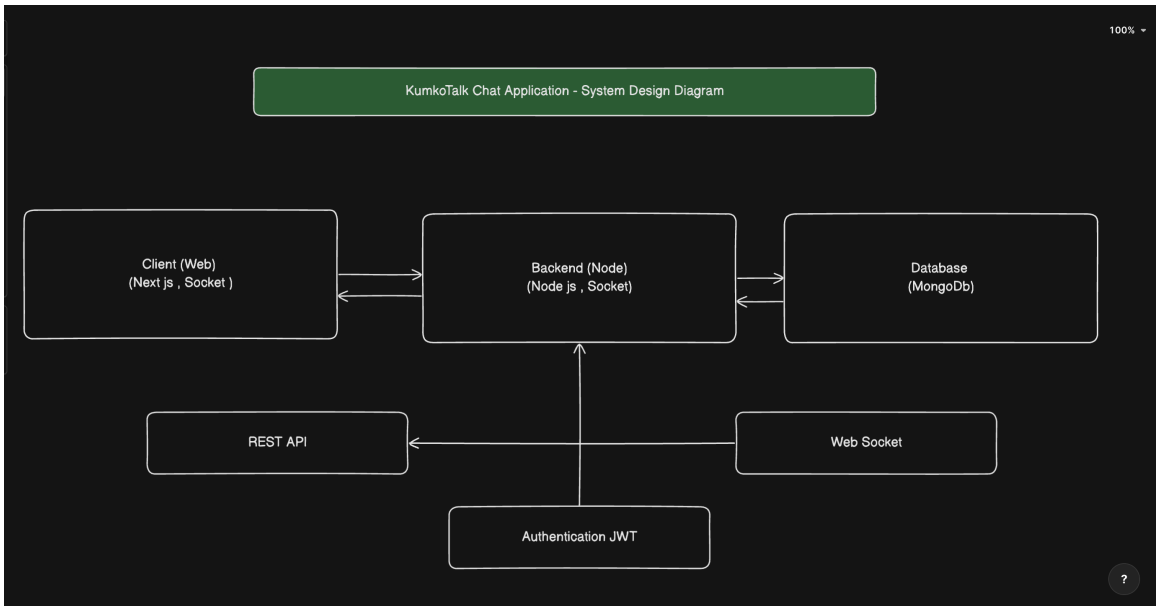
POST /api/users/register
Request Body: { 'username': 'testuser', 'email': 'test@example.com', 'password': 'password123' }

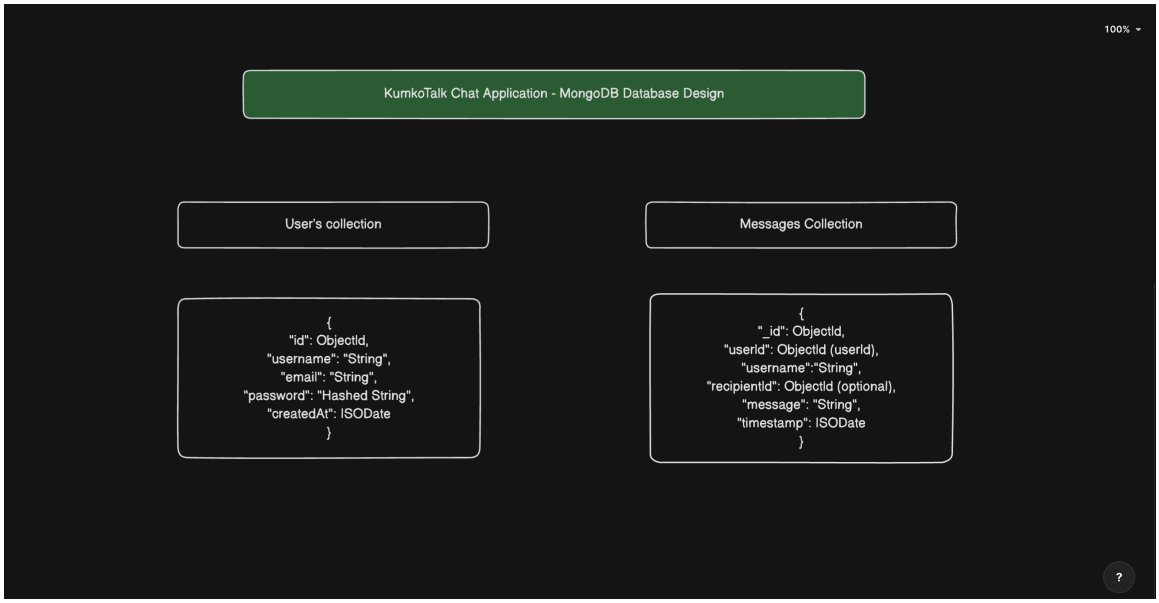
7.2. User Login

POST /api/users/login
Request Body: { 'email': 'test@example.com', 'password': 'password123' }

7.3. Online Users

GET /api/users/online-users
Response: [{ 'id': 'user1', 'username': 'testuser1' }]





The above is the Diagram for System design and Database design