**System Design Document**

This document outlines the design of an online chat application with real-time messaging and video communication features. The system allows users to register, log in, send text messages, and initiate video calls.

**Why Choose Next.js for This Project?**

**Full-Stack Capabilities**

Next.js is a hybrid framework that allows developers to build both client-side and server-side logic in one unified environment. This capability makes it perfect for projects that require both frontend and backend functionalities, such as this chat and video service platform.

**Real-Time Data Handling with SSR and WebSockets**

Next.js, combined with WebSockets via Socket.IO, provides a great environment for handling real-time data like chat messages or video calls. By leveraging Next.js’s server-side capabilities, the data is served faster and more efficiently

**API Routes and Backend Integration**

Next.js comes with built-in support for API routes, which allows the project to handle various backend tasks such as authentication, message storage, and interaction with third-party services like Cloudinary for media storage

**Why MongoDB is Best for a Messaging Website**

**Schema Flexibility**

MongoDB is a NoSQL database, meaning it is schema-less and can handle unstructured or semi-structured data. In a messaging system, user messages, media files, and even different chat structures may not follow a strict relational schema. MongoDB's flexible document model allows you to store diverse types of data in a single collection without having to pre-define complex schemas like in relational databases.

**Document-Oriented Data Model**

MongoDB stores data in BSON (Binary JSON) format, which is a highly flexible and scalable way to store complex hierarchical data. Each message or conversation can be stored as a BSON document, allowing you to nest related data such as user information, timestamps, and even attachments.

**Why MongoDB over Relational Databases for Messaging**

**Handling Unstructured Data:** Unlike relational databases, MongoDB can efficiently handle unstructured or semi-structured data. In a messaging system, the format of messages can vary (text, images, attachments), and MongoDB’s flexible schema is perfectly suited for this dynamic data model.

**Schema-less Nature**: A chat application may require frequent schema changes as new features are added. In a relational database, adding fields or changing relationships between tables requires costly migrations. MongoDB's dynamic schema allows you to store different types of messages or user data without predefined table structures.

**Optimized for JSON-like Documents**: MongoDB stores data in a BSON format, which is natively designed to handle JSON-like data structures. Since modern web applications (including chat services) often work with JSON data on the client-side, MongoDB offers a seamless data flow from frontend to database without the need for heavy transformations.

**Why Combined Tailwind css with Material UI?**

**Tailwind CSS: Utility-First CSS Framework**

**Why Use It:** Tailwind CSS is a utility-first framework that provides low-level utility classes for controlling spacing, colors, typography, positioning, and more. It allows for rapid prototyping and fine-tuned styling without writing custom CSS.

**Strength**: Tailwind is great for customizing layouts quickly, creating responsive designs, and making small, reusable design adjustments. Its utility-based approach reduces the need for context-switching between CSS files and components.

**Material UI: Pre-Built Component Library**

**Why Use It:** Material UI provides a comprehensive set of pre-designed components based on Google’s Material Design principles. It helps maintain consistency and provides advanced UI components like modals, forms, dialogs, and tables, all with out-of-the-box responsiveness.

**Strength:** Material UI helps you build a modern, professional-looking interface with minimal effort. Its components are customizable and accessible, following best practices for UI/UX.

Material UI comes with built-in accessibility and design guidelines that ensure your app is user-friendly and follows best UI/UX practices. Tailwind CSS helps fill in the gaps where you need more control over the exact spacing, positioning, or layout of those components. Tailwind CSS is designed with mobile-first responsiveness in mind, providing an intuitive way to create responsive designs by simply adding utility classes like md:w-1/2 or sm:hidden. When combined with Material UI, this ensures your app is responsive across all screen sizes. Tailwind CSS offers more design flexibility when you need granular control over the layout. You can easily customize the spacing, colors, and typography of Material UI components using Tailwind's utility classes. Tailwind CSS allows for rapid development by using utility classes to define styles directly in your JSX, speeding up the creation of new layouts.

Material UI provides ready-to-use components that help you avoid building basic UI elements from scratch.

**Architecture Overview**

**Registration/Sign-in Service**



**User Submits Form**: The user fills in the registration or sign-in form on the client-side with details username, password, email for the signup and email and password for the signin . Before the form gets submitted validations are performed which validates if email ,username and password fields are valid or not.

**Client-Side to Server Communication**: Upon submission, the form data is sent from the client to the server via an HTTP POST request .This data is sent securely, often over HTTPS.

**Server Validates Request**: The server receives the form data. It takes that data and save it to the database , the database model has fields which required certain types of data. Certain data fields are marked required.The password field is hashed and then stored in to the database. The server checks if the user already exists. If not,the the new user data is saved. . For sign-ins, this can include hashing the entered password and checking if it equals the stored user password. If the saving process went successfully then a response is returned stating that the user was registered successfully.

**JWT Token Creation:** After the user is saved in to the database , a jwt token is created. That jwt token is created using the id and username field from the saved document in the database. The token is sent to the client and stored as cookies in the browser.

**Client Receives and Displays Result**: The client receives the server's response and provides appropriate feedback to the user, such as a welcome message or an error notification. If authentication is successful, the user can proceed to access protected areas of the app.

**Single User Chat Service**

**Architecture Components**

**Frontend:**

**Search Box**: Allows users to search for another user to initiate a chat.

**Message Input**: Users can send messages, and messages are displayed in real-time.

**LocalStorage**: Stores messages temporarily until they are sent to the server after every 4 messages.

**Backend:**

**Socket.IO**: Manages real-time communication between users.

**Database**: Stores user information, chat history, and previous chats.

**Database Schema**:

**User Model**: Contains user details and previous chats.

**StoredChats Model**: Stores chat history after every 4 messages.

**Core Technologies**

**Socket.IO**

Socket.IO is a real-time engine that enables bidirectional communication between clients and the server. It is ideal for chat applications, allowing users to send and receive messages in real time.

**Why use Socket.IO?**

1.Provides real-time communication.

2.Handles reconnections and disconnections automatically.

3.Supports namespaces and rooms, making it easy to scale.

The general workflow of this application states that when the user gets to the chats route , their the **Inbox** component contains the previous chat history of the current user. If one click on any of the previous chat history displayed as list of items then we get redirected to another route where chats specific to conversation is shown. The messages shows have time ,message and user who sent it displayed. The chats are stored in mongodb. Now if the user wants to chat to some other user , he/she can search in the search box for that new user by type his username, if results appear , the conversation can start. For the very first time when the user talkes to someother user, thier document id from the user model gets stored in the previousChat field of the respective users. The messages are sent and received using socket.io. These sent messages gets stored in the local storage temporarily , after the stored message count exceeds 4 in the local storage , the local storage gets cleared and these messages are sent to the database to get stored in the stored chats collection.



**Group Chat Service**

**Architecture Components**

**Frontend:**

**Search Box**: Allows users to search for other users and invite them into a new group chat.

**Message Input**: Users can send messages in real-time, and group messages are displayed with user and time details.

**LocalStorage**: Temporarily stores messages until they are sent to the server after every 4 messages.

**Backend**:

**Socket.IO**: Manages real-time communication between multiple users in the group chat.

**Database**: Stores user information, group chat history, and the list of participants in each group chat.

**Database Schema**:

**User Model**: Contains user details and the group participation information.

**GroupChats Model**: Stores the list of participants and chat history for each group chat.

**StoredGroupChats Model**: Stores group chat messages after every 4 messages.

**Core Technologies**

**Socket.IO**

Socket.IO is essential for real-time communication in group chats, enabling users to send and receive messages in real-time across multiple users.

**Why use Socket.IO?**

Provides real-time, bidirectional communication for group chats.

Handles reconnections and disconnections automatically.

Scales efficiently for multiple users in group chats.

When the user navigates to the group chat route, the Inbox component displays the user’s previous group chats. Each group chat is listed, and clicking on any group chat redirects the user to view the specific conversation.To initiate a new group chat, the user can type usernames into a Search Box. This field sends a request to the server to check if the entered usernames exist. If the user exists, they are added to the group, and the group conversation starts.Messages are sent and received using Socket.IO in real-time, just like in the single chat service. Each message is displayed with the sender's name, the message content, and the timestamp.Messages are temporarily stored in localStorage until 4 messages are sent. After the localStorage holds 4 messages, the messages are sent to the server and stored in the StoredGroupChats collection in the database. LocalStorage is then cleared and ready to store more messages.When a new user is added to a group, their information is updated in the group’s participant list in the GroupChats model, ensuring all participants are recorded and chat history is preserved.



**Video Chat Service**

**Architecture Components**

**Frontend:**

**Call Inbox**: Displays only the video call history from the user's previous chat history. If the user clicks on a call in the inbox, the video call is initiated and the other user gets notified and redirected to the calls page.

**Video Call Interface**: Users can initiate and participate in video calls in real time.

**Backend:**

**Socket.IO**: Manages real-time communication for video calls between users.

**WebRTC (Web Real-Time Communication):** Handles the peer-to-peer connection for transmitting video and audio data between users.

**Core Technologies**

**Socket.IO:**

Socket.IO is essential for real-time event-based communication between the users and the server. It handles notifications when a video call is initiated, manages user availability, and ensures smooth signaling for WebRTC connections.

**Why use Socket.IO?**

Enables real-time communication between clients and server.

Handles reconnections and user disconnections gracefully.

Supports namespaces and rooms, making it ideal for managing individual and group call sessions.

**WebRTC (Web Real-Time Communication):**

WebRTC allows the peer-to-peer connection necessary for transmitting audio and video in real-time, directly between browsers without the need for an intermediary server.

**Why use WebRTC?**

Real-time audio and video communication between browsers.

Peer-to-peer connection ensures low latency and reduces server load.

Secure and scalable, making it suitable for modern video communication apps.

When the user navigates to the Call Inbox, the system displays a list of the previous video call history with other users. The user can click on any of the entries to initiate a new video call. Once clicked, the system starts the call and sends a notification to the other user, redirecting them to the Call Page where the video call begins.During the video call ,a WebRTC peer connection is established between the two users.

