e-Hato

System Design Document

Table of Contents -

1. Project Overview

The chat application is designed to facilitate real-time communication between users. It includes features such as user registration and authentication, sending and receiving text messages, group chat functionality, real-time message updates, and an AI-powered chatbot using the Gemini API. The application uses Next.js for the frontend and Node.js with MongoDB for the backend, leveraging Pusher for real-time message delivery.

2. Features

1. User Authentication: Registration and login with validation.
2. Real-Time Messaging: Instant message delivery using Pusher.
3. Group Chats: Create group chats and manage participants.
4. Search Functionality: Search through chats and contacts.
5. Message Status Indicators: Show whether messages have been seen.
6. AI ChatBot - Chat with AI bot.

3. Architecture

3.1. Frontend

Framework: Next.js

UI Libraries:

- Material-UI for component styling and icons.

- Tailwind CSS for custom styling and responsive design.

State Management: React hooks for local component state and session management.

Real-Time Communication: Pusher.js for real-time event handling.

React Hot Toast : For styled notifications

Frontend Structure -

Pages:

- `/signup`: User Registration

- `/login`: User Login

- `/chats`: Main Chat Interface

- `/contacts`: Add Contacts and Create Chat

Main Components:

- `Form`: Handles user signup and login.

- `ChatList`: Displays list of user chats.

- `ChatDetails`: Shows the conversation in a selected chat.

- `Contacts`: Allows users to select contacts for new chats.

- `MessageBox`: Renders individual messages.

3.2. Backend

Framework: Node.js

Database: MongoDB (using Mongoose)

Real-Time Events: Pusher for managing real-time updates.

Backend Structure -

Controllers:

- `AuthController`: Handles authentication (signup, login).

- `ChatController`: Manages chat-related operations (fetching chats, creating chats).

- `MessageController`: Manages message-related operations (sending messages, fetching messages).

Services:

- `UserService`: Handles user-related logic.

- `ChatService`: Handles chat-related logic.

- `MessageService`: Handles message-related logic.

Middleware:

- Authentication middleware to protect routes.

3.3. Database Design

User Schema - Stores user details

```
username: {
  type: String,
  required: true,
},
email: {
  type: String,
  required: true,
  unique: true,
},
password: {
  type: String,
  required: true,
},
chats: {
  type: [{ type: mongoose.Schema.Types.ObjectId, ref: "Chat" }],
  default: [],
},
```

Chat Schema - Stores details of chats

Each chat is identified by a unique id.

```
members: {
  type: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
  default: [],
},
messages: {
  type: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Message' }],
  default: [],
},
isGroup: {
  type: Boolean,
  default: false,
},
name: {
  type: String,
  default: '',
},
createdAt: {
```

```
    type: Date,
    default: Date.now,
  },
  lastMessageAt: {
    type: Date,
    default: Date.now,
  },
});
```

Message Schema - Stores the messages of a particular chat

```
  chat: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Chat",
  },
  sender: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
  },
  text: {
    type: String,
    default: "",
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
  seenBy: {
    type: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
    default: [],
  },
});
```

4. User Interaction Flow -

1. User Registration/Login:
    - Users visit the registration or login page
    - Users enter their credentials and submit the form
    - Backend validates and processes the credentials.

2. Chat Interface:

- Upon successful login, users are redirected to the chat interface.
- The chat list is populated with existing chats.
- Users can select a chat to view messages.

3. Sending Messages:
   - Users can type messages in the input field and hit send.
   - Messages are sent to the server and stored in the database.
   - Pusher broadcasts the new message to all participants in the chat.

4. Creating Group Chats:
   - Users can select multiple contacts and create a group chat.
   - A group name is set, and the chat is saved in the database.

5. Data Flow

1. User Registration/Login:

   -Frontend sends user data to `/api/auth/signup` or `/api/auth/login`.
   - Backend processes the data, creating or validating the user.
   - Response sent back to frontend with session information.

2. Fetching Chats:
   - Frontend sends a request to fetch chats for the current user.
   - Backend retrieves chats from the database and returns them.

3. Real-Time Messaging:
   - When a user sends a message, the message is sent to `/api/messages`.
   - The backend saves the message in the database.
   - Pusher broadcasts the new message to all clients in that chat.

6. API Endpoints

Method	Endpoint	Description
POST	`/api/auth/signup`	Register a new user.

POST	`/api/auth/login`	Authenticate user login.
GET	`/api/users/:userId`	Fetch user details.
GET	`/api/users/:userId/chats`	Fetch chats for a user.
POST	`/api/chats`	Create a new chat.
GET	`/api/chats/:chatId`	Get chat details including messages.
POST	`/api/messages`	Send a new message.

7. Detailed Working

Connection with MongoDb using mongoose -

Function, `connectToDB` handles connection to MongoDB database using the Mongoose library.

1. `isConnected` Flag - A boolean variable `isConnected` is used to track whether a connection to the database has already been established. If the database is already connected, it prevents redundant connection attempts.

2. MongoDB Connection - The function configures Mongoose to use strict query mode by calling `mongoose.set("strictQuery", true)`. If not already connected, the function attempts to connect to MongoDB using the connection URL stored in the environment variable `MONGODB_URL`.

Log In -

The code uses CredentialsProvider from NextAuth to authenticate users based on their email and password.

When a user attempts to log in, the *authorize* function is called, which connects to the MongoDB database using the *connectToDB* function. It then finds the user in the `*User*` model based on the provided email. If the user is found, it compares the hashed password stored in the database with the provided password using `bcryptjs`. If the password matches, the user object is returned for a successful login; otherwise, an error is thrown.

Sign up -

First a connection to MongoDB database is made using `connectToDB`.

The incoming request (`req`) is parsed to extract the body, which contains the `username`, `email`, and `password` provided by the user.

Checking for Existing User - It checks the database to see if a user with the given email already exists using `User.findOne({ email })`. If an existing user is found, it returns a 400 status response, indicating that the user already exists.

If no user exists with the given email, the provided password is hashed using `bcryptjs` for secure storage. And a new user is created using the `User.create()` method, storing the hashed password along with the `username` and `email`. The user is then saved to the database.

Managing Chat details and messages -

Two API handlers, GET and POST are defined for managing chat details and message status.

The `GET` method is used to retrieve details of a specific chat based on the `chatId` from the request parameters. It first connects to the MongoDB database via `connectToDB`. The chat is fetched from the `Chat` model using `Chat.findById(chatId)`. The response includes- `members` (fetched from the `User` model), `messages` (fetched from the `Message` model). Each message's `sender` and the `seenBy` users are populated from the `User` model.

The `POST` method is used to mark all messages in a chat as "seen" by the current user. It connects to the database and extracts the `chatId` from the request parameters. The body of the request contains `currentUserId`, representing the user who has seen the messages. It updates all messages in the chat (`Message.updateMany`) by adding the `currentUserId` to the `seenBy` field using MongoDB's `$addToSet` operator. The `sender` and `seenBy` fields are populated with the corresponding user data from the `User` model.

Creating a new chat -

First a connection to the MongoDB database is established using the `connectToDB` function.

The incoming request body is parsed to extract the following fields:

- `currentUserId`: ID of the user initiating the chat.

- `members`: Other users involved in the chat.

- `isGroup`: Boolean indicating whether it's a group chat.

- `name`: The name of the chat (for group chats).

If it's a group chat (`isGroup` is true), the query is based on the `isGroup` flag, the chat name, and the members (including the current user). If it's a one-on-one chat, the query checks that the members include only two users (the `currentUserId` and the other member).

If the chat does not already exist, a new `Chat` is created and saved to the database. For group chats, the members and chat details are stored. For one-on-one chats, the two members are stored.

For each member in the chat, the `User` model is updated to include the chat in their list of chats (`$addToSet` ensures no duplicates).

After the chat is created, the Pusher service triggers a real-time event (`"new-chat"`) for each chat member to notify them of the new chat.

Creation of a new message within a chat and real time notification-

First a connection to MongoDB database is established using `connectToDB`.

The request body is parsed to extract:

- `chatId`: The ID of the chat where the message is being sent.

- `currentUserId`: The ID of the user sending the message.

- `text`: The message content.

The `currentUser` is fetched from the `User` model using `findById`.

A new `Message` is created and stored in the database with the following fields:

- `chat`: The chat ID.

- `sender`: The user sending the message.

- `text`: The message content.

- `seenBy`: The sender is automatically added to the list of users who have seen the message.

The chat is updated by:

- Adding the new message ID to the list of messages (`$push`).

- Updating the `lastMessageAt` timestamp to the message creation time.

- The updated chat is retrieved with its populated `messages` (including sender and seenBy) and `members`.

A Pusher event is triggered for the specific chat (`chatId`) to notify about the new message. Another Pusher event is triggered for each member of the chat to update the chat with the latest message.

Retrieving chats for a specific user -

The `userId` is extracted from the request parameters to identify the user for whom the chats will be fetched.

It queries the `Chat` model to find all chats that include the specified `userId` in the `members` field. The chats are sorted in descending order by `lastMessageAt`, ensuring that the most recent chats are listed first.

The query populates:

- `members`: The users participating in the chat from the `User` model.

- `messages`: All messages in the chat from the `Message` model.

- Additionally, it populates the `sender` and `seenBy` fields for each message, linking to users in the `User` model.

Chat Bot Implementation -

1. State Management

- `messages`: Stores the chat history (both user and bot messages).

- `userInput`: Stores the current text input from the user.

- `chat`: Stores the initialized chat instance from the AI model.

2. Google Generative AI (Gemini) Integration

- It uses the Google Generative AI API with the model `gemini-1.0-pro`..

- Chat initialization occurs on component mount (`useEffect`), setting up a chat session.

3. Message Handling

- Sending Messages: The user can send messages by typing and pressing "Enter" or clicking the "Send" button.

- User messages are added to the `messages` state.

- If the chat session is active, the AI generates a response which is also added to the `messages` state.

- Message Display: Messages are displayed with distinct styles for user and bot messages, including timestamps.

7. Security

Password Hashing: Passwords are hashed using bcrypt before storage to ensure security.

Session Management: User sessions are managed using `next-auth`, ensuring secure authentication.