Documentation of Gemini Chat with Backend to support for user chat history

Parts : of frontend
1) Welcome page :
2) First Page :
3) Chat Page :

1) Welcome Page
Purpose: This page serves as the entry point to **GeminiChat**, displaying a welcome message and a sign-in button.

- **HTML Structure**:
- Uses a container (.welcome-container) for layout.
- Contains an <h1> element for the welcome message.
- Includes an <a> tag styled as a button to redirect users to the sign-in page.
- **CSS Styling**: The page is styled using **styles.css**, which handles layout and design.
- **Navigation**: The **"Sign In"** button (<a> tag) links to "../firstPage/index.html".
- **Viewport**: The <meta viewport> tag ensures responsiveness on different screen sizes.
- **Charset**: Uses UTF-8 encoding for compatibility with various text formats.
- **External Stylesheet**: The page links to "styles.css" for styling.
- **File Structure**:
- index.html (current file)
- styles.css (stylesheet)
- firstPage/index.html (linked sign-in page)
- **Accessibility**: Uses semantic HTML (<h1>, <a>) to improve readability.
- **Future Enhancements**: Can add animations, a login form, or authentication integration.

2) First Page

1. Purpose: This page allows users to sign in using Google OAuth 2.0 and redirects them to the chat page after authentication.

2. HTML Structure

- Contains a to structure the sign-in UI.
- Displays a **Google Sign-In button** ( ).
- Uses Google OAuth 2.0 for authentication via the gsi/client library.

3. Google OAuth 2.0 Integration

- **Client ID** is set in the data-client_id attribute.
- The sign-in button automatically prompts users when needed (data-auto_prompt="false").
- The handleCredentialResponse(response) function processes the response.

4. Handling Authentication Response

- The handleCredentialResponse(response) function decodes the JWT token to extract user details.
- It logs user information (ID, Name, Email) to the console.
- Redirects users to chatpage.html with their userId as a query parameter.

5. JWT Parsing

- The parseJwt(token) function decodes the **JWT (JSON Web Token)** returned by Google OAuth.
- Extracts the **Base64-encoded payload** to retrieve user information.

6. Redirection After Login

- Once authenticated, the user is redirected to "../chatPage/chatpage.html" with their Google ID appended as a query parameter (?userId=...).

7. External Libraries

- Loads the **Google Identity Services (GIS) API** via:

```
<script src="https://accounts.google.com/gsi/client" async defer></script>
```

Ensures non-blocking script execution with async defer.

8. Styling

- Uses "styles.css" to control layout and appearance.

9. Security Considerations

- Ensures the **Google OAuth 2.0 client ID** is correctly configured.
- Uses JWT decoding to securely extract user details.

10. Future Enhancements

- Implement **session storage** to persist login state.
- Add a **logout** button to allow users to sign out.
- Improve **error handling** for invalid or expired credentials.

3) Chat Page

1. Purpose : This page allows users to engage in a chat conversation, where messages are saved and retrieved from a backend server. Additionally, it integrates with the Gemini API to provide AI-generated responses.

2. HTML Structure

- **Logout Button**: <button id="logout-btn">Logout</button> to allow users to log out.
- **Chat Container (.chat-container)**: Contains chat messages and an input box.
- **Chat Box (#chat-box)**: Displays conversation messages dynamically.
- **Input Box (#user-input)**: Text input field for user messages.
- **Send Button (#send-btn)**: Sends user messages and triggers Gemini responses.

3. User Authentication

- Extracts userId from the URL using URLSearchParams.
- If userId is missing, redirects the user to "index.html".

4. Chat Loading from Backend

- Function: loadChats(userId)
- Fetches past chat messages from http://localhost:5000/api/chats/{userId}.
- Calls addMessage(chat.content, chat.isUserMessage) to display messages.

5. Message Handling

- Function: addMessage(message, isUser)
- Creates a chat message and appends it to #chat-box.
- Calls formatResponse(text) to format AI responses (e.g., adding bullet points).
- Scrolls the chat box to the latest message.

6. Saving Messages to Backend

- Function: saveChat(userId, content, isUserMessage)
- Sends user and AI messages to http://localhost:5000/api/chats.
- Uses fetch with a POST request to store messages.

7. Fetching AI Responses (Gemini API Integration)

- Function: getGeminiResponse(userPrompt)
- Sends the user message to the Gemini API and retrieves a response.
- Extracts response content from data.candidates[0].content.parts[0].text.
- Handles errors gracefully, returning a default error message if the API call fails.

8. Event Handling

- Clicking #send-btn sends the user input, gets a Gemini response, and updates the chat.
- Pressing **Enter** in #user-input triggers the send button (keypress event).
- Clicking #logout-btn redirects the user to "../welcome/welcome.html".

9. Security Considerations

- Ensures only authenticated users can access the chat page by checking userId.
- Uses proper error handling for API failures (e.g., failing to fetch messages or AI responses).
- Avoids direct exposure of sensitive API keys by storing them in a secure backend instead of client-side.

10. Future Enhancements

- Implement **WebSockets** for real-time message updates.
- Store messages in **local storage** for offline access.
- Enhance UI with **message timestamps** and **user avatars**.
- Improve **error handling** and **authentication security**.

Backend Documentation

This backend is built using **Node.js**, **Express.js**, and **MongoDB**. It provides authentication via **Google OAuth**, allows users to store and retrieve chat messages, and enables communication between the frontend and the database.

```
/chat-app-backend
├── /config
│   ├── db.js              # MongoDB connection configuration
│
├── /models
│   ├── Chat.js            # Chat Schema & Model
│   ├── User.js            # User Schema & Model
│
├── /routes
│   ├── auth.js            # Authentication routes (Google OAuth)
│   ├── chat.js            # Chat-related API routes
│
├── /server.js             # Main server file
├── .env                   # Environment variables (Google OAuth, MongoDB URL)
├── package.json           # Dependencies and scripts
```

2. Database Models

2.1 Chat Model (Chat.js)

Defines the structure for storing chat messages in MongoDB.

Fields:

- userId *(String)* - Identifier of the user.
- content *(String)* - Message content.
- isUserMessage *(Boolean)* - Indicates if the message was sent by the user.
- timestamp *(Date)* - Time of message creation (default: current time).

2.2 User Model (User.js)

Defines the structure for storing user details in MongoDB.

Fields:

- googleId *(String, Unique)* - Google OAuth user ID.
- displayName *(String)* - User's name.
- chats *(Array of ObjectId)* - References to Chat documents.
- createdAt *(Date)* - User creation timestamp.

3. API Routes

3.1 Authentication Routes (auth.js)

Handles Google OAuth authentication.

Endpoints:

1. GET /api/auth/google - Redirects to Google OAuth login page.
2. GET /api/auth/google/callback - Handles Google login and stores user data.

3.2 Chat Routes (chat.js)

Handles chat message storage and retrieval.

Endpoints:

1. POST /api/chats - Saves a chat message.

- **Request Body:** { userId, content, isUserMessage }
- **Response:** { chat object }

2. GET /api/chats/:userId - Retrieves all chat messages for a user.
    - **Response:** [{ chat1 }, { chat2 }, ...]

4. Server Configuration (server.js)

This file initializes the server, connects to MongoDB, and sets up API routes.

Key Features:

- **CORS** enabled for frontend communication.
- **Google OAuth Strategy** for authentication.
- **MongoDB connection** via connectDB().
- **API Routing** for authentication and chat handling.

Running the backend

navigate to backend folder then
npm install
node index.js

Future Enhancements

- Implement **real-time chat** using WebSockets.
- Add **message deletion/editing** features.
- Improve **security** with authentication middleware.
- Optimize **database queries** for better performance

Running the project

go to GeminiChat main folder

use the command to run the server in port 8000 and navigate to welcome.html

python -m http.server 8000 to start the frontend

then start the backend

I intend to use react in future for this project so that project structure becomes more streamlined