Project Title: Real-Time Chat Application

By [Aaryan Keer]

**Table of Contents :**

1 **Introduction**

* Overview of the Project
* Objectives and Goals

2 **System Architecture**

* Architecture Diagram
* Explanation of the Architecture

3 **Solution Design**

* Low-Level Design
  + Component Diagrams
  + Data Models
  + API Details
  + UI/UX Designs
  + Detailed Design
* Security and Permissions
  + Authentication and Authorization Mechanisms
  + Firestore Security Rules

4 **Testing Strategy**

* Unit Testing
* Integration Testing
* End-to-End Testing

5 **Deployment**

* Deployment Steps and Environment

6 **Conclusion**

* Future Work and Enhancements

7 **Appendices and Code Listings**

* Code Listings
* Outputs

**Introduction**

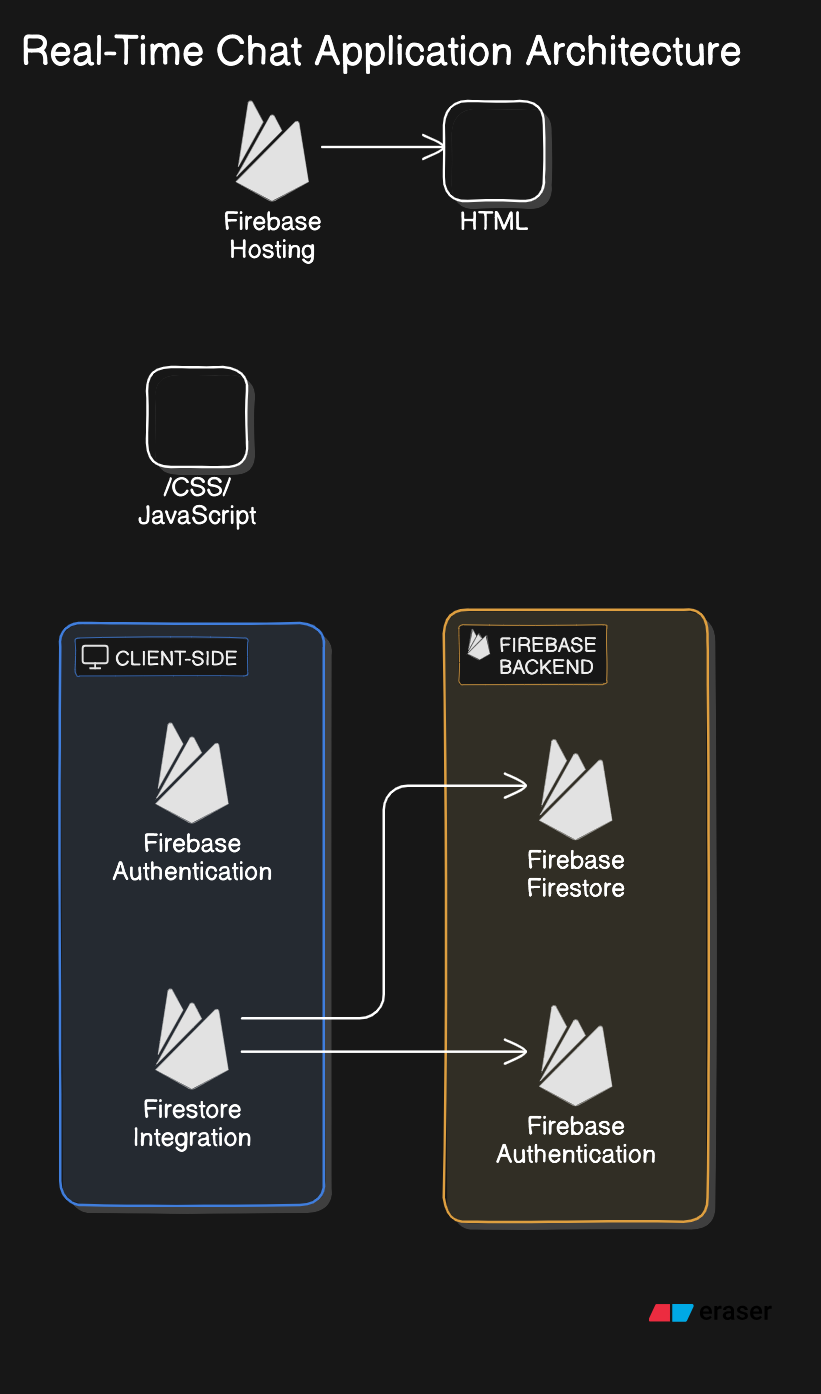
**Overview of the Project:**  
This project involves the development of a real-time web-based chat application designed to provide users with an engaging and interactive communication experience. The application allows users to join chat rooms, exchange messages instantly, and enjoy a seamless chatting environment. The front end of the application is built using HTML and CSS to create an intuitive and visually appealing user interface, while JavaScript handles the dynamic functionality and user interactions. The real-time communication between users is facilitated by WebSockets, ensuring that messages are sent and received instantly without the need for page refreshes.

**Objectives and Goals:**

* **User Interface:** The primary objective is to create a user-friendly and aesthetically pleasing interface that allows users to interact with the chat application effortlessly. The design is responsive, ensuring that it adapts well to various screen sizes, providing a consistent user experience across devices.
* **Real-Time Messaging:** Implementing real-time messaging capabilities is a key goal, enabling users to communicate instantly within chat rooms. The use of WebSockets is critical for achieving this functionality, as it allows for continuous, two-way communication between the client and server.
* **User Authentication:** A secure user authentication system is essential to ensure that users can create and manage their identities within the chat application. The system prevents the impersonation of users and ensures that usernames are unique and cannot be duplicated within the same chat room.
* **Room Management:** The application allows users to create new chat rooms or join existing ones, providing flexibility in communication and collaboration. Managing these chat rooms efficiently is a core objective, ensuring that users can easily navigate between rooms and participate in different conversations.

**System Architecture**

Architecture Diagram:

****

Explanation of the Architecture :

The architecture of the Real-Time Chat Application is designed to provide a seamless, interactive user experience with real-time communication, user authentication, and data storage, all managed through Firebase services. The system is divided into client-side and backend components, each playing a crucial role in the application's functionality.

Client-Side:

* The client-side of the application consists of HTML, CSS, and JavaScript files hosted on Firebase Hosting. This is the interface that users interact with when accessing the chat application through a web browser.
* Firebase Authentication on the client-side is responsible for managing user sign-up and login processes. This ensures that only authenticated users can access chat rooms and send messages.
* Firestore Integration in the JavaScript code enables real-time communication. When a user sends a message, it is immediately written to Firebase Firestore, a NoSQL database provided by Firebase. Firestore listens for changes and automatically updates the chat interface for all users in the same chat room, providing an instant messaging experience.

Firebase Backend:

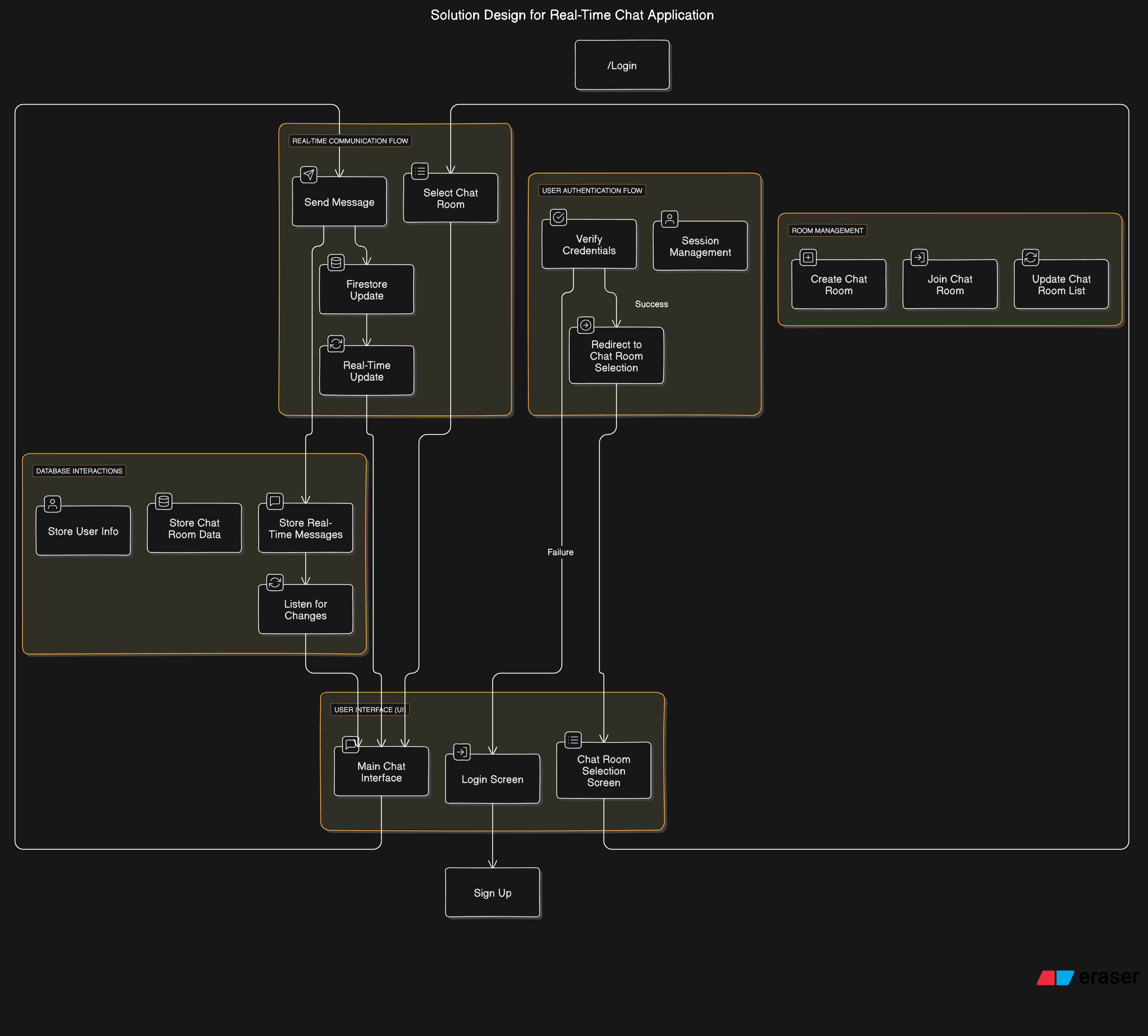
* On the backend, Firebase Authentication verifies the identity of users attempting to access the chat application. This authentication step is crucial for maintaining the security and integrity of user data and chat rooms.
* Firebase Firestore is the central repository for all chat-related data. It stores user information, chat messages, and room details. Firestore’s real-time database capabilities allow it to push updates instantly to all connected clients when new messages are added or users join or leave a chat room.

Data Flow:

* The data flow between the client-side and the Firebase backend is continuous and real-time. When a user sends a message, it is sent to Firestore, which then triggers real-time updates to all clients connected to that chat room. This interaction is bidirectional; updates from Firestore are instantly reflected on the client's interface, providing a smooth and interactive chat experience.

Overall, this architecture leverages Firebase's suite of tools to create a responsive and secure chat application that supports real-time communication, user authentication, and dynamic data management. The use of Firebase Hosting, Authentication, and Firestore ensures that the application is scalable, easy to maintain, and capable of handling multiple users and chat rooms simultaneously.

**Solution Design**

****

Explanation of the Solution Design

The solution design for the real-time chat application is structured to provide seamless communication between users through a secure, real-time messaging system. Below are the key components and their interactions as illustrated in the solution design diagram:

1. User Authentication Flow

* Login: The user starts by logging into the application. This step is crucial for ensuring that only authorized users can access the chat functionalities.
* Session Management: After successful login, the system manages the user's session, maintaining their login state and allowing them to navigate the application without needing to re-authenticate.
* Redirect on Success/Failure: If authentication is successful, the user is redirected to the chat room selection screen. If authentication fails, appropriate error messages are displayed, and the user is prompted to try again.

2. Chat Room Management

* Create Chat Room: Users with appropriate permissions can create new chat rooms. This involves defining the room name and any associated settings.
* Join Chat Room: Users can join existing chat rooms, allowing them to participate in conversations within that room.
* Update Chat Room List: The list of available chat rooms is dynamically updated, ensuring users always have access to the latest room options.

3. Real-Time Communication Flow

* Select Chat Room: After joining a chat room, users select the specific room they wish to enter. This step establishes the context for their chat session.
* Send Message: Users can send messages within the selected chat room. Messages are instantly transmitted to the Firestore database.
* Firestore Update & Real-Time Update: As messages are sent, Firestore updates its records in real-time. The updates are pushed to all users in the chat room, ensuring everyone sees the latest messages without delay.

4. Database Interactions

* Store User Info: User information, including profiles and login details, is securely stored in Firestore. This data is essential for managing user sessions and ensuring personalized experiences.
* Store Chat Room Data: Chat rooms and their associated metadata (e.g., participants, messages) are stored in Firestore. This setup allows for persistent chat histories that users can revisit.
* Store Real-Time Messages: Each message sent in a chat room is recorded in Firestore. The database ensures these messages are available to all users in the chat room in real-time.
* Listen for Changes: The system continuously listens for changes in the Firestore database, such as new messages or updates to chat rooms. This real-time listening ensures that the user interface is always up-to-date.

5. User Interface Flow

* Login Screen: This is the first interface users interact with. It includes fields for entering login credentials and options for navigating to other parts of the application.
* Main Chat Interface: After successful login, users are presented with the main chat interface. This screen shows available chat rooms and recent messages, providing users with easy access to ongoing conversations.
* Chat Room Detail Screen: When a user enters a specific chat room, they are taken to a detailed view of that room. Here, they can see the chat history and send new messages.

6. Sign Up

* Users who don’t have an account can sign up, creating new user credentials and storing their information in Firestore. This process integrates seamlessly with the authentication flow, ensuring new users can quickly get started with the application.

Summary

The solution design ensures a robust, secure, and real-time communication platform. The interaction between Firebase Authentication and Firestore allows for a seamless user experience, with real-time updates and reliable data management. The architecture supports scalability, ensuring the application can handle an increasing number of users and chat rooms efficiently.

Testing Strategy

The testing strategy for the real-time chat application was designed to ensure that each component and the overall system function as intended. The testing process was iterative and involved multiple stages of verification, both during development and after the completion of individual features and the entire application. Below are the key aspects of the testing strategy:

1. Unit Testing

* Component-Level Testing: Each individual page (e.g., login, chat room, signup) was developed and tested separately. During the development of each page, unit tests were conducted to identify and resolve any issues. This step ensured that each component of the application worked correctly in isolation.
* Error Resolution Cycle: After identifying errors during unit testing, the issues were promptly resolved. The process of coding, testing, and error resolution was repeated until each component functioned without issues.

2. Integration Testing

* Page Integration: Once a page was fully developed and free of errors, it was integrated with other existing pages. Integration testing was conducted to ensure that the individual pages worked together seamlessly, with proper communication between different components such as authentication, database interactions, and real-time updates.
* Data Flow Verification: During integration testing, special attention was given to verifying the data flow between client-side components (e.g., user interface) and backend services (e.g., Firestore). This ensured that user data, messages, and chat room details were correctly transmitted and displayed across different parts of the application.

3. System Testing

* Full System Testing: After all components and pages were integrated, comprehensive system testing was performed. This involved running the entire application as a user would, navigating through different pages, and performing typical user actions such as logging in, sending messages, and creating chat rooms.
* End-to-End Scenarios: The system was tested using end-to-end scenarios to simulate real-world use cases. This included scenarios such as a new user signing up, joining a chat room, participating in conversations, and logging out. Each scenario was tested multiple times to ensure reliability.

4. Regression Testing

* Continuous Testing: Throughout the development process, regression testing was performed continuously. Whenever a new feature was added or an existing component was modified, the entire system was tested again to ensure that no new errors were introduced. This helped maintain the stability of the application over time.
* Error Resolution and Re-Testing: Any errors identified during regression testing were resolved, and the affected components were re-tested to confirm that the issues were fixed without impacting other parts of the application.

5. User Acceptance Testing (UAT)

* Real User Simulation: The final stage of testing involved simulating real user interactions to ensure that the application met user expectations. This included testing the user interface, responsiveness, and overall user experience.
* Feedback Incorporation: Any feedback received during UAT was used to make final adjustments to the application, ensuring that the product delivered a high-quality user experience.

Summary

The testing strategy for the real-time chat application was thorough and iterative, with a focus on identifying and resolving issues at each stage of development. By conducting unit, integration, system, and regression testing, and involving real user scenarios in the final stages, the application was rigorously tested to ensure it met all functional and non-functional requirements.

**Deployment**

Deployment Steps and Environment

Deploying the real-time chat application involved several key steps, from setting up the Firebase project to configuring the environment and deploying the web application. Below is a detailed explanation of the deployment process:

1. Setting Up Firebase Project:

* Created a new project on the Firebase console.
* Enabled necessary Firebase services: Firebase Authentication, Firestore Database, and Firebase Hosting.

2. Firebase Configuration:

* Obtained Firebase configuration details.
* Included Firebase configuration in the web application's codebase.

3. Initializing Firebase in the Command Line:

* Installed Firebase CLI using npm install -g firebase-tools.
* Logged into Firebase using firebase login.
* Initialized Firebase project using firebase init.

4. Deploying the Application:

* Built the project (if necessary).
* Deployed to Firebase Hosting using firebase deploy.

5. Testing and Finalizing Deployment:

* Accessed the deployed application.
* Conducted thorough testing.
* Monitored and maintained the application using Firebase console.

Summary

The deployment of the real-time chat application was a structured process that involved setting up a Firebase project, configuring the environment, and deploying the application using Firebase Hosting. By following these steps, the application was successfully hosted and made available for users on the web, with Firebase providing robust backend services to manage authentication, database interactions, and real-time updates.

**Conclusion**

The real-time chat application developed for this project successfully integrates modern web technologies with Firebase's powerful backend services, creating a robust and responsive communication platform. Through the implementation of Firebase Authentication, Firestore Database, and Firebase Hosting, the application offers secure user management, efficient data storage, and real-time messaging capabilities. The structured approach to development, testing, and deployment ensured that each component of the application was meticulously crafted and thoroughly validated, resulting in a seamless and user-friendly experience.

In conclusion, this project demonstrates the potential of using Firebase as a comprehensive solution for building scalable and real-time web applications. The deployment process, which involved setting up a Firebase project, configuring necessary services, and utilizing Firebase CLI, was straightforward and effective, allowing the application to be quickly and reliably hosted. This real-time chat application not only meets the objectives set out at the beginning of the project but also provides a strong foundation for future enhancements, such as adding more features or scaling to support a larger user base. Overall, the project is a successful demonstration of leveraging cloud-based technologies to deliver a fully functional and dynamic web application.

**Future Work and Enhancements**

The current implementation of the real-time chat application provides a solid foundation for basic messaging functionality, but there are numerous opportunities for future enhancements and additional features that could significantly enrich the user experience.

1. **Emojis and Stickers**: Integrating a rich set of emojis and stickers would allow users to express themselves more creatively and effectively during conversations. This feature can be implemented by adding emoji and sticker picker options in the chat interface, with the media stored and retrieved from Firebase Storage.
2. **User Status**: Implementing a user status feature would enable users to set their availability (e.g., online, offline, busy, away), enhancing the real-time nature of the application. This could be managed using Firestore to track and display the status of each user in real time.
3. **Typing Indicators**: Adding typing indicators to show when another user is typing a message would improve the chat experience by making conversations feel more immediate and responsive. This can be implemented using Firestore to track typing activity and update the chat interface in real time.
4. **Read Receipts**: Implementing read receipts, which indicate when a message has been read by the recipient, would add another layer of communication transparency. This feature could be managed using Firestore to store and track the read status of messages.
5. **File Sharing**: Introducing file-sharing capabilities would allow users to send and receive documents, images, videos, and other media directly within the chat. This can be accomplished by integrating Firebase Storage for uploading and retrieving files, and Firestore for managing file metadata.
6. **Group Chat**: Expanding the chat functionality to support group conversations would enable users to create and participate in group discussions. This would involve managing group members and messages within Firestore, and possibly enhancing the user interface to handle multiple participants.
7. **Push Notifications**: Adding push notifications to alert users of new messages when they are not actively using the app would ensure they stay engaged. This can be implemented using Firebase Cloud Messaging (FCM) to send notifications to users' devices.
8. **Customizable Chat Themes**: Allowing users to customize the chat interface with different themes or color schemes could make the application more personalized and visually appealing. This can be achieved by storing user preferences in Firestore and applying them dynamically during runtime.
9. **Search Functionality**: Adding a search feature that allows users to quickly find past messages or conversations would enhance usability. This could be implemented by indexing chat messages in Firestore and providing a search interface to query them.

By implementing these enhancements, the real-time chat application can evolve into a more feature-rich platform that meets a wider range of user needs and preferences, while still leveraging the scalability and reliability of Firebase services.

**Appendices and Code Listings**

Index.html:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Welcome to ChatApp</title>

    <link rel="stylesheet" href="css/style.css">

</head>

<body>

    <header>

        <h1>Welcome to ChatApp</h1>

        <nav>

            <ul class="nav-links">

                <li><a href="signup.html">Sign Up</a></li>

                <li><a href="login.html">Login</a></li>

            </ul>

        </nav>

    </header>

    <main>

        <section class="welcome-section">

            <h2>Join or Log In to Start Chatting</h2>

            <p>Select an option above to sign up or log in.</p>

        </section>

    </main>

    <footer>

        <p>&copy; 2024 ChatApp. All rights reserved.</p>

    </footer>

</body>

</html>

Signup.html :  
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Sign Up</title>

    <link rel="stylesheet" href="css/style.css">

</head>

<body>

    <header>

        <h1>Sign Up for ChatApp</h1>

    </header>

    <main>

        <section class="signup-container">

            <form id="signup-form">

                <label for="username">Username:</label>

                <input type="text" id="username" required>

                <label for="email">Email:</label>

                <input type="email" id="email" required>

                <label for="password">Password:</label>

                <input type="password" id="password" required>

                <button type="submit">Sign Up</button>

            </form>

            <p>Already have an account? <a href="login.html">Login</a></p>

        </section>

    </main>

    <footer>

        <p>&copy; 2024 ChatApp. All rights reserved.</p>

    </footer>

    <!-- Firebase Signup Script -->

    <script type="module">

        import { initializeApp } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-app.js";

        import { getAuth, createUserWithEmailAndPassword } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-auth.js";

        import { getFirestore, doc, setDoc } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-firestore.js";

        import { getAnalytics } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-analytics.js";

        // Your web app's Firebase configuration

        const firebaseConfig = {

apiKey: "YOUR\_API\_KEY",

authDomain: "YOUR\_AUTH\_DOMAIN",

projectId: "YOUR\_PROJECT\_ID",

storageBucket: "YOUR\_STORAGE\_BUCKET",

messagingSenderId: "YOUR\_MESSAGING\_SENDER\_ID",

appId: "YOUR\_APP\_ID",

measurementId: "YOUR\_MEASUREMENT\_ID"

};

        // Initialize Firebase

        const app = initializeApp(firebaseConfig);

        const analytics = getAnalytics(app);

        const auth = getAuth(app);

        const db = getFirestore(app);

        const signupForm = document.getElementById('signup-form');

        signupForm.addEventListener('submit', async (e) => {

            e.preventDefault();

            // Get the user's input

            const username = document.getElementById('username').value;

            const email = document.getElementById('email').value;

            const password = document.getElementById('password').value;

            try {

                // Firebase Authentication: Create a new user

                const userCredential = await createUserWithEmailAndPassword(auth, email, password);

                const user = userCredential.user;

                // Save user info to Firestore

                await setDoc(doc(db, "users", user.uid), {

                    username: username,

                    email: email,

                    uid: user.uid

                });

                alert('Sign up successful! Welcome, ' + username);

                // Redirect to chatroom page

                window.location.href = 'chatroom.html';

            } catch (error) {

                alert('Error: ' + error.message);

            }

        });

    </script>

</body>

</html>

Login.html :  
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Login</title>

    <link rel="stylesheet" href="css/style.css">

</head>

<body>

    <header>

        <h1>Login to ChatApp</h1>

    </header>

    <main>

        <section class="login-container">

            <form id="login-form">

                <label for="email">Email:</label>

                <input type="email" id="email" required>

                <label for="password">Password:</label>

                <input type="password" id="password" required>

                <button type="submit">Login</button>

            </form>

            <p>Don't have an account? <a href="signup.html">Sign up</a></p>

        </section>

    </main>

    <footer>

        <p>&copy; 2024 ChatApp. All rights reserved.</p>

    </footer>

    <!-- Include the Firebase config and login script -->

    <script type="module">

        import { initializeApp } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-app.js";

        import { getAuth, signInWithEmailAndPassword } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-auth.js";

        // Your web app's Firebase configuration

        const firebaseConfig = {

apiKey: "YOUR\_API\_KEY",

authDomain: "YOUR\_AUTH\_DOMAIN",

projectId: "YOUR\_PROJECT\_ID",

storageBucket: "YOUR\_STORAGE\_BUCKET",

messagingSenderId: "YOUR\_MESSAGING\_SENDER\_ID",

appId: "YOUR\_APP\_ID",

measurementId: "YOUR\_MEASUREMENT\_ID"

};

        // Initialize Firebase

        const app = initializeApp(firebaseConfig);

        const auth = getAuth(app);

        // Handle the form submission

        const loginForm = document.getElementById('login-form');

        loginForm.addEventListener('submit', async (e) => {

            e.preventDefault();

            const email = document.getElementById('email').value;

            const password = document.getElementById('password').value;

            try {

                // Sign in the user

                const userCredential = await signInWithEmailAndPassword(auth, email, password);

                const user = userCredential.user;

                // Redirect to the chatroom page after successful login

                window.location.href = 'chatroom.html';

            } catch (error) {

                console.error('Error logging in:', error);

                alert('Failed to log in. Please check your credentials and try again.');

            }

        });

    </script>

</body>

</html>

chatroom.html :  
<<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Chat Rooms</title>

    <link rel="stylesheet" href="css/style.css">

</head>

<body>

    <header>

        <h1>Chat Rooms</h1>

        <button id="logout-btn">Logout</button>

    </header>

    <main>

        <section class="room-selection">

            <h2>Select a Chat Room</h2>

            <ul id="room-list">

                <!-- Chat rooms will be dynamically populated here -->

            </ul>

            <button id="create-room-btn">Create New Room</button>

        </section>

        <section id="chat-section" style="display:none;">

            <h2 id="room-name">Room Name</h2>

            <div id="chat-messages" class="chat-container">

                <!-- Messages will be dynamically populated here -->

            </div>

            <div id="user-status"></div>

            <form id="message-form" class="message-input">

                <input type="text" id="message-input" placeholder="Type your message..." required>

                <button type="submit" id="send-btn">Send</button>

            </form>

        </section>

    </main>

    <script type="module">

        import { initializeApp } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-app.js";

        import { getAuth, onAuthStateChanged, signOut } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-auth.js";

        import { getFirestore, collection, addDoc, getDocs, onSnapshot, doc, getDoc, setDoc, deleteDoc } from "https://www.gstatic.com/firebasejs/10.13.0/firebase-firestore.js";

        // Firebase config

        const firebaseConfig = {

apiKey: "YOUR\_API\_KEY",

authDomain: "YOUR\_AUTH\_DOMAIN",

projectId: "YOUR\_PROJECT\_ID",

storageBucket: "YOUR\_STORAGE\_BUCKET",

messagingSenderId: "YOUR\_MESSAGING\_SENDER\_ID",

appId: "YOUR\_APP\_ID",

measurementId: "YOUR\_MEASUREMENT\_ID"

};

        // Initialize Firebase

        const app = initializeApp(firebaseConfig);

        const auth = getAuth(app);

        const db = getFirestore(app);

        let currentRoomId = null;

        // Track auth state

        onAuthStateChanged(auth, (user) => {

            if (user) {

                // Load chat rooms

                loadChatRooms();

                // Add user to presence when they join

                if (currentRoomId) {

                    updatePresence(true);

                }

            } else {

                window.location.href = 'login.html';

            }

        });

        // Load chat rooms from Firestore

        async function loadChatRooms() {

            const roomList = document.getElementById('room-list');

            const roomsSnapshot = await getDocs(collection(db, "chatRooms"));

            roomList.innerHTML = ''; // Clear existing list

            roomsSnapshot.forEach((doc) => {

                const room = doc.data();

                const li = document.createElement('li');

                li.textContent = room.name;

                li.addEventListener('click', () => joinChatRoom(doc.id, room.name));

                roomList.appendChild(li);

            });

        }

        // Join a chat room

        function joinChatRoom(roomId, roomName) {

            document.getElementById('chat-section').style.display = 'block';

            document.getElementById('room-name').textContent = roomName;

            document.getElementById('room-name').setAttribute('data-room-id', roomId);

            currentRoomId = roomId;

            loadMessages(roomId);

            // Update presence

            updatePresence(true);

            // Listen to presence changes

            listenToPresenceChanges();

        }

        function loadMessages(roomId) {

            const chatMessages = document.getElementById('chat-messages');

            const messagesRef = collection(db, "chatRooms", roomId, "messages");

            // Listen to messages in real-time

            onSnapshot(messagesRef, (snapshot) => {

                chatMessages.innerHTML = ''; // Clear existing messages

                snapshot.forEach((doc) => {

                    const message = doc.data();

                    const p = document.createElement('p');

                    p.classList.add('message');

                    if (auth.currentUser && message.username === auth.currentUser.displayName) {

                        p.classList.add('sent');

                    } else {

                        p.classList.add('received');

                    }

                    p.textContent = `${message.username}: ${message.text}`;

                    chatMessages.appendChild(p);

                });

                chatMessages.scrollTop = chatMessages.scrollHeight; // Auto-scroll to the bottom

            });

        }

        // Listen to presence changes

        function listenToPresenceChanges() {

            if (currentRoomId) {

                const presenceRef = collection(db, "chatRooms", currentRoomId, "presence");

                onSnapshot(presenceRef, (snapshot) => {

                    const usersList = document.createElement('ul');

                    snapshot.forEach((doc) => {

                        const user = doc.data();

                        const li = document.createElement('li');

                        li.textContent = `${user.username} (Online)`;

                        usersList.appendChild(li);

                    });

                    const userStatus = document.getElementById('user-status');

                    userStatus.innerHTML = '';

                    userStatus.appendChild(usersList);

                });

            }

        }

        // Send a message

        const messageForm = document.getElementById('message-form');

        messageForm.addEventListener('submit', async (e) => {

            e.preventDefault();

            const messageInput = document.getElementById('message-input');

            const user = auth.currentUser;

            if (user) {

                const userRef = doc(db, "users", user.uid);

                const userDoc = await getDoc(userRef);

                const roomId = document.getElementById('room-name').getAttribute('data-room-id');

                if (roomId && userDoc.exists()) {

                    try {

                        await addDoc(collection(db, "chatRooms", roomId, "messages"), {

                            username: userDoc.data().username,

                            text: messageInput.value,

                            timestamp: new Date()

                        });

                        messageInput.value = ''; // Clear input

                    } catch (error) {

                        console.error('Error sending message:', error);

                        alert('Failed to send message. Please try again.');

                    }

                }

            } else {

                alert('You need to be logged in to send a message.');

            }

        });

        // Handle presence on window close

        window.addEventListener('beforeunload', () => {

            if (currentRoomId) {

                updatePresence(false);

            }

        });

        // Create a new chat room

        document.getElementById('create-room-btn').addEventListener('click', async () => {

            const roomName = prompt('Enter room name:');

            if (roomName) {

                const docRef = await addDoc(collection(db, "chatRooms"), {

                    name: roomName

                });

                joinChatRoom(docRef.id, roomName);

            }

        });

        // Logout functionality

        document.getElementById('logout-btn').addEventListener('click', () => {

            if (currentRoomId) {

                updatePresence(false).then(() => {

                    signOut(auth).then(() => {

                        window.location.href = 'login.html';

                    }).catch((error) => {

                        alert('Error: ' + error.message);

                    });

                }).catch((error) => {

                    console.error('Error updating presence:', error);

                });

            } else {

                signOut(auth).then(() => {

                    window.location.href = 'login.html';

                }).catch((error) => {

                    alert('Error: ' + error.message);

                });

            }

        });

        // Update user presence in a chat room

        async function updatePresence(isOnline) {

            const user = auth.currentUser;

            if (user && currentRoomId) {

                const presenceRef = doc(db, "chatRooms", currentRoomId, "presence", user.uid);

                try {

                    if (isOnline) {

                        await setDoc(presenceRef, {

                            username: (await getDoc(doc(db, "users", user.uid))).data().username,

                            online: true,

                            lastOnline: new Date()  // Optional: Track last online time

                        });

                    } else {

                        await deleteDoc(presenceRef);

                    }

                } catch (error) {

                    console.error('Error updating presence:', error);

                }

            }

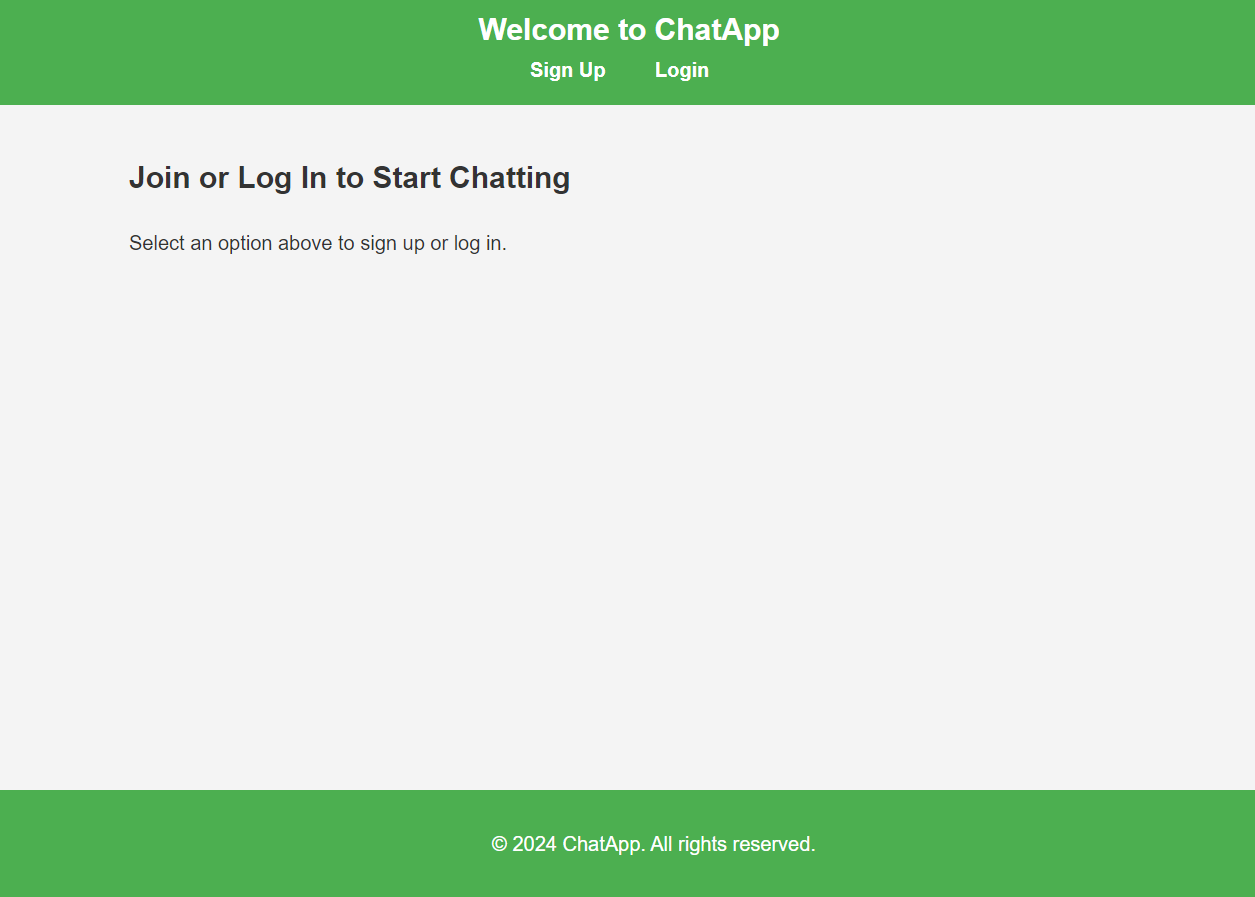
        }

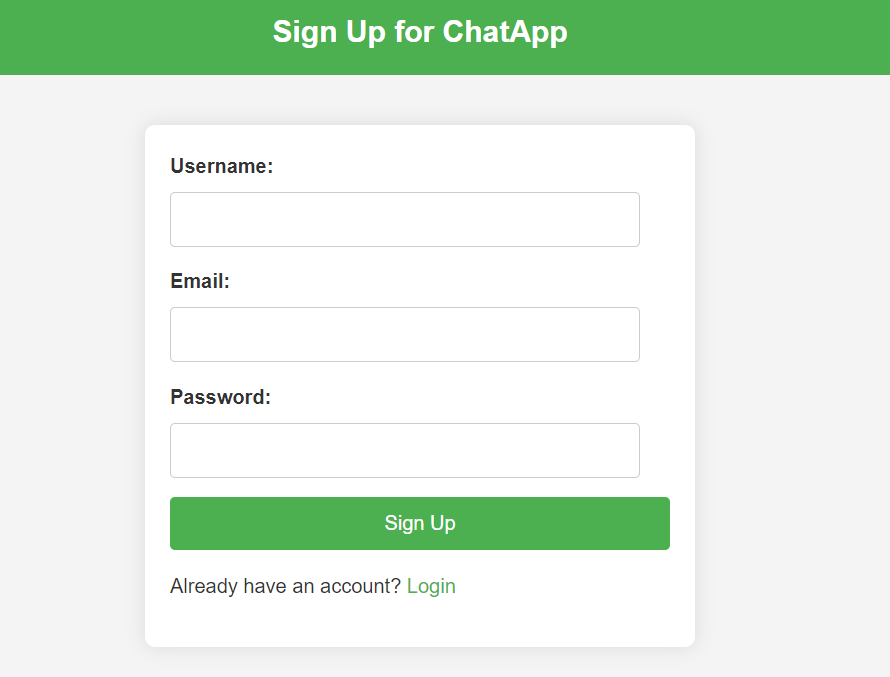
    </script>

</body>

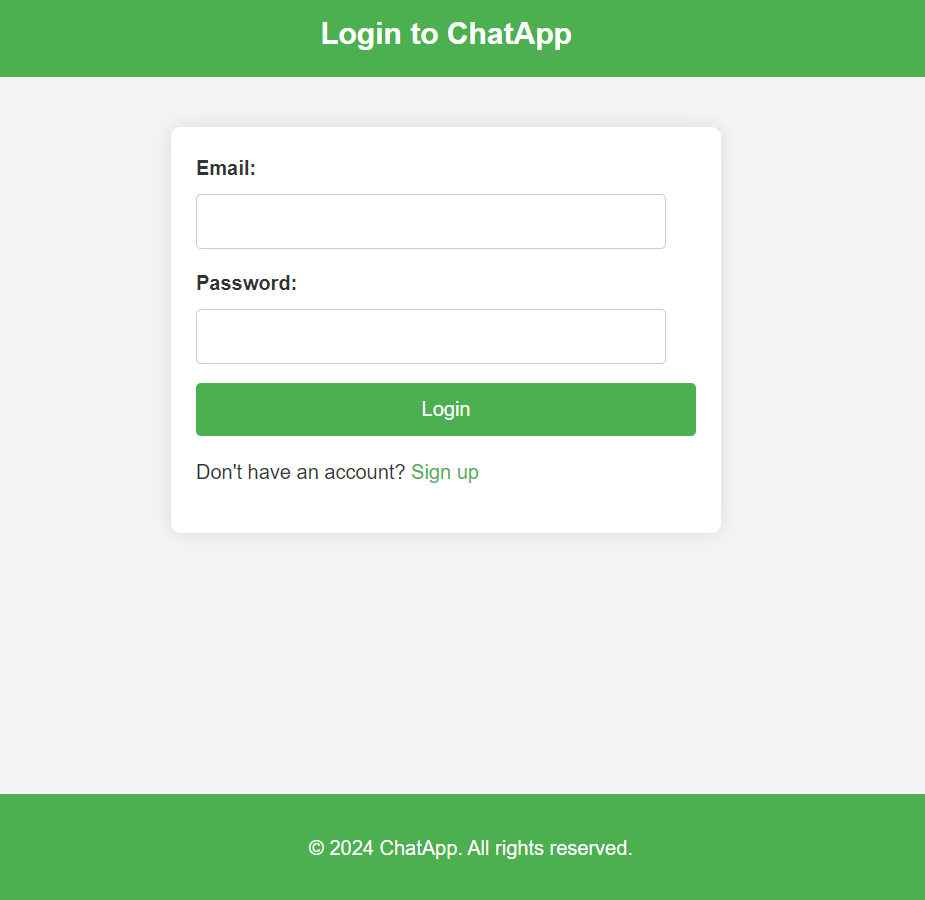
</html>

**Outputs**

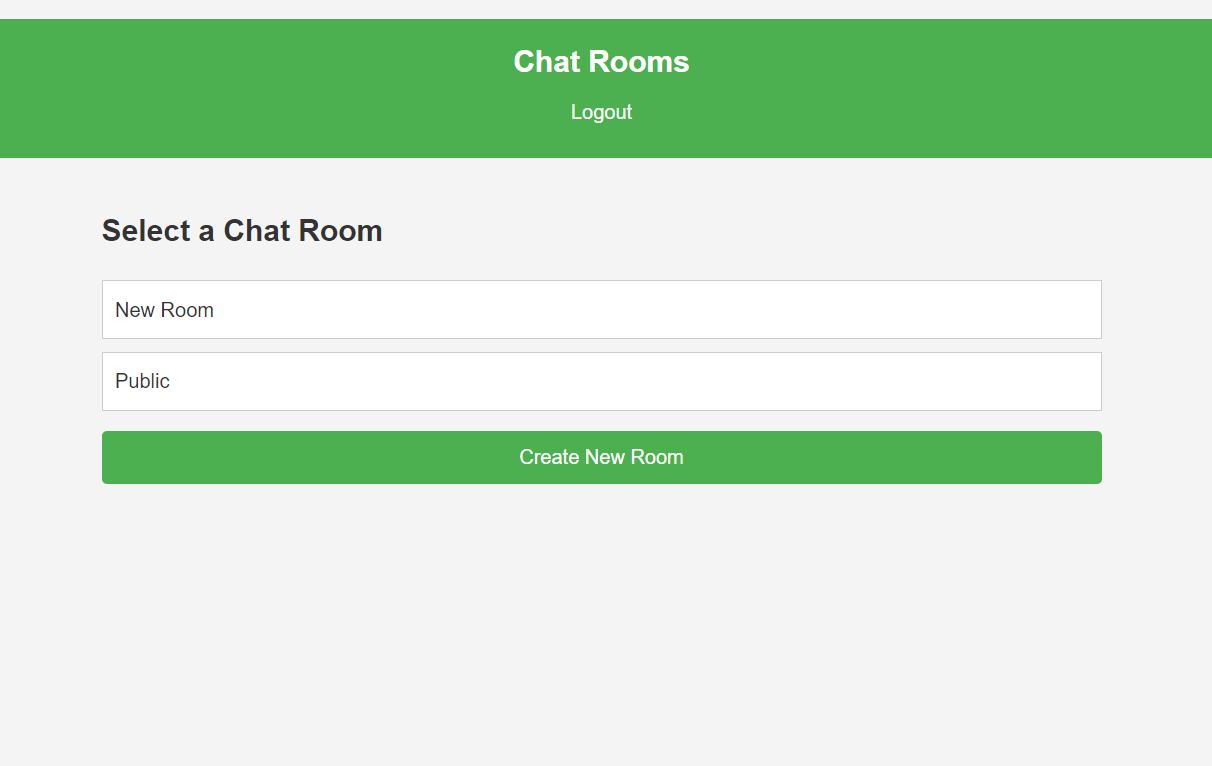
Index.html page :  
  
  
signup.html page :



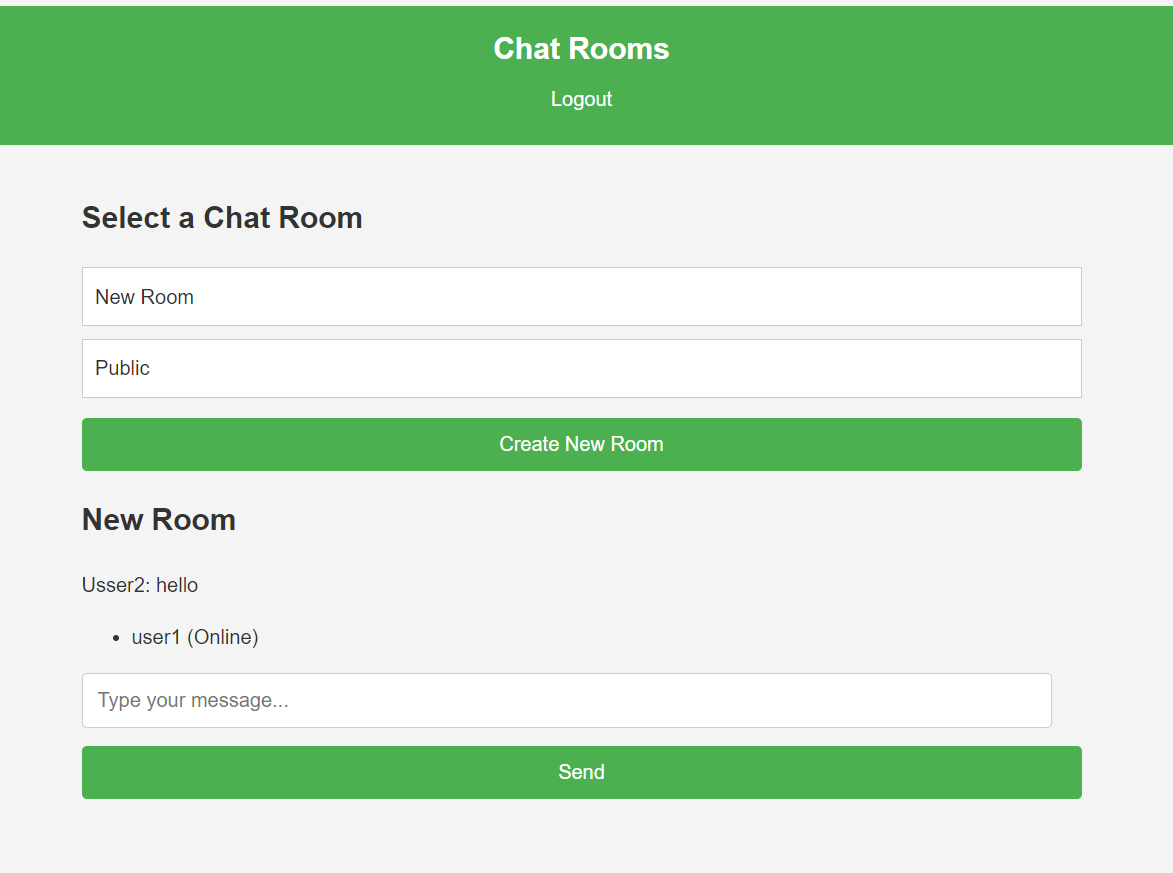
Login.html page :

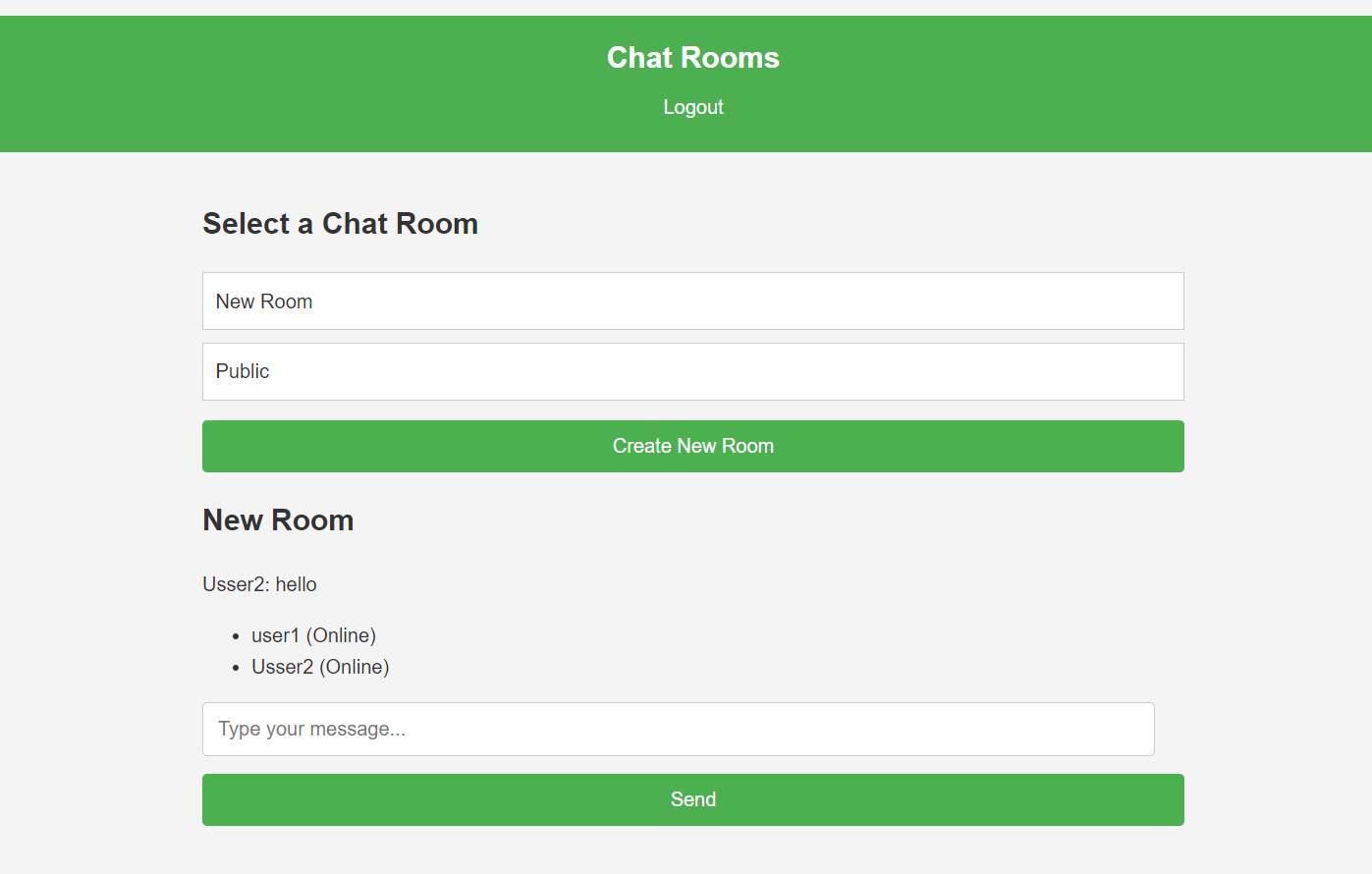


Chatroom.html page :

Shows created chatting rooms  


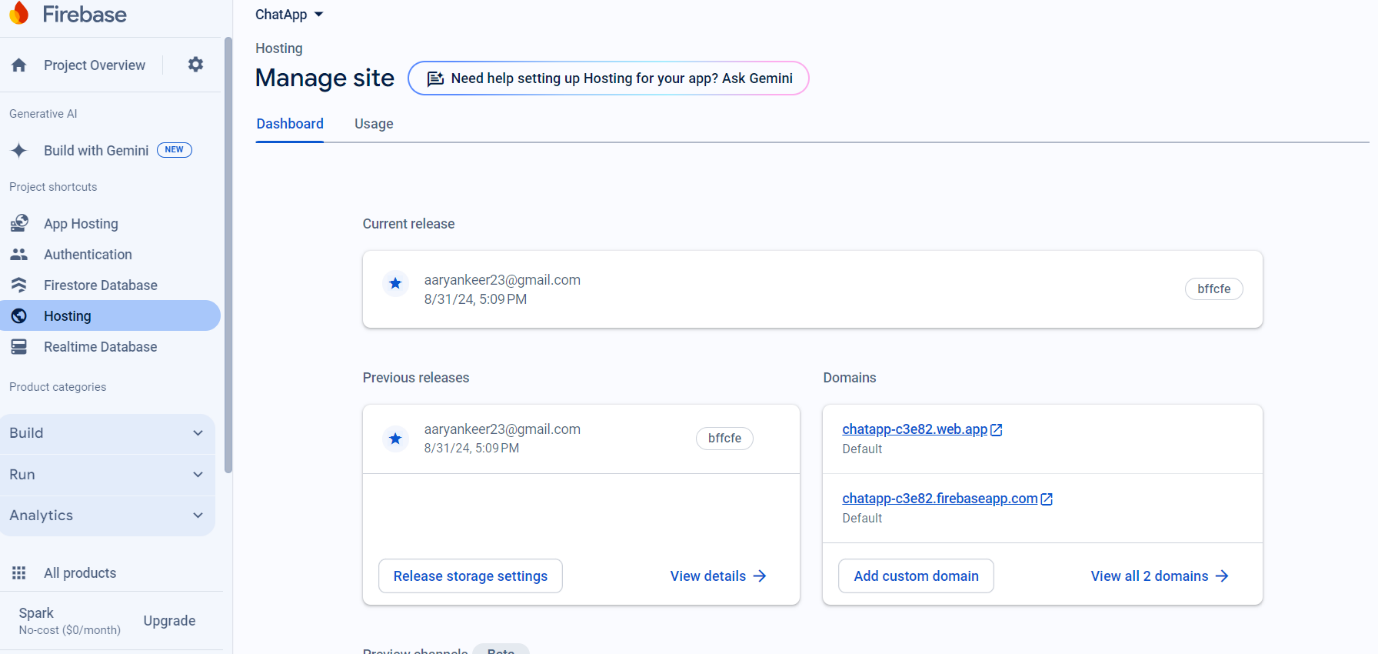
Selected chatting rooms



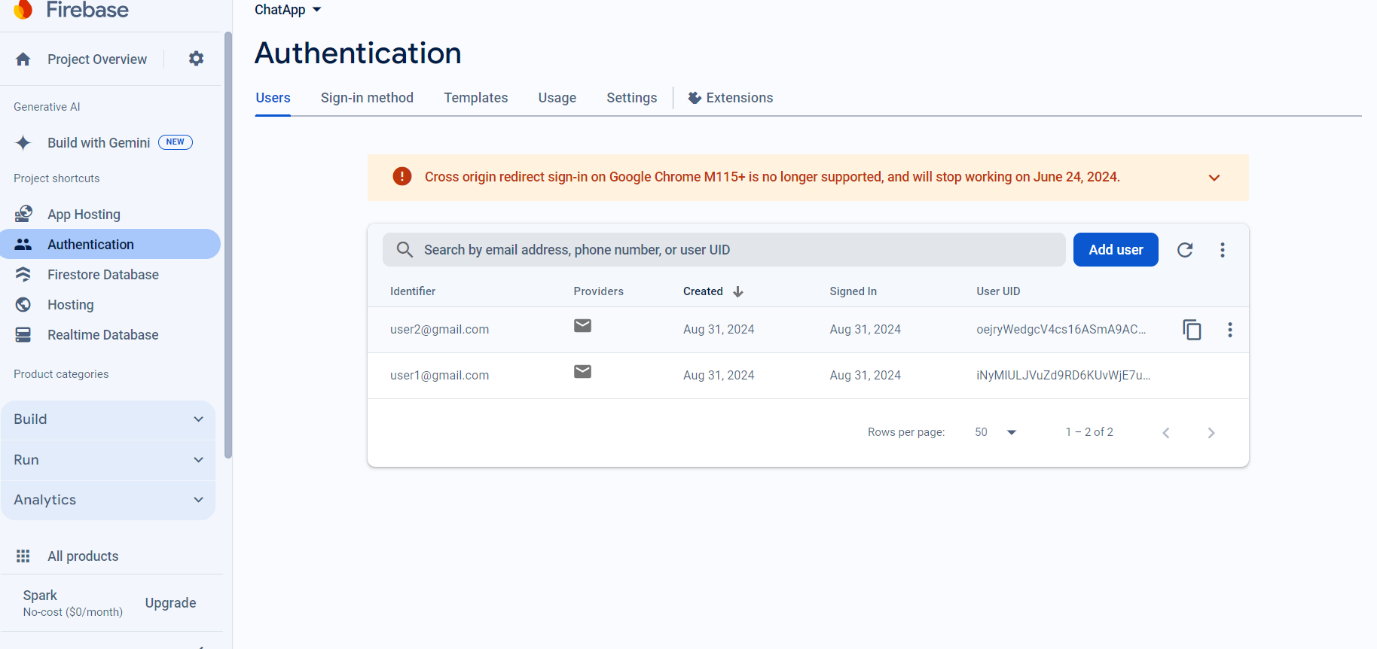


Firebase Backend of Real-Time Chat Application :

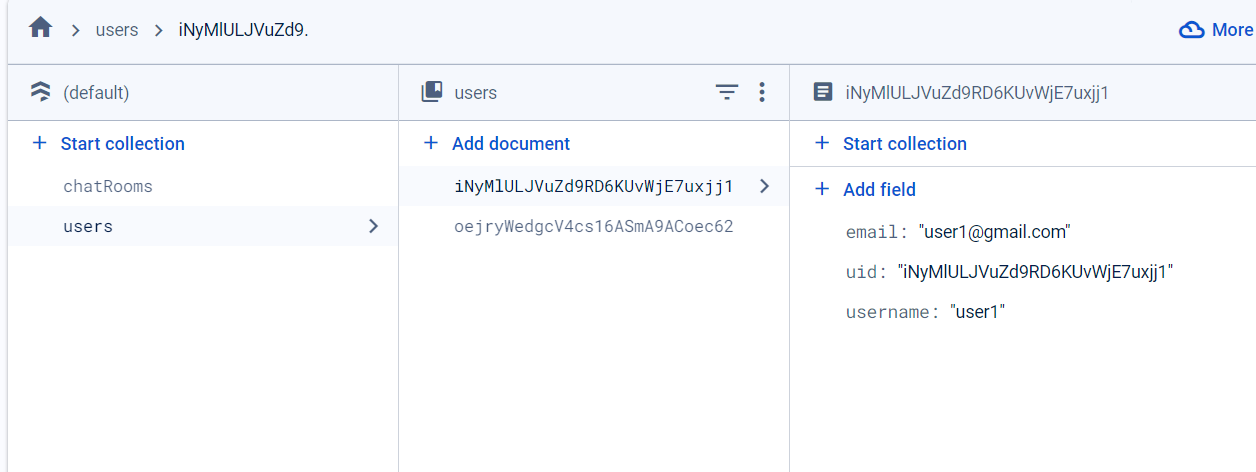
Shows website is hosted



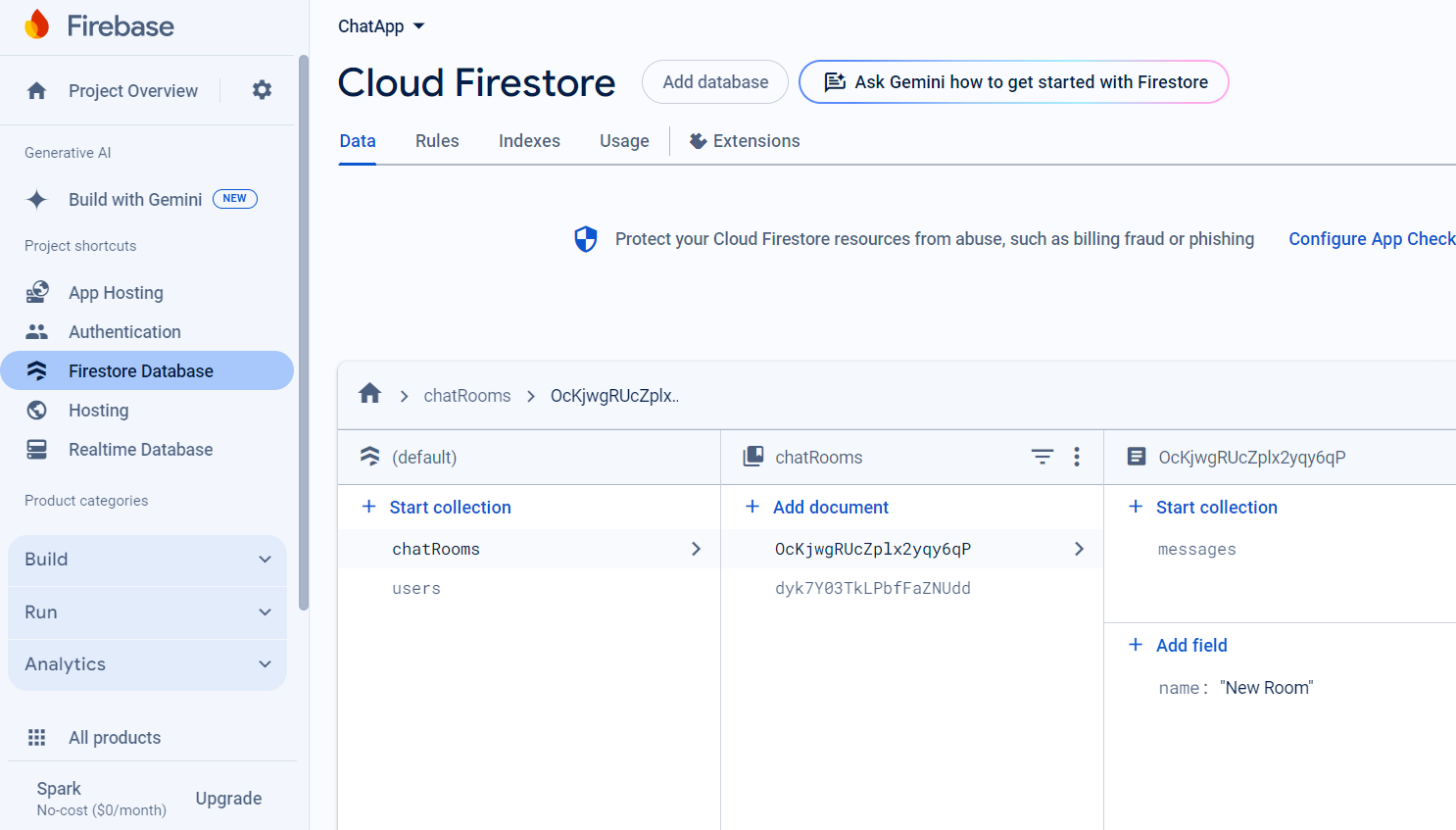
All signed in user



User info is stored inside this collection in firestore db



This is all chatrooms created as a document



Each chatroom document has their own message collections

