

MOBILE COMPUTING FINAL PROJECT FLUTTER CHAT APP

TEAM MEMBERS:

- **Abdelrahman Ihab Shafie
2206183**
- **Mohamed Ahmed Mohamed
Kamel 22011683**
- **Mohamed Ashraf 2206196**
- **Aliaa Omar 2206187**
- **Nada Magdy 2206170**

[**Github Repo Link**](#)

1. Introduction

The chat application is a real-time mobile messaging application built using **Flutter** as the client framework and **Firebase** as the backend infrastructure. The system enables users to register, authenticate securely, create conversations, and exchange messages with real-time synchronization provided by a hybrid architecture using Cloud Firestore and Firebase Realtime Database. The application is structured around modular layers: UI screens, backend services, and data models. Each layer isolates its responsibility to ensure maintainability, scalability, and clarity in system design.

The application aims to demonstrate integration of four essential technical components:

1. **Firebase Authentication (Login & Registration)**
2. **Firebase Realtime Database (Message Engine)**
3. **Firebase Firestore Database (User Profiles & Inbox)**
4. **Real-time Chat Application Logic (Seamless Communication)**

Together, these components form a complete, secure, and scalable chat system with automatic UI updates, persistent storage, and efficient querying.

2. Implementation Details

2.1 Firebase Authentication (Email & Password Login/Registration)

The app uses Firebase Authentication to manage user identities.

- The **Register Screen** collects email, username, and password.
- Before account creation, the username is checked for uniqueness using a Firestore query.
- After successfully creating the Auth account, a **user profile document** is created in Firestore containing:
 - uid
 - email
 - username
- The **Login Screen** authenticates users and redirects them based on authentication state.

Authentication state is monitored by an **AuthGate**, which acts as the navigation controller:

- If authenticated → user is sent to the Chat List.
- If not authenticated → user is redirected to Login.

This ensures only active users can access backend data.

2.2 Database Architecture

A. Firestore Database is used to store all persistent data in a NoSQL format.

- **Users Collection**
 - Document ID: user UID
 - Contains: `email`, `username`, `uid`
 - Used to display names in chats/inbox
- **Chats Collection**

Each chat is a single conversation between two users.

Fields:

- `participants` → array of two UIDs
- `lastMessage` → preview text for inbox
- `lastUpdatedAt` → used to sort chats chronologically

B. Realtime Database (Message Storage)

Instead of a Firestore sub-collection, the actual message history is stored in the **Firebase Realtime Database** to optimize for high-frequency updates.

- **Structure:** Messages are stored as a JSON tree under the `messages` node.
- **Path:** `messages/{chatId}/{messageId}`
- **Fields:** `senderId`, `text`, `timestamp`
- **Benefit:** This keeps the Firestore `chats` documents lightweight (metadata only) while allowing the Realtime Database to handle the rapid stream of new message bubbles.

2.2.1 Firebase Firestore Database (Metadata & Query Engine)

While the Realtime Database handles the raw speed of message delivery, **Cloud Firestore** serves as the structural backbone of the application, managing user identities and the "Inbox" logic. It was selected for its ability to handle complex queries and structured relationships.

How Firestore Was Implemented Firestore operates as the "Source of Truth" for the application state outside of active chat rooms. It manages two critical workflows:

1. The "Inbox" Stream (Chat List) The Chat List screen relies on a sophisticated Firestore query to generate the user's inbox. Instead of downloading all data, the application performs a "Shallow Query" on the `chats` collection:

- **The Query:** `collection('chats').where('participants', arrayContains: currentUid).orderBy('lastUpdatedAt', descending: true)`
- **Efficiency:** This allows the application to load the list of active conversations *without* downloading the thousands of message texts associated with them. The `lastMessage` field in the Firestore document provides the preview text, keeping the initial data load extremely lightweight.

2. User Profile Resolution Firestore is responsible for translating raw User IDs (e.g., `71xE...`) into human-readable names.

- **Implementation:** When the Chat List loads, the `user_service` uses the IDs stored in the `participants` array to fetch the corresponding document from the `users` collection.
- **Data Integrity:** This ensures that even if a user changes their email or authentication details, their profile data remains consistent across the application.

– Why Firestore Was Used

- **Complex Filtering:** The application requires finding chats that *contain* a specific user and *sorting* them by time. Firestore's Composite Indexes handle this logic natively and efficiently.
- **Structured Metadata:** Unlike the Realtime Database's free-form JSON, Firestore's document model enforces a strict structure for User Profiles and Chat Metadata, reducing bugs related to missing fields.

2.2.2 Firebase Realtime Database (Real-Time Message Engine)

In addition to Firestore, the project integrates **Firebase Realtime Database** as the dedicated storage layer for the live message history between users. While Firestore manages user profiles and chat metadata (chat list, last messages, timestamps), the Realtime Database is responsible for handling high-speed message synchronization inside each chat room.

– How Realtime Database Was Implemented

The Realtime Database is used exclusively for storing and streaming individual chat messages. All messages are stored under a single root node named `messages`, organized using the structure:

- `messages/{chatId}/{messageId}`
 - `senderId`
 - `text`
 - `timestamp`

This structure allows the Chat Screen to subscribe to `messages/{chatId}` and receive updates instantly whenever a new message is written.

– The Dual-Write Strategy

The application follows a coordinated strategy whenever a user sends a message:

1. **Write to Realtime Database:** The message contents (senderId, text, and timestamp) are pushed to `messages/{chatId}`. Because the Realtime Database uses WebSocket-based synchronization, the new message appears on both devices almost instantly.
2. **Update Firestore Metadata:** At the same moment, the Firestore chat document is updated with the new `lastMessage` and the updated `lastUpdatedAt` timestamp. This keeps the inbox (Chat List screen) correctly ordered.

– Why Realtime Database Was Used

The decision to use the Realtime Database for message history was based on multiple technical advantages:

- **Speed:** The Realtime Database provides faster real-time synchronization than Firestore for continuous, list-based updates. It is optimized for rapid data streaming.
- **Efficiency:** Instead of storing large message histories inside Firestore—where they would slow down chat list queries—the message data is isolated in the Realtime Database. This ensures Firestore remains lightweight for metadata while message history loads quickly without affecting other chats.
- **Simplicity:** A chat conversation is essentially an append-only list of messages. The Realtime Database's JSON tree structure is ideal for storing this type of sequential data.

– Hybrid Architecture Integration

This hybrid architecture allows the app to use the best database for each task:

- **Firestore:** Used for Metadata, chat list, and profile data (Indexed & Query-based).
- **Realtime Database:** Used for High-speed message streaming.

2.3 Real-Time Functionality (Streams for Chat & Inbox)

The app uses real-time streams from both Cloud Firestore (for the chat list/inbox) and Firebase Realtime Database (for the message history inside each chat).

Chat List Stream

- Provided by `chat_metadata_service`

- Streams all chats where `participants` contains the current user
- Automatically refreshes the inbox when any of:
 - New message is sent
 - `lastMessage` gets updated
 - `lastUpdatedAt` changes

Message Stream

- Provided by `chat_service`
- The Chat Screen subscribes to a real-time message stream
- New messages appear instantly without refresh
- UI auto-scrolls to the most recent message using `scrollToBottom()`

This creates a WhatsApp-like real-time experience.

2.4 The Required Composite Index (Critical Backend Fix)

Because the application uses a **combined query**:

- Filter: `participants` (`arrayContains`)
- Sort: `lastUpdatedAt` (`descending`)

Firestore **requires** a composite index.

Without it:

- Chat list fails to load
- Sorting cannot be applied
- Queries become invalid

With the composite index:

- Inbox loads instantly
- Chats are sorted by most recent activity
- The UI always reflects the latest state

This index is one of the required technical components and a necessary backend optimization.

3. Full System Architecture

3.1 Folder Structure Overview

Root files

- `main.dart` → App initialization, theme, AuthGate
- `firebase_options.dart` → Auto-generated Firebase config

Screens (UI pages)

- Login Screen
- Register Screen
- Chat List (Inbox)
- Chat Screen (Chat room)

Services

- `chat_metadata_service` → create chat, update last message
- `chat_service` → send messages, stream messages
- `user_service` → fetch usernames

Models

- `user_model`
- `chat_model`
- `message_model`

This organization ensures scalability and clarity.

3.2 Core Functional Methods

In ChatListScreen

- `_startChat(receiverId)`
 - Checks if chat exists
 - Creates new chat if not
- `_buildRecentChats()`
 - Builds inbox using a stream of chats

In ChatScreen

- `sendMessage()`
 - Saves message to Realtime Database

- Calls `updateChatMetadata` to refresh inbox
- `scrollToBottom()`
 - Keeps UI focused on recent messages

In `chat_metadata_service`

- `chatsStream(uid)` → retrieves current user chats
 - `updateChatMetadata(...)` → updates `lastMessage` + timestamp
-

4. Justification for Firebase Services

4.1 Why Firebase Authentication?

- Secure login & registration
- Automatic UID generation
- Full integration with Firestore Security Rules
- Eliminates need for custom backend authentication
- Easy handling of auth states (ideal for “AuthGate” navigation)

4.2 Why Hybrid Architecture? (Firestore + Realtime Database)

We utilized a "Best Tool for the Job" strategy:

- **Why Firestore for the Inbox?** The Chat List requires complex querying: filtering by `participants` array AND sorting by `lastUpdatedAt`. Firestore handles this efficiently with Composite Indexes. Doing this in Realtime Database would require downloading all data to the client, which is inefficient.
- **Why Realtime Database for Messages?** The Chat Room is a simple, append-only list of data. Realtime Database is optimized for syncing JSON lists with lower latency and overhead than Firestore. Separating messages prevents the "Inbox" query from becoming slow as chat histories grow large.

4.3 Why Composite Index?

- Required to combine array filtering & timestamp sorting
- Enables WhatsApp-style inbox sorting
- Improves query performance significantly

4.4 Why Real-Time Streams (Firestore & Realtime Database)?

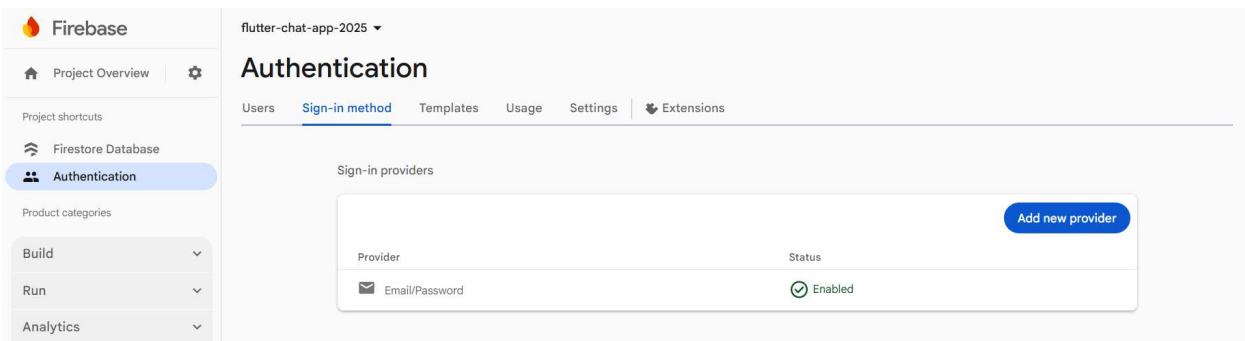
- Delivers new chat messages instantly via Realtime Database listeners, without manual refresh.
- Enables real-time messaging

- Keeps the chat list (inbox) updated automatically via Firestore listeners.
- Highly responsive and efficient

5. Screenshots

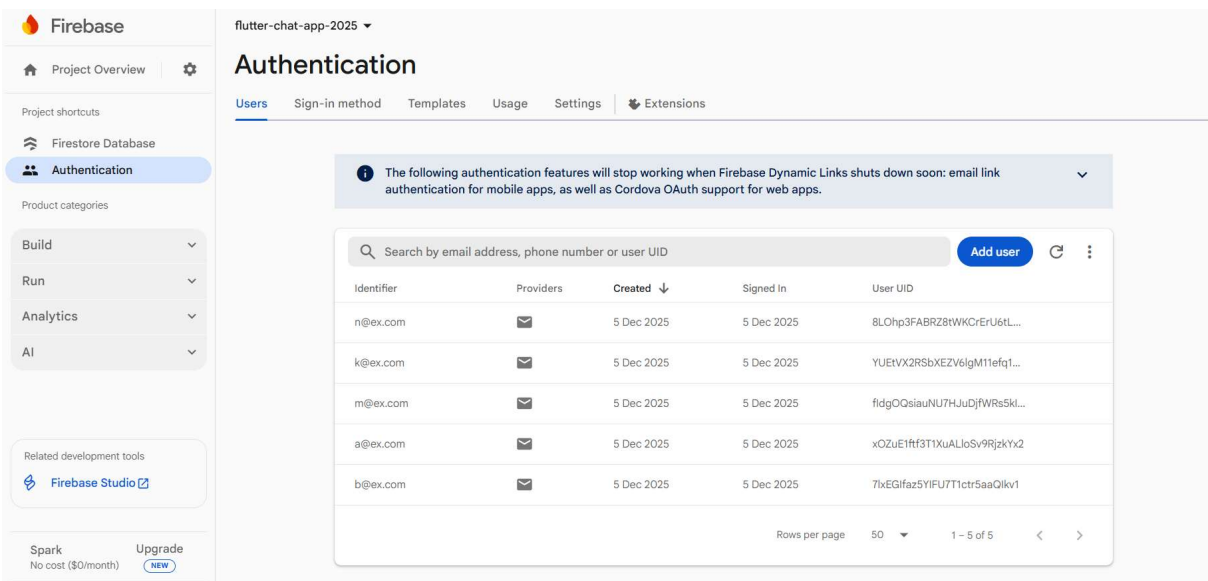
5.1 Firebase Authentication Configuration

– Sign-in Method



Description: Shows the Firebase configuration where the **Email/Password** provider has been explicitly enabled to allow user access.

– Registered Users List

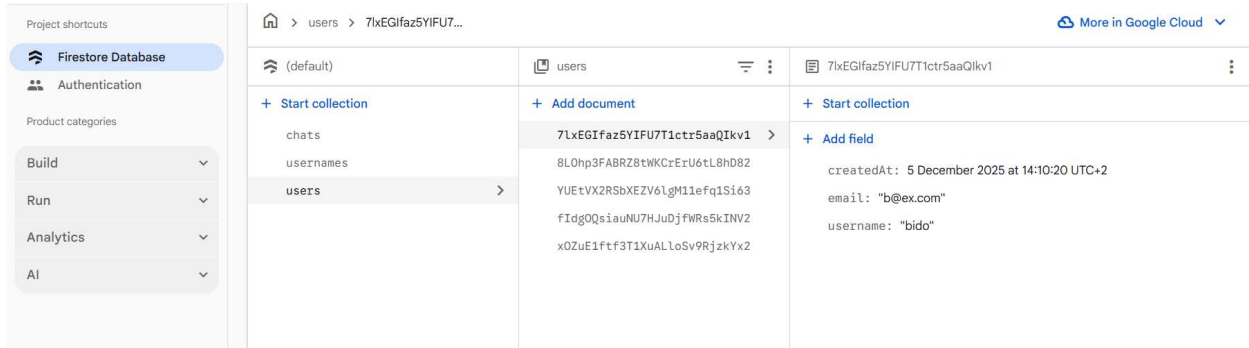


Description: Displays the database of registered users. The **User UID** shown here matches the

Document ID in the Firestore `users` collection, proving the link between the Authentication system and the Database.

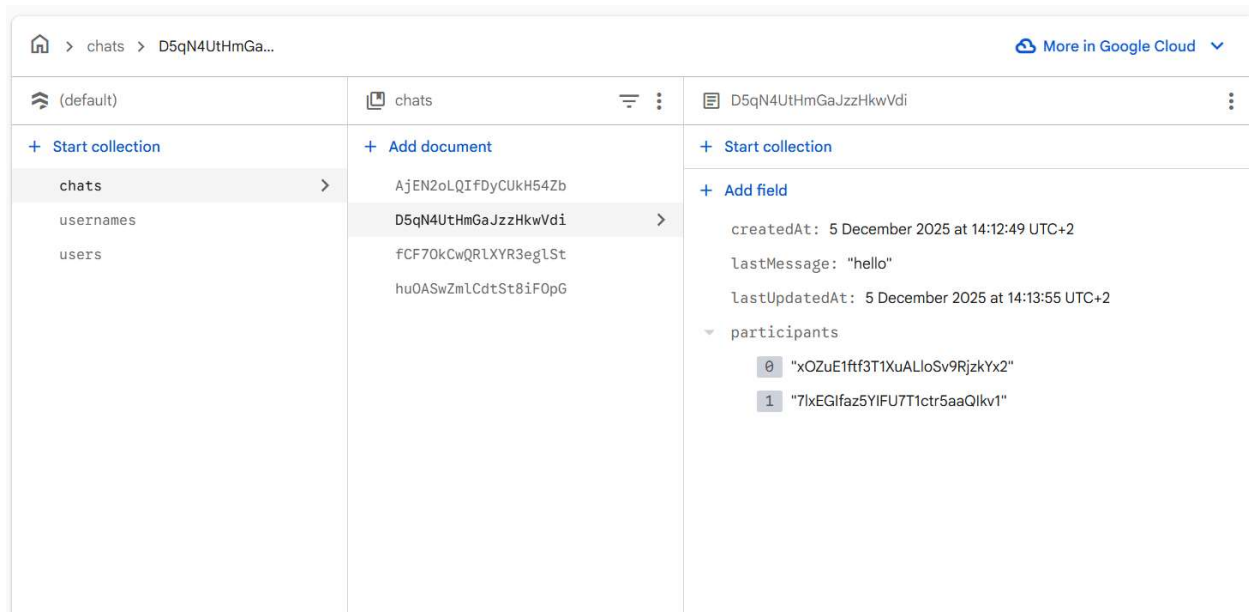
5.2 Firestore Database Architecture

– Users Collection



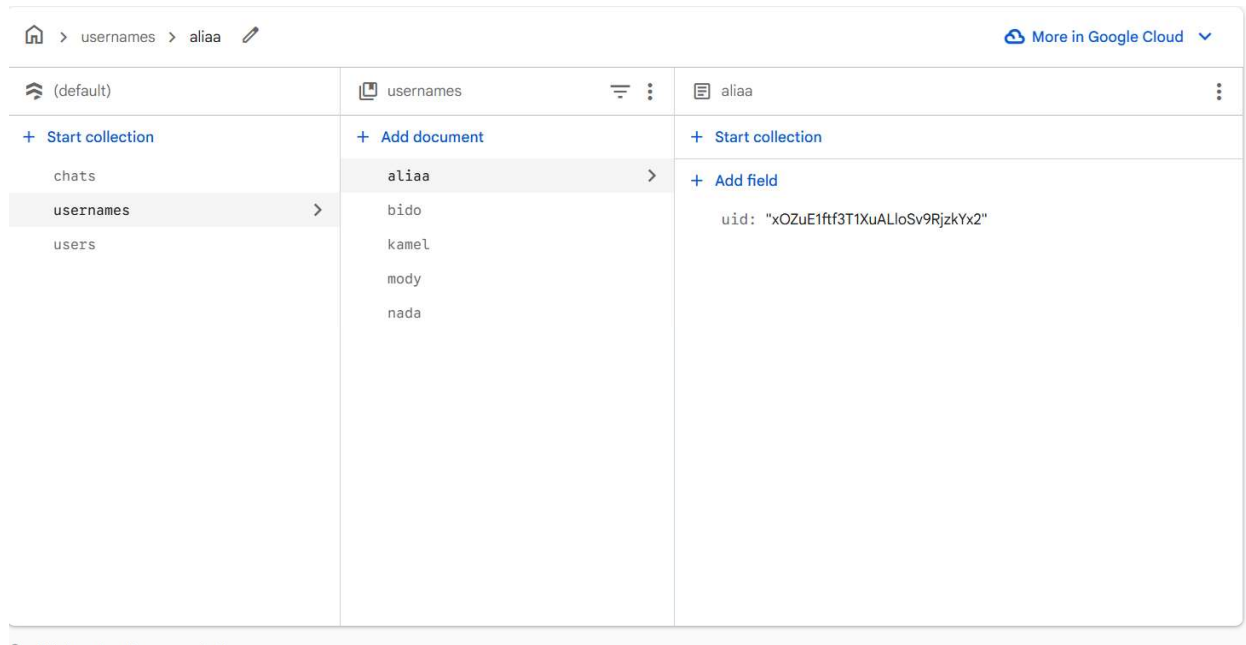
Description: This view demonstrates the user metadata storage. Each document ID corresponds to the Authentication UID, linking the two services. Fields include `username` and `email`, which are fetched by the `user_service` to display human-readable names in the UI.

– Chats Collection



Description: This hierarchical view validates the data model design. The `chats` document stores metadata like `participants` and `lastMessage` for the inbox list.

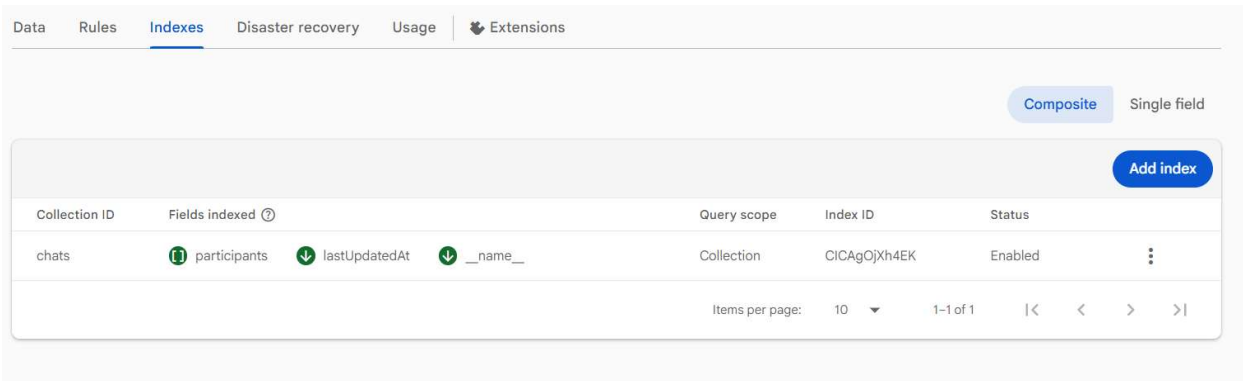
– Usernames Collection (Uniqueness Check)



Description: This collection is a dedicated index used to enforce unique usernames during registration.

- **Structure:** The Document ID is the username itself, and the field is the associated `uid`.
- **Purpose:** Before creating an account, the app queries this collection. If a document with the desired username already exists, the registration is blocked, preventing duplicate display names in the chat.

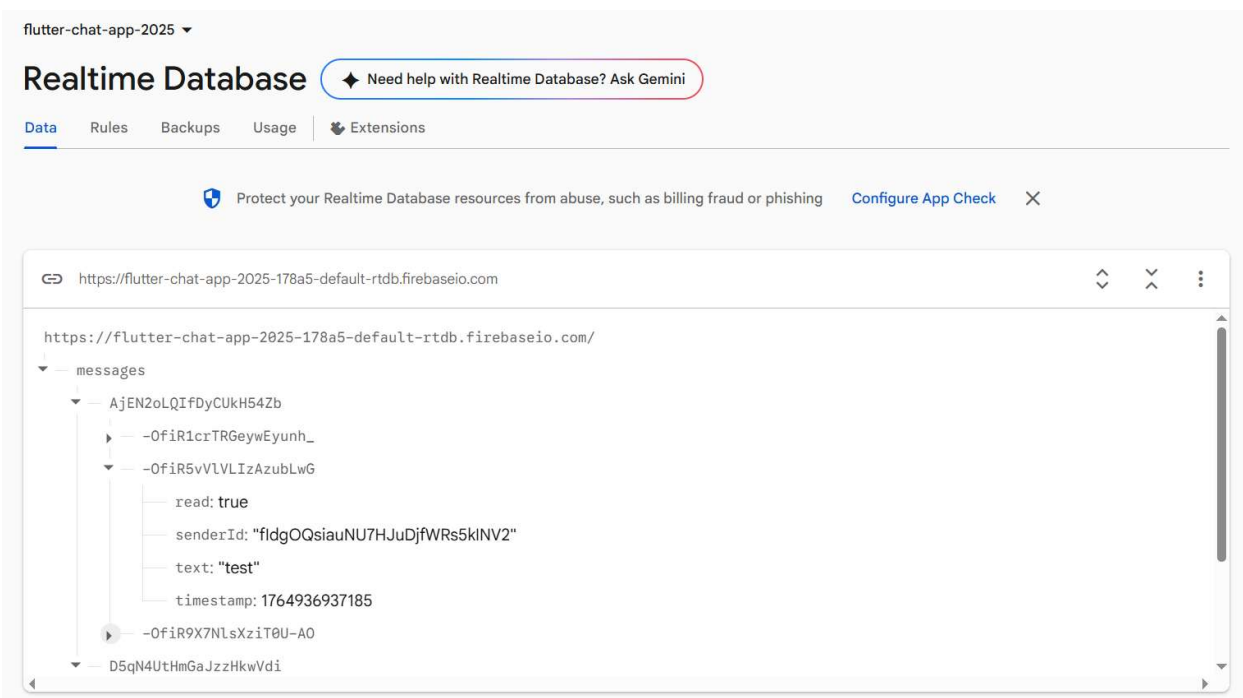
5.3 Composite Index Configuration



Description: This screenshot proves the existence of the custom Composite Index required for the "Inbox" query. The index targets the `chats` collection and combines:

1. `participants` (Arrays) – to filter chats the user belongs to.
2. `lastUpdatedAt` (Descending) – to sort chats by most recent activity. Without this index, the filtered and sorted query in `ChatListScreen` would fail.

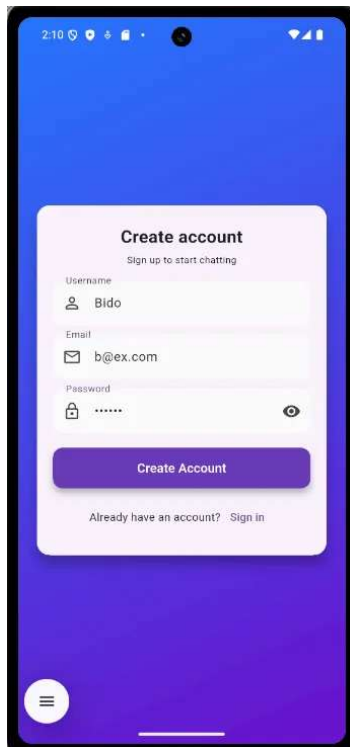
5.4 Realtime Database Structure



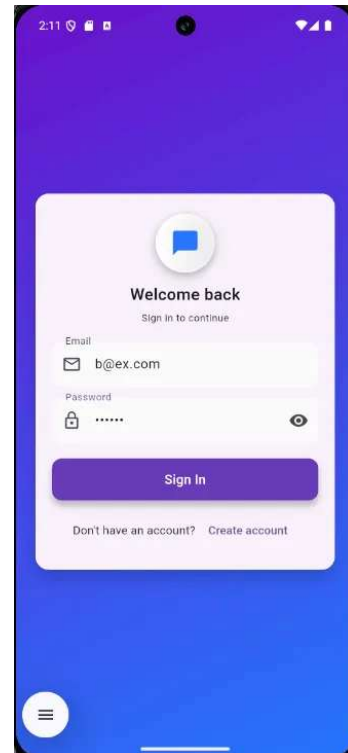
Description: This screenshot confirms the use of the Firebase Realtime Database for storing chat history. The data is structured as a nested JSON tree under the messages node, separated by Chat ID.

5.5 Emulator Screenshots (App Working)

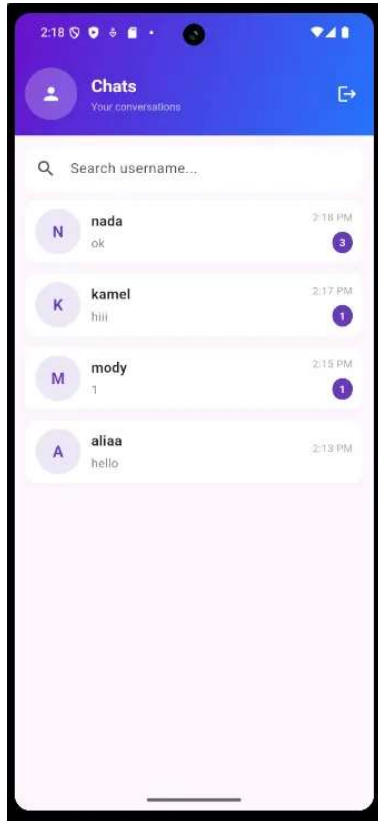
Registration Screen



Login Screen



Chat List Screen (Inbox)



Chat Screen (Real-Time Messaging)



Description:

Demonstrates real-time messaging between two authenticated users. Messages appear instantly because the Chat Screen subscribes to the Realtime Database message stream, while Firestore updates the inbox metadata (lastMessage, lastUpdatedAt).

6. Testing & Verification

The application was tested using two Android emulators:

- Successful login and registration
- Real-time bidirectional messaging
- Correct inbox sorting using lastUpdatedAt
- Accurate metadata updates (lastMessage)
- Composite index functioning as expected

All tests passed as intended.

7. Conclusion

Our application successfully implements a complete real-time messaging system using Flutter and Firebase. The project integrates the required technical components—Authentication, Firestore, Real-time Streams, and Composite Indexing—and demonstrates a clean, modular architecture with efficient data flow. The final system is functional, secure, scalable, and aligned with real-world chat application design principles.