Flutter & Firebase Chat App – Beginner's Guide

Introduction

Welcome to this beginner-friendly guide for building a **Flutter** chat application with **Firebase**. In this guide, we will walk through every step needed to set up Firebase services and integrate them into a Flutter app. We will cover how to configure Firebase (including **Authentication** and **Cloud Firestore**), explain how Firestore stores and retrieves chat data in real-time, and break down the full Flutter code (with **CRUD** operations, **StreamBuilder**, and common widgets like `Scaffold`, `TextField`, `ListView`, etc.). By the end, you will understand how the app works and be able to build your own basic chat app from scratch!

Firebase Setup (Step-by-Step)

To get started, you need to set up a Firebase project and connect it to your Flutter app. Follow these steps:

1. **Create a Firebase Project:** Go to the Firebase Console and click **Add project**. Enter a project name and follow the prompts to create a new Firebase project. (You can disable Google Analytics for now if asked.)

2. **Register your Flutter app with Firebase:** Inside your Firebase project, add a new app:

3. If you are focusing on Android, click the **Android icon** (for iOS, click the **iOS icon**, or the web icon for Web apps). Enter the app's package name (e.g., `com.example.flutterChat`) and the nickname. Register the app.
4. Download the generated configuration file:
    - For Android, this is a `google-services.json` file. Place it in your Flutter project under `android/app/`.
    - For iOS, this is a `GoogleService-Info.plist`. Add it to your Xcode project (under iOS Runner).

5. Alternatively, you can use the FlutterFire CLI to automatically configure Firebase for your project, which generates a `firebase_options.dart` file (as seen in our code) with all the Firebase keys. Either method is fine for setup.

6. **Add Firebase SDKs to your Flutter app:** In your Flutter project, add the necessary Firebase packages to your **pubspec.yaml** dependencies:

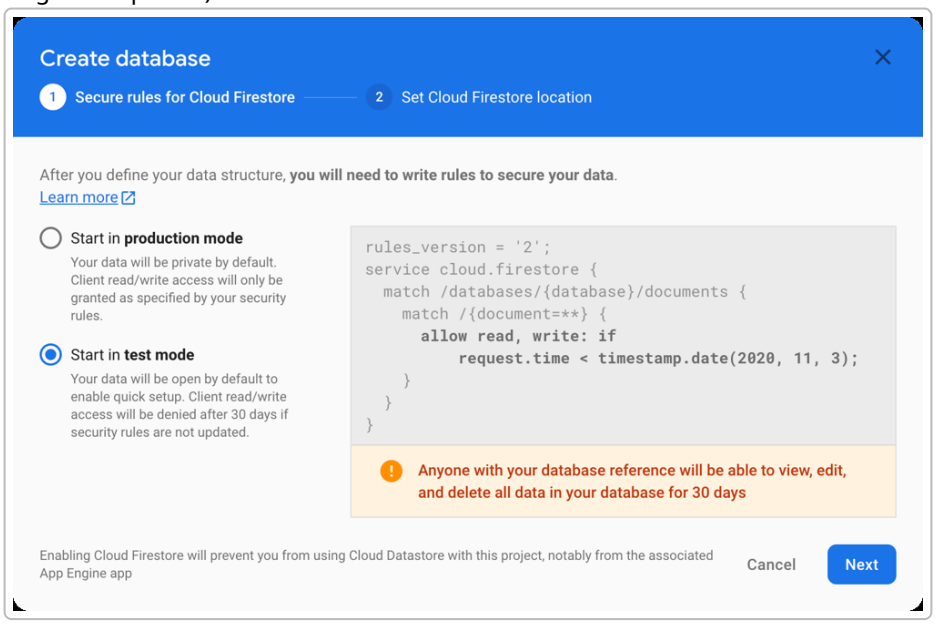
7. `firebase_core` (required for initializing Firebase)
8. `firebase_auth` (for authentication, using anonymous sign-in in this app)
9. `cloud_firestore` (for Firestore database usage)

10. Also add other dependencies used in this app: `shared_preferences` (to save local data like username), `flutter_slidable` (for swipe-to-delete/edit UI), and `intl` (for formatting timestamps).

11. Run `flutter pub get` to install the packages.
    Next, initialize Firebase in your app's entry point. In the `main()` function of Flutter, call `WidgetsFlutterBinding.ensureInitialized();` then `Firebase.initializeApp(...)` with the default options. (This step is already done in our code as shown later.)

12. **Enable Anonymous Authentication:** Our chat app uses Firebase Authentication to let users sign in anonymously (no email/password needed). In the Firebase console, go to **Authentication** and open the **Sign-in method** tab. Enable the **Anonymous** sign-in provider and click Save [1]. This allows users to authenticate with Firebase without credentials. (We will use this to let users pick a username and join the chat without a full sign-up flow.)

13. **Create a Cloud Firestore database:** In the Firebase console, click on **Firestore Database** in the left menu. Press **Create database** and choose a location (select the closest region). When prompted for security rules, select **Start in test mode** for this tutorial (this opens the database for free reads/writes during development).



    Firestore Setup: *When creating a Cloud Firestore database for the first time, choose "Start in test mode" to allow open read/write access during development.* After selecting test mode, click **Next**, choose a database location, and then **Enable**. Your Firestore database will be created with test security rules (which means anyone can read/write for the first 30 days – fine for learning purposes).

14. **Run the Flutter app:** Now that Firebase is configured, you can run your app on an emulator or device with `flutter run`. The first time, it may ask to download the Firebase Android/iOS configurations. Once running, you should see the app's welcome screen where you can enter a username and sign in anonymously.

With Firebase set up, we can now explore how the chat app uses Firestore and FirebaseAuth to function.

Understanding Cloud Firestore for Chat Data

Cloud Firestore is a NoSQL database that stores data in **documents**, which are organized into **collections** [2]. In a chat application, it makes sense to have a collection (e.g. `"messages"`) where each document in that collection is a single chat message. Each message document will contain fields such as the message text, the user who sent it, and a timestamp. Firestore is particularly powerful for chat apps because it offers **real-time updates** – your app can listen to a collection and get notified whenever new data is added or changed [3].

In our app, we use two collections in Firestore: - `messages` – This collection holds all chat messages. Each document represents one message. The fields for a message document include: - `text`: The content of the message (string). - `uid`: The unique ID of the user who sent the message (from Firebase Auth). - `username`: The chosen username of the sender (for display purposes). - `timestamp`: The server timestamp of when the message was sent (used to order messages chronologically). - `typing` – This is an optional small collection used to track which users are currently typing. Each document in `typing` has the document ID equal to a user's ID and a field `isTyping` (boolean), plus a `username` field. When a user is typing, we set their `isTyping` to `true` in this collection, so that other users can see a "... is typing" indicator.

How Firestore stores data: Unlike SQL databases, Firestore doesn't require a fixed schema. When we add a new message using the Flutter app, we call `Firestore.collection('messages').add({...})`. This will create a new document under "messages" with a unique ID and the fields we specify (text, uid, username, timestamp). If the "messages" collection doesn't exist yet, Firestore will create it automatically when the first document is added. The same goes for the "typing" collection.

Real-time updates: The chat app uses Firestore's ability to stream updates. We set up listeners (via the Flutter Firestore SDK) on the "messages" collection. This means whenever a new message is added, edited, or removed in the collection, our app is **immediately notified**. We don't have to constantly fetch data – Firestore pushes the changes to the app, and the UI updates accordingly. This real-time stream is what makes the messages appear to all users instantly after one user sends a message.

Ordering messages: We store a timestamp for each message. In Firestore queries, we **order by** this `timestamp` field to fetch messages in chronological order. In our case, we actually order by timestamp descending (newest first) so that when we display in a list, we can easily reverse it to show latest messages at the bottom of the chat (which is the usual chat UI behavior). This ensures everyone sees messages in the correct order of sending.

CRUD Operations in the Chat App

Our chat application demonstrates all four basic database operations (CRUD) on Firestore data: **Create, Read, Update, Delete**. Here's how each is implemented:

Create (Sending a New Message)

When you type a message and hit the send button (or press enter), the app creates a new document in the `messages` collection. This is done in the code by calling Firestore's `.add()` method. For example, in our code the `_sendMessage()` function does the following:

```
Future<void> _sendMessage() async {
  if (_messageController.text.trim().isEmpty || _auth.currentUser == null) {
    return; // Don't send if message is empty or user not signed in
  }
  try {
    await _firestore.collection('messages').add({
      'text': _messageController.text.trim(),
      'uid': _auth.currentUser!.uid,
      'username': _username ?? 'Guest',
      'timestamp': FieldValue.serverTimestamp(),
    });
    // After sending, reset typing status and clear the input
    await _firestore.collection('typing')
        .doc(_auth.currentUser!.uid)
        .set({ 'isTyping': false });
    _messageController.clear();
  } catch (e) {
    debugPrint('Error sending message: $e');
  }
}
```

Let's break down what happens when creating (sending) a message: - We first check that the message isn't empty and that the user is signed in (anonymous sign-in provides a `currentUser` with a UID). - We call `_firestore.collection('messages').add({...})` with a map of the message data. Firestore will generate a new document with a unique ID under "messages". - Fields we include: - `'text'`: the message text from the input field. - `'uid'`: the user's ID (so we know who sent it). - `'username'`: the user's display name (so others can see who sent it). - `'timestamp'`: we use `FieldValue.serverTimestamp()` to let Firestore set the timestamp to the server time when the message is received (this ensures consistent ordering even if user clocks differ). - After adding the message, we also update the "typing" status for the user to false (since once you send, you're no longer typing). This writes to the `typing` collection with the user's ID. - Finally, we clear the text field controller to reset the input box.

When this operation succeeds, any listeners on the "messages" collection (like our UI) will get the new message in real-time.

Read (Listening for Incoming Messages)

Reading in our chat app is done continuously and in real-time. Instead of manually fetching messages once, we set up a **listener** to Firestore using a Stream. The `StreamBuilder` widget in Flutter listens to

`FirebaseFirestore.instance.collection('messages').snapshots()`. This stream provides **live updates** of the messages collection.

In code (inside our `ChatScreen` widget's build), we have:

```
StreamBuilder<QuerySnapshot>(
  stream: _firestore
          .collection('messages')
          .orderBy('timestamp', descending: true)
          .snapshots(),
  builder: (context, snapshot) {
    if (snapshot.hasError) {
      return Center(child: Text('Something went wrong'));
    }
    if (snapshot.connectionState == ConnectionState.waiting) {
      return Center(child: CircularProgressIndicator());
    }
    // If we reach here, we have data
    final docs = snapshot.data!.docs;
    final messages = docs.map((doc) => Message.fromFirestore(doc)).toList();
    ...
    return ListView.builder(
      reverse: true,
      itemCount: messages.length,
      itemBuilder: (context, index) {
        final message = messages[index];
        final isCurrentUser = message.uid == currentUser?.uid;
        // build each message bubble...
      },
    );
  },
)
```

Here's how the "Read" operation works: - `FirebaseFirestore.instance.collection('messages').snapshots()` gives us a **Stream** of query snapshots. We use `orderBy('timestamp', descending: true)` to get newest messages first. - The `StreamBuilder` listens to that stream. - If any error occurs (e.g. permission issues), `snapshot.hasError` will be true and we display an error text. - If the connection is still loading the initial data, we show a loading spinner (`CircularProgressIndicator`). - Once data is available (`snapshot.data` exists), we extract the list of document snapshots: `snapshot.data!.docs`. - We transform each document into our `Message` model object by calling `Message.fromFirestore(doc)` (which simply converts the doc fields into a `Message` class with `id`, `text`, `uid`, etc.). Now we have a list of message objects. - We use a `ListView.builder` to render each message in the list. We set `reverse: true` on the ListView because we ordered the query descending (newest first) but we want to show the newest message at the bottom of the screen (typical for chat UIs where messages flow upward). Reversing the list view achieves that. - Each list item is a custom widget (`MessageBubble`) which nicely

formats the message (we'll explain this widget later). We also check `isCurrentUser` to know if the message was sent by the currently logged-in user, in order to style it differently and decide whether to show edit/delete options.

Because this is a real-time stream, whenever a new message is added by any user, Firestore will send an update and `StreamBuilder` will rebuild the list with the new data. Similarly, if a message is edited or deleted, the stream will send the updated snapshot and the UI will update automatically to reflect the changes (no manual refresh needed).

Update (Editing a Message)

Our app allows users to edit messages they have sent (perhaps to fix a typo). The update operation in Firestore is done with `doc.update(...)`. In our code, `_editMessage()` handles this:

```
Future<void> _editMessage(Message message) async {
  TextEditingController editController = TextEditingController(text:
message.text);
  final newText = await showDialog<String>(
    context: context,
    builder: (context) => AlertDialog(
      title: Text('Edit Message'),
      content: TextField(
        controller: editController,
        decoration: InputDecoration(hintText: 'Edit your message'),
        autofocus: true,
      ),
      actions: [
        TextButton(onPressed: () => Navigator.pop(context), child:
Text('Cancel')),
        TextButton(onPressed: () => Navigator.pop(context,
editController.text), child: Text('Save')),
      ],
    ),
  );
  if (newText != null && newText.trim().isNotEmpty) {
    try {
      await _firestore.collection('messages').doc(message.id).update({
        'text': newText.trim(),
      });
    } catch (e) {
      debugPrint('Error editing message: $e');
    }
  }
}
```

What happens here: - We open a Flutter `AlertDialog` that contains a `TextField` pre-filled with the current message text. The user can edit it and press "Save". - If the user enters a new text and presses Save, the dialog returns that text (`newText`). - We then call `_firestore.collection('messages').doc(message.id).update({'text': newText})`. The `message.id` is the document ID of the message we want to change (we stored the ID when we created our `Message` model from the snapshot). - This `update` call will change the `text` field of that Firestore document to the new text. - Thanks to our real-time listener, the StreamBuilder will detect this change in the messages collection and rebuild the UI. The edited message will now display the updated text for everyone who is listening.

Delete (Removing a Message)

We also allow users to delete their own messages. Deletion in Firestore is straightforward with `doc.delete()`. In our code, `_deleteMessage()` looks like this:

```
Future<void> _deleteMessage(Message message) async {
  try {
    await _firestore.collection('messages').doc(message.id).delete();
  } catch (e) {
    debugPrint('Error deleting message: $e');
  }
}
```

When a user chooses to delete a message (in our UI they swipe the message and tap the trash icon), we call the above function with the message to delete: - `_firestore.collection('messages').doc(message.id).delete()` removes that document from Firestore permanently. - As soon as this happens, all clients listening to the "messages" collection get a snapshot update with that document gone. The StreamBuilder in our app receives the new list of docs (with one less entry) and rebuilds the ListView. The deleted message will disappear from the UI in real-time for all users.

> **Note:** In a real app, you might want to add security rules to ensure only the owner of a message can edit or delete it. In our test-mode setup, all reads/writes are open, but for production you'd secure the data. Also, you might confirm deletion with the user (e.g., via a dialog) before actually deleting.

Now that we have covered the core database operations, let's look at how Flutter's `StreamBuilder` facilitates those real-time updates in our UI.

Real-time Updates with StreamBuilder

A key piece of this app is the `StreamBuilder` widget. **StreamBuilder** is a Flutter widget that builds its UI based on a stream of data. In our case, the stream is a Firestore query (`.snapshots()` on the messages collection) which emits new data whenever the database changes. When Firestore sends an update (like a

new message or an edit), the StreamBuilder triggers a rebuild of the widgets in its builder function, updating the list of messages on screen.

Why use StreamBuilder? Without StreamBuilder, we would have to manually refresh the UI or use other state management to update the chat list whenever new messages arrive. StreamBuilder provides an easy way to listen to the Firestore stream and automatically rebuild part of the widget tree when new snapshots come in. It keeps our code reactive and in sync with the database.

How StreamBuilder works: We provide it a `stream` (for example, `FirebaseFirestore.instance.collection('messages').snapshots()`) and a `builder` function that tells Flutter how to display the data. The builder function receives an `AsyncSnapshot` object which contains the state of the stream: - `snapshot.connectionState` indicates if the stream is active, waiting, or done. We used this to show a loading indicator while the connection state is waiting for data. - `snapshot.hasData` (and `snapshot.data`) provides the actual data when available. In our case, `snapshot.data` is a `QuerySnapshot` from Firestore containing docs. - `snapshot.hasError` indicates if something went wrong (like a permission error).

Inside the builder, we handle those cases and then build the widget (ListView of messages) from the data. StreamBuilder essentially bridges the Firestore data to our Flutter widgets.

In our app, we actually use **two** StreamBuilders: 1. One for the messages list (as discussed) – this rebuilds the chat messages. 2. One for the typing indicator – `StreamBuilder<QuerySnapshot>(stream: _firestore.collection('typing').snapshots(), ...)`. The typing StreamBuilder listens to the `typing` collection. Every time a user's `isTyping` status changes, it gets a new snapshot. We filter out the entries to find if any *other user* is typing, and if yes, display a small text at the bottom like "`Alice is typing...`". If no one else is typing, it returns an empty widget (`SizedBox.shrink()`).

Using real-time streams means our UI is always up-to-date with the Firestore database with minimal effort. As soon as you send a message, not only does your local list update (because we add it), but any other user's app listening to the same stream updates too. The same happens for edits and deletes. StreamBuilder is crucial for this reactive behavior.

Flutter Widgets Breakdown

Let's go through some of the common Flutter widgets and components used in this chat app and how they work together:

- **MaterialApp**: The root of our app (`MainApp` in `main.dart`) is a `MaterialApp`. This sets up the application with Material Design theming and navigation. We set the `home` of the MaterialApp to our `AuthScreen`.

- **Scaffold**: Both the `AuthScreen` and `ChatScreen` use a `Scaffold`. A Scaffold provides a basic page layout with a top **AppBar** and a **body** area. For example, `ChatScreen`'s Scaffold has an `AppBar` with the title and a logout button, and the body contains the Column with messages and input.

- **AppBar**: The top bar of the chat screen is an `AppBar` widget. We use it to show the current screen title (in our case, we display "Chat - [username]" to show who is logged in) and an action icon button (logout). The AppBar automatically provides a back button if there's navigation, but since we replace the screen on login, we manually handle navigation.

- **Column** and **Row**: These are layout widgets. We use `Column` in the chat screen to arrange the messages list, typing indicator, and input field vertically (one above the other). Inside the input area, we use a `Row` to place the text field and send button side by side horizontally. The `mainAxisAlignment` or `crossAxisAlignment` on these can adjust alignment; for example, in `AuthScreen` we center the column.

- **Expanded**: We wrap the messages ListView in `Expanded` so that it takes up all remaining space in the Column (between the AppBar and the input field at the bottom). This is why the messages list can grow and scroll within the available area, and the input field stays at the bottom.

- **ListView.builder**: To efficiently display a scrolling list of messages, we use `ListView.builder`. This constructor builds children on demand. We specify `itemCount` (number of messages) and an `itemBuilder` function that builds each message widget. We also set `reverse: true` so the list scrolls from bottom (newest) upwards.

- **StreamBuilder**: As explained, it's used to listen to Firestore streams. In the widget tree, StreamBuilder is a parent of the ListView. Whenever the stream updates, the builder provides a new ListView with updated children.

- **TextField**: We use `TextField` widgets for user input:

- In `AuthScreen`, a TextField lets the user enter a username.

- In `ChatScreen`, the TextField at the bottom is where the user types their chat message. We provide decoration (like a label or hint) and use a `TextEditingController` to retrieve what's typed. We also use `onChanged` on the chat TextField to update the typing status (so if the field is non-empty, we mark the user as typing). The `onSubmitted` callback is used to send the message when the user hits the "Enter" key.

- **ElevatedButton**: In `AuthScreen`, we have an `ElevatedButton.icon` for the "Sign In" action. This is a Material Design button with an icon and text. The `onPressed` callback triggers the anonymous sign-in function. We styled it a bit with padding.

- **IconButton**: In `ChatScreen`, the send button (a paper plane icon) is an `IconButton`. It shows an icon and calls `_sendMessage()` when pressed. We also use an `IconButton` in the AppBar for logout (the logout icon on the right).

- **Text**: We use `Text` widgets for all textual content, like the app title, messages, timestamps, "is typing" indicator, etc. Text widgets can be styled with different fonts and colors as we did for message bubbles (e.g., making username bold, coloring user messages text white, etc.).

- **Container** (with BoxDecoration): In `MessageBubble` (the widget for chat messages), we use a `Container` to style the message appearance. The container has padding and a colored background (blue for the current user, gray for others) and rounded corners. We achieved the chat bubble shape by using `BoxDecoration` with `borderRadius` where opposite corners are rounded.

- **Align**: Also in `MessageBubble`, we wrap the container in an `Align` widget to position the bubble to the left or right. If the message is from the current user, we align it to the right; if from others, to the left. This creates the classic chat layout where your messages appear on one side.

- **Flutter Slidable**: This is not a core widget but a package we included (`flutter_slidable`). We used `Slidable` and `SlidableAction` to allow swipe gestures on each message in the list. When you swipe a message (if it's your own message), it reveals options: Edit or Delete. The `Slidable` widget wraps each list item (`MessageBubble`) and provides an `ActionPane` with actions. Beginners can think of it as a convenient way to add slide-to-act functionality. It's an advanced UI touch, so if it's confusing, know that it's primarily for UI polish – the underlying functionality (edit/delete) ultimately calls the Firestore update/delete we explained above.

- **SharedPreferences**: Again, not a UI widget, but worth noting: we use `shared_preferences` to store the user's ID and username locally on the device. This way, if the app is reopened, we check if there were saved credentials and skip the login screen (auto-navigate to chat). This is done in `AuthScreen._checkForSavedCredentials()` and is an example of simple local persistence.

All these widgets work together in the app's widget tree. The flow is: - **AuthScreen** (on launch) shows a centered Column containing a welcome Text, a TextField for username, and a Sign In button. - Upon tapping Sign In, we call Firebase Auth to sign in anonymously. After success, we navigate to **ChatScreen**. - **ChatScreen** (once loaded) builds a Scaffold with AppBar and body. The body's Column has: - Expanded StreamBuilder -> ListView of MessageBubble (shows chat messages). - The typing indicator text (if any user is typing). - The input Row with TextField and send IconButton. - The `MessageBubble` widgets handle styling of each chat message (differentiating sender vs receiver). - The StreamBuilders keep the UI updated with Firestore data live.

Next, let's review the complete source code of this app to see how these pieces come together.

Complete Source Code

Below is the full source code for our Flutter chat app. We include each Dart file and explain the purpose of each section with comments:

`main.dart`

```
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:flutter_chat/auth_screen.dart';
import 'firebase_options.dart'; // Firebase config for different platforms
```

```dart
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform);
  runApp(const MainApp());
}

class MainApp extends StatelessWidget {
  const MainApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Chat',
      theme: ThemeData(primarySwatch: Colors.blue, useMaterial3: true),
      home: const AuthScreen(), // Start the app at the authentication screen
    );
  }
}
```

Explanation: This is the entry point of the Flutter app. We initialize Firebase with `Firebase.initializeApp` (using the options from our generated `firebase_options.dart`). Then we run the app with `MainApp`. `MainApp` is a stateless widget that sets up the MaterialApp with a title and a theme. It uses `AuthScreen` as the home widget, meaning the first screen shown will be the authentication (login) screen.

`auth_screen.dart`

```dart
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'package:flutter_chat/chat_screen.dart';
import 'package:shared_preferences/shared_preferences.dart';

class AuthScreen extends StatefulWidget {
  const AuthScreen({super.key});

  @override
  State<AuthScreen> createState() => _AuthScreenState();
}

class _AuthScreenState extends State<AuthScreen> {
  final TextEditingController _usernameController = TextEditingController();

  @override
  void initState() {
    super.initState();
```

```
  _checkForSavedCredentials();
}

// Checks if user credentials are saved locally to auto-login
Future<void> _checkForSavedCredentials() async {
  final prefs = await SharedPreferences.getInstance();
  final userId = prefs.getString('userId');
  final username = prefs.getString('username');

  if (userId != null && username != null) {
    // Navigate to ChatScreen if credentials are found
    if (mounted) {
      Navigator.of(context).pushReplacement(
        MaterialPageRoute(builder: (context) => const ChatScreen()),
      );
    }
  }
}

// Signs in the user anonymously with Firebase Auth
Future<void> _signInAnonymously(BuildContext context) async {
  if (_usernameController.text.trim().isEmpty) {
    // Username not entered, show a snackbar error
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Please enter a username.'),
        backgroundColor: Colors.red,
      ),
    );
    return;
  }

  try {
    final userCredential = await FirebaseAuth.instance.signInAnonymously();
    final userId = userCredential.user?.uid;

    if (userId != null) {
      // Save username and userId to local storage
      final prefs = await SharedPreferences.getInstance();
      await prefs.setString('userId', userId);
      await prefs.setString('username', _usernameController.text.trim());

      // Navigate to chat screen after successful login
      if (context.mounted) {
        Navigator.of(context).pushReplacement(
          MaterialPageRoute(builder: (context) => const ChatScreen()),
        );
      }
```

```
      }
    } catch (e) {
      debugPrint('Error signing in anonymously: $e');
      if (context.mounted) {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(
            content: Text('Failed to sign in. Please try again.'),
            backgroundColor: Colors.red,
          ),
        );
      }
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Text(
              'Welcome to Flutter Chat',
              style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
            ),
            const SizedBox(height: 20),
            TextField(
              controller: _usernameController,
              decoration: const InputDecoration(
                labelText: 'Enter your username',
                border: OutlineInputBorder(),
              ),
            ),
            const SizedBox(height: 20),
            ElevatedButton.icon(
              onPressed: () => _signInAnonymously(context),
              icon: const Icon(Icons.login),
              label: const Text('Sign In'),
              style: ElevatedButton.styleFrom(
                padding: const EdgeInsets.symmetric(horizontal: 24, vertical:
12),

              ),
            ),
          ],
        ),
      ),
    );
```

```
  }
}
```

Explanation: The AuthScreen is a stateful widget because it needs to handle input and possibly update if we find saved credentials.

- In `initState`, `_checkForSavedCredentials()` looks in `SharedPreferences` to see if we have a stored `userId` and `username` from a previous session. If found, it means the user has already signed in before, so we navigate directly to ChatScreen (skipping the login).

- `_signInAnonymously`: This function handles the login button press. It first checks that a username was entered (if not, shows a brief SnackBar message asking for one). Then it calls `FirebaseAuth.instance.signInAnonymously()`. This Firebase Auth call returns a `UserCredential` with a `user` that has a `uid`. If successful, we take that `uid` and the username the user entered, and save them to `SharedPreferences` for next time. Finally, we navigate to the `ChatScreen` using `Navigator.pushReplacement` (so that the login screen is removed from history).

- The UI (`build` method) is a simple centered column:

- A welcome text.
- A TextField for the username input.
- An ElevatedButton with a login icon and label, which calls `_signInAnonymously` when pressed.
- We use some padding (`SizedBox`) to space out elements, and style the button a bit.

Once the user is signed in (even anonymously, Firebase gives them a unique ID), we move on to the main chat interface.

`chat_screen.dart`

```dart
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'package:flutter_chat/auth_screen.dart';
import 'package:flutter_chat/models/message.dart';
import 'package:flutter_chat/widgets/message_bubble.dart';
import 'package:flutter_slidable/flutter_slidable.dart';
import 'package:shared_preferences/shared_preferences.dart';

class ChatScreen extends StatefulWidget {
  const ChatScreen({super.key});

  @override
  State<ChatScreen> createState() => _ChatScreenState();
}
```

```dart
class _ChatScreenState extends State<ChatScreen> {
  final TextEditingController _messageController = TextEditingController();
  final FirebaseAuth _auth = FirebaseAuth.instance;
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;

  String? _username;
  String? _userId;

  @override
  void initState() {
    super.initState();
    _loadUserData();
  }

  // Load saved username and userId from local storage to use in chat
  Future<void> _loadUserData() async {
    final prefs = await SharedPreferences.getInstance();
    setState(() {
      _username = prefs.getString('username');
      _userId = prefs.getString('userId');
    });
  }

  // Sign out the current user
  Future<void> _signOut() async {
    try {
      await _auth.signOut();
      final prefs = await SharedPreferences.getInstance();
      await prefs.clear(); // Clear local storage on sign out
      if (mounted) {
        Navigator.of(context).pushReplacement(
          MaterialPageRoute(builder: (context) => const AuthScreen()),
        );
      }
    } catch (e) {
      debugPrint('Error signing out: $e');
    }
  }

  // Sends a new message document to Firestore
  Future<void> _sendMessage() async {
    if (_messageController.text.trim().isEmpty || _auth.currentUser == null) {
      return;
    }

    try {
      await _firestore.collection('messages').add({
        'text': _messageController.text.trim(),
```

```
        'uid': _auth.currentUser!.uid,
        'username': _username ?? 'Guest', // Include username in the message
        'timestamp': FieldValue.serverTimestamp(),
      });

      // Reset typing status after sending the message
      await _firestore.collection('typing').doc(_auth.currentUser!.uid).set({
        'isTyping': false,
        'username': _username ?? 'Guest',
      });

      _messageController.clear();
    } catch (e) {
      debugPrint('Error sending message: $e');
    }
  }

  // Updates "typing" status in Firestore (true if user is typing, false if not)
  Future<void> _updateTypingStatus(bool isTyping) async {
    if (_auth.currentUser == null) return;
    try {
      await _firestore.collection('typing').doc(_auth.currentUser!.uid).set({
        'isTyping': isTyping,
        'username': _username ?? 'Guest',
      });
    } catch (e) {
      debugPrint('Error updating typing status: $e');
    }
  }

  // Shows a dialog to edit a message and updates it in Firestore
  Future<void> _editMessage(Message message) async {
    final TextEditingController editController = TextEditingController(text:
message.text);
    final result = await showDialog<String>(
      context: context,
      builder: (context) => AlertDialog(
        title: const Text('Edit Message'),
        content: TextField(
          controller: editController,
          decoration: const InputDecoration(hintText: 'Edit your message'),
          autofocus: true,
        ),
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context),
            child: const Text('Cancel'),
          ),
```

```
          TextButton(
            onPressed: () => Navigator.pop(context, editController.text),
            child: const Text('Save'),
          ),
        ],
      ),
    );

    if (result != null && result.trim().isNotEmpty) {
      try {
        await _firestore.collection('messages').doc(message.id).update({
          'text': result.trim(),
        });
      } catch (e) {
        debugPrint('Error editing message: $e');
      }
    }
  }

  // Deletes a message document from Firestore
  Future<void> _deleteMessage(Message message) async {
    try {
      await _firestore.collection('messages').doc(message.id).delete();
    } catch (e) {
      debugPrint('Error deleting message: $e');
    }
  }

  @override
  void dispose() {
    _messageController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    final currentUser =
_auth.currentUser; // currently logged-in user (could be null if not signed in)
    return Scaffold(
      appBar: AppBar(
        title: Text('Chat - ${_username ?? 'Guest'}'),
        actions: [
          IconButton(
            icon: const Icon(Icons.logout),
            onPressed: _signOut, // logout button
          ),
        ],
      ),
```

```
      body: Column(
        children: [
          // Messages list with real-time updates
          Expanded(
            child: StreamBuilder<QuerySnapshot>(
              stream: _firestore
                  .collection('messages')
                  .orderBy('timestamp', descending: true)
                  .snapshots(),
              builder: (context, snapshot) {
                if (snapshot.hasError) {
                  return const Center(child: Text('Something went wrong'));
                }
                if (snapshot.connectionState == ConnectionState.waiting) {
                  return const Center(child: CircularProgressIndicator());
                }

                // Convert snapshot data to list of Message objects
                final messages = snapshot.data!.docs
                    .map((doc) => Message.fromFirestore(doc))
                    .toList();

                return ListView.builder(
                  reverse: true,
                  padding: const EdgeInsets.all(8),
                  itemCount: messages.length,
                  itemBuilder: (context, index) {
                    final message = messages[index];
                    final isCurrentUser = message.uid == currentUser?.uid;
                    return Slidable(
                      endActionPane: isCurrentUser
                          ? ActionPane(
                              motion: const ScrollMotion(),
                              children: [
                                SlidableAction(
                                  onPressed: (_) => _editMessage(message),
                                  backgroundColor: Colors.blue,
                                  foregroundColor: Colors.white,
                                  icon: Icons.edit,
                                  label: 'Edit',
                                ),
                                SlidableAction(
                                  onPressed: (_) => _deleteMessage(message),
                                  backgroundColor: Colors.red,
                                  foregroundColor: Colors.white,
                                  icon: Icons.delete,
                                  label: 'Delete',
                                ),
```

```
                            ],
                          )
                        : null,
                    child: MessageBubble(
                      message: message,
                      isMe: isCurrentUser,
                    ),
                  );
                },
              );
            },
          ),
        ),
        // Typing indicator widget
        _buildTypingIndicator(),
        // Input field and send button
        Padding(
          padding: const EdgeInsets.all(8.0),
          child: Row(
            children: [
              Expanded(
                child: TextField(
                  controller: _messageController,
                  decoration: const InputDecoration(
                    hintText: 'Type a message...',
                    border: OutlineInputBorder(),
                  ),
                  onChanged: (text) {
                    // Update typing status: true if there's text, false if
empty
                    _updateTypingStatus(text.isNotEmpty);
                  },
                  onSubmitted: (_) {
                    // When user presses enter, send the message
                    _sendMessage();
                    _updateTypingStatus(false); // stop typing after sending
                  },
                ),
              ),
              IconButton(
                icon: const Icon(Icons.send),
                onPressed: () {
                  _sendMessage();
                  _updateTypingStatus(false); // stop typing after sending
                },
              ),
            ],
          ),
```

```
          ),
        ],
      ),
    );
  }

  // Builds a widget to show if other users are typing
  Widget _buildTypingIndicator() {
    return StreamBuilder<QuerySnapshot>(
      stream: _firestore.collection('typing').snapshots(),
      builder: (context, snapshot) {
        if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
          return const SizedBox.shrink(); // no data or no one typing
        }

        // Filter out the current user's typing status and get others who are
typing
        final typingDocs = snapshot.data!.docs.where((doc) {
          return (doc['isTyping'] == true) && (doc.id != _userId);
        }).toList();

        if (typingDocs.isEmpty) {
          return const SizedBox.shrink(); // no one else is typing
        }

        // Combine all usernames who are typing into one string
        final typingUsernames = typingDocs.map((doc) {
          return doc['username'] as String;
        }).join(', ');

        return Padding(
          padding: const EdgeInsets.symmetric(vertical: 2, horizontal: 8),
          child: Text(
            '$typingUsernames is typing...',
            style: const TextStyle(fontStyle: FontStyle.italic, color:
Colors.grey),
          ),
        );
      },
    );
  }
}
```

Explanation: This is the main chat screen where most of the functionality lies. It's a stateful widget because we handle dynamic data and user interactions. Let's go through it section by section:

- **State variables:** We create a `_messageController` for the input field, and instances of `FirebaseAuth` and `FirebaseFirestore` for convenience. We also have `_username` and `_userId` which we will load from SharedPreferences to display and use as needed.

- **initState -> _loadUserData():** When the chat screen is initialized, we load the saved `username` and `userId` from SharedPreferences (these were stored at login). We store them in state so we can use `_username` to display in the AppBar title and for sending messages, and `_userId` to know who the current user is (used in typing indicator logic).

- **_signOut():** This function signs the user out of FirebaseAuth (which for anonymous users basically just ends their session), clears all stored preferences (so next time they'll need to enter username again), and navigates back to the AuthScreen.

- **_sendMessage():** (Discussed earlier in CRUD Create) – It adds a new message to Firestore and updates the typing status to false. We ensure not to send empty messages. If `_auth.currentUser` is null for some reason (shouldn't happen if we're on chat screen), we also don't proceed.

- **_updateTypingStatus(isTyping):** This writes to the `typing` collection a document for the current user (using their UID as the doc ID) with `isTyping` true/false and stores the username (so others can see who is typing). We call this whenever the text field changes. If no user is signed in, it returns.

- **_editMessage(message):** (Discussed in CRUD Update) – Shows a popup dialog with a TextField to edit the message. If the user enters a new text and saves, it updates the Firestore document for that message with the new text.

- **_deleteMessage(message):** (Discussed in CRUD Delete) – Deletes the message document from Firestore.

- We also override `dispose()` to clean up the `_messageController`.

- **build():** The UI is constructed here. We use a Scaffold with an AppBar:

- The AppBar title shows "Chat - [username]" (default to "Guest" if somehow username is null).
- The AppBar also has a logout IconButton that calls `_signOut` when pressed.

The body of the Scaffold is a Column with three main parts: 1. **Expanded StreamBuilder (messages list):** This takes the largest portion. We set up `StreamBuilder<QuerySnapshot>` on the `messages` collection (ordered by timestamp). The builder logic is as we described: show error text if error, loading spinner if waiting, otherwise build the ListView of messages. - We map each document to a `Message` model instance using `Message.fromFirestore`. - The ListView.builder iterates through the list of messages. For each message: - Determine if it's the current user's message (`isCurrentUser`). - Use a `Slidable` widget to wrap the message. If it's the current user's message, we provide an `ActionPane`

with two actions: edit and delete (each action button calls our `_editMessage` or `_deleteMessage`). - The child of the Slidable is our custom `MessageBubble` widget which actually displays the message text UI. We pass in the `message` and whether `isMe` (current user). - Note: If the message is not from the current user, `endActionPane` is set to null (so no slide actions available). - We also set `reverse: true` so the ListView scrolls from bottom. And we add a padding around the list items.

1. **Typing indicator:** `_buildTypingIndicator()` returns a widget that shows who is typing. We place it right below the Expanded list. This is a StreamBuilder on the `typing` collection. Inside, it filters out the current user's doc and checks if anyone else has `isTyping == true`. If yes, it takes their usernames, joins them, and displays a text like "Alice is typing...". If none, or if there's no data, it returns an empty space. This widget is only as tall as needed for the text (or zero height if no one is typing). We put a small vertical padding for spacing.

2. **Input Row:** At the bottom, we have a padded Row containing:

    - A TextField (expanded to take all horizontal space except the button). This is where the user types a new message. It has a hint text "Type a message..." and an outlined border. We connect it to `_messageController` to manage the input text.
    - The `onChanged` callback of this TextField calls `_updateTypingStatus(text.isNotEmpty)`. This means as soon as the user starts typing (text.isNotEmpty becomes true), we mark them as typing in Firestore. If they erase everything (text.isEmpty), we mark not typing.
    - The `onSubmitted` callback calls `_sendMessage()` and then `_updateTypingStatus(false)` – this handles the case when the user presses "Enter" on the keyboard to send the message. We send the message and immediately mark them as not typing (since the message is submitted).
    - An IconButton with a paper plane (send icon). Pressing this explicitly calls `_sendMessage()` and then `_updateTypingStatus(false)`. This covers the tap-to-send case (and also ensures the typing status resets if they press the button instead of hitting enter).
    - Essentially, both pressing enter and tapping the send icon will send the message and clear the typing status.

3. **_buildTypingIndicator():** (Explained above) – returns a StreamBuilder that listens to all users' typing status. It filters out the current user's status and shows others who are typing. If multiple users, it joins their names (e.g., "Alice, Bob are typing..."). This function is used in the build method to insert the typing indicator widget in the column.

This `ChatScreen` handles displaying messages, sending new ones, editing and deleting, logout, and indicating typing – all via Flutter widgets reacting to Firestore and user input.

`models/message.dart`

```
import 'package:cloud_firestore/cloud_firestore.dart';

class Message {
  final String id;
```

```
  final String text;
  final String uid;
  final String username;
  final Timestamp timestamp;

  Message({
    required this.id,
    required this.text,
    required this.uid,
    required this.username,
    required this.timestamp,
  });

  // Factory constructor to create a Message from a Firestore document
  factory Message.fromFirestore(DocumentSnapshot doc) {
    final data = doc.data() as Map<String, dynamic>;
    return Message(
      id: doc.id,
      text: data['text'] ?? '',
      uid: data['uid'] ?? '',
      username: data['username'] ?? 'Unknown',
      timestamp: data['timestamp'] ?? Timestamp.now(),
    );
  }
}
```

Explanation: The `Message` class is a simple data model to represent a chat message in Dart. It has fields corresponding to what we store in Firestore: - `id` is the document ID in Firestore. - `text`, `uid`, `username`, `timestamp` are the content of the message.

The `Message.fromFirestore` factory takes a `DocumentSnapshot` (from a Firestore query) and extracts the data to create a `Message` object. We provide default values in case some fields are missing (e.g., if `timestamp` is null, we default to `Timestamp.now()`, although normally `FieldValue.serverTimestamp()` ensures it's set). This model helps us convert from the loosely typed data in Firestore to a strongly-typed Dart object we can use in our UI easily (like `message.text`).

`widgets/message_bubble.dart`

```
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import '../models/message.dart';

class MessageBubble extends StatelessWidget {
  final Message message;
  final bool isMe;
```

```
const MessageBubble({super.key, required this.message, required this.isMe});

@override
Widget build(BuildContext context) {
  return Align(
    alignment: isMe ? Alignment.centerRight : Alignment.centerLeft,
    child: Container(
      margin: const EdgeInsets.symmetric(vertical: 5, horizontal: 10),
      padding: const EdgeInsets.all(10),
      decoration: BoxDecoration(
        color: isMe ? Colors.blue[300] : Colors.grey[300],
        borderRadius: BorderRadius.only(
          topLeft: const Radius.circular(12),
          topRight: const Radius.circular(12),
          bottomLeft: isMe ? const Radius.circular(12) : Radius.zero,
          bottomRight: isMe ? Radius.zero : const Radius.circular(12),
        ),
      ),
      child: Column(
        crossAxisAlignment:
            isMe ? CrossAxisAlignment.end : CrossAxisAlignment.start,
        children: [
          if (!isMe)
            Text(
              message.username,
              style: const TextStyle(
                fontWeight: FontWeight.bold,
                fontSize: 14,
                color: Colors.black87,
              ),
            ),
          const SizedBox(height: 5),
          Text(
            message.text,
            style: TextStyle(
              color: isMe ? Colors.white : Colors.black,
              fontSize: 16,
            ),
          ),
          const SizedBox(height: 5),
          Text(
            DateFormat('hh:mm a').format(message.timestamp.toDate()),
            style: TextStyle(
              color: isMe ? Colors.white70 : Colors.black54,
              fontSize: 12,
            ),
          ),
        ],
```

```
          ),
        ),
      );
    }
}
```

Explanation: `MessageBubble` is a stateless widget that represents a single chat message in the UI (the speech-bubble appearance). It takes a `Message` object and a boolean `isMe` (true if this message was sent by the current user).

- We use an `Align` widget to position the bubble: right-aligned if it's the current user (`Alignment.centerRight`), or left-aligned if it's someone else (`Alignment.centerLeft`).
- The `Container` inside has a margin for spacing between messages and padding inside the bubble for the text. The `decoration` gives it a colored background and rounded corners:
- We set the color to blue-ish if it's the current user (`Colors.blue[300]`), or gray for others (`Colors.grey[300]`).
- The `borderRadius` uses `Radius.circular(12)` on all corners except one to create a bubble shape. Specifically, if `isMe` is true (current user), we round all corners except bottom-right (giving a tail-like flat edge on that side). If `isMe` is false, we round all except bottom-left. This asymmetry makes each bubble look like it's pointing from the sender.
- Inside the container, we use a `Column` to stack the message sender name (if not current user), the text, and the timestamp.
- We only show the `username` Text if `isMe` is false (i.e., show the sender's name for other people's messages, but don't show a name on your own messages).
- The message text is shown in a `Text` widget. We style it white for current user (on a blue background) or black for others (on a light gray background). We increase the font size a bit for readability.
- Below that, we show the timestamp. We format the `Timestamp` to a human-readable time using the `intl` package's `DateFormat('hh:mm a')` (e.g., "07:45 PM"). We style the timestamp smaller and a bit faded (white70 or black54) so it's visible but not too prominent.
- Overall, `MessageBubble` takes care of the presentation details of a message, making the chat UI look nicer.

That's it! You now have a working Flutter chat app powered by Firebase. To recap, we set up Firebase (Auth and Firestore), used anonymous authentication for quick login, and used Cloud Firestore to store and sync messages in real-time. We went through the code which handles creating, reading (streaming), updating, and deleting messages. We also implemented a typing indicator and swipe-to-edit/delete functionality for a better chat experience.

Feel free to modify and expand this app – for example, you could integrate user sign-up, add pictures or chat rooms, or enforce security rules so only authenticated users can read/write. Flutter and Firebase together make it straightforward to build scalable, real-time applications. Happy coding!

---

[1] Authenticate with Firebase Anonymously | FlutterFire
https://firebase.flutter.dev/docs/auth/anonymous-auth/

[2] [3] Cloud Firestore | FlutterFire
https://firebase.flutter.dev/docs/firestore/usage/