EXPERIMENT NO 6

NAME-Nitish Bhosle

CLASS-D15A

ROLL NO-04

AIM- To connect Flutter UI with firebase.

THEORY-

Firebase helps developers to manage their mobile app easily. It is a service provided by Google. Firebase has various functionalities available to help developers manage and grow their mobile apps.

Steps to Add firebase to our Flutter app using Firebase CLI

1. Install the Firebase CLI and log in (run firebase login)
2. From any directory, run this command:
    - dart pub global activate flutterfire_cli

3. Then, at the root of your Flutter project directory, run this command:
    - flutterfire configure --project=questitnextjs

4. This automatically registers your per-platform apps with Firebase and adds a lib/firebase_options.dart configuration file to your Flutter project.

5. To initialise Firebase, call Firebase.initializeApp from the firebase_core package with the configuration from your new firebase_options.dart file:

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

import 'firebase_options. dart';

await Firebase.initializeApp(

options: DefaultFirebaseOptions.currentPlatform,

);

6. Add the dependencies in the

pubspec.yaml file Firebase_core : ^version

Firebase_auth : ^version

SYNTAX

1. Firebase Core

Purpose Serves as the foundation for all Firebase services in your Flutter application.

Implementation:

- Initialized in `main.dart` using `Firebase.initializeApp()` to set up the Firebase SDK before app startup.

- Ensures Firebase services are properly configured and ready to use throughout the app.

Reference Code

dart

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

2. Firebase Authentication

Purpose: Manages user authentication, allowing users to sign in/sign up and maintaining authentication state.

Implementation:

- Provides multiple authentication methods via `AuthService` including:
  - Email/password authentication
  - Google Sign-In integration
- Tracks authentication state changes with `authStateChanges()` stream
- Handles user sign-out across multiple auth providers

Reference Code:

dart

```
// Email/password auth
Future<User?> signInWithEmail(String email, String password) async {
  try {
    final userCredential = await _auth.signInWithEmailAndPassword(
      email: email,
      password: password,
    );
    return userCredential.user;
  } catch (e) {
    print(e);
    return null;
  }
}

// Google Sign-In
Future<User?> signInWithGoogle() async {
  try {
    final googleUser = await _googleSignIn.signIn();
    if (googleUser == null) return null;
    final googleAuth = await googleUser.authentication;
```

```
    final credential = GoogleAuthProvider.credential(
      accessToken: googleAuth.accessToken,
      idToken: googleAuth.idToken,
    );
    final userCredential = await
_auth.signInWithCredential(credential);
    return userCredential.user;
  } catch (e) {
    print(e);
    return null;
  }
}
```

3. Cloud Firestore

Purpose: Provides a NoSQL database for storing and synchronizing application data in real-time.

Implementation:

- User Profiles: Stores user information in the `users` collection
- Matching System: Records user swipes in the `swipes` collection
- Conversations: Manages user matches and messages in the `conversations` collection
- Real-time Data: Uses streams to provide live updates for messages and matches

Data Structure:

- users: Stores user profiles with personal information, preferences, and interests
- swipes: Records user swiping behavior (like/dislike) with timestamps
- conversations: Contains messaging between matched users

- Each conversation has a subcollection of `messages`

Reference Code:

dart

```
// Creating/updating user profile
Future<void> setUserProfile(UserModel user) {
  return
_db.collection('users').doc(user.uid).set(user.toMap());
}

// Recording swipe action
Future<void> recordSwipe(String swiperUID, String swipedUID,
String action) {
  return
_db.collection('swipes').doc('${swiperUID}_$swipedUID').set({
    'swiperUID': swiperUID,
    'swipedUID': swipedUID,
    'action': action,
    'timestamp': FieldValue.serverTimestamp(),
  });
}

// Creating a match conversation when both users like each
other
Future<void> checkForMatch(String user1UID, String user2UID)
async {
  final doc = await
_db.collection('swipes').doc('${user2UID}_$user1UID').get();
  if (doc.exists && doc['action'] == 'like') {
    final conversationID = user1UID.compareTo(user2UID) < 0
        ? '${user1UID}_$user2UID'
        : '${user2UID}_$user1UID';

    await
_db.collection('conversations').doc(conversationID).set({
```

```
      'users': [user1UID, user2UID],
      'createdAt': FieldValue.serverTimestamp(),
    });
  }
}
```

4. Real-time Messaging

Purpose: Enables instant messaging between matched users.

Implementation:

- Uses Firestore for real-time message delivery
- Stores messages in subcollections within conversation documents
- Efficiently retrieves and displays messages with StreamBuilder
- Handles message sending with server timestamps

Reference Code:

```dart
// Sending a message
Future<void> sendMessage(String conversationID, String senderUID, String message) {
  return _db.collection('conversations').doc(conversationID).collection('messages').add({
    'sender': senderUID,
    'message': message,
    'timestamp': FieldValue.serverTimestamp(),
  });
}

// Retrieving messages in real-time
```

```
Stream<List<Map<String, dynamic>>> getMessages(String
conversationID) {

  return _db

      .collection('conversations')

      .doc(conversationID)

      .collection('messages')

      .orderBy('timestamp')

      .snapshots()

      .map((snapshot) => snapshot.docs.map((doc) =>
doc.data()).toList());

}

`
```

5. State Management with Firebase

Purpose: Integrates Firebase data with the app's state management system.

Implementation:

- Uses Provider pattern to make Firebase services accessible throughout the app
- Manages authenticated user state with UserProvider
- Efficiently updates UI based on real-time Firebase data changes
- Implements proper error handling for Firebase operations

Reference Code:

```dart
// App wrapper that handles authentication state

return StreamBuilder<User?>(

  stream: FirebaseAuth.instance.authStateChanges(),

  builder: (context, snapshot) {

    if (snapshot.connectionState == ConnectionState.waiting) {

      return Scaffold(body: Center(child:
CircularProgressIndicator()));
```

```
    }
    final user = snapshot.data;
    if (user == null) {
      return LoginScreen();
    } else {
      // User is authenticated, fetch and provide user data
      // ...
    }
  }
);
```

Summary

Your application is a dating/matching app that effectively uses Firebase for:

1. User Management: Authentication with multiple sign-in methods and user profile storage
2. Matching Algorithm: Records and processes user preferences and swipe actions
3. Real-time Communication: Facilitates instant messaging between matched users
4. State Synchronization: Keeps UI in sync with backend data changes

OUTPUT