Cloud Computing

Introduction

The task involves developing a mobile application using Flutter, and the backend would be developed with Firebase. The goal is to develop an app with cloud capabilities that provide basic features like authentication, database, and storage. By using Firebase as a Backend-as-a-Service (BaaS), the application benefits from an extendable, serverless backend platform, which lessens the load on the regular server tasks.

Firebase real-time NoSQL database allows storing, retrieving, and updating data with efficiency and provide unified and transparent access across devices and sessions. Firebase hosts all the available application data that provides basic app capability.

Aside from demonstrating the theme of cloud computing, the project also utilizes web scraping. Web scraping is utilized to obtain and display external data dynamically within the app. While the scraped data isn't reflected in Firebase, it is processed and displayed instantly within the user interface, demonstrating how real-time data can be obtained from the web into a cloud-based mobile application.

The project is part of the overall development of web service and cloud computing. It demonstrates how mobile applications can utilize cloud platforms like Firebase and how technologies like web scraping can be incorporated to improve workflows based on the cloud.

Objective

The main objective of this project is to demonstrate the integration of a cloud backend with a Flutter frontend. This entails integrating Firebase services for user authentication, real-time data management, and cloud storage.

Another primary objective is to integrate web scraping to show how real-time, external data can be dynamically integrated into a mobile application, further

showcasing the capabilities of cloud-based technologies in developing interactive, data-driven applications.

Tools and Technologies

Flutter: For cross-platform mobile app development
Flutter plugins: firebase_core: ^3.4.0,firebase_auth: ^5.2.0,google_sign_in: ^6.2.2, google_fonts: ^6.1.0,popup_card: ^0.1.0,http: ^1.2.2,html: ^0.15.5+1,image_picker: ^1.1.2,permission_handler: ^11.3.1,firebase_storage: ^12.2.0,cloud_firestore: ^5.4.0
Firebase: For backend services (including Firestore, Auth, Cloud Storage)
Dart: Programming language used in Flutter
Git & GitHub: Version control and collaboration

Code Explanation

1. **Sign-up/Login Page**

```
Future<void> _signInWithGoogle(BuildContext context) async {
 try {
   final GoogleSignInAccount? googleUser = await
GoogleSignIn().signIn();
   if (googleUser == null) return; // user cancelled sign-in

   final GoogleSignInAuthentication googleAuth =
   await googleUser.authentication;

   final credential = GoogleAuthProvider.credential(
     accessToken: googleAuth.accessToken,
     idToken: googleAuth.idToken,
   );

   await FirebaseAuth.instance.signInWithCredential(credential);

   // success
   final user = FirebaseAuth.instance.currentUser;
   ScaffoldMessenger.of(context).showSnackBar(SnackBar(
     content: Text('Signed in as ${user?.displayName}'),
   ));
   Navigator.pushReplacement(
     context,
     MaterialPageRoute(builder: (context) => const genshinMain()),
   );
 } catch (e) {
```

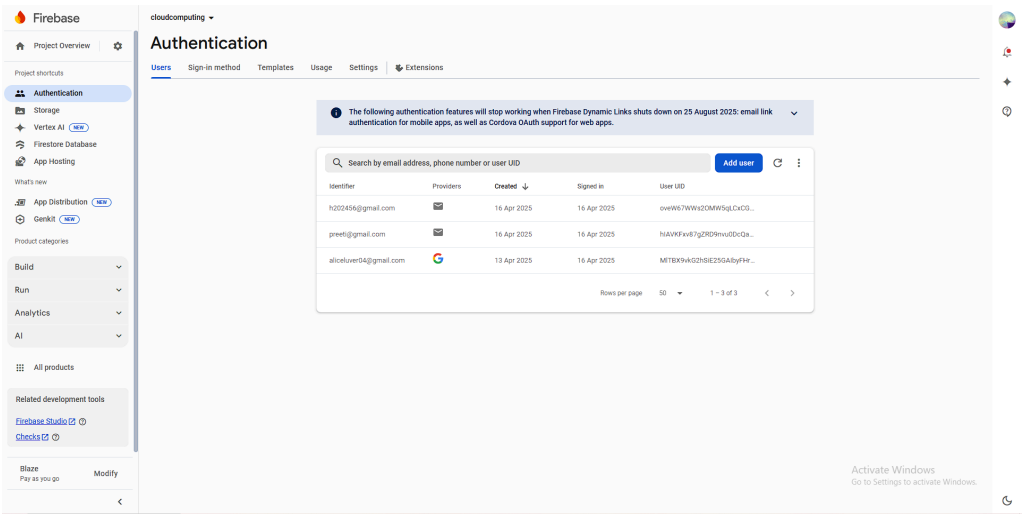
```
    ScaffoldMessenger.of(context).showSnackBar(SnackBar(
      content: Text('Error: $e'),
    ));
  }
}
```

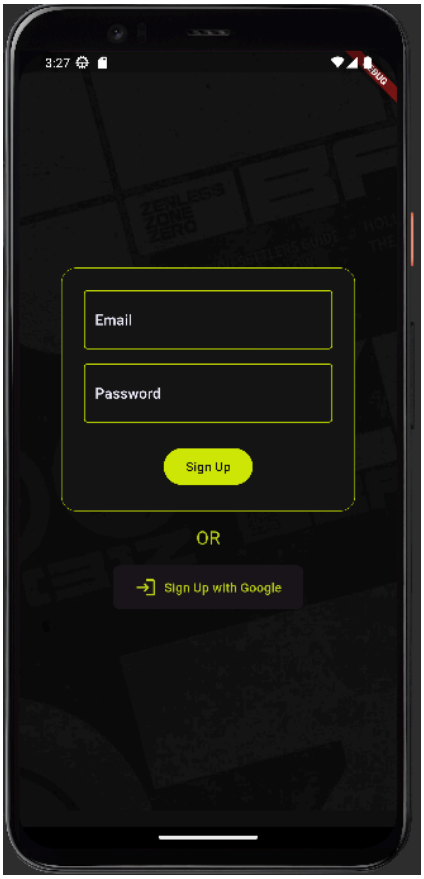
This function handles Google Sign-In in the app using Firebase Authentication. It prompts the user to choose a Google account and, if successful, retrieves the authentication tokens required to sign in to Firebase. A Firebase credential is created using these tokens, and the user is signed in. On successful login, a confirmation message is shown using a SnackBar, and the user is navigated to the main page (genshinMain). If an error occurs during the process, an error message is displayed.

```
final email = emailController.text.trim();
final password = passwordController.text;

try {
final userCredential = await FirebaseAuth.instance
    .createUserWithEmailAndPassword(email: email, password: password);

print('User signed up: ${userCredential.user?.uid}');
Navigator.pushReplacement(
 context,
 MaterialPageRoute(builder: (context) => const genshinMain()),
);
} on FirebaseAuthException catch (e) {
String message;
if (e.code == 'email-already-in-use') {
message = 'This email is already in use.';
final loginCredential = await FirebaseAuth.instance
    .signInWithEmailAndPassword(email: email, password: password);

print('User logged in: ${loginCredential.user?.uid}');

// ✅ Navigate after successful login
Navigator.pushReplacement(
 context,
 MaterialPageRoute(builder: (context) => const genshinMain()),
);
} else if (e.code == 'weak-password') {
message = 'The password provided is too weak.';
} else if (e.code == 'invalid-email') {
```

```
message = 'Invalid email address.';
} else {
message = 'An error occurred. Please try again.';
}

print('Signup error: $message');
// Optionally show a snackbar or dialog with the error
}
```

This code manages user registration using email and password with Firebase Authentication. It first attempts to create a new user with the entered email and password. If the account is successfully created, the user is redirected to the main page (`genshinMain`). If the email is already in use, it automatically tries to log the user in with the same credentials. The code also handles other common errors, such as weak passwords or invalid email formats, and prints appropriate messages for debugging or displaying feedback to the user.





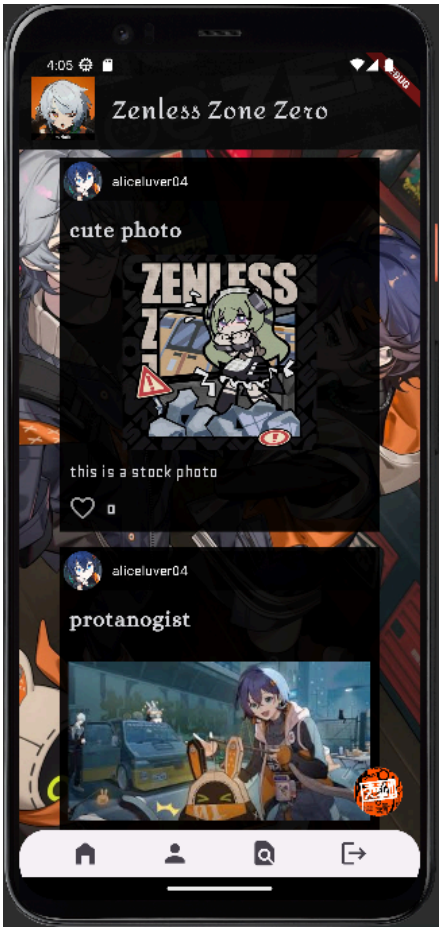
2. Feed Page

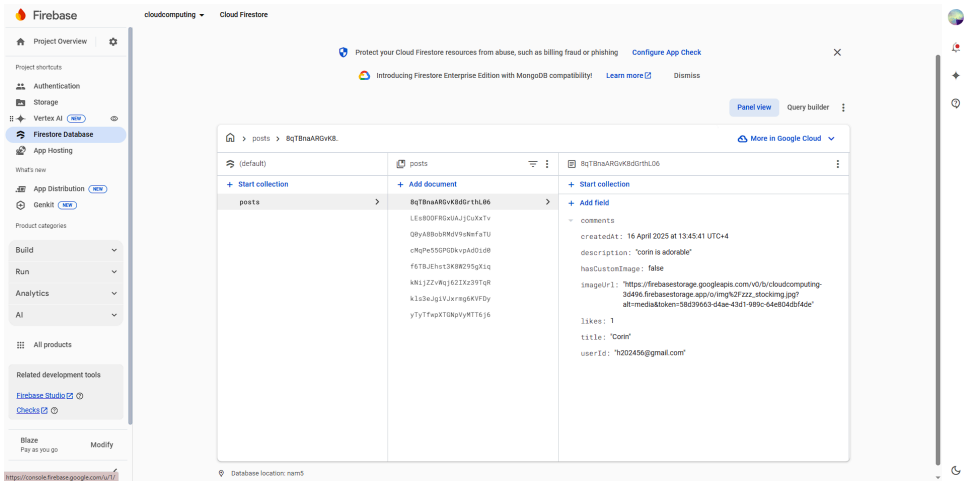
```
StreamBuilder<QuerySnapshot>(
 stream: FirebaseFirestore.instance.collection('posts').orderBy('createdAt',
descending: true).snapshots(),
 builder: (context, snapshot) {
   if (snapshot.hasError) {
     return Center(child: Text('Error: ${snapshot.error}'));
   }

   if (snapshot.connectionState == ConnectionState.waiting) {
     return Center(child: CircularProgressIndicator());
   }

   final posts = snapshot.data!.docs;
   return SingleChildScrollView(
     child: Column(
       children: [
         Column(
           children:
           List.generate(posts.length, (index) {
             final post = posts[index];
             final data = post.data() as Map<String, dynamic>;
             final docId = post.id;
             return  Padding(
               padding: const EdgeInsets.symmetric(vertical: 8.0),
               child: PostWidget( postText: data['title'], userid:
data['userId'], postDes: data['description'] , imgurl:
data['imageUrl'],num:data['likes'].toString(),iddoc:docId),
             );
           }),
         ),
         SizedBox(height: 50,)
       ],
     ),
   );
 },
)
```

This code uses a `StreamBuilder` to listen in real-time to updates from the `posts` collection in Firestore. It fetches the posts ordered by the `createdAt` timestamp in descending order, ensuring the most recent posts appear first. While the data is loading, a circular progress indicator is shown, and if there's an error, an error message is displayed. Once data is available, it loops through each post document, extracts the relevant fields (like title, description, image URL, user ID,

likes, and document ID), and passes them into a custom `PostWidget`. All posts are displayed inside a scrollable column, providing a feed-like layout.

2.1 Display Post in post page





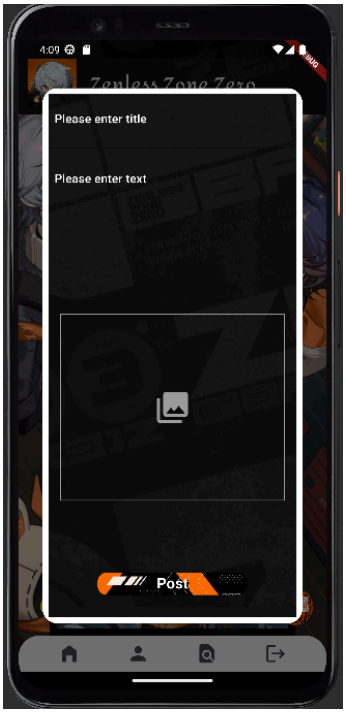
2.2 Add Post in post page

```
Future<void> _uploadPost(BuildContext context) async {
 if (postTitle.text.isEmpty || postDescription.text.isEmpty) {
   ScaffoldMessenger.of(context).showSnackBar(
     SnackBar(content: Text('Please fill all text fields')),
   );
   return;
 }

 _isUploadingNotifier.value = true;

 try {
   String imageUrl;
```

```
    // Handle image upload or use default
    if (_imageNotifier.value == null) {
      imageUrl = defaultImageUrl; // Use default image
    } else {

      final storageRef = FirebaseStorage.instance
          .ref()
          .child('img/${DateTime.now().millisecondsSinceEpoch}.jpg');
      await storageRef.putFile(_imageNotifier.value!);
      imageUrl = await storageRef.getDownloadURL();
    }

    // Save post data
    await FirebaseFirestore.instance.collection('posts').add({
      'title': postTitle.text,
      'description': postDescription.text,
      'imageUrl': imageUrl,
      'userId': FirebaseAuth.instance.currentUser!.email,
      'createdAt': FieldValue.serverTimestamp(),
      'likes': 0,
      'comments': [],
      'hasCustomImage': _imageNotifier.value != null, // Track if custom image
was used
    });

    // Reset form
    postTitle.clear();
    postDescription.clear();
    _imageNotifier.value = null;

    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Post created successfully!')),
    );
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error: ${e.toString()}')),
    );
  } finally {
    _isUploadingNotifier.value = false;
  }
}
```

This function handles the process of uploading a new post to Firebase Firestore, including an optional image upload to Firebase Storage. It first checks if the title and description fields are filled. If an image is selected, it uploads the image and retrieves the download URL; otherwise, it uses a default image URL. Then it creates a new document in the `posts` collection with the title, description, image

URL, current user's email, server timestamp, and initial values for likes and comments. It also tracks whether a custom image was used. Upon success, it clears the form and notifies the user, ensuring a smooth and responsive user experience.



2.3 Update likes in post

```
final docRef = FirebaseFirestore.instance.collection('posts').doc(this.iddoc);
// replace with your doc ID

try {
 final snapshot = await docRef.get();
 final currentLikes = snapshot.get('likes') ?? 0;

 await docRef.update({'likes': currentLikes + 1});


 // Just toggle the icon (doesn't persist across refresh)
} catch (e) {
 print('Error updating likes: $e');
}
```

This code handles the functionality of liking a post in a Firebase Firestore collection. It retrieves a reference to a specific document using its ID (`iddoc`), then fetches the current number of likes from that document. After obtaining the current like count, it increments the value by one and updates the document. This

allows users to register a like on a post, and the updated like count is stored in Firestore. The `try-catch` block ensures that any errors during the update process are caught and printed for debugging.

3. **User Page**

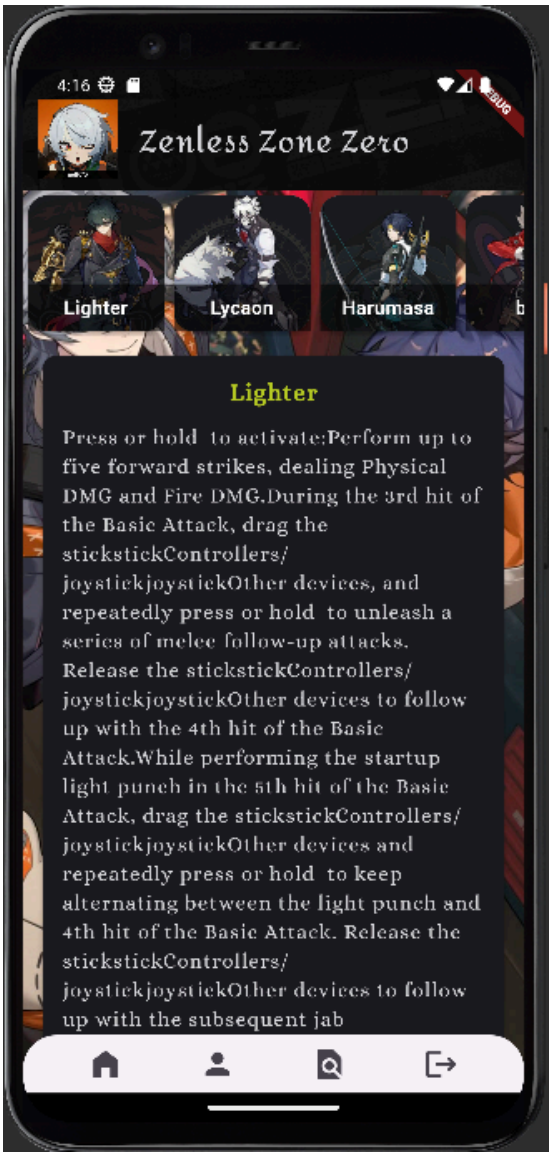
```
StreamBuilder<QuerySnapshot>(
 stream: FirebaseFirestore.instance
     .collection('posts')
     .where('userId', isEqualTo: currentUserEmail) // Filter by
logged-in user
     .snapshots(),
 builder: (context, snapshot) {
   if (snapshot.hasError) {
     return Center(child: Text('Error: ${snapshot.error}'));
   }

   if (snapshot.connectionState == ConnectionState.waiting) {
     return Center(child: CircularProgressIndicator());
   }

   final posts = snapshot.data!.docs;

   return SingleChildScrollView(
     child: Column(
       children: [
         Column(
           children: List.generate(posts.length, (index) {
             final post = posts[index];
             final data = post.data() as Map<String, dynamic>;
             final docId = post.id;

             return Padding(
               padding: const EdgeInsets.symmetric(vertical: 8.0),
               child: PostWidget(
                 postText: data['title'],
                 userid: data['userId'],
                 postDes: data['description'],
                 imgurl: data['imageUrl'],
                 num: data['likes'].toString(),iddoc:docId
               ),
             );
           }),
         ),
         SizedBox(height: 50),
       ],
     ),
   );
 },
)
```

This `StreamBuilder` widget is used to display posts in real-time that belong specifically to the currently logged-in user, based on their email (`currentUserEmail`). It listens to a Firestore `posts` collection and filters documents where the `userId` field matches the user's email. While the connection is initializing, a loading spinner is shown, and if an error occurs, it displays an error message. Once data is received, it iterates through each post and creates a `PostWidget` for each, showing the title, description, image, and number of likes. This ensures the user sees only their own uploaded posts dynamically and in real-time.

4. Wiki page

```
Future<void> fetchTdData() async {
 final url = Uri.parse('https://zenless-zone-zero.fandom.com/wiki/Lighter');
 final response = await http.get(url);

 if (response.statusCode == 200) {
   final document = html_parser.parse(response.body);

   // Find the table with both class names
   final table = document.querySelector('table.wikitable.skill-table');

   if (table != null) {
     final tdElements = table.querySelectorAll('td');

     setState(() {
       tdTexts = tdElements.map((td) => td.text.trim()).toList();
       isLoading = false;
     });
   } else {
     setState(() {
       tdTexts = ['Table not found'];
       isLoading = false;
     });
   }
 } else {
   setState(() {
     tdTexts = ['Failed to load page'];
     isLoading = false;
   });
 }
}
```

This function `fetchTdData()` demonstrates how web scraping is implemented in the Flutter app using the `http` and `html` packages. It sends an HTTP GET request to a specific wiki page (`Lighter` page from the Zenless Zone Zero fandom). If the page loads successfully (status code 200), it parses the HTML to find a table with both `wikitable` and `skill-table` classes. It then extracts all `<td>` (table cell) elements from that table and stores their trimmed text in the `tdTexts` list, which is used to update the UI via `setState()`. If the table is not found or the request fails, it displays appropriate error messages instead. This showcases how external data can be dynamically fetched and displayed in a cloud-capable app without being stored in Firebase.



Service Model Classification of Packages Used (PaaS vs SaaS)

Package/Service	Service Model	Description
firebase_core	PaaS	Initializes and connects the Flutter to Firebase.
firebase_auth	SaaS	Provides user authentication as a managed service.
google_sign_in	SaaS	Enables Google OAuth sign-in via Google's hosted authentication service.
firebase_storage	PaaS	Cloud file storage and retrieval with scalable backend support
cloud_firestore	PaaS	NoSQL database service with real-time sync, part of Firebase platform.
google_fonts	SaaS	Accesses fonts hosted on Google Fonts' web service.
Flutter SDK	PaaS	Development framework offering tools and environment for app building.
Firebase Platform	PaaS	Backend-as-a-Service offering a full suite of tools for app development.

Conclusion

This project showcases the integration of modern development tools and cloud services to build a responsive and dynamic Flutter application. Features such as user authentication, real-time database updates, image uploads, and data scraping were efficiently implemented using Firebase and external packages. Overall, the project highlights how cloud-connected apps can enhance user experience, ensure scalability, and streamline the development process.

Github link

https://github.com/preetiRawat16/cloudapp