Final Report on Marapp

Introduction

This report presents a detailed overview of the development process of our mobile application, including the challenges faced, solutions implemented, and the final execution steps. The application was developed using Dart and Flutter, with Firebase as the backend service.

Implementation Details

The application was built with the following technologies:

- **Frontend:** Flutter (Dart)
- **Backend:** Firebase (Firestore Database, Authentication), Imgur (image storage), OpenCage, EmailJs
- **Additional Services:** Alternative solutions for maps and image storage due to Firebase limitations

Key Features

- User authentication
- Data storage and retrieval using Firebase Firestore
- Image storage workaround due to Firebase free-tier limitations
- Alternative map solution instead of Google Maps API

Challenges and Solutions

1. Firebase API Limitations

Issue: One of the primary challenges was Firebase's free-tier limitations, particularly with image storage and mapping services. The basic plan imposed restrictions that affected the development.

Solution: We implemented a workaround by using Imgur for image storage, which provided more flexibility. For maps, we integrated an alternative service to avoid the limitations imposed by Firebase.

Code Example:

```
Future<String> uploadImage(File imageFile) async {
  var request = http.MultipartRequest('POST', Uri.parse('https://api.imgur.com/3/image'
  request.headers['Authorization'] = 'Client-ID YOUR_CLIENT_ID';
  request.files.add(await http.MultipartFile.fromPath('image', imageFile.path));
  var response = await request.send();
  var responseData = await response.stream.bytesToString();
  var jsonResponse = jsonDecode(responseData);
  return jsonResponse['data']['link'];
}
```

This function uploads an image to Imgur and returns the URL of the uploaded image.

2. User Permissions

Issue: Managing user permissions within Firebase Authentication was more complex than anticipated.

Solution: We created custom Firebase rules to manage user permissions effectively, ensuring that data access was restricted appropriately.

Firebase Rules Example:

```
{
  "rules": {
    "users": {
      "$uid": {
        ".read": "$uid === auth.uid",
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

This ensures that users can only read and write their own data in Firestore.

3. Absence of Firebase Functions

Issue: Firebase's free-tier does not support Cloud Functions, which would have automated tasks and triggers.

Solution: We used client-side logic to replace Cloud Functions and implemented background tasks within the app to achieve similar results.

Code Example:

```
void sendWelcomeEmail(String userEmail) async {
  final url = Uri.parse("https://api.emailjs.com/api/v1.0/email/send");
  final response = await http.post(
    url,
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode({
      'service_id': 'your_service_id',
      'template_id': 'your_template_id',
      'user_id': 'your_user_id',
      'template_params': {
        'user_email': userEmail,
      },
    }),
  );
  if (response.statusCode == 200) {
    print("Email sent successfully");
  } else {
    print("Failed to send email");
  }
}
```

This function replaces Firebase Cloud Functions by sending an email using EmailJS from the client-side.

4. Query Complexity

Issue: Firestore's querying system was restrictive, leading to complex queries.

Solution: We optimized the database schema and designed queries to improve performance and reduce the need for complex joins and filters.

Optimized Query Example:

```
Future<List<DocumentSnapshot>> getUserPosts(String userId) async {
  QuerySnapshot snapshot = await FirebaseFirestore.instance
      .collection('posts')
      .where('userId', isEqualTo: userId)
      .get();
  return snapshot.docs;
}
```

This query retrieves all posts by a specific user while keeping the query simple and efficient.

5. Learning Dart and Flutter

Issue: Adapting to Dart and Flutter posed a learning curve for the team.

Solution: We made use of online tutorials, official documentation, and community resources to accelerate our learning. We also broke the development into smaller tasks to get more familiar with the language.

Implementation Choices

Our implementation was driven by the need to balance features, performance, and constraints:

- **Flutter:** Used for its cross-platform capabilities.
- **Firebase:** Chosen for its real-time data handling, despite the limitations.
- **Third-party Services:** To bypass Firebase's restrictions on maps and image storage, third-party services were integrated.

How to Execute the Application

Prerequisites

To run the app, you need to install:

- Flutter SDK
- Android Studio (or another IDE like Visual Studio Code)
- Dart SDK

Steps to Run the App

1. **Download the Source Code**: Available on AulaWeb.
2. **Extract the Project**: Unzip the project folder.
3. **Install Dependencies**: Run:

   ```
   flutter pub get
   ```

4. **Run the App**:

   ```
   flutter run
   ```

Alternative to Android Studio

If you're not using Android Studio:

- Set up Flutter CLI and environment variables.
- Run the app with `flutter run`.

Conclusion

Through a mix of creative solutions and optimizations, we overcame the challenges associated with Firebase's free-tier limitations, learned Dart effectively, and implemented alternative solutions for image storage and maps. This project allowed us to develop a fully functional mobile app with a robust backend, learning valuable lessons in both frontend and backend development.