MoveInSync Task

RideSharing Platform-TripSync

Soumen Kumar
Roll No-B22ES006

GitHub Repo Link

Click Here.

1 Introduction

This report outlines the development of a ride sharing and tracking application built using Kotlin, XML, and Firebase for backend authentication and database management. The application caters to three distinct user roles: Traveler, Traveler Companion, and Admin, each with specific functionalities and access levels.

2 Project Overview

The application aims to provide a comprehensive platform for ride sharing, tracking, and management. It incorporates features such as ride sharing, real-time tracking, geofencing, feedback collection, and administrative oversight.

3 User Roles and Functionalities

3.1 Traveler

- Share ride details (TripId, Driver Name, Driver Phone Number, cab number, etc.) via SMS for ongoing trips.
- Review audit trail of shared rides.

3.2 Traveler Companion

- Track the ride of the traveler in real-time.
- Receive notifications for trip completion (not implemented).
- Get nearby notifications when the cab enters the geofence of the traveler's drop-off location (not implemented).
- Share feedback on the experience with Admin.

3.3 Admin

- View all rides shared by users.
- Access overall experience feedback from users.

4 Implementation Details

4.1 Technology Stack

- Frontend: Kotlin and XML for Android application development
- Backend: Firebase for authentication and real-time database

4.2 Project Structure

The project follows a standard Android application structure:

```
app/
|-- manifests/
|   `-- AndroidManifest.xml
|-- kotlin+java/
|   `-- com.sk.tripsync/
|       |-- AdminActivity
|       |-- CompanionActivity
|       |-- directionsResponse.kt
|       |-- FeedActivity
|       |-- Feedback_Activity.kt
|       |-- HistoryActivity
|       |-- HisTrips.kt
|       |-- LoginActivity
|       |-- MainActivity
|       |-- Review
|       |-- ReviewActivity
|       |-- ReviewAdapter
|       |-- Ride
|       |-- RideAdapter
|       |-- RidesActivity
|       |-- SplashActivity
|       `-- TravelerActivity.kt
`-- res/
    |-- anim/
    |   `-- rotate.xml
    |-- drawable/
    |   |-- app_icon_background.xml
    |   |-- default_cab_icon.xml
    |   |-- default_location_icon.xml
    |   |-- ic_launcher_background.xml
    |   `-- ic_launcher_foreground.xml
    |-- layout/
    |-- mipmap/
    `-- values/
        |-- colors.xml
        `-- strings.xml
```

5 Key Features Implementation

5.1 Main Activity

The main activity serves as the entry point for users, where they are authenticated via Firebase Authentication. After a successful login, users are redirected to one of the following windows:

- **Traveler Window**
- **Admin Window**
- **Companion Window**

5.2 Traveler Window

The Traveler Window provides the following functionalities:

- **Map View**
  - **API Used**: OpenRoute Service and osmdroid.
  - **Description**: The map view uses osmdroid for displaying the map and OpenRoute Service for route calculations. The map shows the user's location and allows interaction, such as zooming and panning.
- **Buttons**

- **Feed Button**
  - **Functionality**: Allows users to enter 'from' and 'to' coordinates to find a cab. The coordinates are processed to find available cabs within the area.
  - **Route Finding**: Uses OpenRoute Service API to calculate the optimal route between the entered coordinates.
- **History Button**
  - **Functionality**: Displays all previous and ongoing rides.
- **Closest Taxi Determination**:
  - **Algorithm Used**: To find the closest taxi, a nearest neighbor search algorithm is used. This algorithm iterates through a list of available cabs, calculates the distance from the current location to each cab, and selects the cab with the minimum distance.
  - **Process**:
    1. Retrieve the current location of the user.
    2. For each available cab, calculate the distance from the user's location to the cab's location.
    3. Select the cab with the smallest distance as the closest one.
  - **Space Complexity (SC)**: $O(n)$, where n is the number of cabs. The space required is proportional to the number of cabs stored and processed.
  - **Time Complexity (TC)**: $O(n)$, where n is the number of cabs. Each cab's distance needs to be computed, making the time complexity linear with the number of cabs.

  Process of Using OpenRoute Service API
  - **Description**: The OpenRoute Service API provides routing capabilities. It calculates the optimal path between two geographical points.
  - **API Call**: Send a request to the API with the 'from' and 'to' coordinates. The API returns a set of coordinates representing the optimal route.
  - **Response Handling**: Parse the API response to extract route coordinates and draw the route on the map.

5.3 Admin Window

The Admin Window features:

- **Rides Button**
  - **Functionality**: Displays a list of all rides taken by users, including details like ride IDs, driver information, and ride status.
- **Review Button**
  - **Functionality**: Shows feedback provided by companions. This feedback helps in assessing ride quality and driver performance.

5.4 Companion Window

The Companion Window includes:

- **Trip ID Input Box**
  - **Functionality**: Allows companions to enter a trip ID to view details of the associated trip.
- **Feedback Provision**
  - **Functionality**: After reviewing trip details, companions can provide feedback about the ride. The feedback is stored and managed using Firebase Realtime Database, ensuring real-time updates and accessibility.

6 Reducing Search Area by Dividing Map into Cells

6.1 Description

When dealing with a large map area, searching through all available cabs can be inefficient. To improve the efficiency of locating the nearest cab, the map is divided into smaller, manageable cells. Each cell represents a specific geographic region, and only relevant cells are searched based on the user's query.

6.2 Steps

1. **Divide the Map into Cells**:
   - The map area is partitioned into a grid of cells. Each cell covers a specific geographic region.
   - The size of each cell can be adjusted based on the density of cabs and the size of the map area.
2. **Index Cabs by Cell**:
   - Each cab is assigned to a cell based on its location.
   - Maintain a data structure (e.g., a map or list) to store the cabs within each cell.
3. **Search within Relevant Cells**:
   - When a user queries for the nearest cab, determine the cell or cells that intersect with the user's location or the search area.
   - Only search within these relevant cells to find the nearest cab.
4. **Combine with Spatial Indexing (Optional)**:
   - Use spatial indexing techniques like KD-trees or Quad-trees within each cell to further optimize the search process.
   - This allows for faster querying within each cell.

6.3 Example

Consider a map divided into a 10x10 grid of cells, where each cell covers an area of 1 square kilometer. If a user searches for the nearest cab from a specific location, the system identifies the cell or cells that include the user's location and only searches within those cells. This significantly reduces the number of cabs to be checked compared to searching through all available cabs on the map.

6.4 Performance Metrics

- **Space Complexity (SC)**: $O(n)$, where n is the number of cabs. The space complexity includes storing cabs in cells and maintaining the data structure.
- **Time Complexity (TC)**:
  - **Cell Lookup**: $O(1)$ for identifying relevant cells if cell data is indexed.
  - **Cab Search within Cells**: $O(k)$, where k is the number of cabs in the relevant cells. The search time depends on the number of cabs in the cells being queried.

7 Discussion on Key Concepts Using TravelerActivity Code Snippet

7.1 Handling System Failure Cases

7.1.1 Implement Fault-Tolerant Mechanisms

In the `fetchRoute` method, the code uses Retrofit to make an API call to OpenRouteService. By using the `enqueue` method, the code ensures the call is asynchronous and will not block the main thread, which is crucial for maintaining UI responsiveness. To further enhance fault tolerance, we could implement retry logic for failed API calls using a library like Retrofit's `RetryInterceptor`.

```
override fun onFailure(call: Call<DirectionsResponse>, t: Throwable) {
    Log.e(TAG, "API call failed: ${t.message}")
    Toast.makeText(this@TravelerActivity, "Failed to fetch route. Retrying...", Toast.LENGTH_SHORT).
    show()
    // Retry logic can be implemented here
}
```

Listing 1: Fault-tolerant API call

7.1.2 Employ Backup and Recovery Strategies

Data integrity and backup can be managed by storing critical information, such as trip details, locally (e.g., using Room or SharedPreferences) before making network calls. In case of a failure, the app can recover from the local copy.

7.1.3 Develop Comprehensive Error Recovery Procedures

The `onFailure` method in the API call handler already logs errors. To improve, we could add user notifications, retry mechanisms, or fallback strategies. For example:

```
override fun onFailure(call: Call<DirectionsResponse>, t: Throwable) {
    Log.e(TAG, "API call failed: ${t.message}")
    Toast.makeText(this@TravelerActivity, "Failed to fetch route. Retrying...", Toast.LENGTH_SHORT).
    show()
    // Retry logic can be implemented here
}
```

Listing 2: Error logging and user notification

7.2 Object-Oriented Programming Language (OOPS)

7.2.1 Choose a Robust OOPS Language

Kotlin is chosen for this project, which is a modern OOP language with powerful features like null safety, extension functions, and coroutines, making it suitable for developing maintainable and scalable applications.

7.2.2 Leverage OOPS Principles

- **Encapsulation:** The `TravelerActivity` class encapsulates data (like `trips`, `service`, and UI elements) and behavior (methods for handling map interactions, API calls) in a single entity, promoting modularity.
- **Inheritance:** While not shown directly in the snippet, using abstract classes or interfaces for common behaviors across activities can promote code reuse.
- **Polymorphism:** Retrofit uses interfaces to define API calls, allowing different implementations without changing the code that uses these interfaces.

7.3 Error and Exception Handling

7.3.1 Develop a Robust Error and Exception Handling Framework

The provided snippet handles errors in API responses (`onResponse` and `onFailure` methods). This can be extended with custom exception classes to handle specific errors more gracefully.

```
class NetworkException(message: String): Exception(message)
```

Listing 3: Custom Exception Class

7.3.2 Provide Meaningful Error Messages

The `Toast` messages provide immediate feedback to the user. For more detailed debugging, consider logging errors with additional context or using a centralized logging service.

```
Log.e(TAG, "API call failed with response code: ${response.code()} - ${response.message()}")
```

Listing 4: Detailed Error Logging

7.3.3 Regularly Review and Update Error-Handling Strategies

Monitor system logs to identify common failure points and update error-handling logic accordingly. Implementing a feedback mechanism where users can report issues can also provide insights into failure cases.

In summary, the provided code snippet demonstrates handling system failures, leveraging object-oriented principles, and implementing error and exception handling in a Kotlin-based Android application. By incorporating these strategies,we can create a more robust, maintainable, and user-friendly application.

8 Conclusion

This project demonstrates the successful implementation of a ride sharing and tracking application with distinct user roles and functionalities. The use of Kotlin, XML, and Firebase provided a robust foundation for building a scalable and feature-rich mobile application.

9 Future Work

- Implement trip completion notifications for Traveler Companions.
- Develop geofencing functionality for nearby notifications.
- Enhance the admin dashboard with data analytics and reporting features.

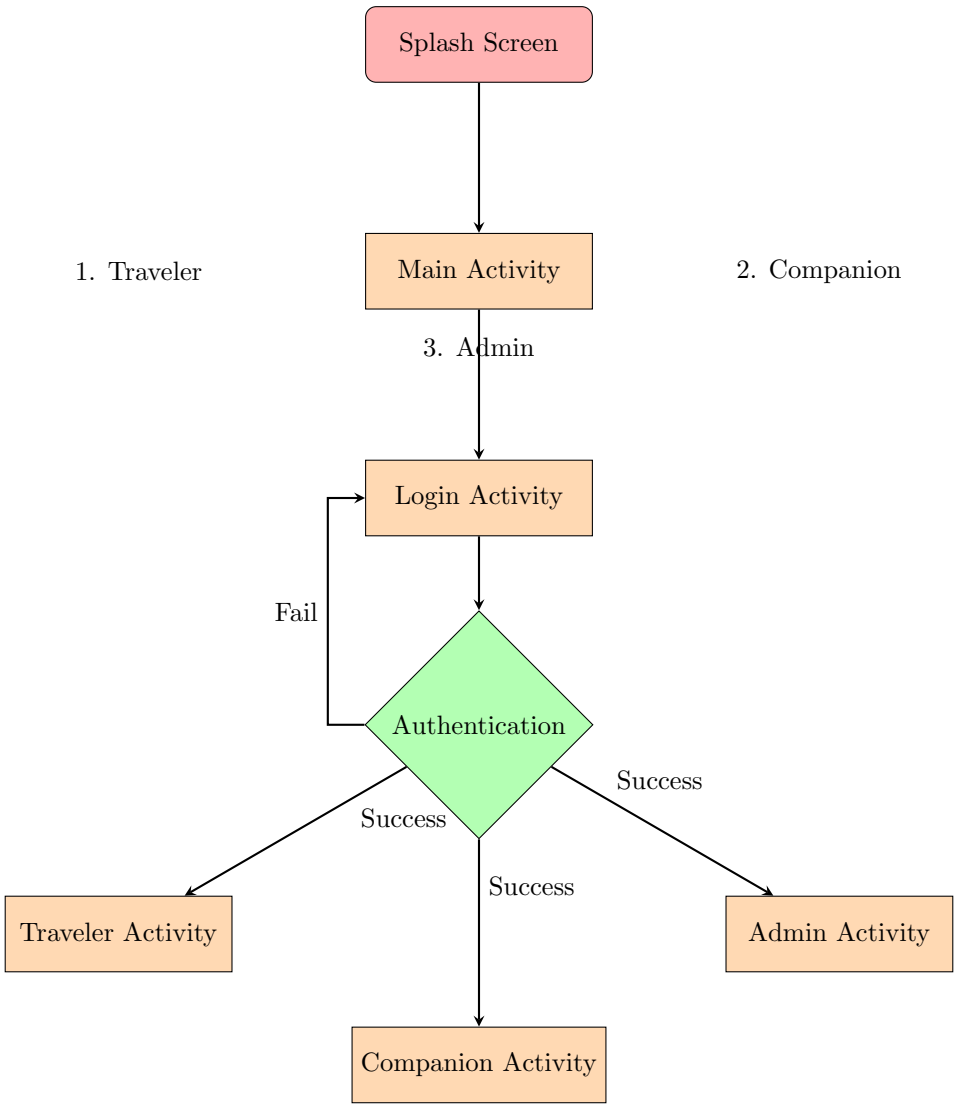

Figure 1: App Flow Diagram

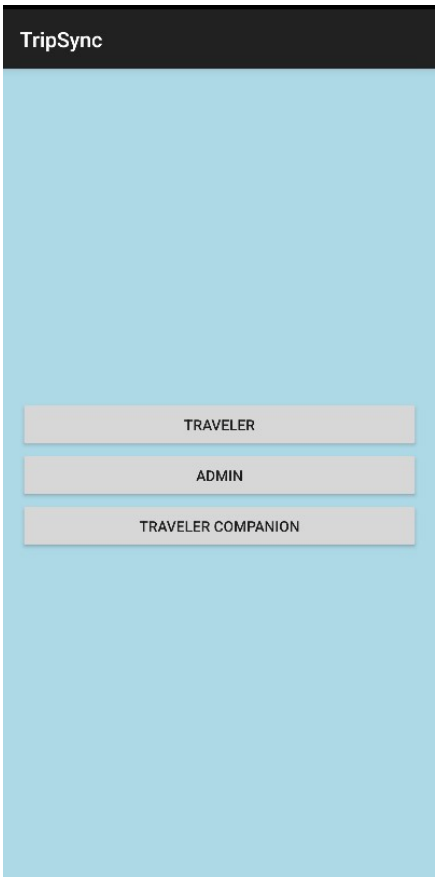

Figure 2: Image 1

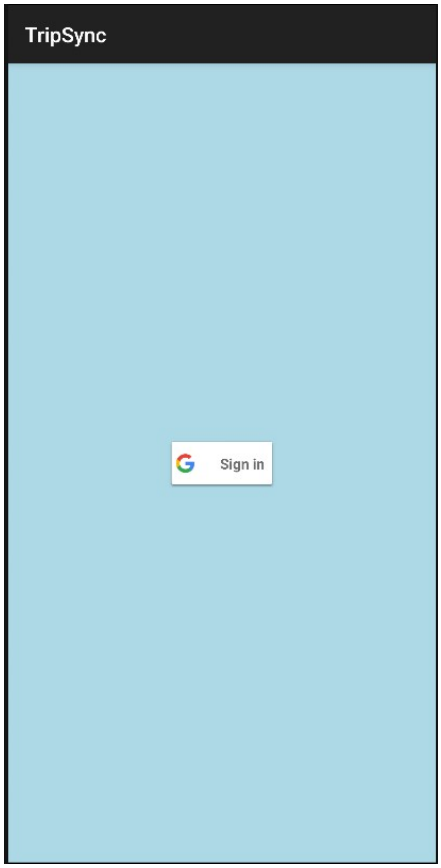

Figure 3: Image 2

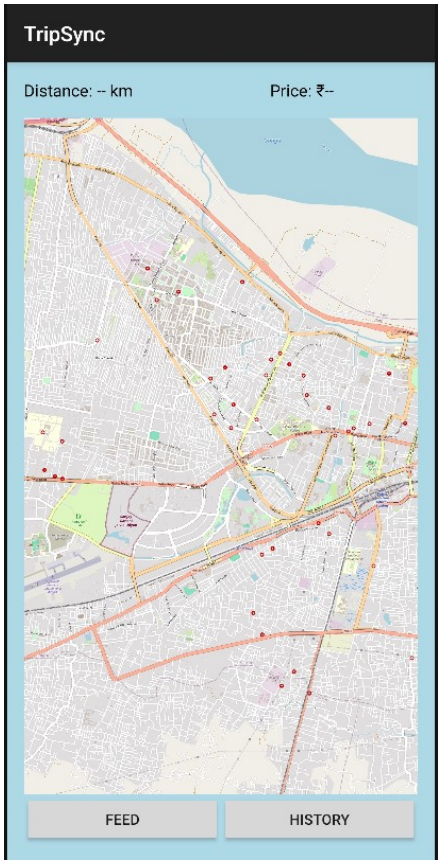

Figure 4: Image 3

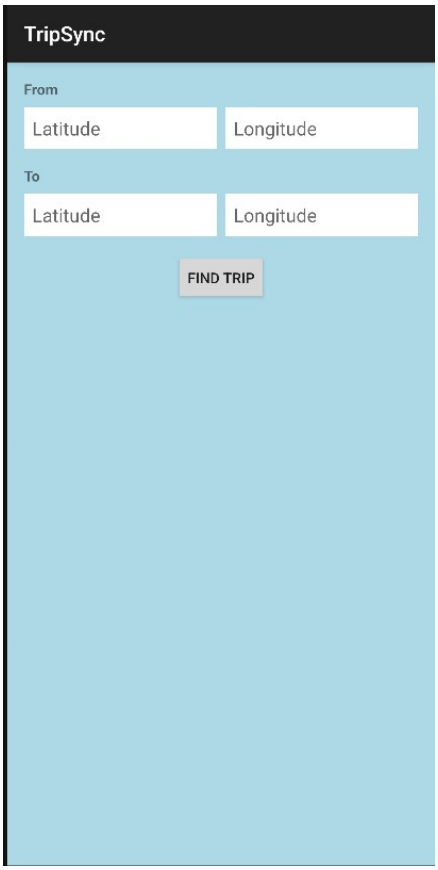

Figure 5: Image 4

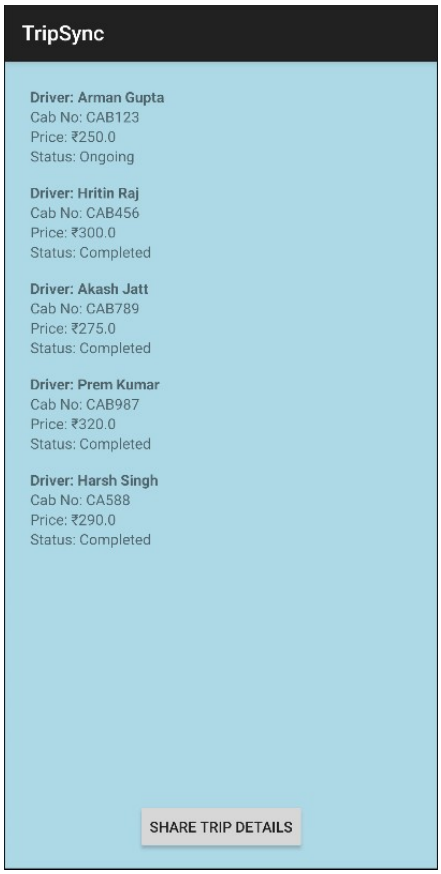

Figure 6: Image 5

Figure 7: Screenshots (Part 1)

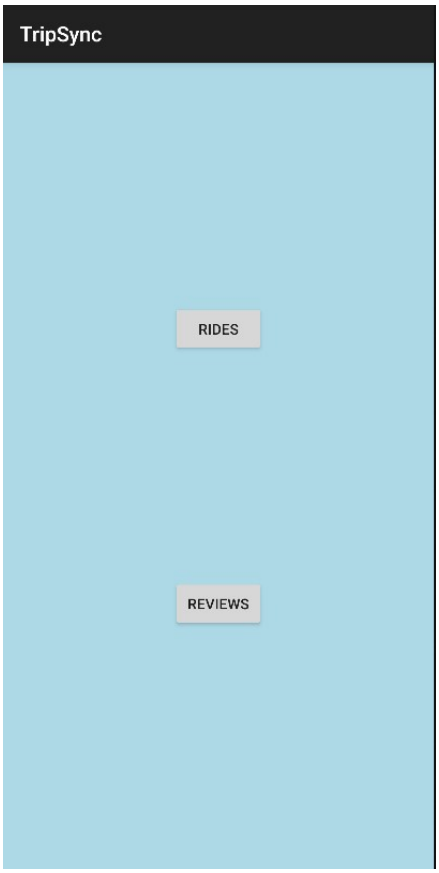

Figure 8: Image 6

Figure 9: Image 7

Figure 10: Image 8

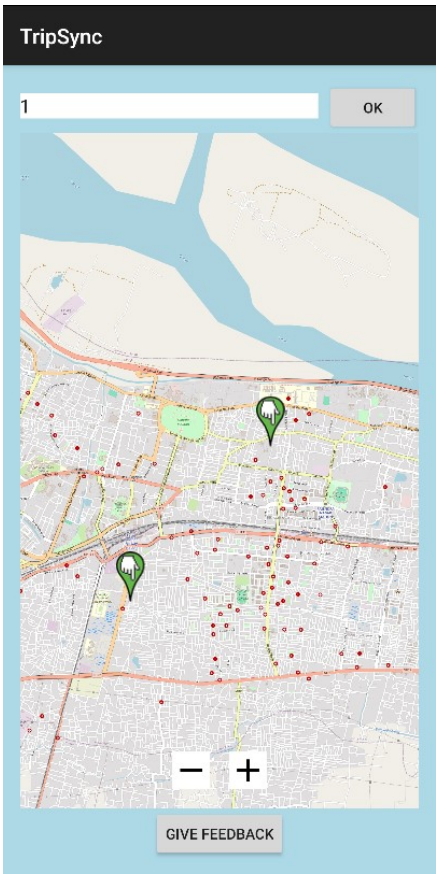

Figure 11: Image 9

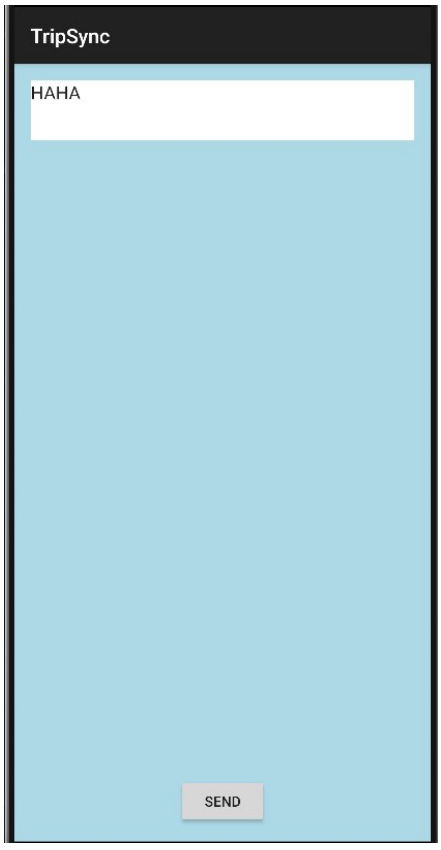

Figure 12: Image 10

Figure 13: Screenshots (Part 2)