50.001 Team 22

Campus Courier

Members

1006633 Du Bowei
1006924 Lim Jia Hui
1006933 Zhuang Yang Kun
1007011 Nicole Cheah Ching Suan
1007012 Mohamed Zuhairi Bin Mohamed Amran
1007037 Yang Rui
1007150 James Bryan Budiono

Table of Contents

1. Background....3
a. Our Problem and Our Proposed Solution....3
b. Resources Used....3
c. How Our Solution Addresses Sustainability Goals, Diversity and Inclusion....3
2. Our Prototype and Its Features....4
a. Login and Register....4
b. Posting Requests....5
c. Requestor's Home Page....6
d. Finding Requests....7
e. Supplier's Home Page....8
f. Completed Requests....9
g. History Page....10
h. Report System....11
i. Profile Page....12
3. System Design and Implementation....13
a. System Architecture....13
b. OOP Principles Used....13
c. Design Patterns Used....13
d. Parts of Our Project That Did Not Use Java....14
e. Other Key Technical Features....14
4. 2D Aspect: Our Data Structure(s) Chosen and Reasons For Our Choices....15
5. Possible Future Work....17
6. Conclusion....17

1. Background

a. Our Problem and Our Proposed Solution

Campus Courier is a student-centric app designed to streamline the acquisition of essential items for SUTD students who are facing time constraints due to their demanding academic workload, making it difficult for them to obtain essential items. Whether it's groceries, household items, or project materials, students can post requests for essential items they need. Other students can then accept these requests, purchase the items, and arrange for delivery, providing an avenue for them to earn some side cash while helping their peers.

Users can seamlessly switch between roles as requestors and suppliers, each offering a tailored user interface. Requestors benefit from real-time tracking of their posted requests, while suppliers manage their accepted requests through a to-do checklist. Ensuring accountability and reliability, both parties must confirm delivery to mark the request as completed and archive it in the history section.

The app allows for efficient browsing through a search function based on string input and filtering by location and category, streamlining the process of finding and fulfilling requests. The app also incorporates a penalty system, allowing users to report non-cooperative individuals, ensuring a safe environment.

Beyond its practical utility, Campus Courier cultivates a culture of mutual support and collaboration among students, enriching the university experience and fostering a sense of community within the SUTD campus.

b. Resources Used

We used Android Studio to code, and Google Firestore Database and Google Cloud Storage to store our users' data.

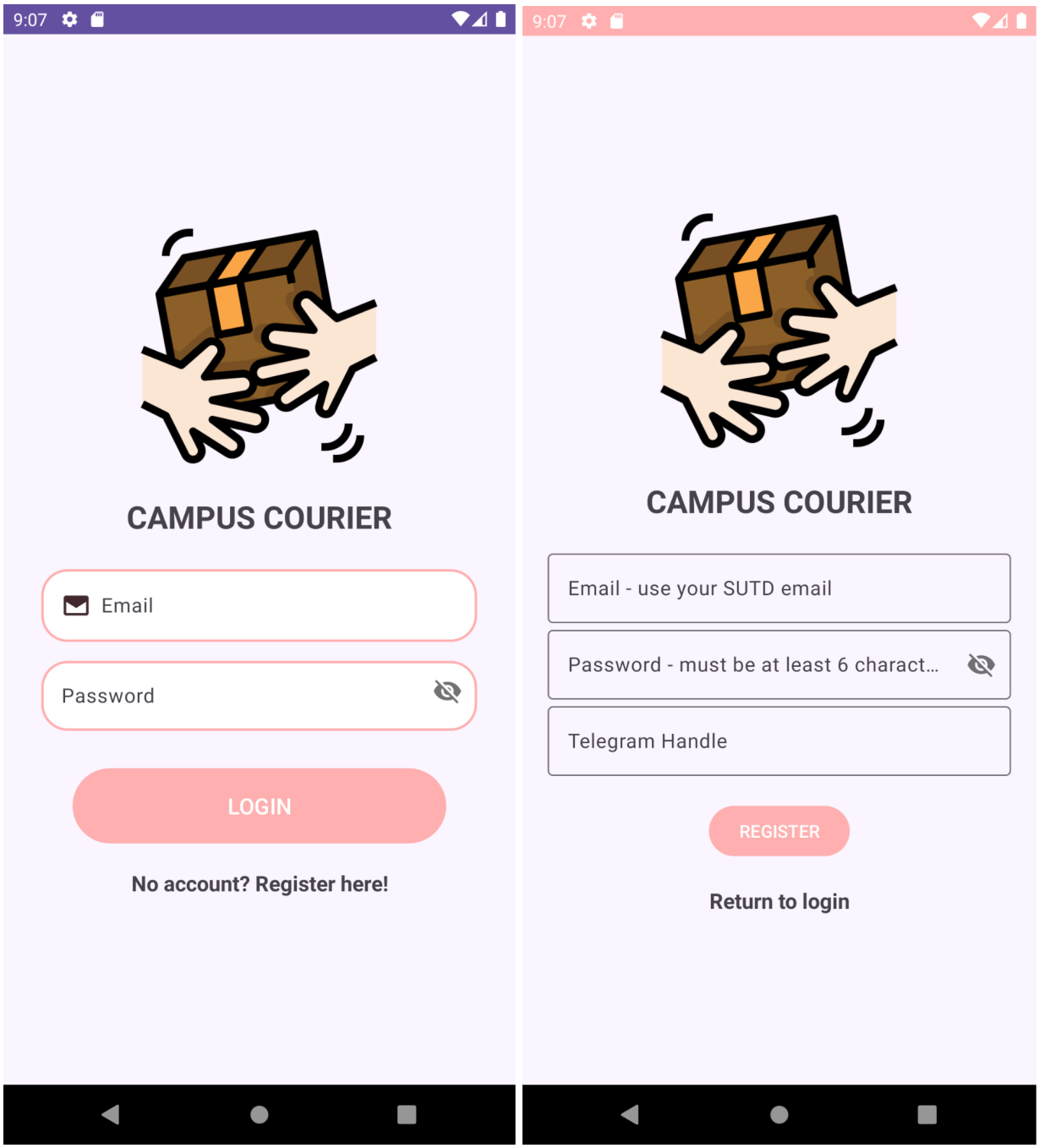
c. How Our Solution Addresses Sustainability Goals, Diversity and Inclusion

Campus Courier not only promotes environmentally sustainable practices by reducing carbon emissions through fewer trips, but it also prioritizes accessibility by catering to individuals who face physical challenges in obtaining essential items. By combining these two aspects, we aim to create a more inclusive and eco-friendly solution that benefits both individuals and the planet.

2. Our Prototype and Its Features

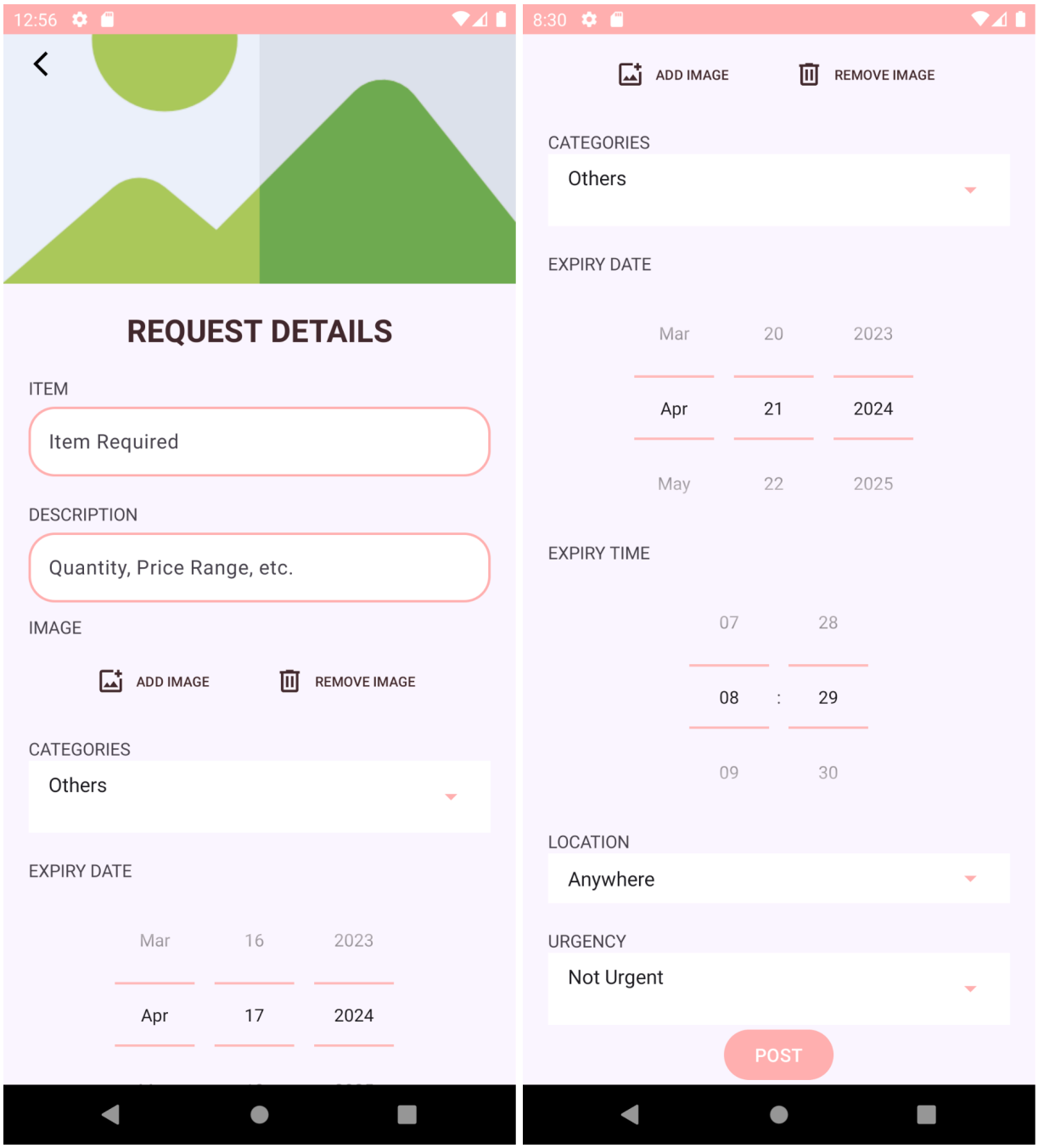
a. Login and Register

Users can login to their existing account or register for an account when they first enter our app. If they have already logged in before, they will automatically be re-directed to the “Home” page.



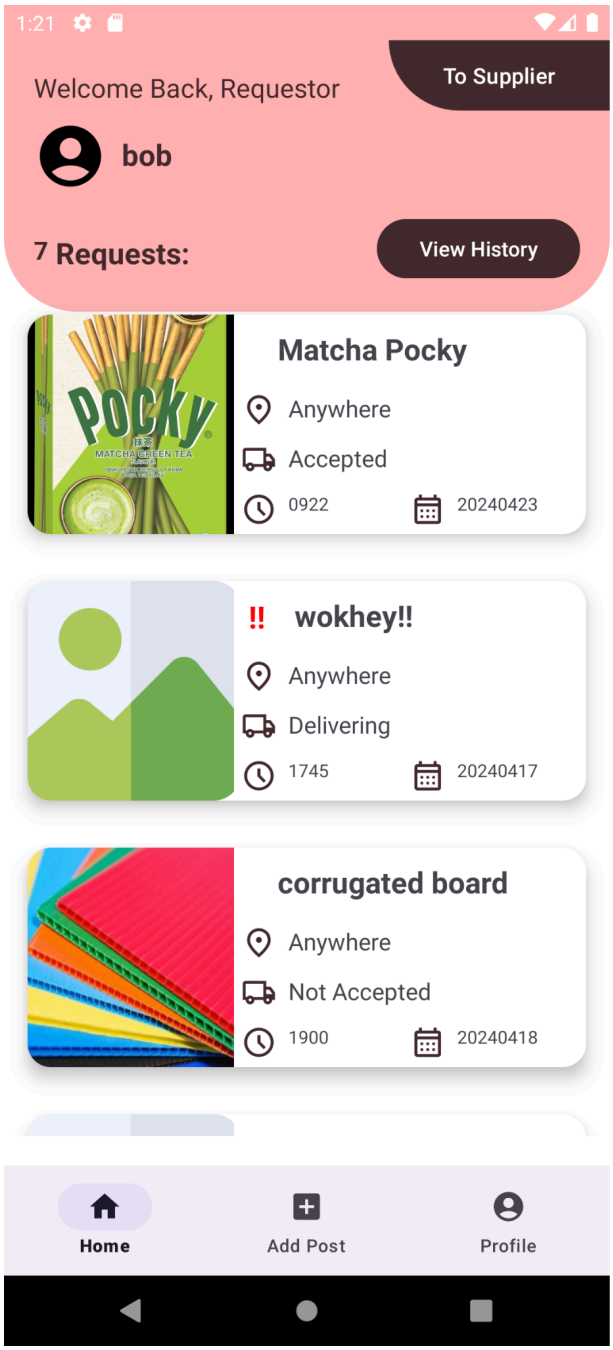
b. Posting Requests

As requestors, users can post new requests by simply filling in the information of their request, such as the description, category, and expiry date and time.



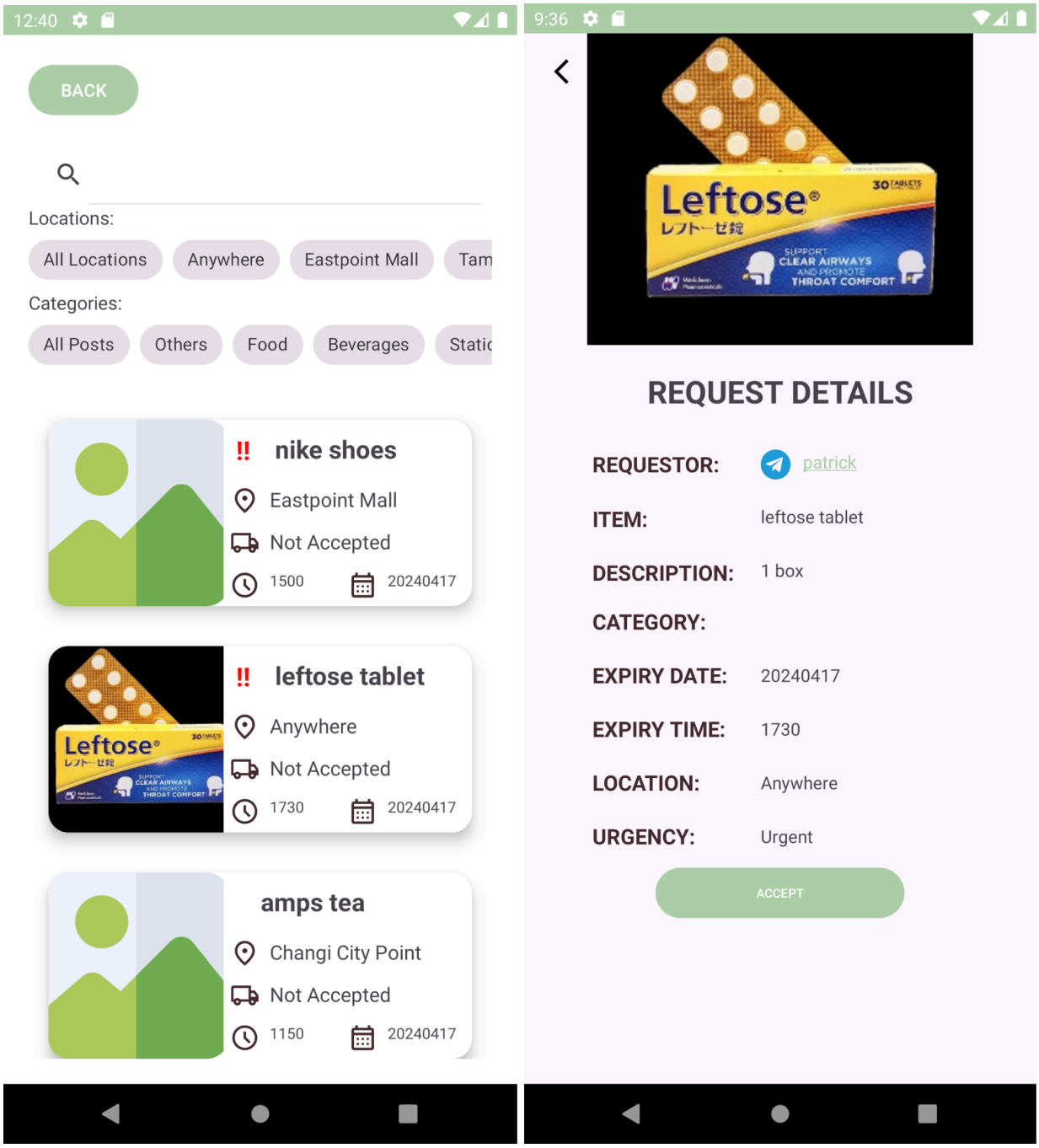
c. Requestor's Home Page

Their requests will then appear in their "Home" page. The status of their request (Not Accepted/Accepted/Delivering) will be displayed and updated in real-time.



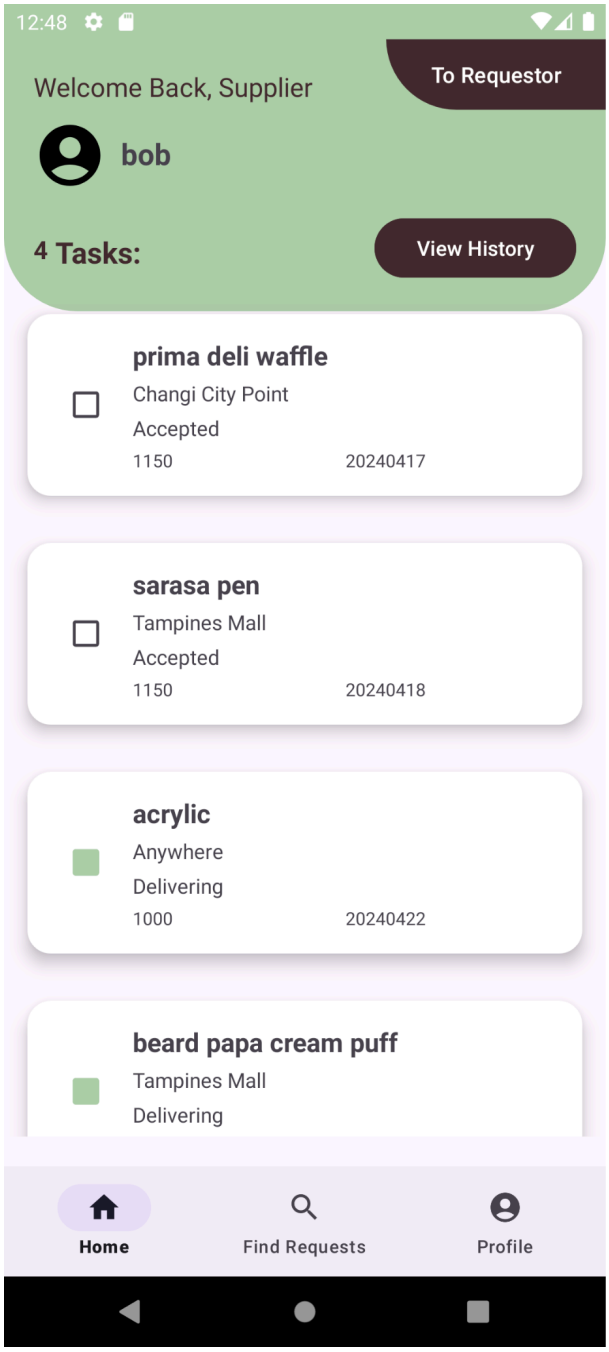
d. Finding Requests

As suppliers, users can browse for requests to accept through a search function based on string input and filtering by location and category.



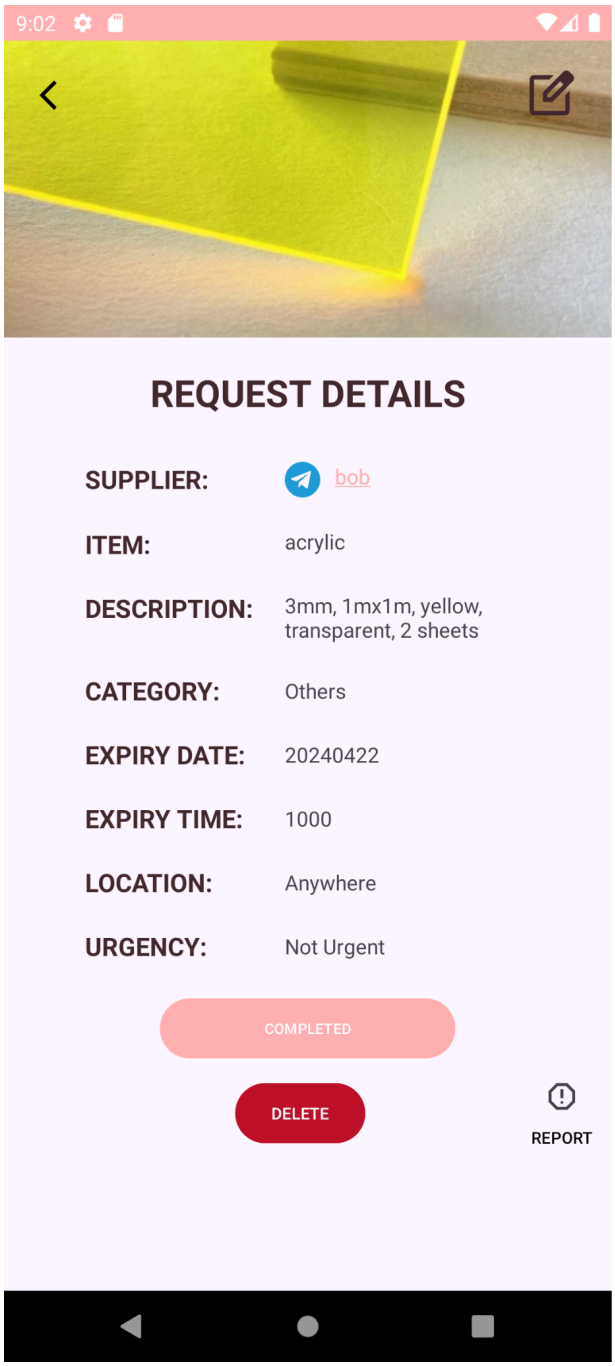
e. Supplier's Home Page

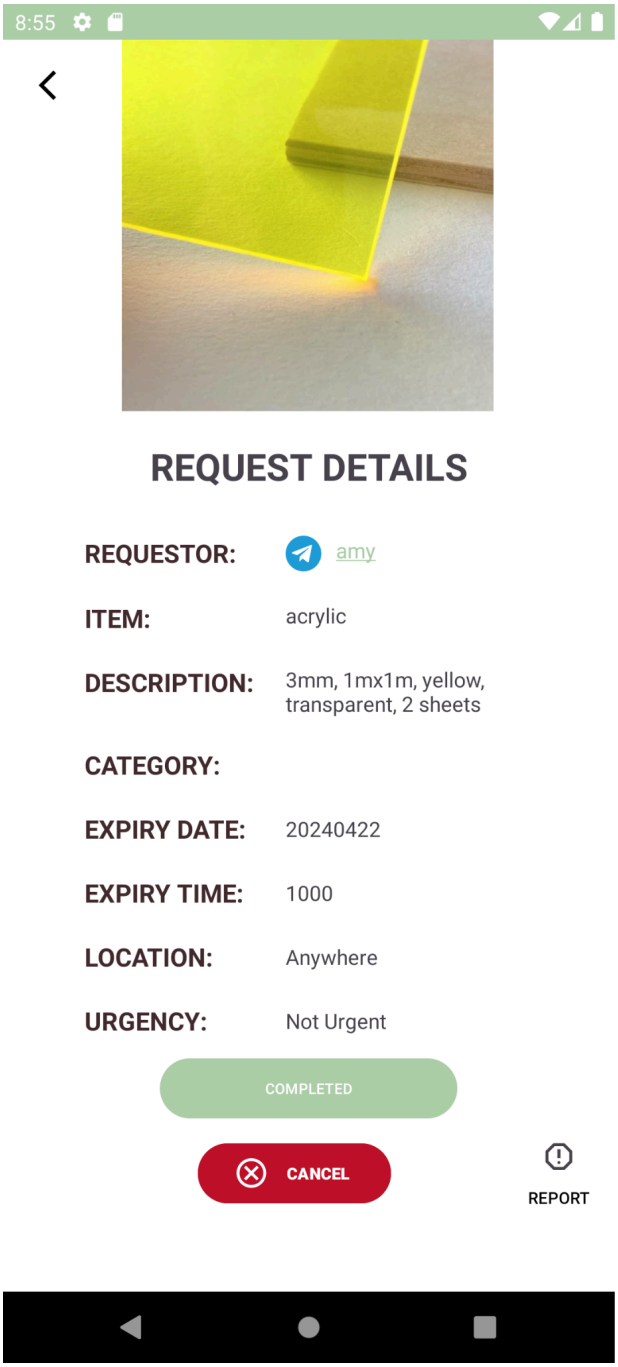
Their accepted requests will appear as a to-do list in their "Home" page. When they check a box, the status of that request will change from "Accepted" to "Delivering". The status will be automatically updated on the requestor's end.



f. Completed Requests

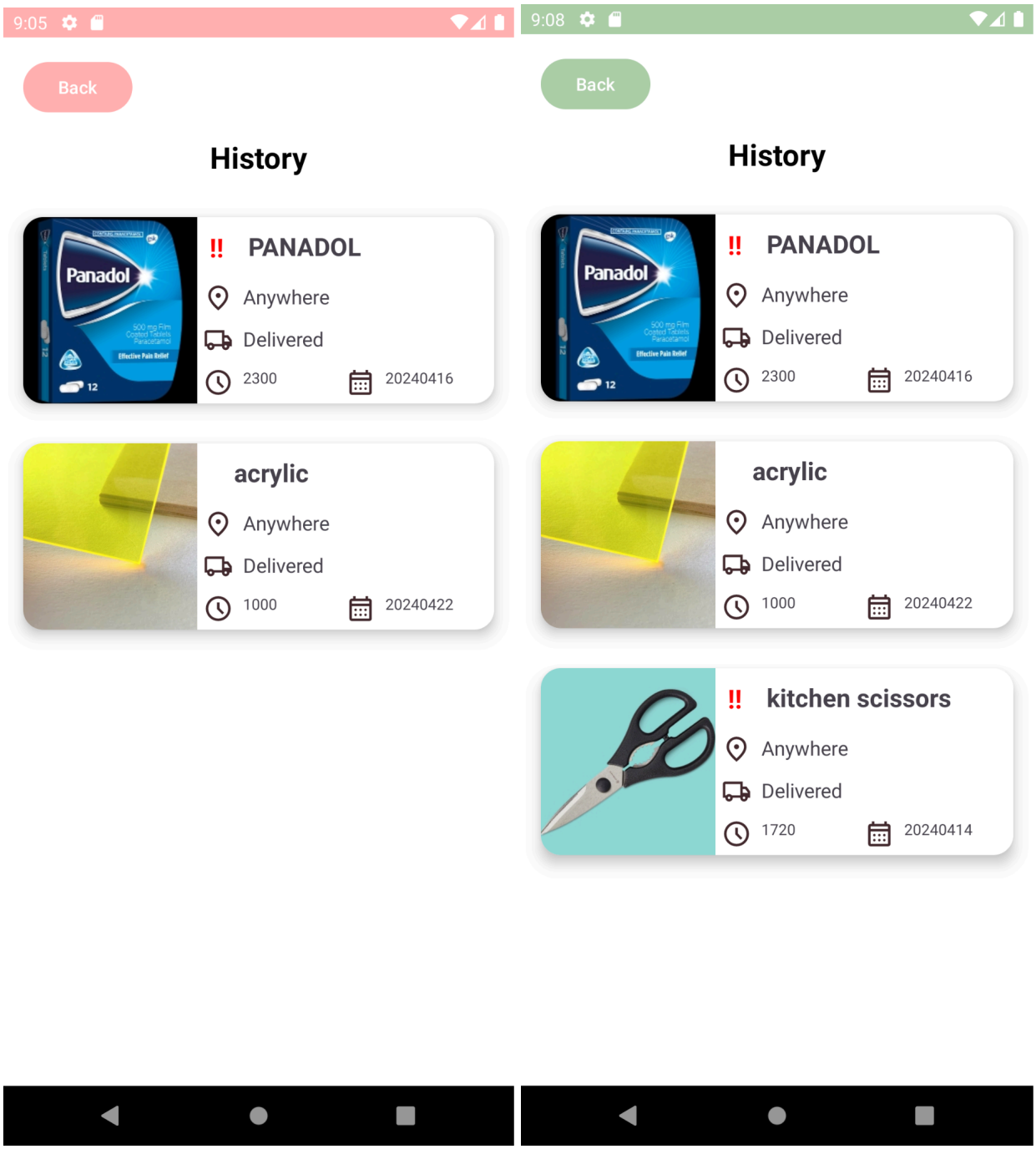
Both the requestor and the supplier must press the “COMPLETED” button on their respective ends for the request to be marked as successful and archived in each user’s “History” page.





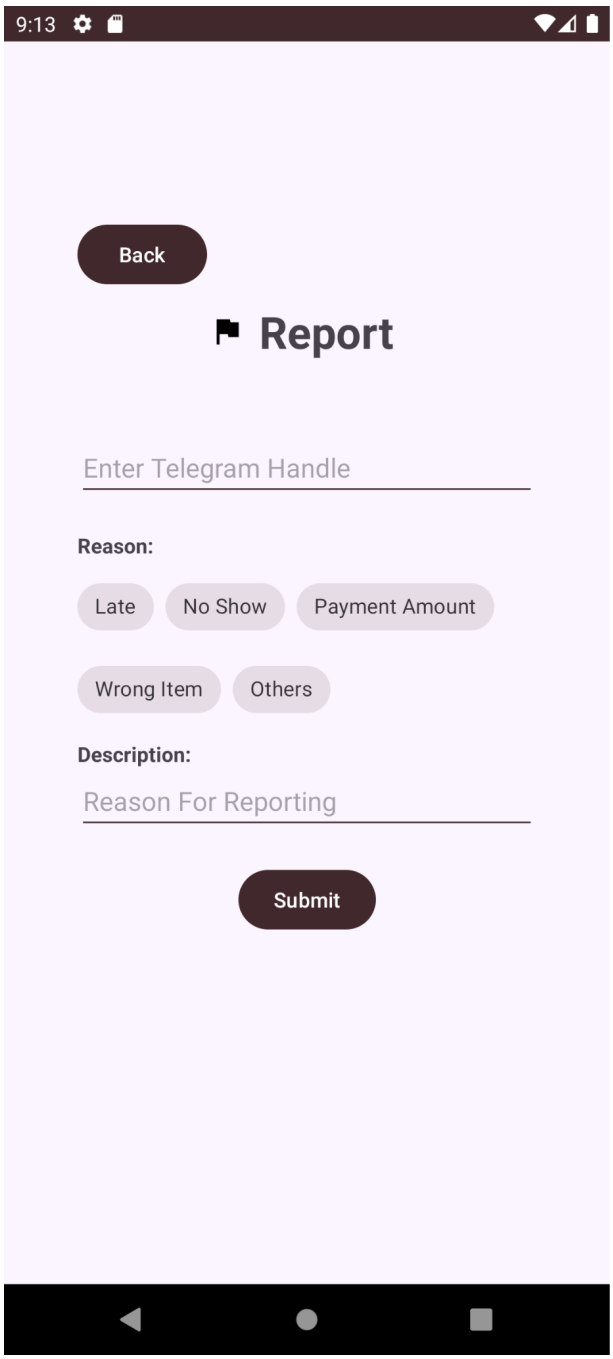
g. History Page

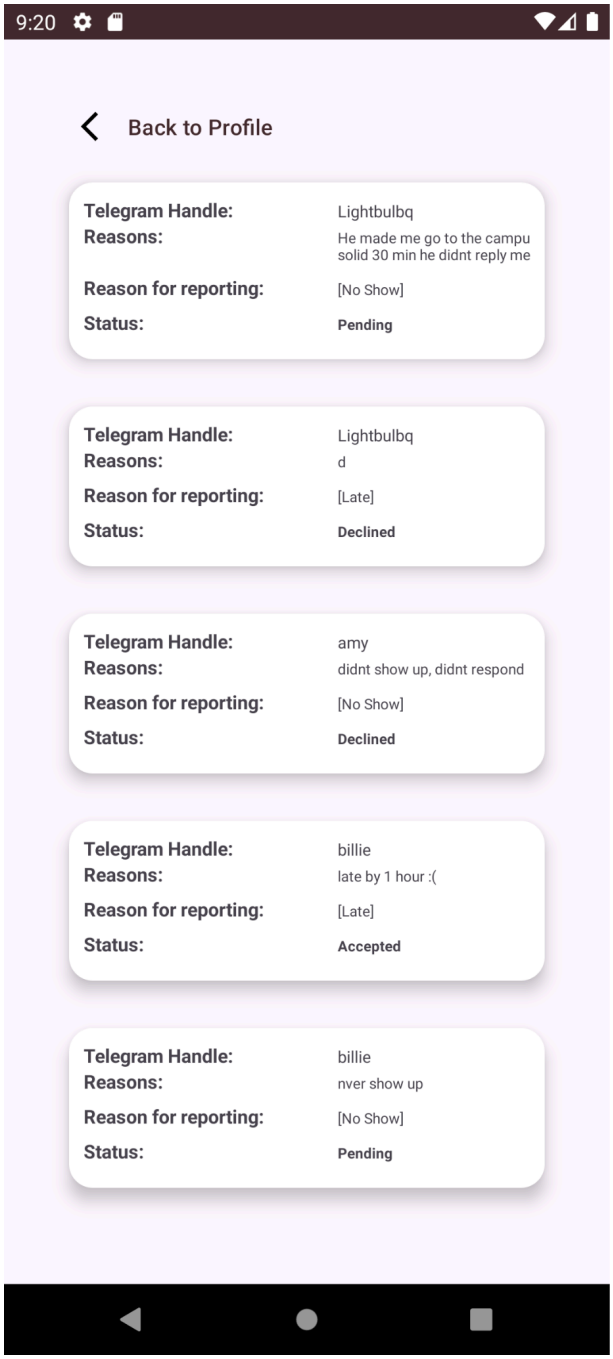
All previous successful transactions will be archived in the “History” page.



h. Report System

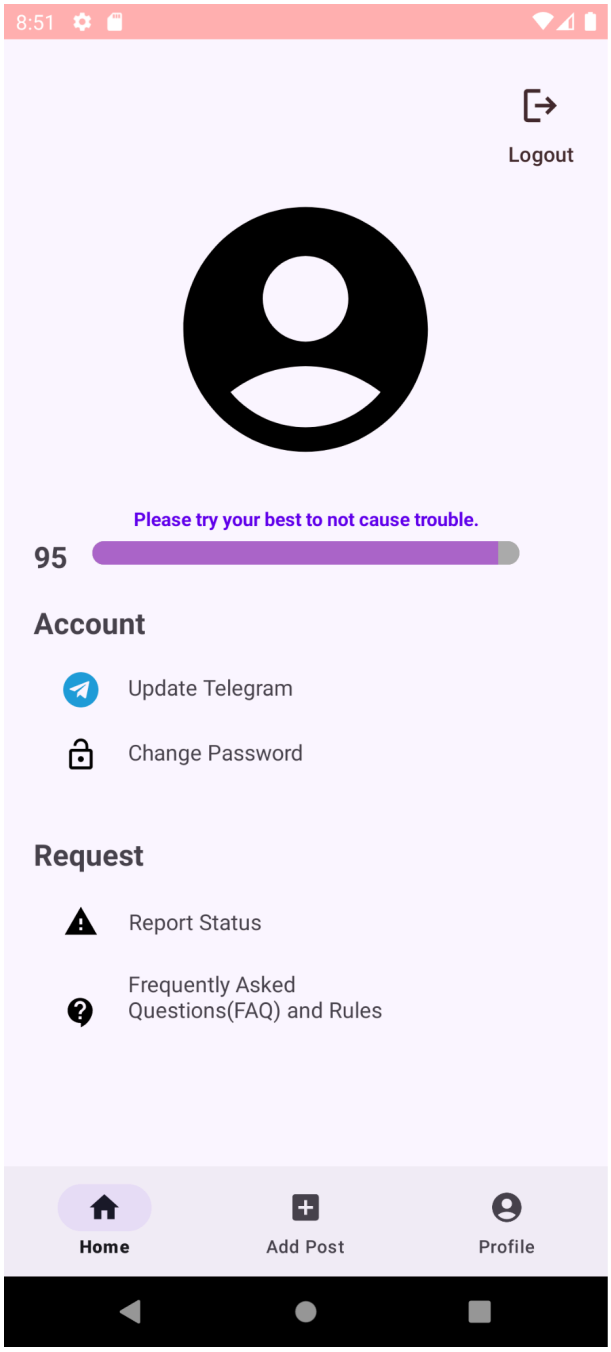
Our app also incorporates a penalty system, allowing users to report non-cooperative individuals and track the status of their reports (Pending/Accepted/Declined). If a report is successful, the user being reported will have their points deducted.





i. Profile Page

Users can keep track of their remaining points at their “Profile” page. They can also update their Telegram handle, change their password, track their report status, and view the FAQs from the “Profile” page.



3. System Design and Implementation

a. System Architecture

Client-Server Architecture: Our app follows a client-server model where the client-side interacts with the server-side for various functionalities such as posting requests, accepting requests, and tracking deliveries.

b. OOP Principles Used

Interfaces in OOP define a contract that classes can implement, specifying a set of methods or properties that the implementing class must provide. View.OnClickListener() is an interface that defines a method onClick(View v), which is called when the associated view (in our case, a button) is clicked. This allows us to define the behaviour or actions that should be performed in response to the user clicking the button, such as navigating to another screen.

c. Design Patterns Used

Our Requests class and Users class implement Serializable, which suggests the use of the Serialization design pattern. This pattern is used to convert an object into a byte stream that can be persisted to a file, sent over a network, or stored in a database. By implementing the Serializable interface, instances of the Requests class and Users class can be serialized, allowing for easy storage and transmission of Requests objects and Users objects.

In addition to implementing Serializable, we also included an adapter class responsible for managing data and creating the layout cards for each post in our RecyclerView. This adapter class plays a pivotal role in handling the visual representation of data within the RecyclerView. The ViewHolder class, which extends RecyclerView.ViewHolder is a crucial component within this adapter. It holds references to the views comprising each item in the RecyclerView.
Within the adapter class, we implemented two key methods:

1. onCreateViewHolder(): This method is invoked when the RecyclerView needs a new ViewHolder instance to represent an item. Here, we utilize LayoutInflater to inflate the layout for each item and create a new instance of the ViewHolder class to hold the views specific to that item.
2. onBindViewHolder(): When a ViewHolder is bound to new data, typically during scrolling or data updates, this method is called. Within onBindViewHolder(), we update the contents of the views in the ViewHolder to accurately reflect the data corresponding to the given position in the dataset. This involves setting text, images, or other UI elements based on the data associated with each item.

By effectively implementing these adapter methods, the RecyclerView efficiently manages view creation, recycling, and updating as users interact with the list. This approach ensures optimal performance and a good user experience, particularly in Android apps displaying lists with dynamic data.

d. Parts of Our Project That Did Not Use Java

In our project, we used XML to create how the app looks and feels for users. We designed screens like login pages and user profiles using XML tags and settings for things like buttons, text sizes, and images. XML also helped to manage things like colours and styles across the app, making everything look consistent and easy to update. It was especially useful for organizing lists of items and menus. Overall, XML made it easier to design a good-looking and user-friendly app.

e. Other Key Technical Features

We used Firebase Authentication API for implementing user authentication and authorization.

4. 2D Aspect: Our Data Structure(s) Chosen and Reasons For Our Choices

We mainly used Google Firestore Database and Google Cloud Storage to store our users' data.

Firestore Database is a NoSQL document-oriented database that organises data into collections and documents. Collections act as containers for documents, which are individual records stored as JSON-like objects. Each document contains fields and their corresponding values, allowing for flexible data modelling.

We also used Firebase Storage as it provides cloud storage for our images. It stores your images in a Google Cloud Storage bucket, making them accessible through Firebase and Google Cloud. This allows our users to upload images from mobile clients via the Firebase SDKs for Cloud Storage.

We also used Firebase Authentication as it provides a straightforward way to authenticate users using email/password. This allows users to create accounts and sign in securely. Firebase Authentication also smoothly integrates with the other Firebase services we used, such as Firestore Database and Firebase Storage. This allows us to enforce access controls based on the authenticated user's identity and manage user-specific data securely.

We organized our data into three distinct collections within the Firestore Database: 'posts', 'report', and 'users'. The 'posts' collection stored individual requests as documents, with each field capturing necessary details such as request status, expiry date, requestor ID, supplier ID, and document ID. This approach of storing requests as separate documents within the 'posts' collection facilitated easy retrieval of request-related information. The 'users' collection held user profiles as separate documents, containing email addresses, Telegram handles, demerit points, and posts made by each user, which streamlined user management and authentication processes. Additionally, the 'report' collection tracked all reports made, ensuring efficient data storage and retrieval.

In our code, we made sure that the Android app was properly integrated with Firebase by adding the necessary Firebase dependencies in the project's Gradle files and configuring Firebase services through the Firebase console. We then initialized the Firestore database in the Android app by obtaining an instance of FirebaseFirestore. We then obtained a reference to the Firestore collection where the data was to be stored using Firebase Instance.collection("name of collection path"). We could then access the Firebase through this call, and carry out the corresponding functions (to update, add or delete) on our collections.

Our use of Firebase helper functions was pivotal, especially in implementing RecyclerView functionalities. These functions used different layout cards for various sections like admin reports, report status, and requests, enhancing the visual representation of data. By connecting to our database, we could fetch specific data for each request and update the UI with real-time information stored in Firebase.

Another data structure we used was ArrayList for our RecyclerViews. We used recycler views multiple times within our app for various reasons to list down the requests made on the requestor side, and the list of available requests to be accepted on the supplier side. This allowed us to display the requests in layout cards that we made, making our data more presentable and easier to understand. We used an array list as the data structure that holds the data to be displayed in the RecyclerView, as it dynamically grows or shrinks to accommodate changes in the dataset, such as adding or removing items to the list. The RecyclerView adapter then interacts with this ArrayList to populate the RecyclerView with views representing each request in the list.

This comprehensive approach not only optimized our data management workflow but also guaranteed a dynamic and responsive user interface. Users could access up-to-date and relevant information without encountering delays or inconsistencies, enhancing their overall experience with the app.

5. Possible Future Work

Firstly, real-time updates will ensure that users experience immediate feedback and information refreshes as they interact with the app. This includes instant notifications or updates on changes, such as new requests, updates to existing requests, or status changes. By optimizing data synchronization and leveraging efficient communication protocols, we can minimize latency and provide users with a seamless and responsive interface.

Secondly, transitioning from external messaging platforms like Telegram to an in-app chatting system offers several advantages. Users can communicate directly within the app, enhancing convenience, privacy, and security. We can implement features such as real-time messaging, multimedia sharing, message threading, and notifications to create a robust communication platform tailored to our users' needs. Integration with existing user profiles and authentication systems will further enhance the user experience.

Lastly, introducing in-app money transfer functionality revolutionizes how users engage in transactions. By enabling direct payment transfers between users within the app, we eliminate the need for manual intervention or external payment gateways. Users can securely transfer funds, manage transaction histories, and receive instant notifications, streamlining the entire payment process. Implementing robust security measures, compliance with financial regulations, and user-friendly interfaces will be paramount in ensuring a smooth and trustworthy payment experience.

6. Conclusion

Campus Courier represents a concerted effort to address the time constraints and accessibility challenges faced by students at SUTD in obtaining essential items. Through our user-centric app, students can seamlessly connect as requestors and suppliers, facilitating the acquisition of necessities while fostering a collaborative and supportive community within the university.

Our project leverages Google Firestore Database, Google Cloud Storage, and Firebase SDK for Java to create an efficient platform. The integration of Firebase Authentication ensures secure user interactions.

Moving forward, we envision further enhancements to our app. Real-time updates will provide immediate feedback and information refreshes, enhancing the overall user experience. Using Firebase Cloud Messaging can enable real-time updates and notifications, enhancing user engagement. Transitioning to an in-app chatting system will improve communication privacy and convenience. Additionally, implementing in-app money transfer functionality will revolutionize transaction processes, offering users a seamless and secure payment experience.

Campus Courier embodies our commitment to sustainability, inclusivity, and technological innovation. By embracing these principles and continuously refining our app, we aim to create a lasting impact on student life at SUTD and beyond.