Project: College Carpool

Author: Stephen Cassedy

Course: CASE4

Student Number: 13391736

Supervisor: Marija Bezbradica

Abstract

College Carpool is a mobile application that I designed for my final year project in DCU. It is a carpooling app focused on college students, limited to DCU where they can message each other, organise rides, request real-time lifts and even (optionally) donate money to each other through contactless payment. It is designed to help students of DCU to commute to college and could also help a few drivers not too get lost on the way in with an inbuilt navigation system.

College Carpool Technical Manual

1 Introduction

1.1 Overview

College carpool is an application I have designed and implemented for my Final Yea Project for Computer Applications. It has been a year-long project that I have worked on continuously. The mobile application (app) is a ride-sharing and planning oriented system by which users can communicate and share rides. Aimed at Dublin City University (DCU), this application is something that I have developed from the bottom up and developed specifically towards what I think a college student would desire in a modern app.

1.2 Motivation

The reasons that I have chosen to make this app are mainly student oriented. The side-effects may also lead to less traffic congestion in the campus area as well as an increased availability of parking. During my time in DCU, I have become aware of the problem of both commuting to college and getting parking. As I live close to the college, it has not become a real problem for me but for my fellow students. This app acknowledges the fact that while many people need to commute to college, not everyone is able to drive. While trying to decide on a project to do, I tried to imagine something that could have a practical use for students and may also benefit the university. I noticed that some of the people in my class had begun to carpool as I'd often see them coming in from the M1 motorway just 500m up the road from college. This group also had become friends. Then the idea came; why not develop a system that students could use to organise rides to college? Through this, students may also make new friends and connections which would lead to an overall better experience with college. This was my motivation to pick this project. The system would incorporate any DCU student, past and present so that the college 'family' could continue to keep in touch both during and after college to enrich the connectivity that our college encourages. A bi-product of ride sharing would be the reduced amount of traffic in the area as not everyone would need to drive and could possibly set up a rota system. Less cars means less traffic and more parking, something I think we can all sympathize with.

1.3 Glossary

- Firebase – An application development platform hosted online that contains a suite of tools such as a database, crash reporting, and the ability to program server side code in small node.js functions.
- Data Snapshot – An instance of a database section from a specific point in time. Part of the firebase suite.
- Firebase console – The user interface where a developer can monitor the status of a firebase linked application.
- Google Directions API – A web service that allows a user to make a request consisting of various parameters that will return a JSON file that contains information on a journey.

- Google Places API – A web service that allows the user to access places located near a location.
- Google console – A place where a developer can manage his API keys and monitor the quantity of calls that are being made by the applications linked to each key.

2 Research

2.1 Firebase

Before I started development, I knew I would need certain to accommodate my app. A list of these requirements would be a database, authentication system and a way that would enable me to allow users to communicate seamlessly. I began searching for a viable solution that would satisfy these requirements. For a long time, it seems like MongoDB would allow me to manage my system. I then came across firebase, a development platform that had only become well-established in May 2016 after being acquired by Google in October 2014. As I looked more into the platform, it seemed that it satisfied all my needs along with many extra features that supported a growing app. I also saw using this as a way experiment with a modern technology that I had no experience with before. The main attraction as far as I was concerned was the ability to have a real-time database that required no serve side code to run. This allowed me to begin my development without the need to get bogged down in setting an environment up. I spent a few days playing around with features on the firebase console but was limited in what I could do. I believe one of the most attractive features of firebase is its versatility in the sense that it can be used across multiple platforms (web, android, iOS, unity and c++) so is perfect for an app that is expected to grow.

2.2 Directions

Directions are a key part of the app. They enable three things: the satellite navigation system, viewing other user's active journeys and the viewing of friends planned journeys. I spent time researching how I was going to be able to draw and represent a journey on a map so that it was easily understandable and immediately clear what it represented. It seemed that Google maps supported polylines, a sequence of latitude and longitude coordinates that could be linked together, and markers which could be used to represent waypoints or stop offs of a journey. This is how my system would represent user journeys. When it came time to drawing polylines, google had sufficient APIs that allowed me to quickly learn how to manipulate map features. Through my time working with the API, I found flaws with it and how my system was to be implemented that I did not foresee at the start but I will mention these later.

2.3 Android

I had not previously worked with android for application design so began reading about how an android application should be designed. Due to java being the primary programming language that I learned in college and the primary language used in android development, I took to it quite quickly. Basic concepts were that an application could be split into activities and fragments. Activities represent different screens for different functionality while fragment represent different layout of an activity. Effectively an activity can be split into multiple fragments if the developer wishes. Java classes remained the same as ever except I now had access to phone hardware features such as GPS and near frequency controller (NFC) cards to work with. I also had to come to terms with how views work in the android system and how I could move user interface elements

around. This was all a steep learning curve but once the ground work was put in, it became very intuitive to use for the most part.

2.4 Hardware

A feature of my app is a mock payment system where a user can take advantage of contactless payment between phones. This was made possible through by NFC cards. These are small chips that exist in some phones (growing in popularity) that can emit or receive a radio wave. The cards must be within a very short distance of each other. For use with phones, the backs of each phone where the card is placed. I had no experience with the cards before and only got a phone capable of working with the cards during development so I had to research how they worked. The android support documentation was very helpful in learning to work with them.

2.5 Location

I knew from the start of the project that location was going to play a major key in the implementation of my app. I had to do some pre-requisite research into how to get a phone's location. Viable options seemed like Geolocation, Location Manager and Location Services. After research into the area, I decided to opt for the location service's fused location API method as the android support documentation said that it was the most recent, most accurate and least power consuming method of obtaining GPS location.

3 Design

3.1 Backend Overview

As I mentioned before, my application was integrated with the firebase platform. It made development a lot easier in my opinion as it allowed the use of its suite of features with ease of use. Mainly using the database to store user profiles and their associated elements, information was stored in JSON format where everything is mapped in a parent and child. Firebase also handled my authentication with its inbuilt authentication system. Firebase functions were a new feature that added to firebase that I thought I would use in my app. The feature only came out in March 2017. These allowed me to write JavaScript code directly into firebase that could manage aspects of my database. Other aspects of my backend were run with the Google Directions API and the Google Places API.

3.2 Back End Features

3.2.1 Authentication

Firebase utilizes a FirebaseAuth instance to manage users who are currently logged in. A user must obtain an instance by signing in with valid credentials. Firebase manages my authentication system. This means that user passwords are out of my control. I have implemented the 'sign up with e-mail and password' method of authentication. I had also previously implemented the ability to sign in with google but decided to remove it. Processes behind google login included getting a secure hash algorithm certificate to accompany the OAuth 2.0 security protocol. I obtained this from my Google API console. I had intended to pull user profile images from their google account and display it on the map to represent

where a user was. It made sense to have google authentication as every android phone has a Google account associated with it. The only problem with my plan was that many users ever upload a photograph to accompany their account and therefore the idea didn't work out. Implementing a feature such as mapping a user to a map was also outside of the project scope and, although it would have been a nice feature, it wasn't necessary. I opted to stick with my own custom e-mail and password login. Authentication involves going to the firebase console and navigating to the authentication section. Here you will be given a list of viable authentication options. These include phone number, GitHub, Google, Facebook, e-mail and password and Twitter. An anonymous option is also available but I believe this is more for testing purposes. Firebase allows me to use the 'createUserWithEmailAndPassword' method to add a user to the authenticated users list. The firebase backend will then remember this user if they are in the user list and authenticate them when they attempt to sign in with the 'signinWithEmailAndPassword' method. Firebase will then give them access to Firebase features such as the real-time database or storage for example.

3.2.2 Cloud Functions

I used firebase functions in the design of my app. These are small programs that are run from the firebase and not directly from the client device. Doing this reduces the amount of code that to be run by the client and allows functionality to be handled by the server. It allows all code for the functions to be stored in the same place to. Function scripts are written in node.js and offer monitoring for events such as changes in storage, database and authentication. I found it very hard to find good documentation on these functions due to how new they are. Firebase do 'firecasts' on YouTube where firebase employees do run throughs of firebase features and I found these more helpful. I use firebase functions to monitor database changes. I do this by monitoring 'onWrite' events which monitor changes that occur in specific parts of the database. Sample code is provided below:

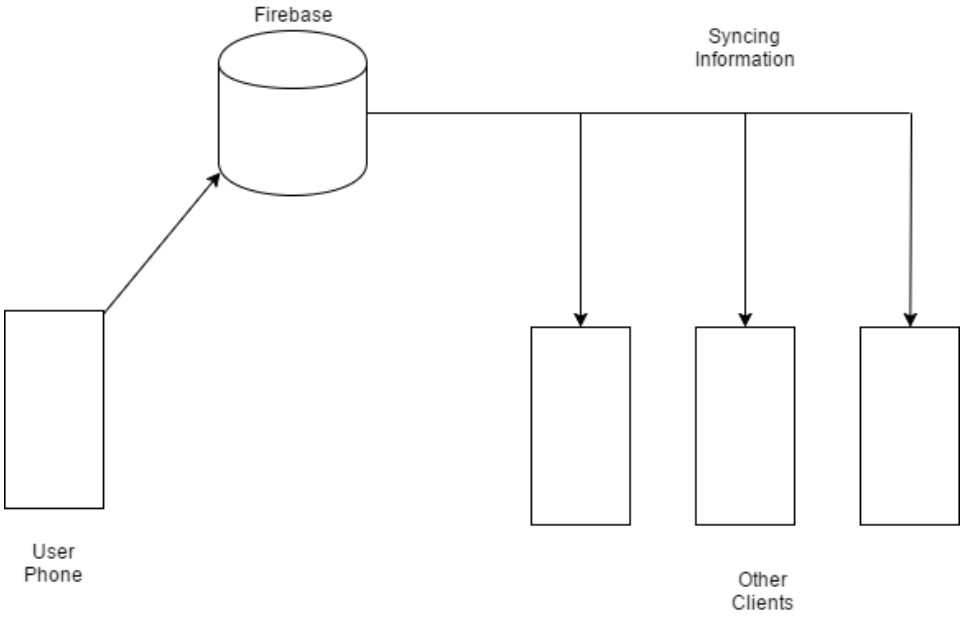
```
/*Listen For A Friend Request*/
exports.friendRequest = functions.database.ref("/UserProfile/{receiverID}/FriendRequest/{requestID}/")
    .onWrite(event => {

        const request = event.data.val();
        console.log("Friend Request: " + request);
        const userName = request.userName;
        const receiverID = event.params.receiverID;

        sendFriendRequestNotification(receiverID, userName);
    });

    function sendFriendRequestNotification(receiverID, userName){
        var database = admin.database();
        var ref = database.ref("UserProfile/" + receiverID);
        var fcmToken;
        ref.once("value", function(snapshot){
            fcmToken = snapshot.val().fcmToken;
            console.log("FCM Token: " + fcmToken);
            const payload = {
                notification : {
                    title: "New Friend Request",
                },
                data : {
                    "type" : "friendRequest",
                    "username" : userName
                }
            };
            sendNotification(fcmToken, payload);
        });
    }
```

In this case, I monitor for a change in "/UserProfile/{receiverID}/FriendRequest/{requestID}/" where the chain brackets represent wildcards. Wildcards are variables names that can change. For example, using the wildcard for the receiver I.D. will mean that it will monitor for changes in any user's profiles.

I use functions to listen to changes for the following reasons: Monitoring messaging, monitoring ride requests and their response (either accepted or declined) and finally listening for friend requests. I use this in conjunction with the firebase messaging service (FMS) which I will talk about later as it is handled by the client phone. I had never worked with JavaScript before so I had trouble learning some of the basics. Once I came to terms with basic operations though I was fine as the script for managing all my functions is only 190 lines.

3.2.3 Real-time Database

Firebase provided me with a real-time database to use. In this I store user information in the form of user profiles. The main attraction for me with this is it allows me sync information in real-time. This was useful due to the nature of my app. It depends on real-time information to allow users to communicate and share rides. Without it, it would defeat the purpose of my app. Things that depend on the real-time nature are the messaging system, user latitude and longitude and active journeys. The database is useful as it syncs data in real time to all users so data is readily available to everyone. This is highlighted in the below diagram:



3.2.3.1 JSON Database

The firebase database is structured in JSON format. It doesn't run off SQL. Information is stored as key-value pairs where a key represents a parent node and a value represents a child of this node. There is no specific structure apart from this as JSON is nearly like a giant text file in its basic form. It cannot be searched like a relational database. When I wish to search for it, I do it by search for specific keys and navigation down through the layers until I find the value I am looking for. The max depth for a firebase is 32 layers deep so there is plenty of room. My max depth is 7 layers deep so there is plenty of room for expansion.

3.2.3.2 My Database Structure

My database is mainly structured around the idea that mainly all information is stored under user profiles. Each user profile is stored by a unique key. This key is the current Firebase user I.D. Every time a user authenticates to the Firebase server, they are given an authentication instance. A unique user is generated for each unique person that authenticates to the server. The authentication instance contains information about the current user. On the client side, this is represented by a FirebaseUser object. From this user, we can get the user I.D. to store the user information under. Incidentally, this is also the object that would have contained the user's Google image if they had signed in with Google authentication. All user information is stored under the individual's user I.D. Sample database structure provided below:

```
UserID:
        JourneyPlanner:
                TimeStamp 1:
                        Day: 26
                        Month: 5
                        Year: 2017
                journeyWaypoints:
                        List Element 0:
                                latLng:
                                        lat: 53.4256
                                        lng: -6.1316
                                name: Portmarnock
                        List Element 1:
                                latLng:
                                        lat: 53.5116
                                        lng: -6.3971
                                name: Ashbourne
                TimeStamp 2:
```

Messaging:
senderID:
TimeStamp:
Message: Hello
Sender: senderID

Email: stephen.cassedy2@mail.dcu.ie
FirstName: Stephen
SecondName: Cassedy
Lat: 52.323
Lng: -6.321

Storing my data this way allows me to keep all information relating to each user in a specific place. I have written firebase rules which stop users accessing areas of the database that they are not meant to be in. This is done from the firebase console. An example of the rules would be that a user can read and write to his own section of the database to any area but another user may only read or write to a specified area. Sample rule provided below:

```
"UserProfile": {
  "$uid" : {
    ".read" : true,
    ".write" : true,
    "FriendRequest" : {
    ".write" : true
    },
    "Messaging" : {
      ".write" : true,
          ".read": true
    }
```

These rules allow me to store data securely. Firebase cloud functions run with administrative privileges so that they can access all parts of the database. The database can be monitored from the Firebase console. Information can also be changed by the admin supervising it. This proved useful for testing as I often required a data change to trigger functions in the client device.

3.3 Front End Overview

Since my application adapts a client-server model, the front end is the application itself. It is designed to run on Android OS standard development kit (SDK) version 19 and over. Its target SDK is version 24 as that is the version of the phone I tested it on while developing. The version I developed for is important in some case as different calls from the android library can't be used for certain SDK versions. The front end is integrated with the firebase though Google play services. This is done through the Firebase console. Upon selecting the application that you want to link to Firebase, a JSON file is generated for the project. You can store this JSON file in the application build folder along with editing some of the application gradle files and it will be linked.

3.3.1 Application Structure

The application is broken down into its activities and fragments. Activities are used to manage the main parts of the app and interact with large user interface (UI) components while

fragments only manage very specific parts of the UI that may require a different program to run. While these two types of class manage interactions with the UI, there can still be plain old java objects (POJOs) used to assist them throughout the program. These POJOs act as normal by encapsulating information about an object. Using POJOs also allowed me to store information easily into Firebase the database supported pushing classes it with the class object being set as the key and the member variables being set as the children.

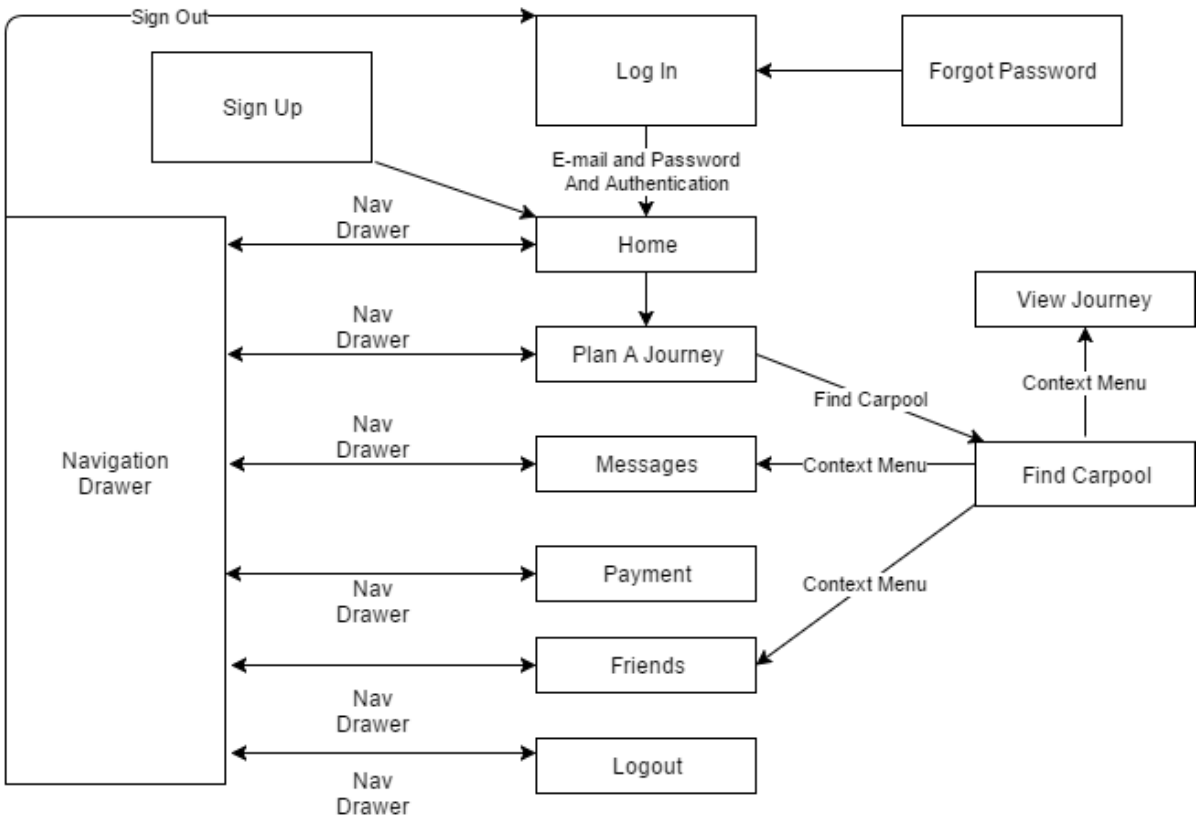
3.3.2 XML

Extensible Mark-up Language (XML) is what makes up the UI in an android app. It describes the layout of the UI components. I didn't spend as much time developing the UI as I should have as I was more concerned with functionality. This is one regret I have as I believe the way an app looks adds a lot to how it is received by the user.

3.3.3 Other Components

Other components of my app include receivers and broadcasters. These are android classes that can be used to transport data around my app. I found them useful when I needed to transfer data from services, non-UI threads, to UI threads. Examples of this in my code are from the FCM service and from the satellite navigation service. The FCM uses broadcasters to send data about received messages and ride requests while the navigation service uses a receiver to send updated variables from the service to the main UI thread in the navigation activity.

3.3.4 Brief Overview



4 Implementation

4.1 Back End Implementation

4.1.1 Cloud Functions

Cloud functions allow me to run node.js code on the Firebase server. This means that the client application has a lot less work to do. I use them to send FCM messages to client phones based on database events. The functions that I have implemented are

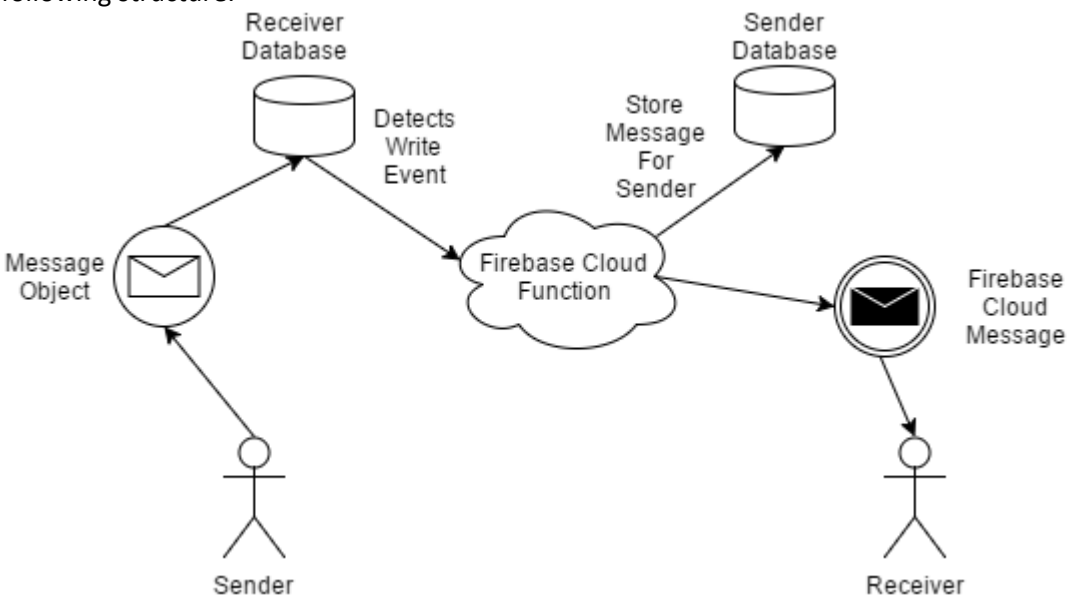
- Messaging
- Ride Requests
- Friend Request

4.1.1.1 Messaging Backend

My app allows users to message each other instantaneously. I have a cloud function that monitors a write event to a path of "UserProfile/UserID/Messaging/SenderID/timeStamp". This means that messages from a sender will be stored in the recipient's database section and that they will be alerted whenever a sender writes to that location. Database rules allow other users to write to that location of the database. Once a write even is detected, the function triggers. It will first capture the data snapshot where the event occurred. This is a message object consisting of the message, a sender name, a boolean stating whether the message was already copied and a sender's user I.D. The reason for the boolean is that after it detects a write event, it will also write the message to the sender's message list. This would then trigger an infinite loop of write events as the function would continuously detect a new write event. The boolean stops the loop from repeating. Storing the message in the sender's profile adds structure to the database as it allows the message list to be easily referenced for future use by both sender and receiver.

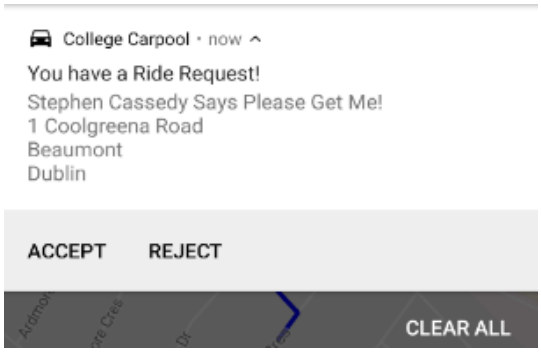
The function will down send an FCM message to the receiver's phone. It can contact this phone using a Firebase cloud messaging token. This token is a unique sequence of characters that is generated per Firebase authentication instance. It changes regularly each time the app

is restarted and is monitored by a service I have implemented. This pushes the updated token to the user profile whenever it changes so there is always a valid token to use. The FCM consists of a JSON formatted message split into two parts; notification information and data. Notification information is the title of the notification that the user will receive while the data is the other information that we can send with the FCM. In the case of messaging, I send a notification saying that the user has a new message from "Stephen" for example while the data contains the message body and the sender I.D. A service that handles incoming requests will take control of the message once it reaches the client phone. The process is follows the following structure:

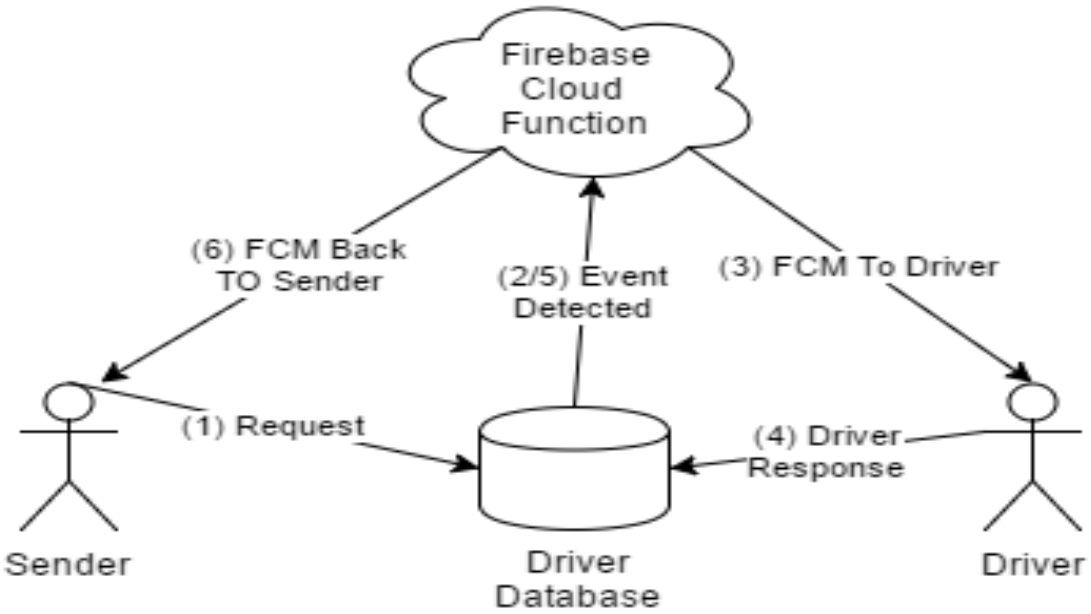


4.1.1.2 Ride Requests

Cloud functions handle ride requests sent between users. To make requests as accessible as possible, due to the nature of the request (receiver will be driving at the time). Notifications seemed like the obvious answer as they allowed a response in a one-click response. The idea is that a driver will get a request notification with the requesters address and name and then be able to either accept or reject the request. The request function works in a similar way to the messaging function except that requests are written to the driver/receiver's database section at "UserProfile/ReceiverID/RideRequests/senderID/Requests/Request". A request contains information about the user's latitude, longitude, user name and user I.D. Once again, the function will wait for a write event to this location and send on a notification to the driver. The FCM service will then handle the FCM message and present the user with an option to accept or reject the request. Once the driver chooses and option, a write even will occur at "UserProfile/ReceiverID/RideRequests/senderID/Response". This will trigger another cloud function associated with the ride request function that will send a notification back to the sender of the request indicating to them whether their request has been accepted or rejected. The transaction looks like this:



The data flow is represented as such:



Upon the driver response, the request will be removed from the driver's database. If the driver exits navigation while the ride request is still pending and removes the notification from their notification tray, it doesn't matter as if the user sends a request when the driver resumes navigation, it will overwrite the ride request under the sender's I.D. and just send the notification again.

4.1.1.3 Friend Request

Friend requests work the same way as the previous two except they are sent to "UserProfile/receiverID/FriendRequest/senderID". Once again, a write even is triggered and the receiver is notified. The friend request can be found in the friend request section of the app. With this function, I chose not to implement a cloud function that would notify the user that their request was accepted. It seemed like a redundant function as the app is not oriented around the friend system although it can be useful to view friend's planned journeys.

4.2 Firebase Authentication

I chose to implement a login and registration system that allows users to sign in with e-mail and password. This is one of the ways a user can authenticate to the Firebase server once they have signed up. Users without an account will not be allowed to sign in and therefore cannot use the app. I chose the e-mail login as I could limit the e-mail people could use to login. Given the circumstance I am in, and without being able to authenticate to DCU's servers, I cannot check if an e-mail is truly associated with a DCU user account. The best I can do is check that the user's e-mail ends in "@mail.dcu.ie". The motivation behind using DCU e-mail as an authentication method is to add a layer of security to the app. I am not oblivious to the fact that carpooling could have potential risks such as damage to the driver's car or the driver themselves. I am basing the idea off the success of apps such as Uber where anyone individual can operate as a taxi driver. Making sure that users are from DCU may lead to carpooling to be safer as all involved are or have been members of DCU which will hopefully lead to a sense of respect between users. If a situation did arise, I would hopefully be able to isolate the driver involved. Students may also be more open-minded to offering rides to fellow students so I think the idea works better.

Authentication is all handled on the server-side. The user signs up with their credentials and if no user with the same e-mail has already signed up, they can create an account. I have put restrictions on login credentials such as the "@mail.dcu.ie" on the e-mail and the password must be greater than 6 characters to increase strength. Unless a user specifically signs out in the app, an authentication instance will remain present. If the user logs out, it will be destroyed and the user will be forced to login to their account again.

4.3 Front End Implementation

4.3.1 Authentication

The UI authentication consists of 3 parts: Login, registration, and a forgot password feature. These are all represented by their individual activities and layout files that encapsulate their functionality.

4.3.1.1 Login

The first screen a user will find themselves at is the login screen. Here they have two text fields to fill in which are indicated by e-mail and password. The user can enter their details and sign-in here. Once they press the login button, the device will authenticate with the Firebase server and the user will receive an authentication instance and they will be directed towards the app's main 'home' screen.

4.3.1.2 Registration

It may be the user's first time opening the app. In this case, they can navigate towards the sign-up screen with the button of the same name. Here they will see 4 fields for them to enter text into indicated by first name, second name, e-mail and password. These are the basic credentials of each user. Upon successfully entering their details and being able to make an account (e-mail has not already been taken), their login details will be saved to the android device's shared preferences. Shared preferences allow a phone to store small key-value pairs in internal storage. I thought this might be useful as it saves the user the trouble of entering the same login details over and over. In the future, when the user goes to log-in, they're details will already be filled in to the text fields.

4.3.1.3 Forgot Password

Although it is unlikely that a user will lose their details as they are stored in shared preferences, a user may switch phone and somehow lose their credentials. Firebase supports a feature which allows users to reset their password via e-mail. From the login screen, they need to enter the forgot password screen. Here they will be presented with a field for them

to enter their e-mail into. They can enter their password and click the forgot password button. This will use the firebase 'sendPasswordResetEmail' function to send an e-mail to the inputted e-mail that will request the user to follow a link to reset their password. If the user remembers their password they can still enter it as their old password will be valid until the point that they follow the link to reset it.

4.4 Home Screen

The first main screen that the user will be presented with it the home screen. This is also where the user will be brought to automatically is they have not logged out specifically since their last use of the app. The purpose of this screen is to allow users to view active journeys that are taking place.

4.4.1 Permissions and Location Service

When the user signs in for the first time, they will have to allow the app access to the phones location. This requires the ACCESS_FINE_LOCATION permission which will allowed me to give a very accurate reading of the user's location. As of Android 6.0, permissions must be accepted at runtime by the user depending on their sensitivity. Access to a phone's location is considered a sensitive permission so therefore the user must give permission for it. This is done by checking if the permission is already granted and if not, calling the android method 'requestPermissions'. This will start a dialog on screen that will pause the main UI thread until the user has either accepted or rejected the permissions request. College Carpool cannot function without this permission. The app also requires that location services are enabled. The GPSChecker class creates a location request which will request the phone location from the location settings API. This is just a trigger to initiate a dialogue as if the request can't be satisfied, a dialogue will appear on-screen requesting the user to enable their location services. Once again, the app will not work entirely if the user doesn't have location enabled.

4.4.2 Google Location Service

Many features of the app depend on its ability to provide an accurate location for the user. As I have mentioned before, after considering multiple ways of obtaining the device location, I opted for the location service's fused location API method. This method, once to set to high accuracy mode is documented to be accurate to be within 5 metres in ideal conditions although I have experienced variations in performance. To use this service, I had to make a Google API Client. I made a class called "GoogleClientBuilder" that would manage any Google clients that I would need through the project. The client allowed me access to the google play services library from which I could access the location services. Each client could only contain one instance of a google API. The location services API allowed me to call fused location which is a mixture of GPS, cellular and Wi-Fi location data to assess where your device is. The main benefit is that with it, I could request location updates with a Location Request object that could define the intervals that the client would request location updates and how accurate a mode to use. High accuracy is what I use but the downside of this mode is that battery consumption goes up. As long as the client is connected to the location service, it will continue to request location updates. Every time a new location is acquired, I push that location to the user's profile so that it can be referenced in the future. Since the database reacts so fast, it doesn't affect the rate at which location changes are detected and actually proves beneficial to do so when it came to designing the navigation service.

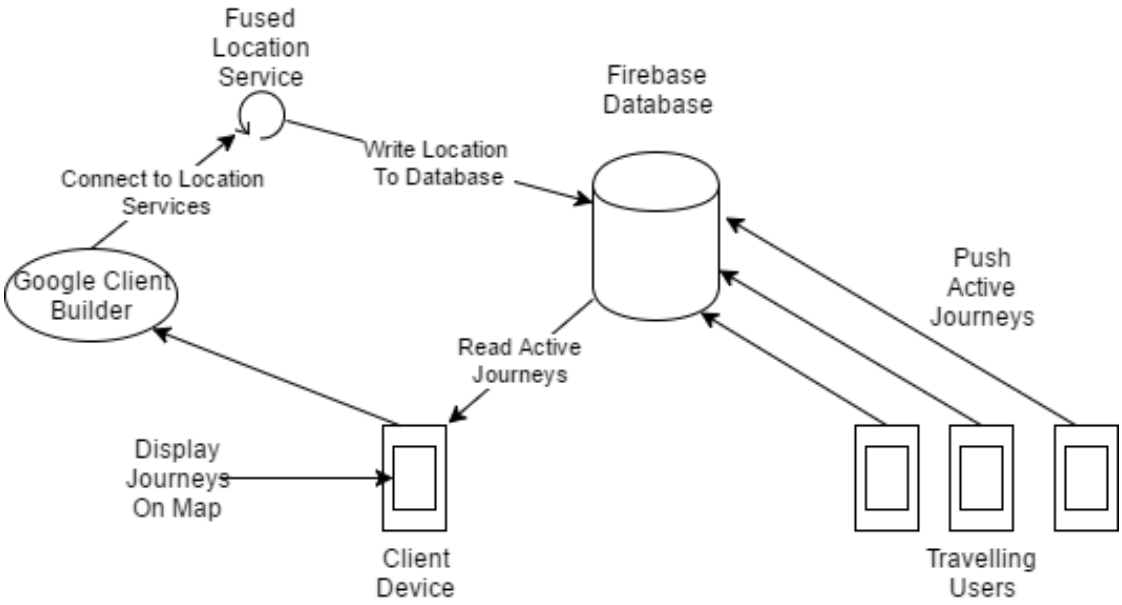
4.4.3 Google Map

The screen contains a support map fragment that consists of a Google map. It enables me to show the user activity of other users on the map and is the primary place where users can search for a real-time carpool. Upon opening the app, it will display the polylines of all active

journeys. If a user clicks on one of these lines, they will be able to see the waypoints that the polyline is passing through and in turn f they click on a waypoint they will be able to either request a ride or message the user directly.

4.4.4 Active User Map

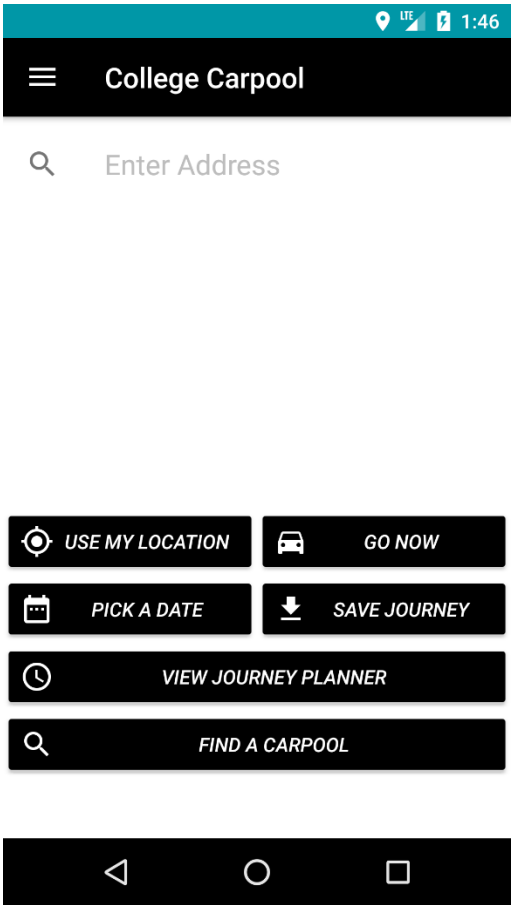
The active user map is a class I made to manage displaying the active journeys that were taking place. It reads the journeys by setting up a value event listener for the database at "ActiveJourneys/{userID}/" where {userID} represents the I.D. of any user who is travelling at that moment. The journey is stored in the database in encoded format to make it less cumbersome to store, retrieve and plot on the map. The class maps the polylines by using the PolyUtil library provided by Google to decode the lines it retrieves from the database. Associated with this line in the database are the journey waypoints too so it retrieves these also. As the class draws the lines, it associates objects with the setTag() method. This method is one that is new to the polyline library for Android. I was previously using maps to associate data such as waypoints but the setTag() method was a fantastic addition to the library. To get it I had to update all of my google dependencies in March 2017 as it was only in the newest version of google-play-services-map. This class works as highlighted below:



The user can navigate to other activities with the navigation drawer that appears in the top left of every screen (except the friend activity) in the app. Alternatively they can navigate to the plan journey activity via the button at the bottom of the home screen.

4.5 Planning a Journey

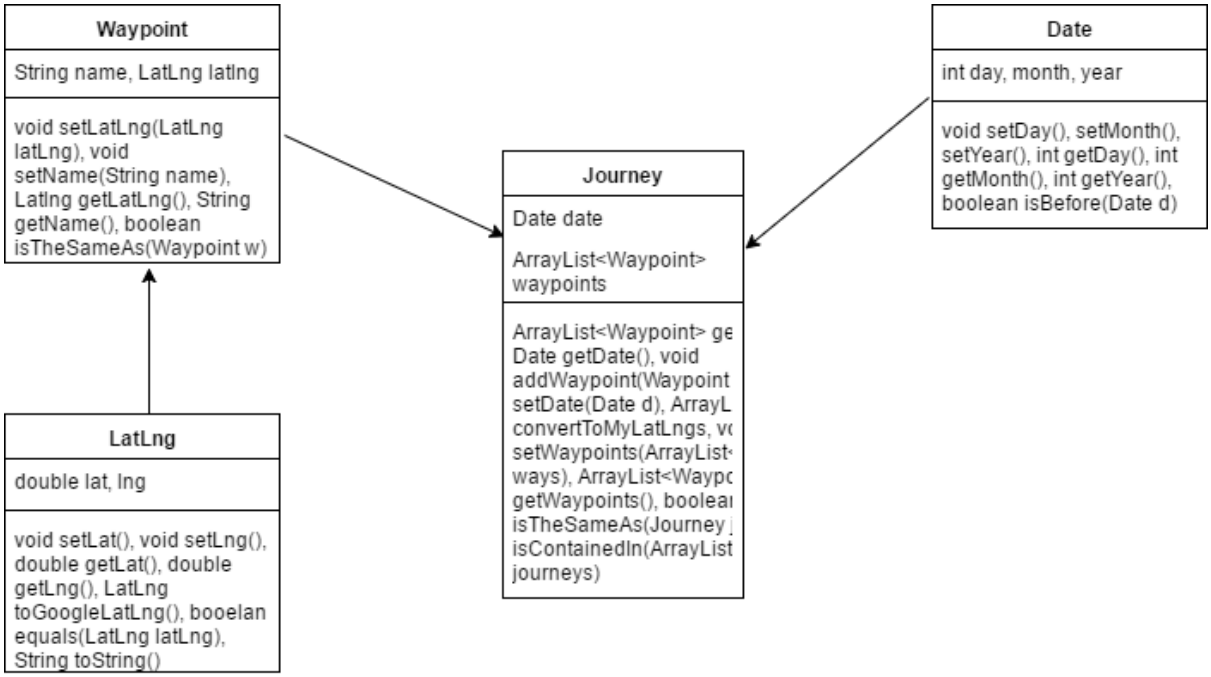
College Carpool incorporates the ability to plan a journey ahead of time. This is useful for both the driver planning a journey and any other people who may wish to travel with them. The plan journey activity is designed as shown here:



This allows users to plan a journey by searching for destinations at the top of the screen and adding them to a list in the white space between the buttons and the search bar. The list of stops is stored in a list view. This is built with a custom adapter I made that numbers the stops etc. Upon a long click on a list item, a context menu is generated. This contains the following options: Remove Stop, Move Up and Move Down. These will all edit the list and re-organise journey order if needs be.

4.5.1 Search Bar

The search bar is a place autocomplete fragment that will return any places that match the search term. I have limited the search to return results from Ireland only. The search will return a place object. This object contains information that I use such as latitude/longitude coordinates and the place name. I display the place name while converting the list of places to a journey object. A journey object is a POJO that I made to encapsulate the idea of a journey including a data and a waypoint object which in turn holds a waypoint name and a latitude/longitude object that I made. Journey object make up the base of how I transfer different users journeys around and store them. Below is a class diagram of Journey:



This diagram represents how some major classes are related to each other.

4.5.2 Use My Location

The 'Use My Location' button uses a Google API client to make a call to the Google places API. Once this client finishes building, it returns a PlaceLikelihoodBuffer that contains possible locations that the client device is near. Each place in the buffer has a probability that ranks how likely it is that you are in that place. I sort through these and return the highest probability place I am in and add it to the list when this button is pressed. This was originally a feature from before I had the satellite navigation set up and I needed to request directions from my location but I decided to leave it in to make it easier for users to select where they are in case they are planning a future journey from their location.

4.5.3 Pick A Date

This button launches a date picker dialogue that allows the user to select a date that they wish to travel on. I have programmed it so that it cannot accept a date that is in the past as a valid date for travelling. This is done by getting the current date with a java Calendar object and converting it to one of my date objects. I can then call the "isBeforeDate(Date d)" method from the date class to determine whether it is a valid date.

4.5.4 Go Now

The go now button launches the satellite navigation activity and service. Pre-requisites are that at least one stop is entered so that the journey has a final destination.

4.5.5 Save Journey

Saves the journey to Firebase if a valid date s selected and at least 2 waypoints are in the journey.

4.5.6 View Journey Planner

This will launch the view journey planner activity where a user can view and edit their planned journeys list. It is effectively a schedule.

4.5.7 Find A Carpool

Find a carpool will launch and activity that will allow users to search the database for a journey that will pass by a stop of their selection.

5 Navigation Service

The navigation service is a major part of the application that I did not foresee before I started getting deeper into the project. It ended up being one of the core features in the functionality of the app and took a lot of time to implement.

5.1 Polyline URL Builder

One of the first steps involved in getting a navigation service working is to create a Google Directions API request. I needed to be able to do this the obtain a JSON file from the web service that contained directions between my waypoints. I designed a class that could take a list of latitude and longitudes and be able to do output a valid URL for requesting directions. It forms the URL by taking the first elements of the last as the origin and the last elements as the destination. In between are the waypoints. The URL is set so that the waypoints will be optimized. This means that when the JSON file returns, it will have attempted to have ordered the waypoints between the start and end of the journey in the most efficient way possible. Google use an implementation of Dijkstra's algorithm to solve this problem known as the travelling salesman problem. I have tested whether it returns the waypoints in the correct order numerous times and it seems to be correct most of the time. If locations exist close together and lie along the same route, it may show preference to neither. I also needed a google API key for the directions service. Without an API key, I was extremely limited to the amount of calls I could make. I was ignorant of this when I started development and an infinite loop in calls to the API used my calls up very quickly. After this I learned that I had to use an API key to gain access to more calls. Originally, I was allowed 1000 calls every 24 hours but after registering my bank card with Google, I gained access to 150,000 calls per 24 hours. This allowed me to experiment with the code without fear that I would run out of calls and stop development for the day. It did lead to Google charging my account for use though.

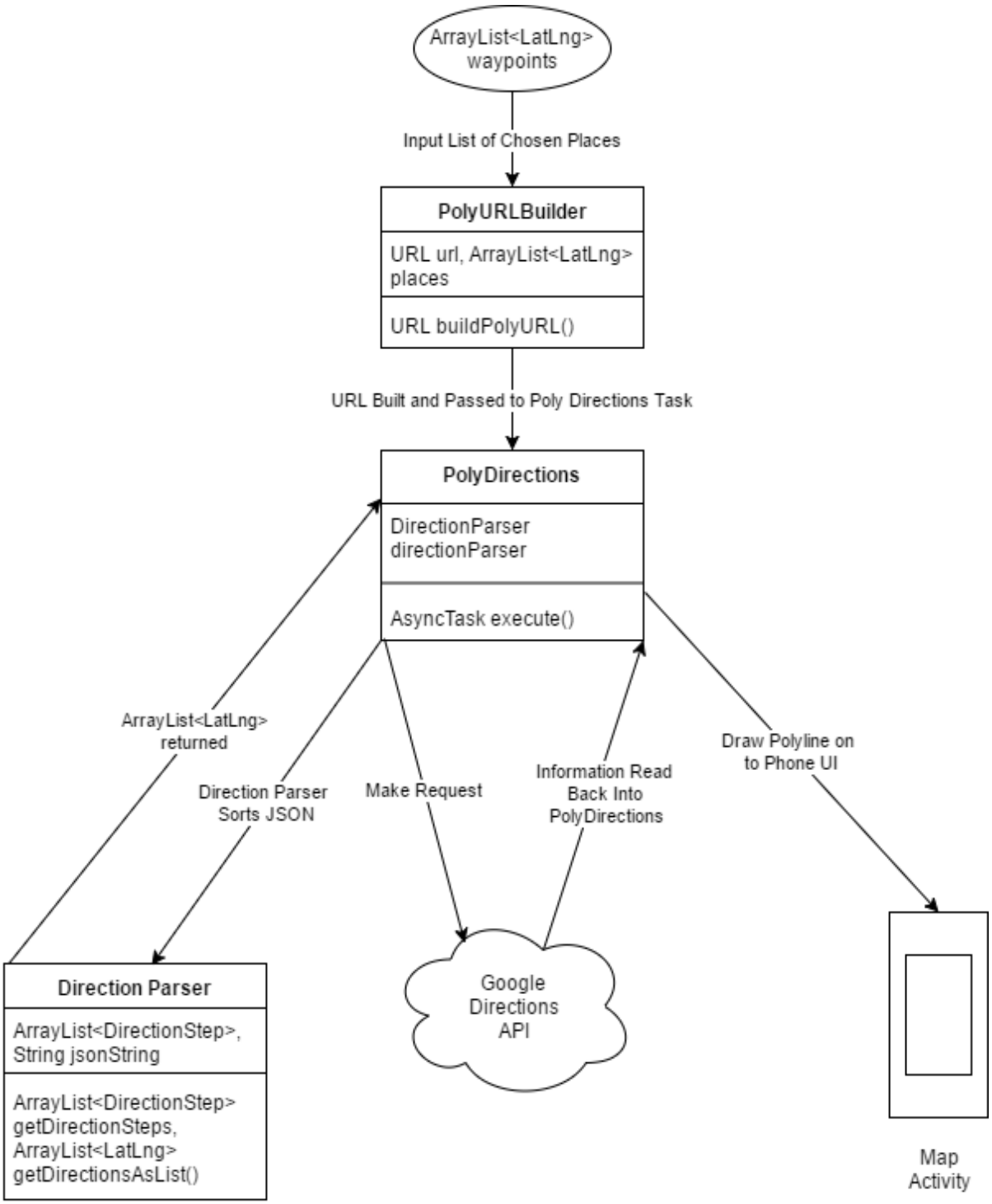
5.2 Polyline Directions

Once the URL is built to obtain the JSON file, I needed to be able to request the directions from the web-service. I wrote an asynchronous class that took the URL as a parameter and would return a list of latitude/longitudes or draw the polyline straight on a map depending on what activity it was launched in. I decided to make the task asynchronous so that it would separate from the main UI thread and not block it up. Reasons it may block the UI thread included the web call may not happen immediately due to poor internet and then the JSON file had to be parsed which could take some time depending on the size of the file. For journeys with a length of over 100km the UI will begin to suffer. This class will open a connection to the webpage and begin to read in its contents line by line with a Buffered Reader and storing it in a string variable. Sorting the JSON string is handled by a direction parser class. The direction parser will pull an ArrayList of latitude/longitudes out of the string. Depending on the activity that the user is in, it will either return the arraylist and end or it will execute the onPostExecute() method wherein it can draw the polyline on to the activity's Google map. For the navigation system, it will only return the ArrayList.

5.3 Direction Parser

My direction parser class is responsible for filtering information out of the JSON returned by the poly directions class. It cycles through the JSON string and breaks it down into JSON arrays and objects. Every Google directions API request is broken down into routes, legs and steps with steps making up legs and legs making up routes. Therefore, a step is the lowest atomic unit of a journey. I pick information that I want from the step objects such as html instructions, manoeuvres, start

latitudes and end latitudes. I store these variables in an object called a direction step. I made this to encapsulate on step of a leg. When the direction parser has finished, I have a list of all of the steps that make up the journey. They are stored in an ArrayList<DirectionStep> in the direction parser class.



5.4 Navigation Activity

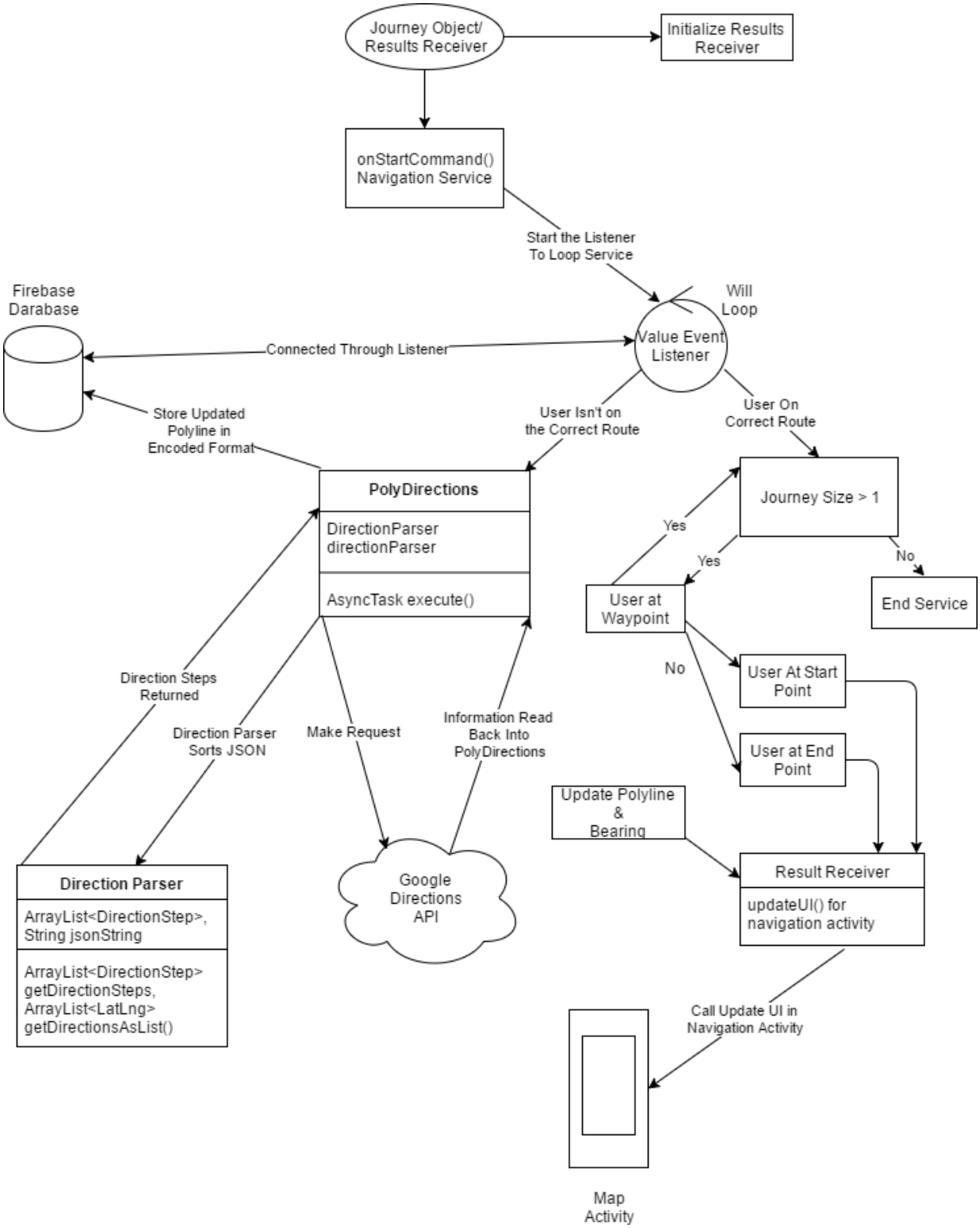
Once the backend for getting the Google directions as a polyline was set up, I could move on to designing the activity that could run the navigation feature. The activity obviously had to have a Google map in it. The main feature of the activity would be a handler method for the UI as it would need to be update regularly as the driver reached various waypoint or pickups or if the driver changed their route and forced the polyline to be re-calculated. It would also initialise a text to speech voice engine that could read out strings that were fed into it. This takes time to load up so it had to be done in the onCreate() method. The activity would start by receiving an intent (a parcel of information from the previous activity) that consisted of a Journey. It extracts the waypoints from the journey and sets up a custom results receiver (Navigation Receiver). This receiver is used to send information from the navigation service back to the navigation activity. Whenever results are returned from the receiver, the updateUI() method is called. This handles map behaviour and the changes that occur at different points if the journey. It also handles broadcasts about ride requests that are received from the FCM messaging service that runs in the background. Once the rest of the activity was set up, I started the navigation service containing the waypoint latitude/longitudes and result receiver as intents.

5.5 Navigation Service

I attempted to reduce the strain put on the UI by creating a separate service thread for most of the work regarding the navigation to be carried out. I came up with a way of looping the service so that it would work as I wanted. I set up a Firebase value event listener on the current user profile. This would be triggered anytime the user's location changed. All the logic for the service is handled in here and is run through every time location changes. Location updates are at every 1000 milliseconds at the time of writing. The main principles the service works off are:

- Add my current location to the start of the journey and re-calculate polyline from my current location. If it's already there, overwrite it but don't re-calculate polyline.
- Check if the user is on the correct route. This involves calculating the distance the user is from the polyline coordinates. If the user is greater than 50m from the route, the user is deemed to be off route and the polyline is re-calculated. Else continue with the function.
- If polyline re-calculated, push updated journey to public access in Firebase.
- If the user is on the correct route, assign the current direction step from the direction parser the current step variable.
- Check if the journey waypoints are greater than 1, with index of 0 being the user's location after previously adding/overwriting it. Else the journey has been completed so shutdown the service and remove the value event listener to stop listening from updates.
- Check if the user is at a waypoint by cycling through the journey waypoints and evaluating whether the user's position is within 60m of any waypoint. If they are, remove the waypoint and check if the journey size is still greater than 1. If not, stop the service.
- Check if the user is at the start location of the current step. If so, assign the html instruction for the current step to a local string variable. This is to be read out by the text to speech engine in the navigation activity.
- Also check if the user is at an end step. Due to the distance metrics and the unpredictability of how long in distance a step may be, a user can often be at both a start step and an end step. If the user is at an end step, increase the position of the current step in the direction steps list.
- Manage what happens if a user is at both a start and end step. In this case, I perform a calculation to test which part of the step the user is closer to. If user is closer to the start step, set a boolean atEndStep to false and vice versa if at an end step.
- Update the polyline by checking if you have passed within 10m of any of the points and get the bearing of user. This will point them in the direction they should go.
- Pass results to results receiver by sending a bundle object to it with latest information.

Navigation Service Design:



6 Near Frequency Controller

6.1 Concept

The concept of adding near frequency controller (NFC) functionality came about when I was looking for something extra to add to the app. I thought a payment system that didn't require cash was a safer way to do things, especially if you are ferrying strangers in your car. I also took inspiration from the success of the Dublin Bus Leap cars system.

6.2 Research

I had no experience with NFC cards before so I began reading documentation in them. I learned that they support reader/writer mode, P2P mode and card emulation mode. Since I was only producing a 'dummy' payment system that had no real backend to it, I opted for the P2P mode. This is the same functionality that is used for Android Beam. If the app was being used in the real world, card emulation mode would be used to simulate a bank card. I also learned that Android mainly supports communication in the form of NDEF messages.

6.3 Setup

To begin working with the NFC card, I had to make an activity that implemented 2 NFC adapter call back methods. The NFC adapter allowed me to gain access to the card while the call backs monitored both when another NFC card came in communication range and when the NDEF message has successfully been completed.

6.4 Wallet

The wallet displayed at the top of the screen is only a double that I'm storing in the database. With the NFC, I alter the local value and when it changes, I push the new value to firebase and update the user's UI. Although basic, it does show how a payment system could operate. I allow users to enter 'Donation' amounts using number wheels. Each number wheel represents euros and cents respectively. I was advised to use donations by a user who I tested the app on as they thought the previous text of "To Pay: "was confusing during the payment process. I also think donation is more suited to the project as I am not encouraging users to pay fares for rides. I simply have it there as a sign of gratitude and to possibly cover petrol costs in case the person getting a ride cannot reciprocate.

6.5 Transaction

To perform an NFC transaction, the phones must be places close together back to back. This triggers the push NDEF message call back which in turn will trigger an NDEF message to be created by the user's phone. This then collects the payment amount collected from the number wheel and stores it in a MIME type message. This message is embedded in an NDEF Record which in turn is inside an NDEF message. To send the message, the user will be prompted to beam the message on-screen. Once they do this, the message will be beamed across and received by the other device. This then triggers the on NDEF push complete call back for the user while the receiver will receive the message which triggers the on new intent function. The NDEF message sent to the receiver takes the form of an intent. The app knows how to receive the message as I've defined it in the apps manifest that incoming NDEF messages will be collected by intent-filters. I then store that intent in a parceable array. From this array, I can translate the parceable object to an NDEF message and from this I can get the original MIME message from its NDEF records. The amount sent over is then added to the receiver's wallet and pushed to the database.

If the user doesn't have the payment activity open at the time of transaction, the NDEF message will tell the phone to open the app. I did this by including an application record in the message that would point the receiving phone to launch the app by package name, e.g. "tets.collegecarpool.alpha".

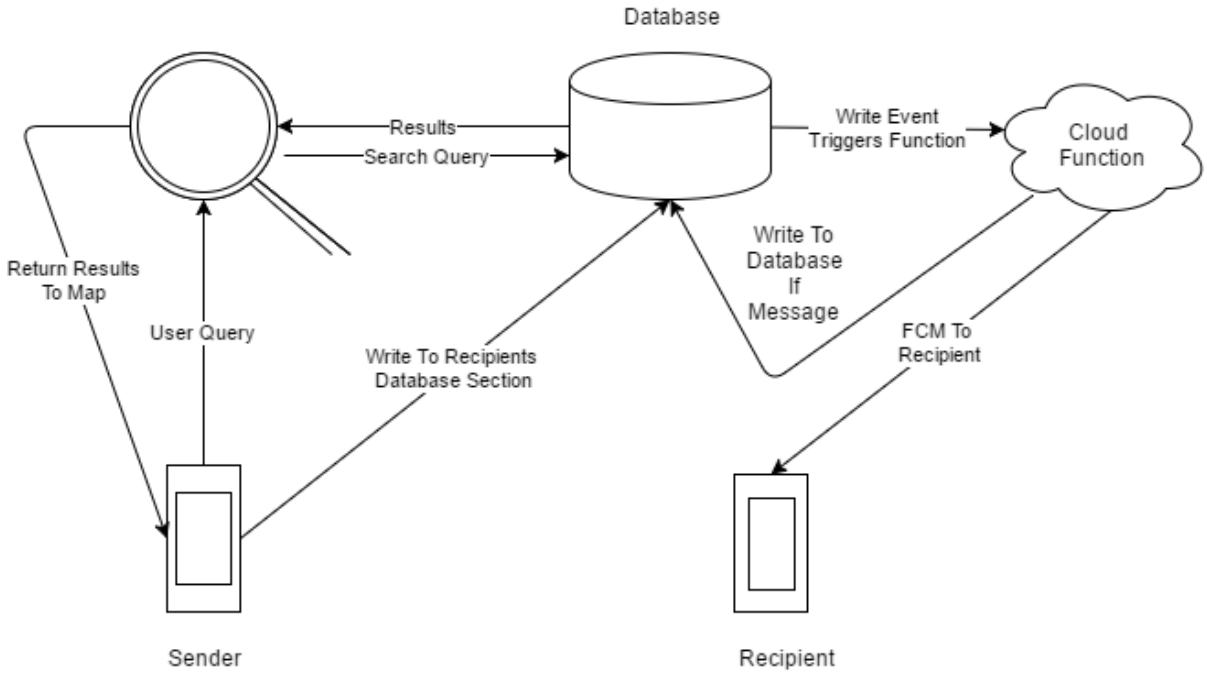
7 Friends and Messaging

7.1 Friends

I integrated a friend system into the app in the later stages of development. I thought being able to have friends may enhance user experience with the app as it allowed people to group together. A user can add a friend by going to the 'Find carpool' activity via the journey planner activity. They can search for a journey that they wish to go on and if they find a match, they can select the item from the context menu and either send a friend request or message the journey owner. This is done by searching the entire Firebase and storing journeys with matching keywords. The journey is then stored in a map consisting of the user I.D. who owns it and the journey object. When the user selects a list item and clicks add friend, I search the map for the corresponding value to the journey they selected and therefore I get the user I.D. too from the map entry. Having this allows me to directly write to the other person's section of the database which will trigger a friend request cloud function and alert the user that they have received a request. The motivation to become a friend with someone is that you have access to their journey planner. This allows you to see where your friends are planning on going and when so that you might be able to get in contact with them early and give them plenty of notice.

7.2 Messaging

Messaging works in a similar way to the friend request. A user can send a message from 3 locations: the home screen after clicking on a driver's polyline, from the friends list where the recipient is already a friend or from the 'find carpool' activity. In all of these cases the user can have access to the recipient's user I.D. All they need to do is write to the recipient's database and the cloud function will handle the rest.



8 Firebase Messaging Service (FMS)

The last major component of my front end was the firebase messaging service. This service is what allows my push notifications to be handled by the app rather than just appear in my notification tray and nothing else. This is a service that ships with firebase cloud messaging but that you must implement yourself to add functionality. The purpose I designed the service for is to interpret remote messages that are received through your firebase messaging.

From the remote message, there are 4 types of message I can receive: message, rideRequest, rideResponse and friendRequest. These are all defined in the index.js script if the cloud functions script and are the 'data' section of the FCM sent from the node.js code. I have special handlers for each type of FCM I receive which are called inside 'if' statements that separate the messages by their types.

Each individual handler will generate a unique notification using the Android notification manager to suit the purpose of the notification. For instance, the ride request notification will have an Accept/Reject field while the message will have a title saying who the message sender was.

The FMS service is enabled using FCM tokens. An FCM token is a unique string of characters that will allow Firebase to communicate directly with the client device rather than sending notifications to all clients. The token will refresh at regular intervals such as restarting the app or when a Firebase authentication instance is started again. To manage the changes and constantly keep a valid token stored for each user, I implement a service that will begin running as soon as the user completes log in and authenticates to the server. This service implements FirebaseInstanceIdService which issues a call back method 'onTokenRefresh()' whenever a user's FCM token changes. I keep user the token up-to-date by pushing it to firebase as soon as its created and letting the service run for the entirety of the time the app is running.

9 Code Snippets

9.1 Navigation Service

```
/*Check If One LatLng (User) is Close to Another*/
private boolean isNear(LatLng userLatLng, LatLng other, float threshold){
    float [] distance = new float [1];
    Location.distanceBetween(userLatLng.latitude, userLatLng.longitude, other.latitude, other.longitude, distance);
    return distance[0] < threshold;
}
```

This code snippet calculates if a user is within a set distance from a latitude/longitude. It uses the PolyUtils library for polylines to calculate whether one latitude/longitude is within a set threshold from another. While this code may seem simple, It helped me to refactor out nearly 45 lines of code from the original navigation system I had written.

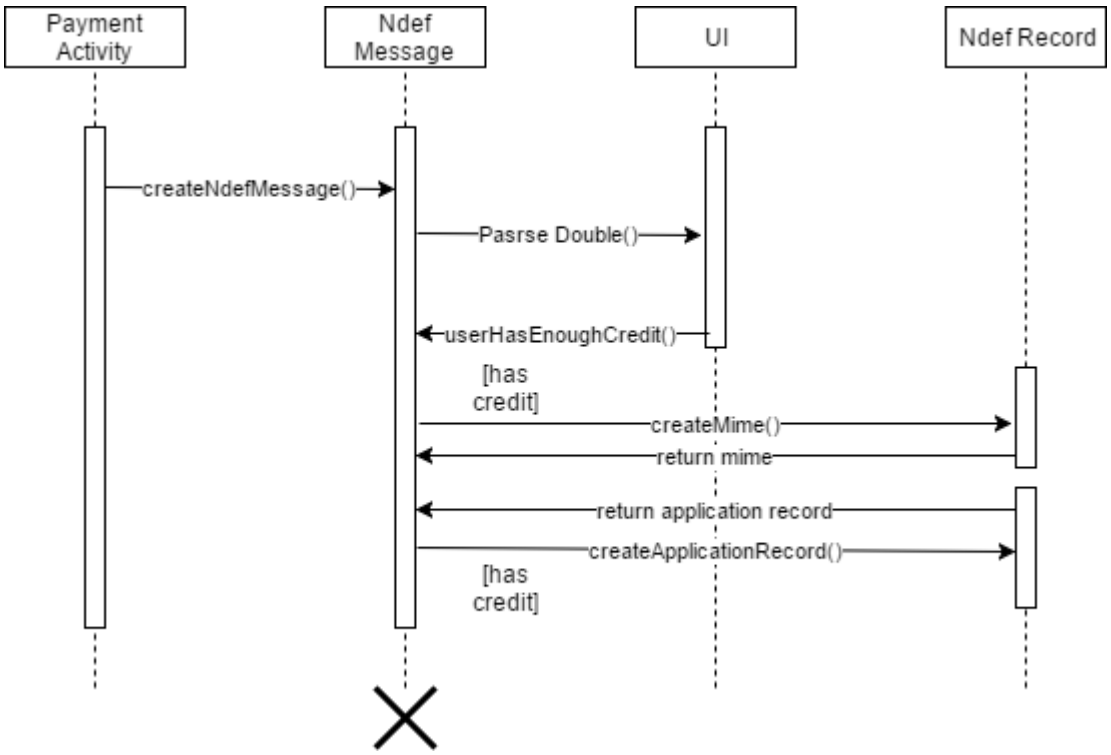
```java
public class NavigationReceiver extends ResultReceiver {

    private NavigationActivity navigationActivity;

    public NavigationReceiver(Handler handler, NavigationActivity navigationActivity) {
        super(handler);
        this.navigationActivity = navigationActivity;
    }

    /*Call updateUI() on the navigation MaP*/
    @Override
    protected void onReceiveResult(int resultCode, Bundle resultData) {
        navigationActivity.updateUI(
                (ArrayList<LatLng>) resultData.getSerializable("WaypointLatLngs"),
                (ArrayList<LatLng>) resultData.getSerializable("PolyLatLngs"),
                resultData.getBoolean("JourneyFinished"),
                resultData.getBoolean("RemovedCloseWaypoint"),
                resultData.getBoolean("UserAtStartStep"),
                resultData.getBoolean("UserAtEndStep"),
                resultData.getBoolean("PolyLineRecalculated"),
                resultData.getString("Instruction"),
                resultData.getFloat("Bearing"));
    }
}
```

This code snippet represents the navigation receiver. This piece of code allowed me to transfer data between the ongoing navigation service and the navigation activity. By passing the navigation activity to the result receiver, it enables me to collect information straight from the bundle that the service sends at the end of each loop. I can then call updateUI() with all of the new parameters from here and have it execute on the UI thread.

9.2 NFC Payment

```java
@Override
public NdefMessage createNdefMessage(NfcEvent event) {
    String message = String.valueOf(euro) + "." + String.valueOf(cent);
    cost = Double.parseDouble(message);
    /*This incorrect loop cause money to be taken away irregardless of message being beamed*/
    Log.d(TAG, "Personal Balance is " + String.valueOf(getPersonalBalance()) + " and Cost is " + String.va
    if (userHasEnoughCredit(cost)) { //This is the point where its going wrong
        if(tapDetected) {
            tapDetected = false;
            //removeCostFromWallet(cost);
            //displayBalance();
            return new NdefMessage(new NdefRecord[]{
                    createMime("application/test.collegecarpool.alpha", message.getBytes()), //Inserts the
                    createApplicationRecord("test.collegecarpool.alpha") //Embeds Android Application Reco
            });
        }
        else {
            String tapWarning = "Restart Activity";
            return new NdefMessage(new NdefRecord[]{
                    createMime("application/test.collegecarpool.alpha", tapWarning.getBytes()),
                    createApplicationRecord("test.collegecarpool.alpha")
            });
        }
    } else {
        Log.d(TAG, "ENTERED ELSE NDEF STATE");
        String ErrorMessage = "Not Enough Credit";
        return new NdefMessage(new NdefRecord[]{
                createMime("application/test.collegecarpool.alpha", ErrorMessage.getBytes()),
                createApplicationRecord("test.collegecarpool.alpha")
        });
    }
}
```

This code snippet demonstrates how the NDEF message is created in the payment activity. First the method checks if the user has enough credit to pay what they have entered into the selector wheels. The tap detected boolean condition is there as a flag to stop the NDEF message sending repeated amounts to the recipient phone. This was a problem I faced while trying to work with the NFC cards; Information would continuously be written to the receiving device if the connection was in any way moved and restarted. The threshold for when the connection is registered as being broken is very small and therefore cause an issue when two people were trying to hold their phones together. With this flag, it will only be set to true again if the message delivered successfully. This error also lead me to have to put in a condition where the user may have to restart the payment activity due to a deadlock because of the boolean. We can clearly see the createMime() call fir generating the MIME data while the createApplicationRecord() is what issues an intent to start the application if an NDEF message is received while the user's device is outside of the app. Flow is represented as such:



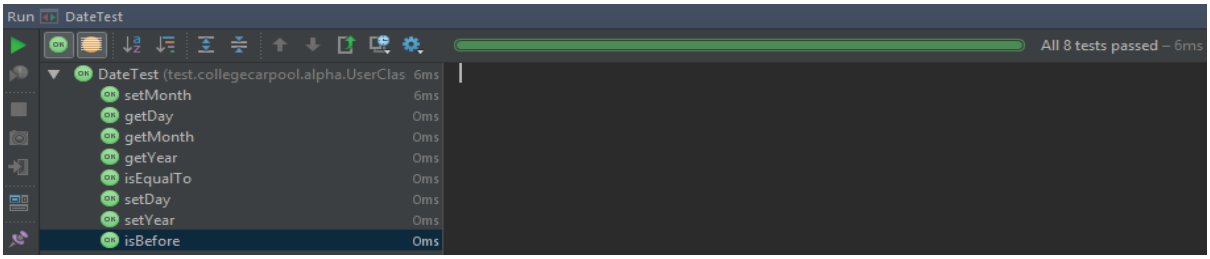
10 Testing and Validation

10.1 Unit Testing

Unit testing was the most basic testing that I performed on the lower level components of my program. These included classes such as the date, waypoint, time, user profile, journey and time classes. Since these classes make up the core system of functionality in my system, I thought that it would make sense to thoroughly test them. Since I had never written unit tests before, I had to research how to use them. It primarily seems to involve using the 'assertEquals()' command to compare the results from methods with a value that you manually assert it to be equal to. This allowed me to test getters, setters, sorting algorithms and mapping algorithms to a reasonable extent. The results of my unit tests

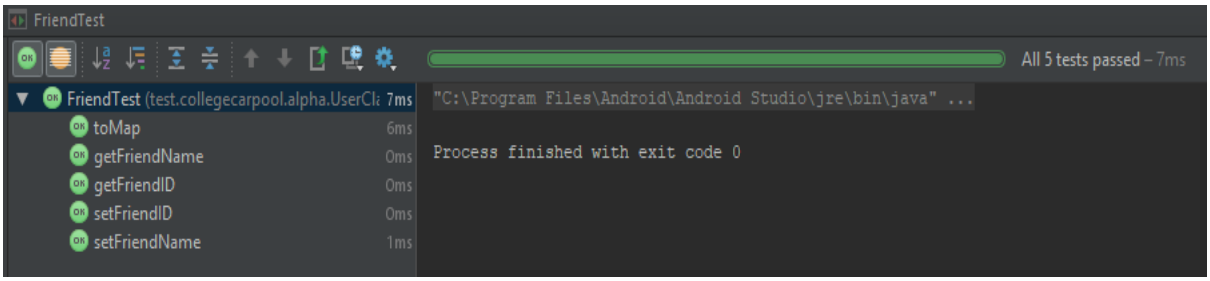
10.1.1 Date Class Test

Methods Tested:
setMonth() : passed
getMonth() : passed
setDay() : passed
getDay() : passed
setYear() : passed
getYear() : passed
isEqualTo() : passed
isBefore() : passed //Originally failed and had to re-write function



10.1.2 Friend Test

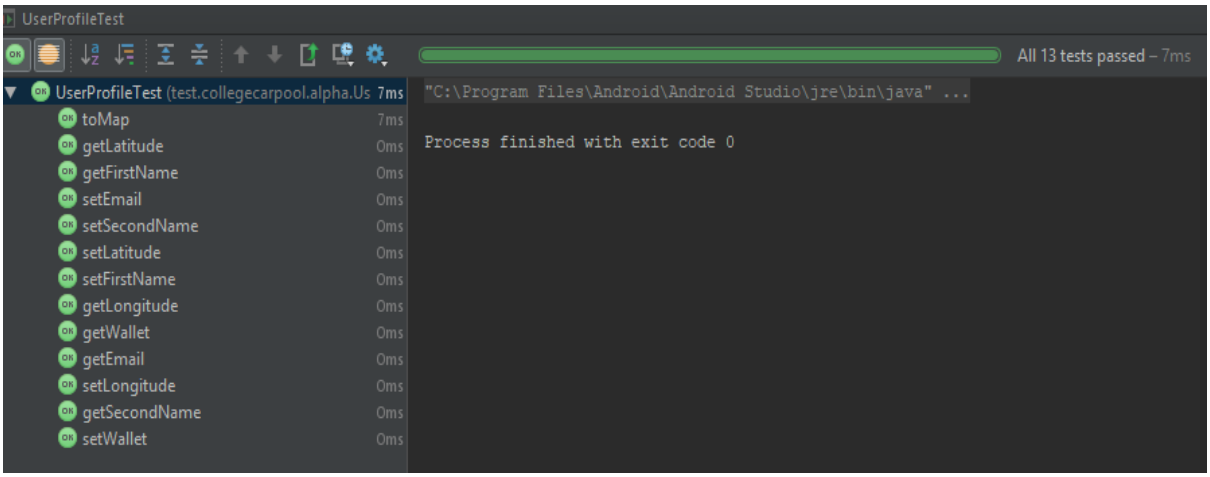
Methods Tested:
toMap() : passed
getFriendName : passed
getFriendID : passed
setFriendID : passed
setFriendName : passed



10.1.3 User Profile Test

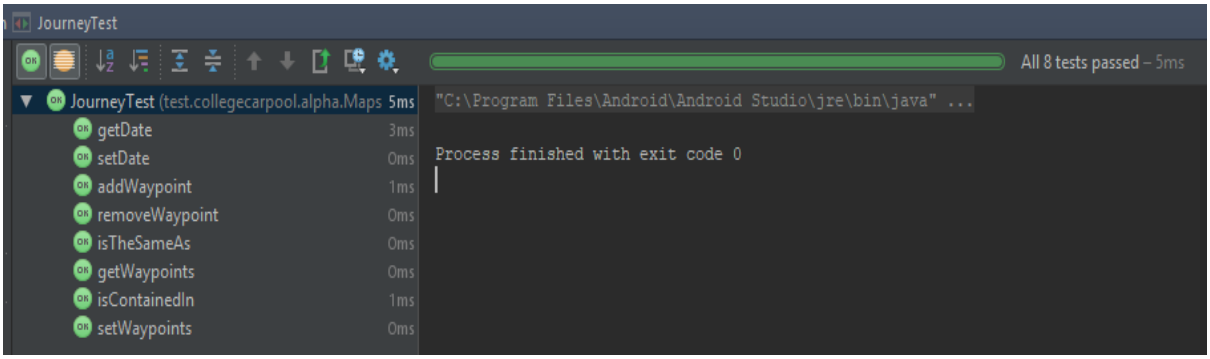
Methods Tested:
toMap() : passed
getLatitude() : passed
getLongitude() : passed

getFirstName() : passed
setEmail() : passed
setSecondName() : passed
setLatitude() : passed
setLongitude() : passed
setFirstName() : passed
getSecondName() : passed
getWallet() : passed
setWallet() : passed
getEmail() : passed



10.1.4 Journey Test

Methods Tested:
getDate() : passed
setDate() : passed
addWaypoint() : passed
removeWaypoint() : passed
isTheSameAs() : passed
getWaypoints() : passed
isContainedIn() : passed
setWaypoints() : passed

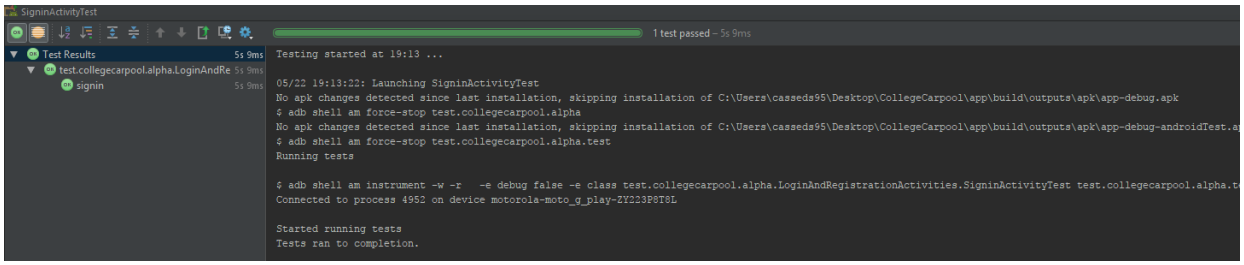


10.2 Espresso UI Testing

I only used espresso for a brief time before I found a better solution. Espresso is a UI response tester that you can program to interact with UI features.

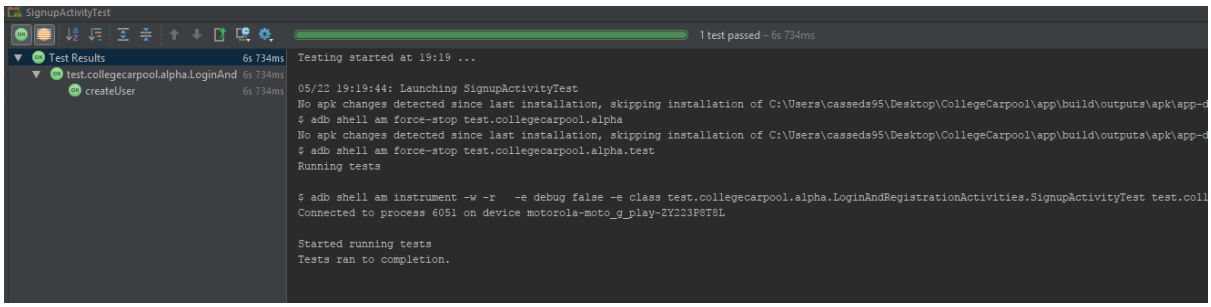
10.2.1 Sign-in Activity

I first tested the sign-in activity. I did this by allowing espresso to type text previously determined text into the edit text fields that accepted the e-mail and password of the user. The system knows where to find the views it must enter text into by using an android layout resource that I coded into the test. It will then perform a click action on the login button.



10.2.2 Signup Activity

I also tested the signup activity. Here there are 4 edit text views for first name, second name, e-mail, password. Once again, I used the espresso testing for entering text into the desired fields. It would then perform a click action in the signup button.



An example of what the espresso code looks like:

```
@Before
public void initString(){
    email = "test@mail.dcu.ie";
    firstName = "Test";
    secondName = "Test";
    password = "test123";
}

@Test
public void createUser() throws Exception {
    onView(withId(R.id.first_name))
            .perform(typeText(firstName), closeSoftKeyboard());

    onView(withId(R.id.second_name))
            .perform(typeText(secondName), closeSoftKeyboard());

    onView(withId(R.id.email))
            .perform(typeText(email), closeSoftKeyboard());

    onView(withId(R.id.password))
            .perform(typeText(password), closeSoftKeyboard());

    onView(withId(R.id.btn_Signup)).perform(click());
}
```

10.3 Firebase Test Lab

Coding the espresso tests was repetitive and slow. I began looking for ways to test my application quicker when I remembered that Firebase had a function that allowed me to test my app remotely. To do this I had to upload my app's APK to the Firebase server. Here, I could select a variety of phones and API versions that I could test-run the app on. This performed the same job as espresso testing but was far quicker and involved way less overhead as the testing was done automatically. All I had to do was provide the test lab with valid login credentials and it would login to the app and begin testing UI features such as button clicks, pager fragments and autocomplete fragments. The testing suite gave fully comprehensive reports back on performance including screenshots, logs and even an MP4 video of the test. This drastically sped up the speed at which I could test the app and gave me access to testing a wider range of SDK versions and phone models. On the next pages is a model of what the output produces.

The testing also reported crashes and a debugger for them. It also included the phone's logs so that I had a centralised location where I could monitor failures and successes from multiple devices. Using this feature revealed bugs in my code that I had not previously seen such as unlinked items in the navigation menu.

10.4 User Testing

The last form of testing that I did was user testing. Due to the nature of my app, this was arguably one of the most key features as its success depended on its usability. Users I tested the app on were family members, close friends and fellow college students. I had a questionnaire (see appendices) that I gave to these people so that they could fill it out and give me feedback on what they thought of the system.

10.4.1 Issues

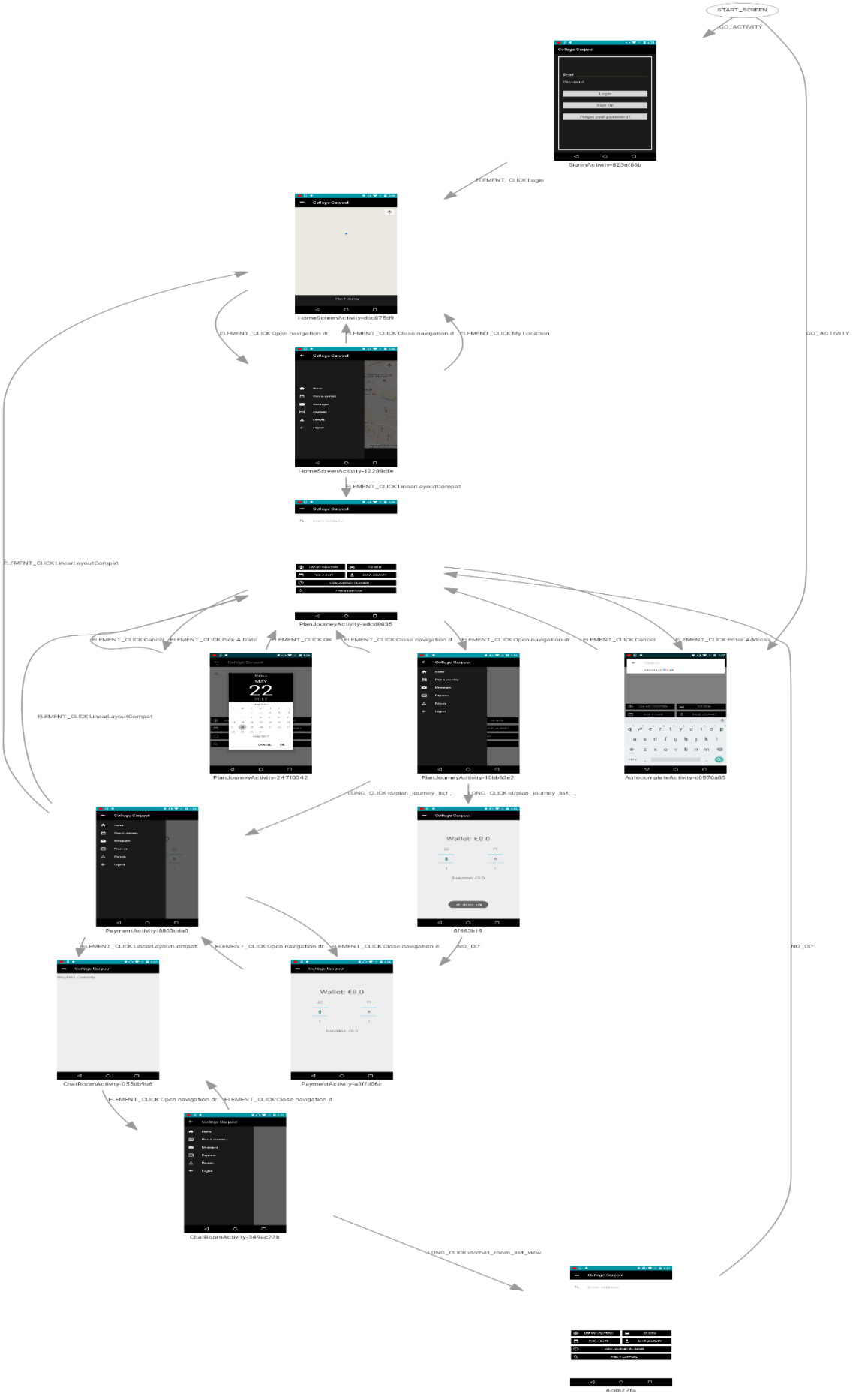
Issues that people came across were the lack of security measures associated with the app. Unfortunately, this was beyond the scope of the project to add security other than only allowing DCU students to use it. Given the circumstances, I took this criticism but unfortunately could not act to help it.

Some people also seemed opposed to the idea even though they agreed it was fit to purpose. This might have come down to the dislike of sharing rides with people they didn't know too well although I think that if this was on a larger scale, results would be more favourable.

10.4.2 Successes

Many people believed the app to be fit for purpose. They thought that the functionality worked well and they enjoyed the ability to message each other. I added the friend system under instruction from one of the test group. I thought this added another layer to the app and made it more familiar to most users.

The navigation system also got complimentary reviews from the test group. Thankfully nobody got lost as the system could guide them from location to location with ease.


START_SCREEN
GO_ACTIVITY
SignInActivity-823ef86b
ELEMENT_CLICK Login
HomeScreenActivity-dbc875d9
ELEMENT_CLICK Open navigation dr...
ELEMENT_CLICK Close navigation d...
ELEMENT_CLICK My Location
HomeScreenActivity-12289dfe
ELEMENT_CLICK LinearLayoutCompat
ELEMENT_CLICK LinearLayoutCompat
PlanJourneyActivity-edcd8035
ELEMENT_CLICK Cancel
ELEMENT_CLICK Pick A Date
ELEMENT_CLICK OK
ELEMENT_CLICK Close navigation d...
ELEMENT_CLICK Open navigation dr...
ELEMENT_CLICK Cancel
ELEMENT_CLICK Enter Address
ELEMENT_CLICK LinearLayoutCompat
PlanJourneyActivity-24703d42
PlanJourneyActivity-10bb63e2
AutocompleteActivity-d8570a85
LONG_CLICK id/plan_journey_list_
LONG_CLICK id/plan_journey_list_
PaymentActivity-8803cda0
Wallet: €8.0
8f663b19
ELEMENT_CLICK LinearLayoutCompat
ELEMENT_CLICK Open navigation dr...
ELEMENT_CLICK Close navigation d...
NO_OP
NO_OP
ChatRoomActivity-055db9b6
Wallet: €8.0
PaymentActivity-a3ffd06c
ELEMENT_CLICK Open navigation dr...
ELEMENT_CLICK Close navigation d...
ChatRoomActivity-349ac22b
LONG_CLICK id/chat_room_list_view
4c8877fa
GO_ACTIVITY

11 Problems

Through the development of the app, I came across numerous problems that hindered my project's development.

11.1 11.1 Navigation Service

The navigation service was by far the most tedious part of the app to design. This was partly due to the nature of the app and the fact that it was extremely difficult to test as I would have to code a new feature and then go out and road-test it every time I made a change. This was obviously not the most efficient system but it was the only one I had. The main problem I was having was that I had to balance the metrics of the system right so that an end step would not be triggered too early and a start step wouldn't be triggered too late. Having to juggle all the variables associated with getting the directions just right led to me having a mass of redundant code in the navigation service that I couldn't tamper with. Refactoring was the only solution so I decided to do an overhaul of all the existing code. I took certain parts of the code out that I knew I could re-use and made a structured plan to tackle the service now that I had a better understanding of how it should work. One of the single biggest problems that I made in the entire course of the project was in the "isNear()" function in the navigation service. I have mentioned this function before as the method that made my service so much more manageable but for a long time it was the problem. I was using PolyUtils to compare the distance between two latitude/longitudes and returning a boolean based on whether a threshold distance was reached. The threshold I was comparing with was of type int while the number that represented the distance between the two points was a float. This did not throw an error in the compiler but was warping the results so that they were skewed for a long time. Once I fixed this bug, the service was a lot more accurate and only needed slight tweaking to get the directions to be passed to the text-to-speech engine at the right time.

11.2 Unreliable API

During my time working with the Google Directions API for the app, it has been nothing but brilliant. The same couldn't be said for the Google Places API. It was slow to react, often crashed my program due to its connection timing out and rarely managed to get the correct location when attempting to pinpoint the user's location. I would not use this API if I had to develop the app again. Just the day before I had to carry out the video walkthrough of the app, the API went offline and my app continued to crash across all user's phones. This was one feature that did not impress me.

11.3 Limited API Calls

During the initial stages of development of the navigation service, I was making far too many requests to the Google Directions API as my program had some unforeseen loops in it. This lead to me running out of API calls for that day and so my development had to stop. This happened more than once so I finally registered my bank card details with Google to allow me access to 150, 000 API calls a day rather than the standard 1000.

12 Future Work

I believe my app can expand. It is an idea that can be taken to greater heights and is only limited by the people that use it. There are certain features that I would look to improve in the future such as the friend system and the messaging system. I would also completely re-design the UI if I had more time. The payment system could also be implemented properly rather than the mock system I currently have in place.