CA326 - Technical Specification

Project Title

Carpool App for DCU Students on Android

Students

- Daragh Prizeman (19459734)
- George Eskander (19451972)

Supervisor

Darragh O’Brien

Date Completed

04/03/2022

Table Of Contents

1. Introduction

1.1. Overview

Our project is a carpool app for DCU students on Android devices.

This app allows students who drive regularly to and from DCU campus, to sign up as a driver, and offer any empty seats in their car to help other students with a lift to or from campus.

On the other hand, students who are in need of a cheaper method of commuting to and from campus, can select the passenger role and enter their desired trip locations. Passengers can then view a list of nearby drivers ordered by the estimated arrival time of their journey. Once a passenger requests to join a driver's trip, the driver will be notified in real-time, without the need to refresh.
The driver can then either accept or deny the passenger's request. If they accept the request, the passenger is added to the trip, updating the trip information such as arrival time, distance etc. This also adds the new passenger's location to the map for the driver and all other passengers.
The app makes use of Google Maps Places and Directions API, to calculate the fastest route between locations and show to users their route on a map.

1.2. Motivation

As we are both current DCU students who commute to and from campus every day using public transport (buses and trains), so we know from first hand experience that commuting can be expensive, and sometimes lonely. We thought of creating this app to provide students with a cheaper, more social method of getting to and from DCU campus. This carpool app would also help to reduce pollution by reducing the number of cars on our roads, if students were to travel together.

1.3. Glossary

API - stands for Application Programming Interface; they are used for communication between different applications.

React Native - UI framework used for creating cross-platform mobile applications without having to write lots of "native" code, it is similar to React but with native components instead.

Django - backend framework for creating web and mobile applications through Python.

Encryption - The conversion of data into code to protect information from unauthorised access.

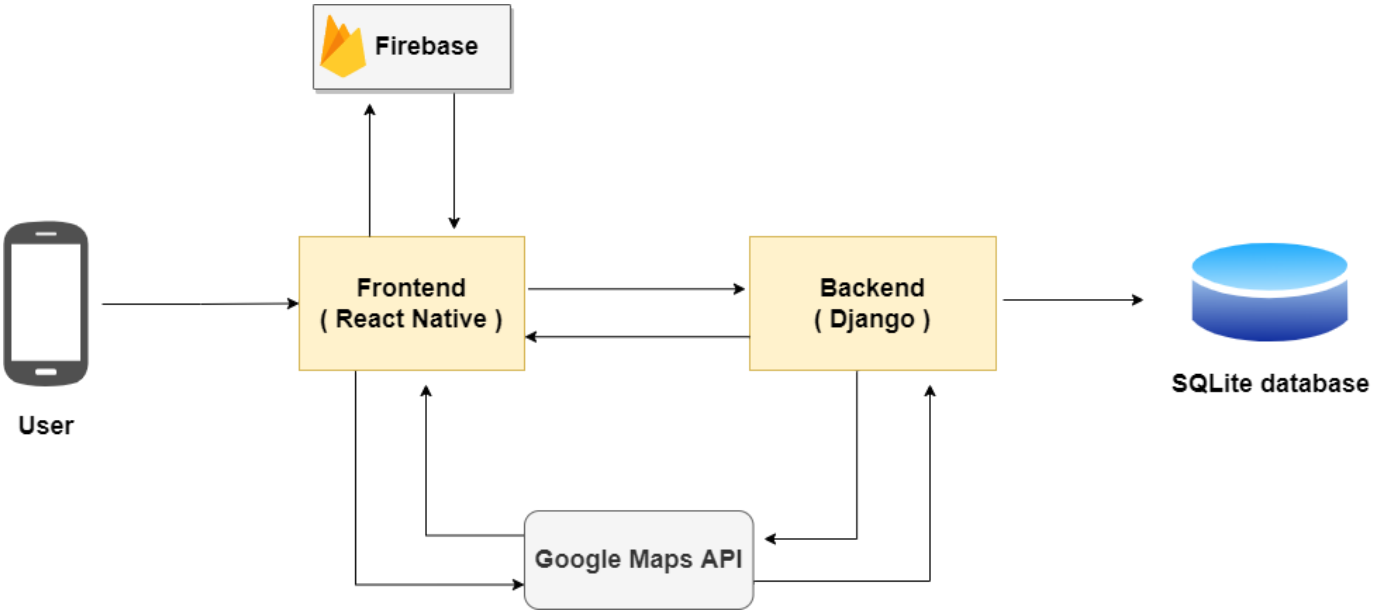
Firebase - cloud messaging service. We used it's real-time database feature.

Heroku - cloud platform which we used to host our backend server.

SDK - stands for Software Development Kit; they are used to create applications for specific platforms.

2. System Architecture

Below is a diagram representing the system architecture of our app



Backend

We developed the backend of our carpool app using Django. We implemented Django's REST framework to handle user authentication and password encryption. We connect a SQLite database to our backend to store the models we created for this application. We deployed the backend server online using Heroku. We make requests to this server, from the frontend using the following URL: http://blooming-shelf-28383.herokuapp.com + /endpoint. For the full set of endpoints for our backend API, see 'src/carpool-app/backend/carpool/urls.py' in our git repository.

Frontend

We developed the frontend of our carpool app using React Native. The frontend communicates with the backend by sending data to and from the backend server. Once the frontend was fully developed, we published the app as an SDK using Expo.

3rd party components

Google Maps API

The frontend makes use of the Google Maps Places, Distance Matrix, and Directions API to show users a map of their route.

The backend makes a HTTP request to Google Maps Directions API and we use the response to calculate arrival and departure times for each waypoint in the trip, and also the distance and duration between each waypoint. This calculation can be seen in the get_route_details() function inside our ‘views.py’ file, which is located in the ‘src/carpool-app/backend/carpool/’ directory in our git repository.

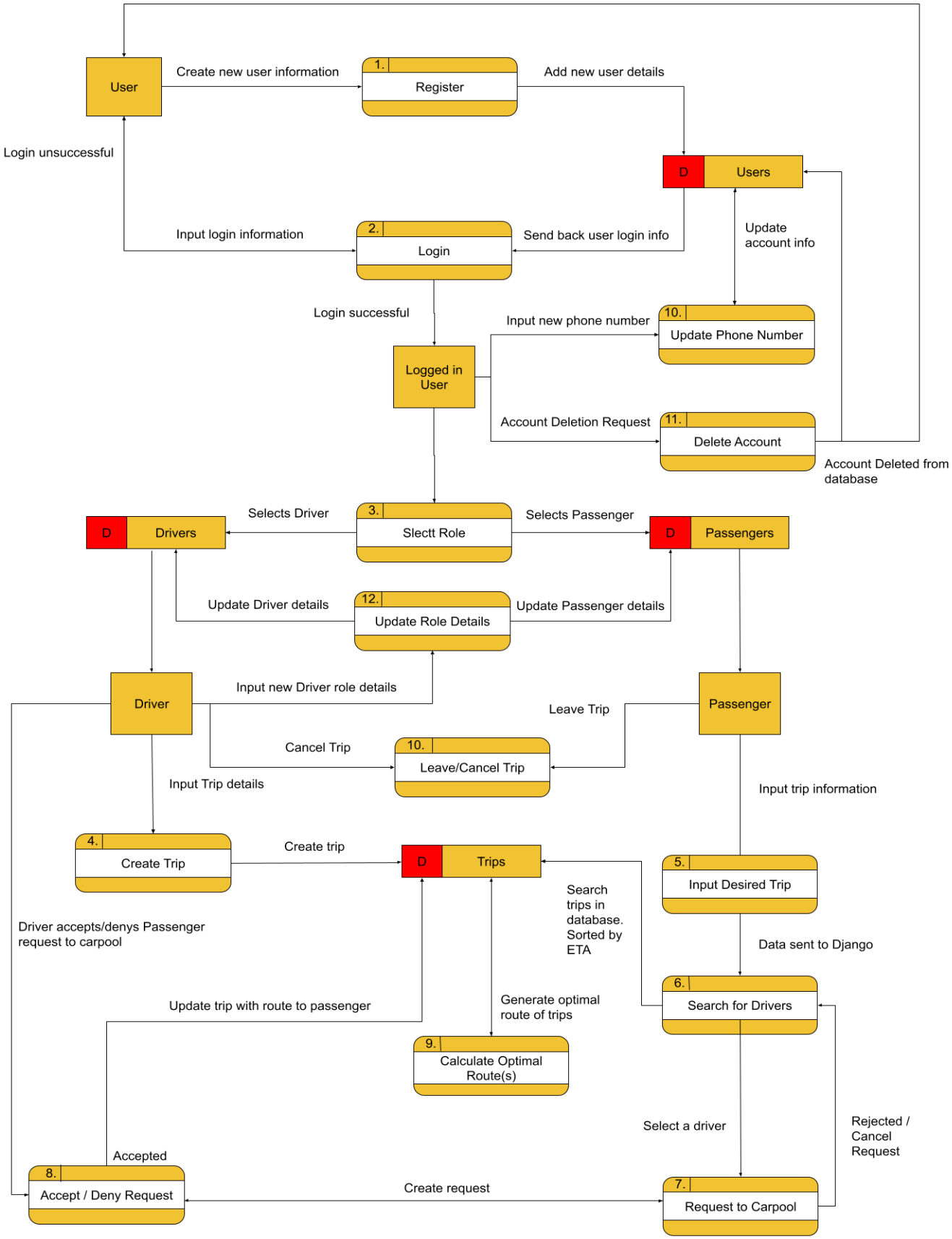
Firebase

We made use of Firebase’s real-time database in the frontend to allow us to show passenger requests to drivers in real time. This also allows the passenger to be instantly updated when their request is accepted.

3. High Level Design

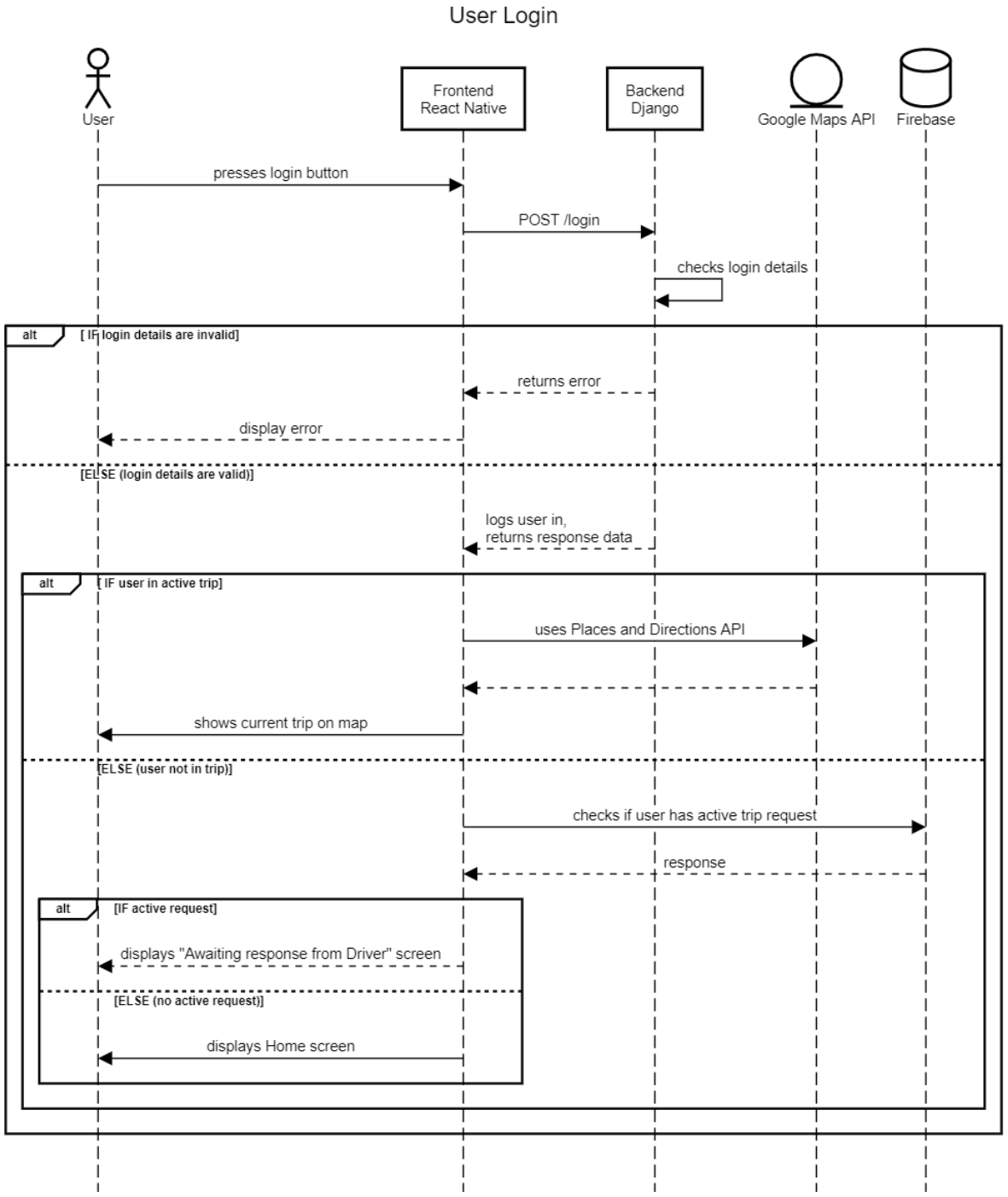
3.1. Data Flow Diagram

(see next page.)


User
Create new user information
1.
Register
Add new user details
Login unsuccessful
D
Users
Input login information
2.
Login
Send back user login info
Update account info
Login successful
Input new phone number
10.
Update Phone Number
Logged in User
Account Deletion Request
11.
Delete Account
Account Deleted from database
D
Drivers
Selects Driver
3.
Slectt Role
Selects Passenger
D
Passengers
Update Driver details
12.
Update Role Details
Update Passenger details
Driver
Input new Driver role details
Leave Trip
Passenger
Cancel Trip
10.
Leave/Cancel Trip
Input Trip details
Input trip information
4.
Create Trip
Create trip
D
Trips
5.
Input Desired Trip
Search trips in database. Sorted by ETA
Driver accepts/denys Passenger request to carpool
Data sent to Django
6.
Search for Drivers
Update trip with route to passenger
Generate optimal route of trips
9.
Calculate Optimal Route(s)
Select a driver
Rejected / Cancel Request
Accepted
8.
Accept / Deny Request
Create request
7.
Request to Carpool

3.2. Sequence Diagram for user login

The below sequence diagram shows the relationship between the frontend and backend of our application and how they incorporate third party components to show the correct screen to the user after login. Note that this is showing the relationship between components for user login only, however other user functions work in a similar way.



4. Testing Strategy

4.1. Unit Testing

We implemented unit tests for each of our backend models and views that were necessary to ensure they were functioning as expected. These unit tests can be found in the ‘test.py’ file located in the ‘src/carpool-app/backend/’ directory of our git repository.

4.2. Graphical User Interface Testing (GUI)

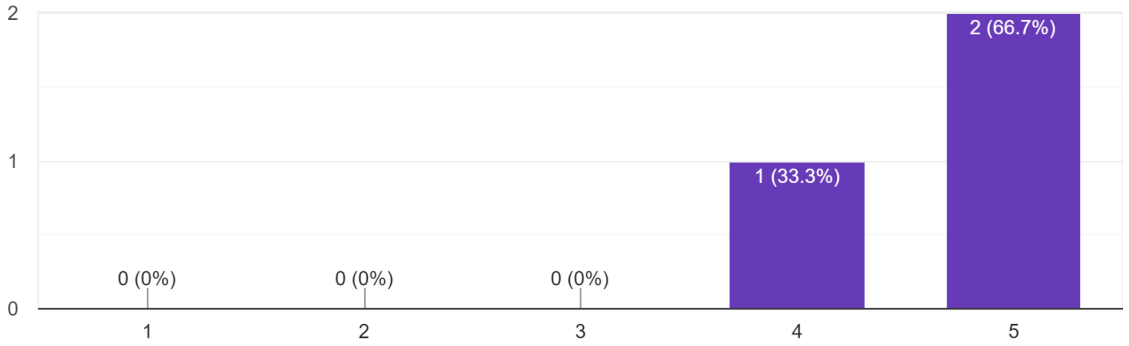
Before getting real users to test the usability of our app, we conducted our own manual GUI tests, to ensure that each functionality of each component was functioning as intended. When we found that a particular component wasn’t functioning as expected on the user interface, we looked back at the code and fixed the problem. Then we tested it again to ensure it's working.

4.3. User testing

Once we had the frontend of our app ready, we got a few other DCU students to conduct a usability evaluation of our app. We first got ethical approval to conduct this user testing. You can see our plain language statement, user tasks, and feedback questionnaire in the Appendices of ‘ethics.pdf’ file located in the ‘docs’ folder in our git repository. Unfortunately, we only had time to conduct user testing on 3 other DCU students. This is something that we will do earlier in future projects. However, the feedback we got was overwhelmingly positive. See some examples below.

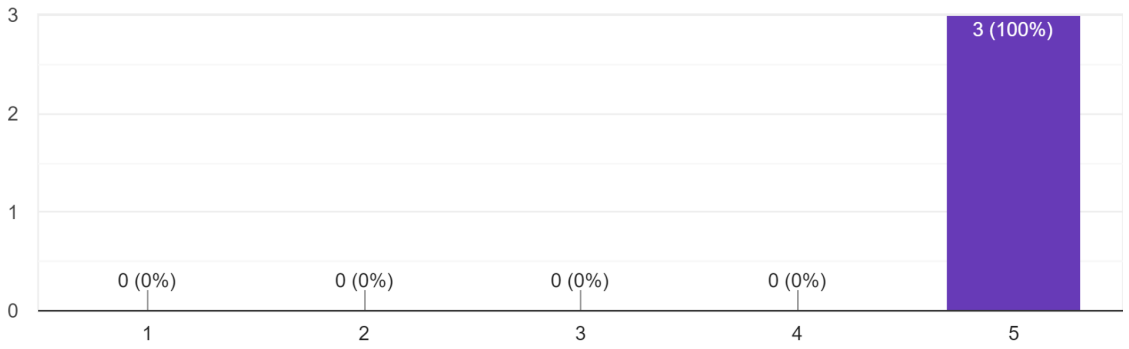
How would you rate the ease of use of the app?

3 responses



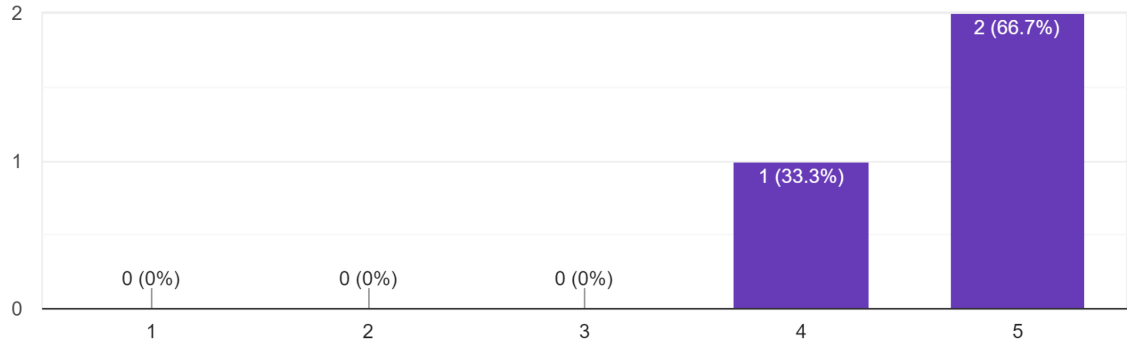
How would you rate the overall design of the app?

3 responses



How likely are you to use this app regularly, if it was available to you?

3 responses



Please give a reason for your answer to the above question.

3 responses

i commute to college from outside dublin every day, would be a very useful service.

Easy to use, functions correctly, well designed.

I would use the app often to save money commuting

Anything else you would say about the app?

2 responses

sell it to a taxi company!

This app is a great idea and very easy to use.

The above responses, although limited, show that there is an appetite for this app by DCU students who commute regularly.

5. Problems and Resolutions

5.1. Solving screen space for setting up a trip

Since we had no previous experience with developing or designing mobile apps, this was a challenge. Setting up a trip can be different depending on whether you are a driver or passenger.

In order to solve this problem, we had to make the UI more dynamic or flexible depending on the input users give. For picking locations, we decided to put it at the top, with everything else hidden below until both locations are entered.

```
// Component for selecting campus buttons and To/From DCU buttons at top of Trip Screen
function CampusDirectionSelector({campusSelected, setCampusSelected, isTripToDCU,
setIsTripToDCU}) {
       const dispatch = useAppDispatch();

       return (
       <View style={{borderBottomColor: "blue"}}>
       {isTripToDCU === undefined ?
                     // from or to dcu buttons
                     :
                     // campus buttons depending on to/from DCU
              }
       </View>
       )
}
```

```
// function LocationInputGroup

{(!showWaypointCollapsible && (!trips.locations.startingLocation.info.isEntered ||
trips.locations.destLocation.info.isEntered) ||
(trips.locations.startingLocation.info.isEntered ||
!trips.locations.destLocation.info.isEntered) && !showWaypointCollapsible) &&
   <CampusDirectionSelector
       campusSelected={campusSelected}
       setCampusSelected={(value: string) => {setCampusSelected(value)}}
       isTripToDCU={isTripToDCU}
       setIsTripToDCU={(value: boolean | undefined) => {setIsTripToDCU(value)}}
   />
}

{(!showWaypointCollapsible || (trips.locations.startingLocation.info.isEntered ||
trips.locations.destLocation.info.isEntered)) ?
   isTripToDCU ?
       ((!showWaypointCollapsible && trips.locations.destLocation.info.isEntered) ?
       <CreateGoogleAutocompleteInput
           locationObjName={"startingLocation"}
           placeholder="Enter your starting point..."
```

```
            style={{rounded: 5}}
        />
        : null)
        :
        ((!showWaypointCollapsible && trips.locations.startingLocation.info.isEntered) ?
        <CreateGoogleAutocompleteInput
            locationObjName={"destLocation"}
            placeholder="Enter your destination..."
        />
        : null)
    : null
}
```

Once both locations are picked, we needed components that could be hidden or reduced in size or could cover all the other components. These were done for the other inputs such as departure time and number of seats, and waypoints show up, these are collapsible components. Similarly for ‘show trips’ in the passengers screen we have a swipeable panel. For waypoints, when pressed we hide all other inputs except for the create trip button for the driver.

This way we had a functional and user-friendly UI for setting up trips.

5.2. Optimising trip search for passengers

We needed a way to optimise trip search taking into account drivers and passengers constraints and their estimated time of arrival (ETA). We decided to focus on constraints such as time constraints, route constraints, and trip constraints (such as available seats).

To solve this problem, we had to first consider all the factors and possible cases for searching a trip. Factors we found included: time of departure, ETA between all passengers, the trips destination, and the ETA to the passenger, trips number of seats, and the direction i.e. to or from DCU.

First we check, whether or not the passenger is starting from one of DCU’s campuses, after that we check for trips that include the

passengers campus location depending on whether it's to or from DCU. We then sorted these trips found by time of departure first, after that depending on to/from DCU, we set up the new potential route details of those trips. These are then finally sorted by ETA.

See **get_trips** in views.py

5.3. Showing trip requests to drivers in real-time.

Initially, we were using only a SQL database in Django to store all trip information. However if a driver was on the 'Driver' screen, it wasn't showing new passenger requests live as they are requested. To solve this problem, we set up Firebase's real-time database. When a passenger makes a request, it stores the trip request in Firebase using the following function.

```
// This hook is called when a passenger presses the request button on a trip
// Function takes in tripID and passenger data.
// creates a trip request in the Firebase database. Also sets user request
status to "waiting" indicating they have an active request.
export async function storeTripRequest(tripID, passengerData) {
    const db = getDatabase();
    let status: boolean;
    return (
        get(ref(db, `/trips/${tripID}`)).then((snapshot) => {
            if (snapshot.val() !== null) {
                status = snapshot.val().status === "waiting" &&
snapshot.val().availableSeats !== 0;
                if (status) {
                    update(ref(db, `/tripRequests/${tripID}/`),
{[`/${passengerData.passengerID}`]: {...passengerData}});
                    update(ref(db, `/users/`) ,
{[`/${passengerData.passengerID}`]: {tripRequested: {tripID: tripID,
requestStatus: "waiting", status: ""}}});
                }
                return status;
            }
            else {
                status = false;
            }
            return status
        })
    )
```

```
}
```

Then we set up a listener (see below) for the driver to update everytime a new passenger request is added to Firebase.

```
onValue(ref(db, `/tripRequests/${trips.id}`), (snapshot) => {
                tempFbTripsVal = {
                    ...tempFbTripsVal,
                    data: {
                        ...tempFbTripsVal.data,
                        tripRequests: snapshot.val() !== null ?
snapshot.val() : {}
                    }
                };

                setFirebaseTripsVal(tempFbTripsVal);
            })
```

Now the driver is updated live on screen when a new passenger makes a request.

5.4. Synchronising trip data among drivers and passengers

Since realising we needed to have real-time data, we found problems such as when users logged out while requesting a trip and got accepted/denied by a driver or getting map data after being accepted while in the app.

Since a lot of our data sits in our Django backend, we needed a way for Firebase to tell Django to fetch the correct data for the user depending on the data in Firebase.

We solved this by linking the ids of passengers, drivers, and trips etc. in our SQLite db with the data in Firebase. Anything linked with data in Django in Firebase has a status for telling Django what to fetch while also being dependent on what's in Firebase.

5.5. Preventing user errors

There were many possible cases for user error we found throughout the project, such as deleting your account while driving for an ongoing trip or requesting a trip while having an ongoing trip as a driver (and vice versa).

To solve most cases of these user errors, we decided to make an alert modal which can't be escaped until a user has given an answer. Here is an example of the alert we made for a driver cancelling their trip, which can be found in 'components/DriverCurrentTrip.tsx' in the frontend folder.

```
{/* Shows "Are you sure?" modal, when driver presses cancel trip */}
 {isCancelTripPressed &&
    <TripAlertModal
       headerText="Are you sure you want to Cancel Trip?"
       bodyText="This action is irreversable."
       btnAction={{
          action: () => {
              cancelTrip()
          },
          text: "Yes
       }}
       otherBtnAction={{
          action: () => {
              setIsCancelTripPressed(false)
          },
          text: "No"
       }}
   />
}
```

Specifically for when a user has an ongoing trip, navigation is restricted to the role they have a trip as and prevents the user from deleting their account until that trip has completed.

Another case we found is the driver not completing the trip, since passengers can't leave until the trip is complete, we let one of the users of the trip end the trip once it passes an hour over the ETA.

Sample of navigation restrictions, see **index.tsx** in src/carpool-app/frontend/ directory.

```
{(user.status === "available" || user.status === "driver_busy") &&
 <Tab.Screen name="Driver" component={DriverScreen}
   options={
       {tabBarIcon: () => {return <Ionicons name="car-outline" size={25}
color="grey"/>}}
   }
   listeners={({navigation, route}) => ({
     tabPress: (e) => {
       e.preventDefault();
       let routeName = navigationRef.current?.getCurrentRoute().name;
       let routeCondition = routeName === "Passenger";

       if (user.status === "available") {
         if (routeCondition) {
             setTabAlert(true)

             // warn that changes may be lost
             setShowStatusAvailableFromPassengerAlert(true);
         }
         else {
           if (trips.role === "") {
             dispatch(updateRole("passenger"));
           }
           navigation.navigate("Passenger")
         }
       }

       if (routeName === "Passenger") {
         if (user.status === "available") {
             setTabAlert(true)
             // warn that changes may be lost
             setShowStatusAvailableFromPassengerAlert(true);
         }

         if (user.status === "passenger_busy") {
           if (routeName !== "Passenger") {
             setTabAlert(true)

             // alert you have an ongoing trip as driver
             setShowStatusPassengerBusyAlert(true);
             navigation.navigate("Passenger")
           }
         }
       }
       else {
          navigation.navigate("Driver")
       }
     }
   })}
 />
}
```

6. Installation Guide

Firstly, ensure that you have ‘Expo Go’ installed in your android device. This is available from the Google Play Store.

To use our carpool app, all you need to do is scan the QR code below on your Android device:

Or visit the following link on your android device:

exp://exp.host/@prizemd2/frontend?release-channel=default

This will take you directly to our app without any installation needed, as our ‘frontend’ folder is published using Expo and the backend server is hosted on Heroku.

For the list of packages we used in this project

See **“requirements.txt”** in carpool-app/backend/ directory

And

See **“package.json”** in carpool-app/frontend/ directory