

# Student Carpooling App

## Technical Specification



### Members:

- Hannah O'Connor - 16382283
- Catherine Mooney - 16416052

# **Table of contents**

## **1. Introduction**

- 1.1 Overview
- 1.2 Glossary

## **2. System Architecture**

- 2.1 Description
- 2.2 Diagram

## **3. High-Level Design**

- 3.1 Context Diagram
- 3.2 State Machine Diagrams
- 3.3 Class
- 3.4 Component Diagram
- 3.5 Data Flow Diagram
- 3.6 Firebase Data Organisation

## **4. Problems and Resolution**

- 4.1 Searching Active Chats
- 4.2 Removing Polylines and markers from map
- 4.3 Data changes not showing in real time
- 4.4 Google Maps and Directions API

## **5. Installation Guide**

# 1. Introduction

## 1.1 Overview

*Student Carpooling* is an android app built and developed specifically for Ireland's university students. The app is available to any student who has an active and recognisable Irish university email address, along with an Android mobile device. It based on the typical idea of carpooling, but instead, works to connect student communities exclusively.

The app comprises of two user modes, passenger and driver, with some similar but mostly different functionality for each mode.

The main features of this application include:

- Creating a trip
- Creating a trip notice
- Requesting to join a trip
- Map Activity
- Real time Driver Tracking
- Route automation
- In app messaging system
- User rating system
- Automatic login(app remembers if user is driver or passenger from previous login)
- Password Recovery
- Navigation drawer to access all the main functions of app ( including logout)
- Splash Screen
- Automatic push Notifications

Student Carpooling was developed using Android SDK with Java on Android Studio, the official integrated development environment (IDE) for Android app development. The UI is designed for mobile devices only and targets API level 28, with a minimum target of 26. We had originally planned our app development using a minimum API level of 14, with the hopes to target broad range of android OS devices. However, with the integration of google maps and directions, we found a lag in performance across lower API Level devices with google play services and the map activity was not able to load correctly. We found that with the increase of the min sdk, we could ensure strong performance on the device running the app.

Our user interface (UI) was designed to be as user friendly and accessible as possible. For students, fun and bright colors were used in order to achieve an extremely welcoming and modern feel. We wanted our users to have the most simple yet enjoyable experience possible when navigating through the app.

With the incorporation of optional profile images within the profile activity, an image loader library, 'Glide', recommended by google, was utilised alongside Firebase Cloud Storage. This API provided bindings to download our user images, stored within Cloud Storage, and

load and display them within the app. Following that and the designing of our UI, we decided to opt for circular image views. However, after discovering that there wasn't a view component for circular images within android studio, we included an API that allowed us to. (<https://github.com/hdodenhof/CircleImageView>)

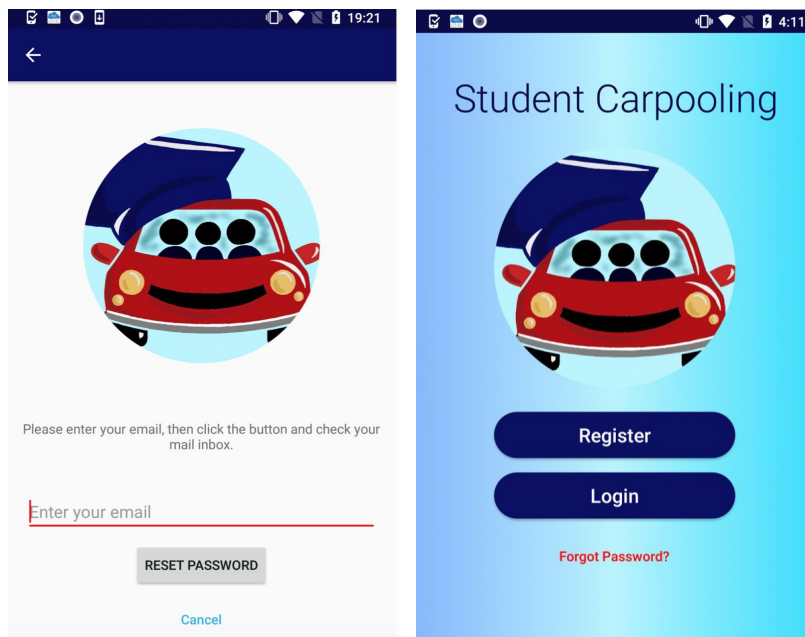
A Drawer navigation was chosen to provide simple and easy access to all of the app functionality and to reduce the users' short term memory load.

The app's theme consists of orange, yellow and red tones sided with dark blues and navy to establish a high contrast between colours. We chose a mixture of cool and warm tones to stimulate the users visual senses. We carefully chose this selection as we wanted the app to be extremely original and have a more youthful appeal. Originally, we had chosen a range of blues and white as our base colors, primarily considering anyone who had difficulty distinguishing between colors. In spite of that, through researching, we found that most other applications within the play store used that same colour palette and it seemed like a safe option. However, after anticipating and debating, we decided that we wanted to strive for originality instead..

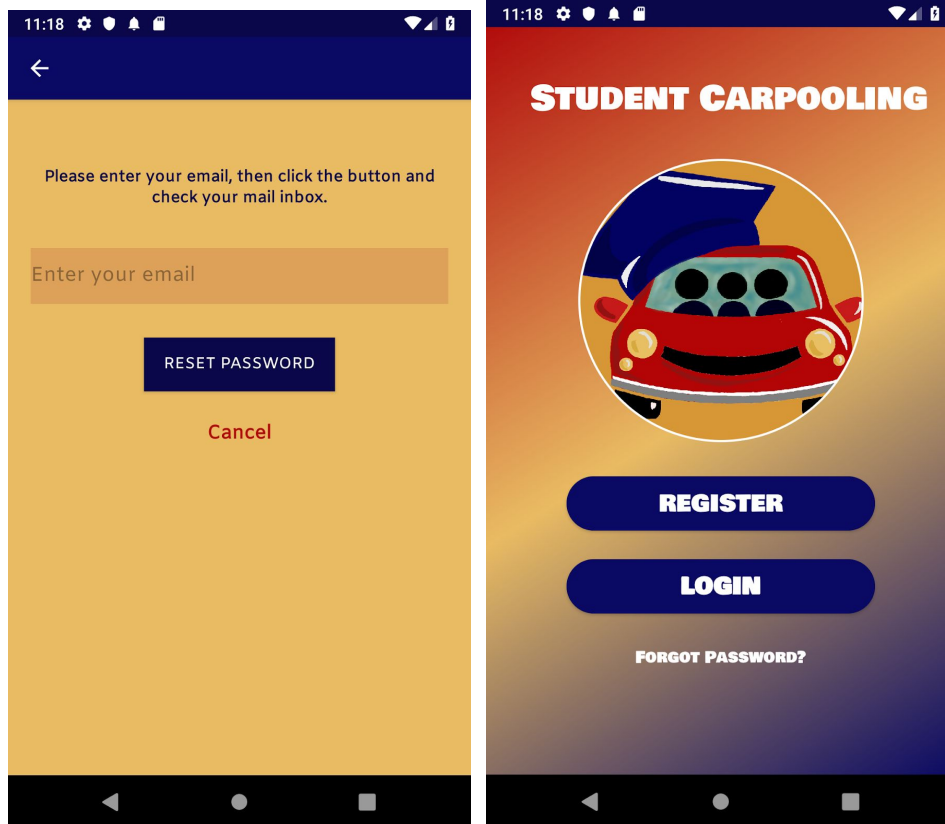
Also, within our user testing, we asked our participants which design they found more appealing and nearly all chose the final design. So with that the positive feedback, we decided to opt for the sunset tone over the blue. We have shown this below.

## Original UI versus Final UI

### Original:



### Final:



## 1.2 Glossary

- **Android OS:** Google's Linux-based open source operating system for mobile devices.
- **Firebase:** is a mobile application cloud-based development platform owned by google, it acts as the server and provides tools in order to develop our app.
- **Google Maps API:** A service in order to enabled google maps within our application and allows for location markers too.
- **Geo-Fire:** GeoFire is an open-source library for Android that allows you to store and query a set of keys based on their geographic location using the Firebase realtime database.
- **Firebase Authentication:** A service that can authenticate our users using through email and passwords. When a new user signs up with an email, it sends a verification email to that address before storing to database. It handles sending password reset emails too.
- **Firebase Real-time Database:** No SQL, cloud hosted database that syncs and stores data across our app users within real time.
- **Google Play Services:** Required in order to use google APIs such as Google Maps within our app.
- **Place Autocomplete API:** A service that provide autocomplete functionality and predictions for string geographic searches

- **Google Directions API:** A service in which calculates directions between locations and adds polylines to the map.
- **Firebase Cloud Storage:** Firebase SDK to allow for the storage of user-generated content, such as photos.
- **Glide API:** Glide is a fast and efficient open source media management and image loading framework for Android that wraps media decoding, memory and disk caching, and resource pooling into a simple and easy to use interface.

## 2. System Architecture

### Description of system Architecture

The current system architecture has stayed inline with the proposed system architecture discussed within our functional specification, along with the use of additional third party APIs. The system architecture of our applications follows a two tier architecture, with the major components being the application itself and firebase, which acts as our backend. The client tier connects the android application to the firebase database, and the firebase sdk provides static hosting for the app.

The data stored, from our application content, within firebase real time database, is retrieved within the app through the attachment of listeners to a specific database reference and those listeners update the application whenever triggered by a change in that database reference. This powers the majority of the application's functionality. The real time database stores the app's data as json strings, which are then synchronised to the client in real time. The synchronising of the data in real time is enabled by communication between the database and the client are handled via websockets.

Through the use of Firebase Authentication library, the application was able to verify it's student users by sending of a verification email upon signing up. We were able to format the registration page in such a way that the email domain name matches those stored within our application. If the user begins typing in this field, the domain name result is shown and completed automatically, and if incorrect, the app will inform the user through the use of a simple toast message. This is what establishes the student exclusivity of our app, which is the core and fundamental distinction among other conventional carpooling apps.

For the app's map activities, after a driver creates a trip, their destination is converted to geographic coordinates and saved within the database. When a passenger user wishes to join that trip, it follows the same process. The user will be asked to enter just their desired

pick up location within the search dialog, which auto completes the inputted string. This was achieved through the use of the Google Place Autocomplete API, which ensured that a recognised place would be entered, and also worked to combat any errors in spelling. In addition, it allowed us to apply filters to those search results, the first filter being locations in Ireland only and the second being an address so that more of an exact location could be achieved for those passenger pick up points to make the driver more informed. The string addresses were converted into real geographic coordinates for the map activity, through the use of the geocoder class, which is a part of Android Studio.

In order to include route automation, we implemented the Google Directions API. This API allowed us to create a google api context which what enabled the app to calculate the distance. A direction API request is then initialized using that google api context and the driver's current location is set as the origin. Each passengers pick up coordinates are then retrieved from the database and added to the map, attaching an onwindowclickedlistener to each, so that when a passenger marker is clicked, an alert dialog is shown to the driver, asking if they wish to calculate the route to that marker. If 'yes' is clicked, the function to calculate the route is called, the retrieved result to then passed to the function to draw the route. In order to specify the route result with its associated polylines, we created a Route class, which creates a route object that comprises of the legs from the directions result, which gives use the duration and distance information and links it to it's generated polyline.

We implemented the driver tracking feature by getting the drivers current device location once a trip has been started by that driver, and through the use of the open source api, geo fire, we can update the drivers last known location at each one second interval. Some problems, however, were found with this api, which was solved to the syncing of an older version. Within the passengers view, the updated driver location is shown each second, and it appears the driver is moving through the map. We created this activity so that when the driver is less than 100 kilometres from their destination, the rating system is prompted and member of a trip can rate their fellow carpoolers.

The ratings gathered for each user are stored within their user info. A function was created within the user profile activity to calculate the average rating, and display this using an android studio rating bar widget, for other users to see, and enhance user experience. In terms of notifications, we used another third party api called one signal. When a user signs back into our app, a unique notification key for the users device is generated and added to their user information within the database.

We have created scheduled notifications for drivers to remind them of when their trip is about to start, and an hour in advance. This feature was relatively easy to implement. First we got the trip date and time from the database for each particular upcoming trip, we converted the time to millisecs and then used it to create a java data object, adjusting the time to an hour in advance. We then implemented a countdown timer object to send this notification once the timer was complete.

We had a complication with the notifications however, as we would of liked to open the activity for which the notification is related to, but due to our time constraints, this feature has not been implemented. The notification will be received within the notification bar of the device and when clicked the home pages for either the driver or user will be shown, the number of unchecked notification however are shown in the app logo and will disappear once the user enters the app.

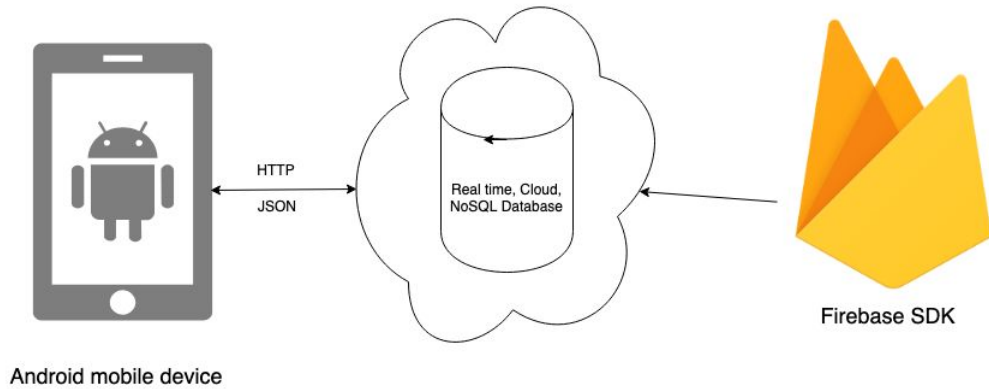
We ensured that if a user loses internet connection, the application will work to some extent offline through the caching of the data that's synchronized. Although Internet connection is listed as a requirement of the app, you may not be able to perform functions, but the data that's already saved will remain available. We wished to enhance the user experience, so that when internet connectivity is lost, the app stays responsive.

### **System Architecture Diagram**

The below diagram shows the architecture of our application. The elements involved show the front end, that is the android application. For the backend, Firebase is used to host the application and includes the various SDKs required for the operations of Student Carpooling

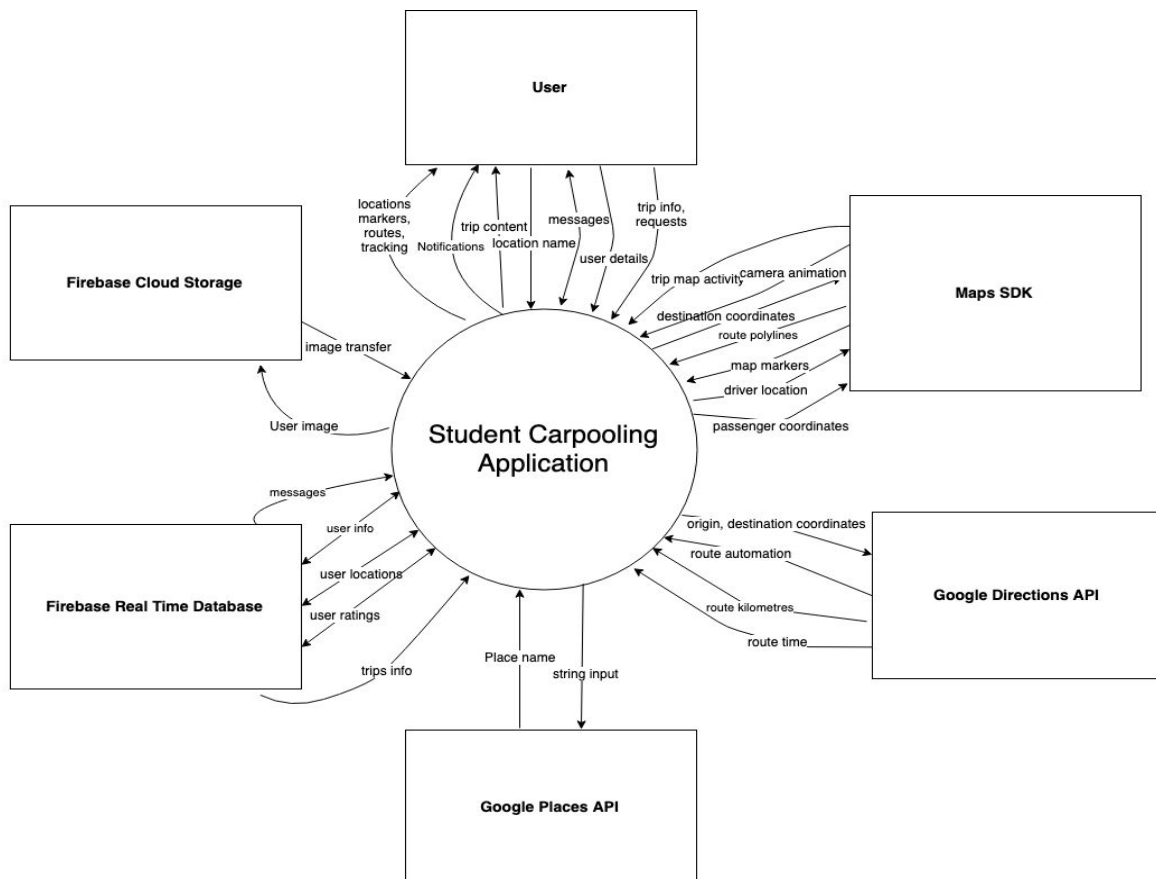


## Two Tier Architecture



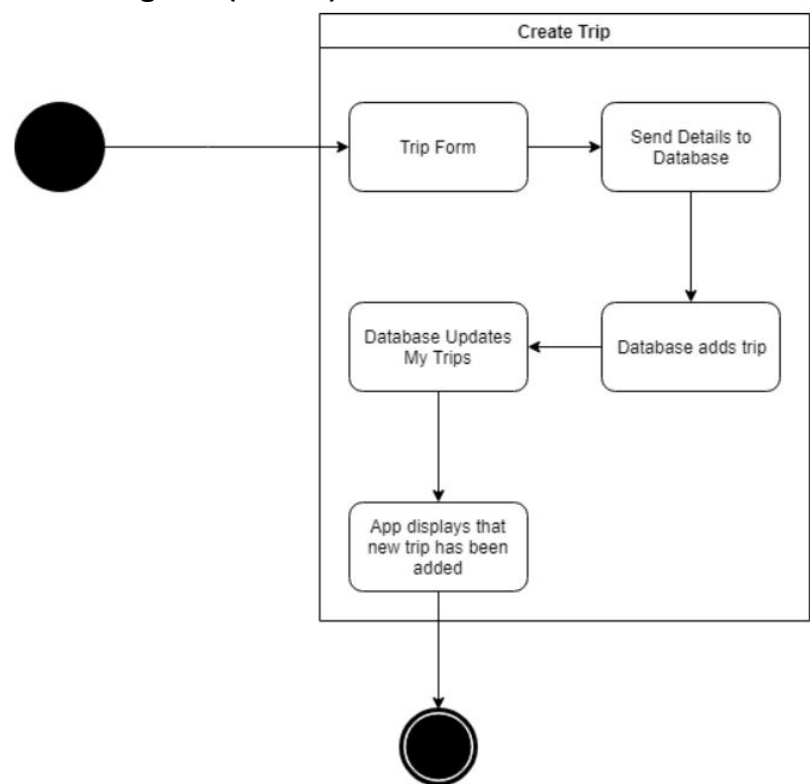
## 3. High-Level Design

### 3. 1 Context Diagram<sub>m</sub>

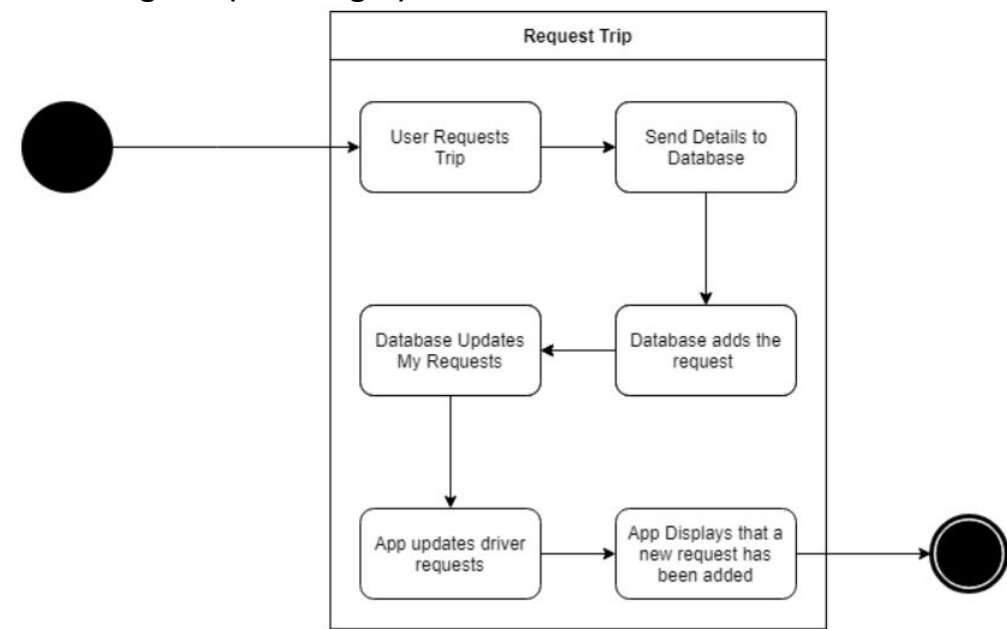


### 3. 2 State Machine Diagrams

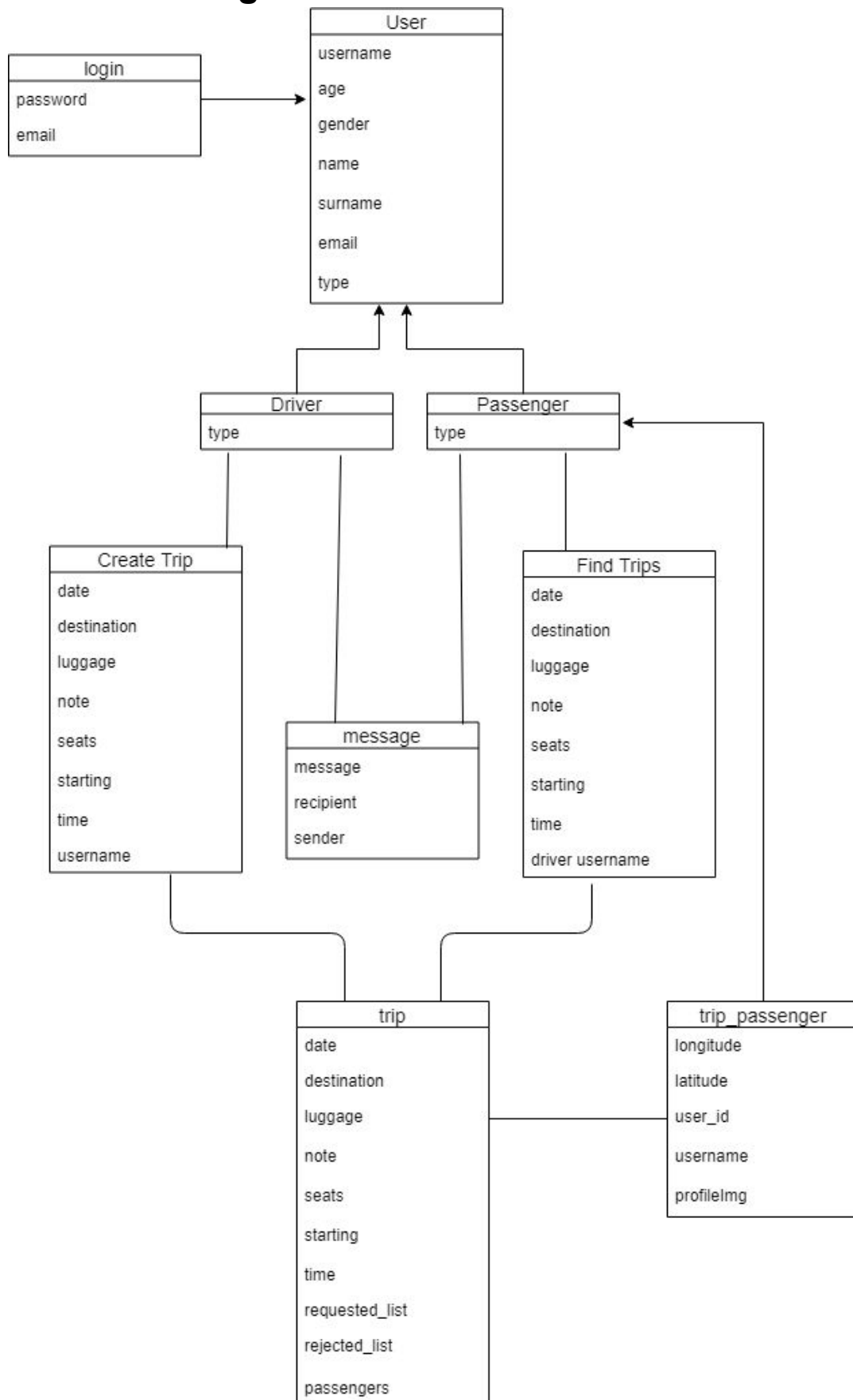
State Diagram (Driver)



State Diagram (Passenger)

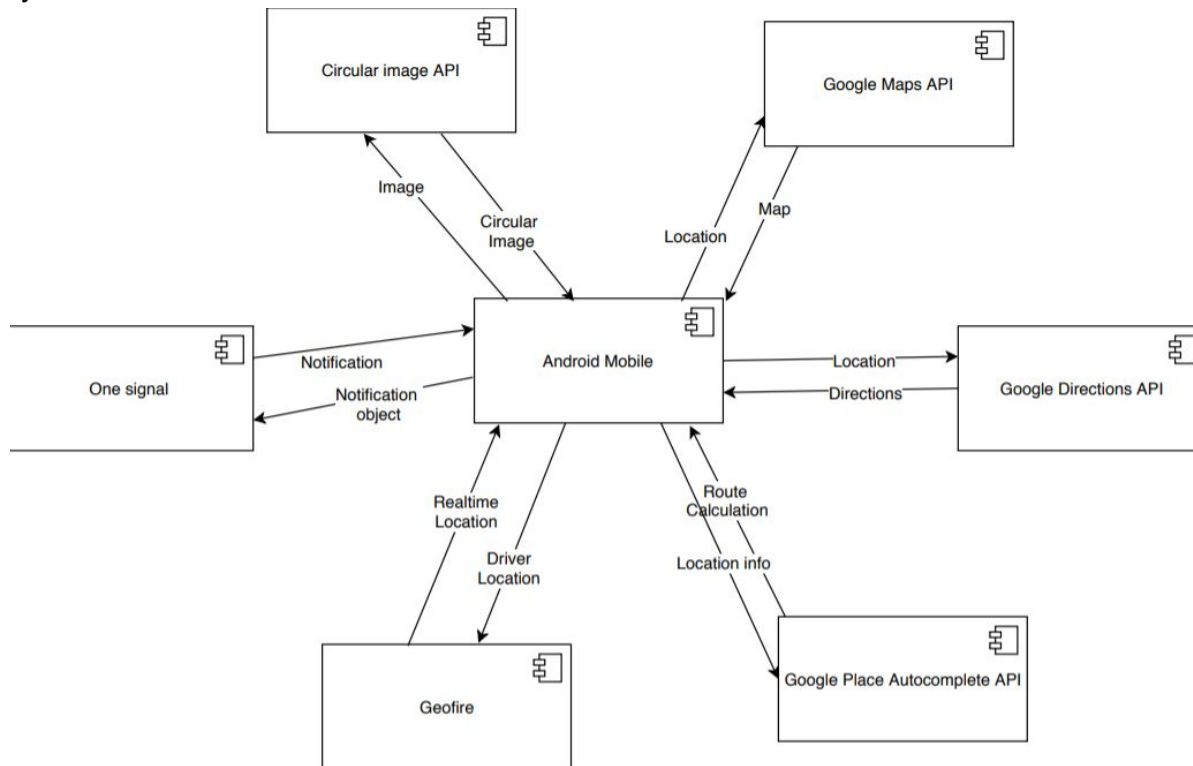


### 3. 3 Class Diagram

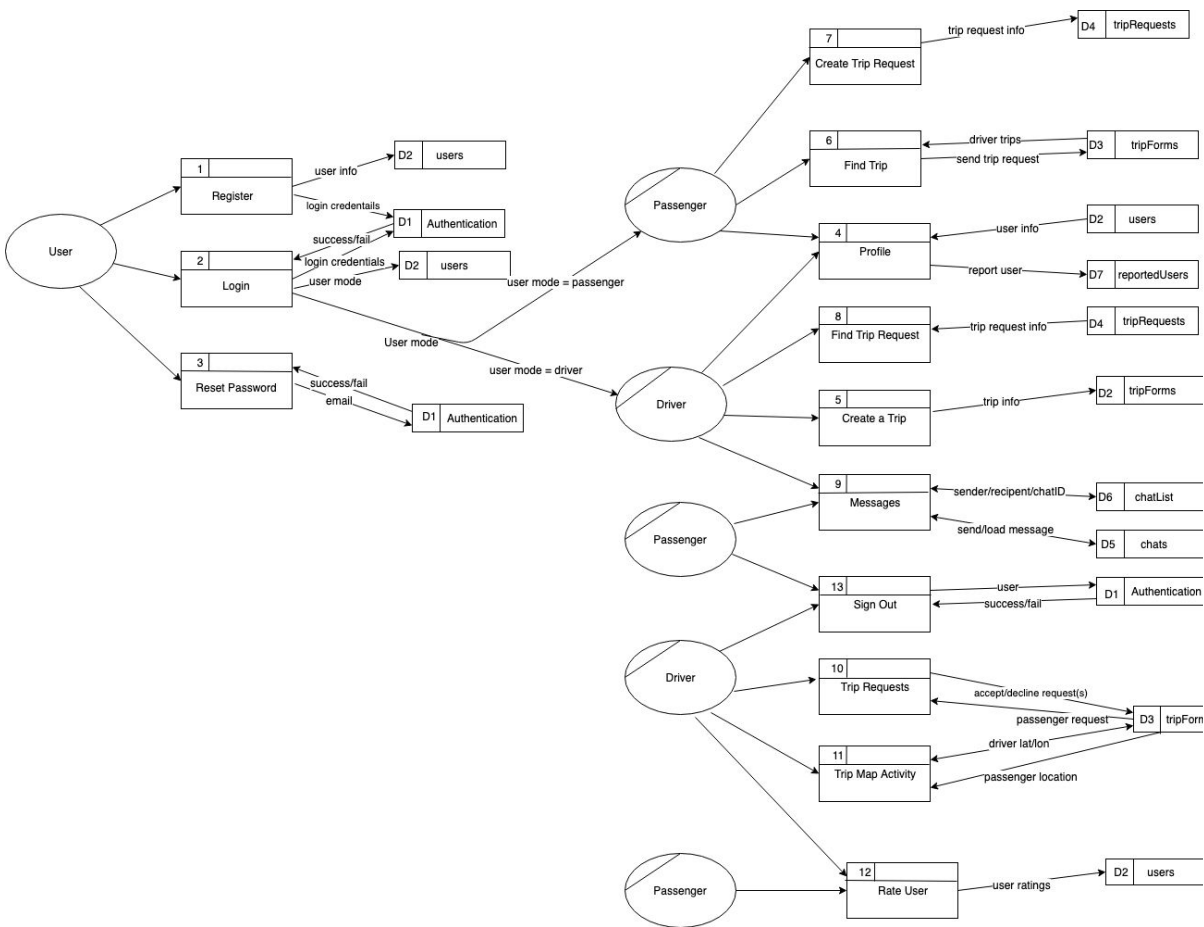


### 3. 4 Component Diagram

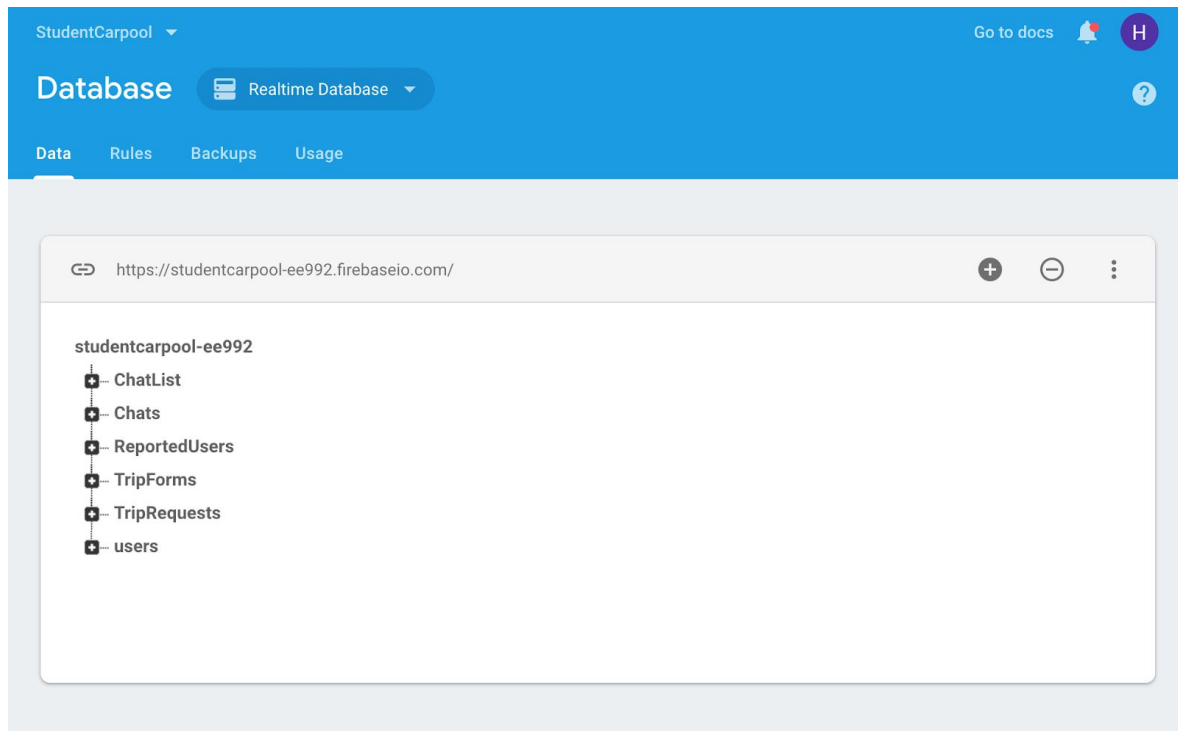
The component diagram shows the interaction between the various components of the system



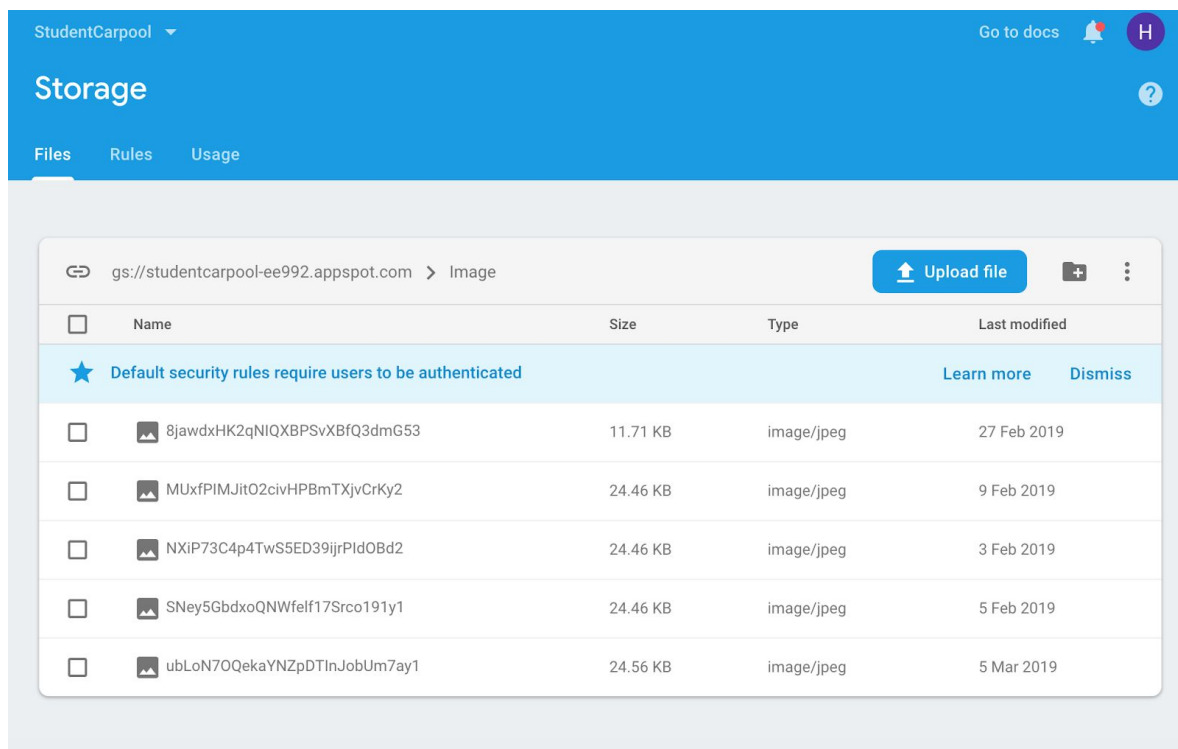
### 3.5 Data Flow Diagram



## 3.6 Firebase Database Organisation



## Firebase Cloud Storage Organisation



- The name of the image corresponds to the user's unique ID

## **4. Problems and Resolution**

### **4.1 Searching Active Chats**

#### **Problem**

When implementing our messaging system, the plan was to include a list of active chats that the current user had. In an attempt to implement this, we created a database reference 'Chats' whenever a message was sent by a user, it contained the sender and receiver id, along with the content of the message. We added a listener to that database reference and searched for if the user was a sender or receiver of a message and if so, show their correspondent within their list of active chats. However, due to the large amount of data generated from messages sent back and forth, it crashed our app and would only work when tested with a small amount of messaging data.

#### **Resolution**

To resolve this issue, we decided to create another database reference called 'ChatList' so that when a chat was started by either one of the users. This reference includes a unique chat id, original sender and recipient user ids. So, in that way we could create functions to check if the user was a receiver or sender within any of the children of that reference, rather than having to loop through and check each individual message.

### **4.2 Removing the Polylines and markers from the map**

#### **Problem**

Within the driver map activity, once the polylines were added to determine a route from the driver's location, there was no way of removing them or recalculating a route from their current location to another marker. Also, when a driver reaches one of their passengers and picks them up, their passenger's original marker was left on the map.

#### **Resolution**

To combat the issue with the polylines, we included a refresh button. So when that button is clicked, the map is cleared and all of the current markers and polylines are reset.

However, that didn't solve the issue with the old passenger markers, so instead, we took that idea and adapted it slightly. We created a handler to call the function that added the passenger's markers every 10 min and within that function, it checked for if the passenger was picked up by comparing the driver's current latitude and longitude to the passenger's pick up coordinates and if close in distance, we set the value of the child 'isPickedUp' within the passenger's child of the trip's information of that unique trip to true. So if that value is true, a marker isn't added.

## **4.2 Data changes not showing in real time**

### **Problem**

We encountered a problem within our recycler view, where the the adapter was not being refreshed automatically whenever there was a change in the one of the list of trip or message objects. The content shown was not being updated unless the user had clicked back into the activity, meaning they were not being updated in real time, despite using the real time database.

### **Resolution**

After researching more into the Firebase Real time database listeners, we found a resolution to this issue. We were required to use an `addValueEventListener`, rather than `addListenerForSingleValueEvent`. This is due to the fact that the `addValueEventListener` keeps listening for the database reference it's attached to, whereas with the `addListenerForSingleValueEvent` only executes when there is a data change and stops after executing the method once.

## **4.3 Google Maps and Directions API**

### **Problem**

When testing the app on earlier devices, lower than 26, the google map was not loading. We originally believed that we had implemented it incorrectly. However, when we testing the app on a real android device and virtual devices with an API level of 26 and higher, the map activity loaded perfectly. We believe that this may have been an issue with Google Play services not being updated to the latest version. Also, with Google Directions API, we encountered issues when first attempting to implement it, it just didn't seem to be working at all.

### **Resolution**

In terms of the loading of the google maps, we found that the only solution to this issue we could think of was to raise the minimum API level to 26, in that way we could ensure strong compatibility across the devices using our app. We resolved the issue with the Directions API, through implementing an earlier version of this API and by also added a new and separate google api key, that was unrestricted. As our original key used for our places api and map activity was restricted to android apps.



## 5. Installation Guide

### Prerequisites

Java 9

Android Studio Version 3.3.2

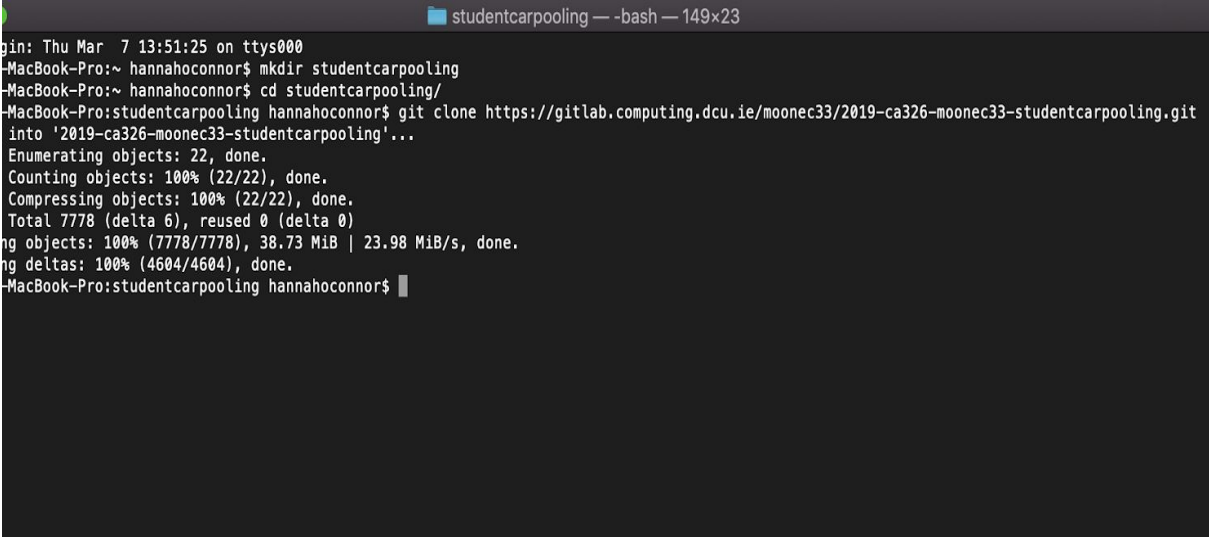
Android 8.0 (API Level  $\geq 26$ )

Internet Connection

### Setting up the project

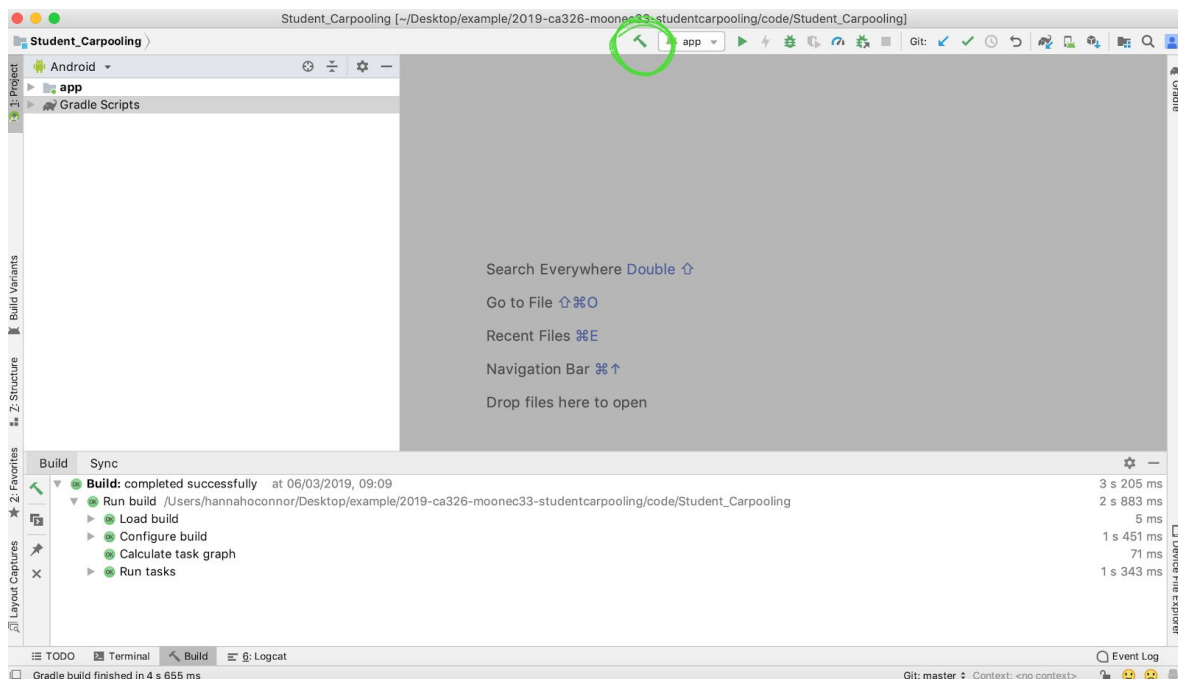
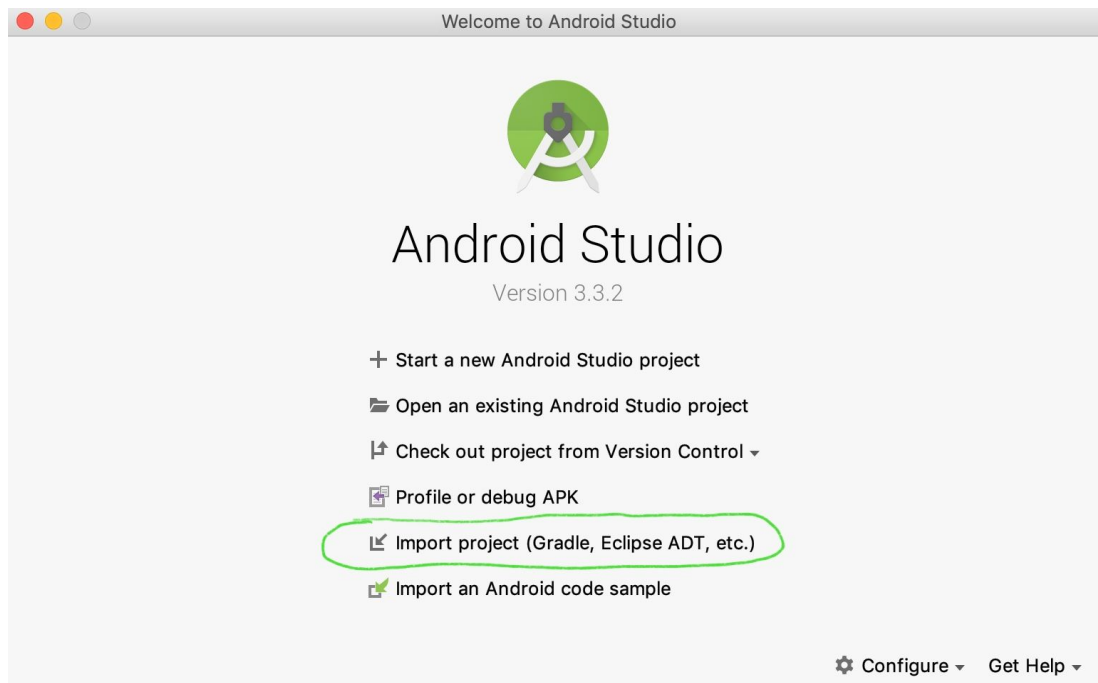
- Open your terminal
- Create the directory you wish to store the project
- Change to your newly created directory
- Enter: git clone

<https://gitlab.computing.dcu.ie/moonec33/2019-ca326-moonec33-studentcarpooling.git>



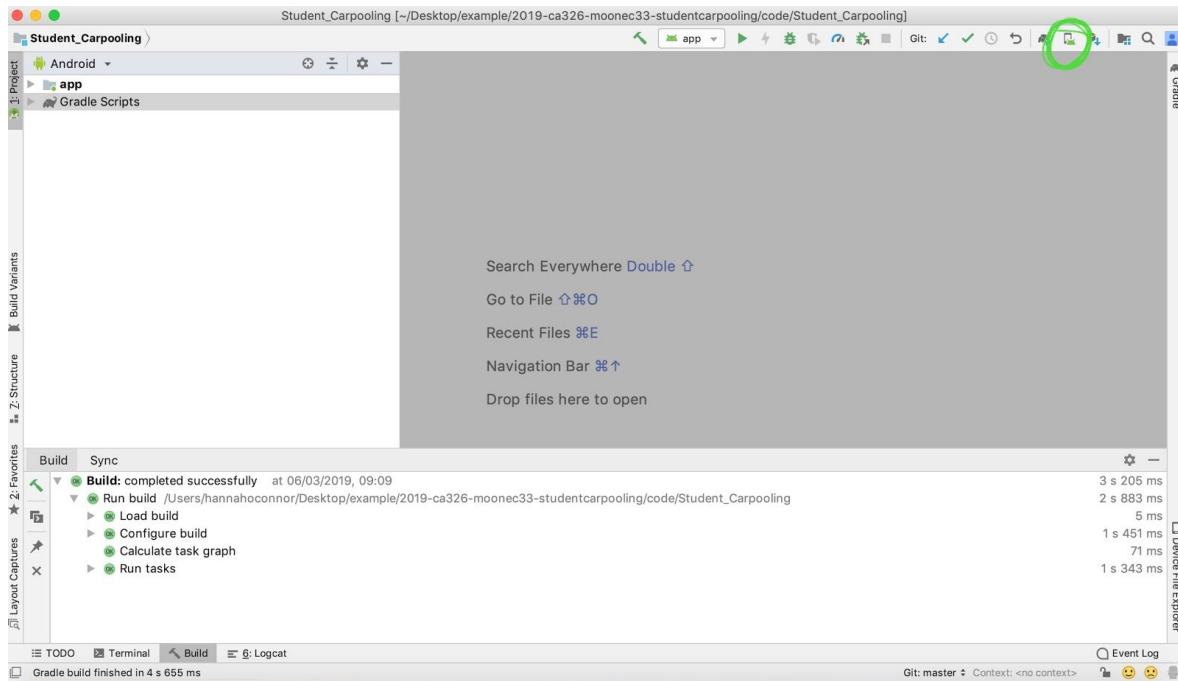
```
studentcarpooling — -bash — 149x23
gin: Thu Mar 7 13:51:25 on ttys000
MacBook-Pro:~ hannahconnor$ mkdir studentcarpooling
MacBook-Pro:~ hannahconnor$ cd studentcarpooling/
MacBook-Pro:studentcarpooling hannahconnor$ git clone https://gitlab.computing.dcu.ie/moonec33/2019-ca326-moonec33-studentcarpooling.git
into '2019-ca326-moonec33-studentcarpooling'...
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Compressing objects: 100% (22/22), done.
Total 7778 (delta 6), reused 0 (delta 0)
Receiving objects: 100% (7778/7778), 38.73 MiB | 23.98 MiB/s, done.
Resolving deltas: 100% (4604/4604), done.
MacBook-Pro:studentcarpooling hannahconnor$
```

- Open Android Studio
- Click Open Import Project and locate the folder in which you stored the project

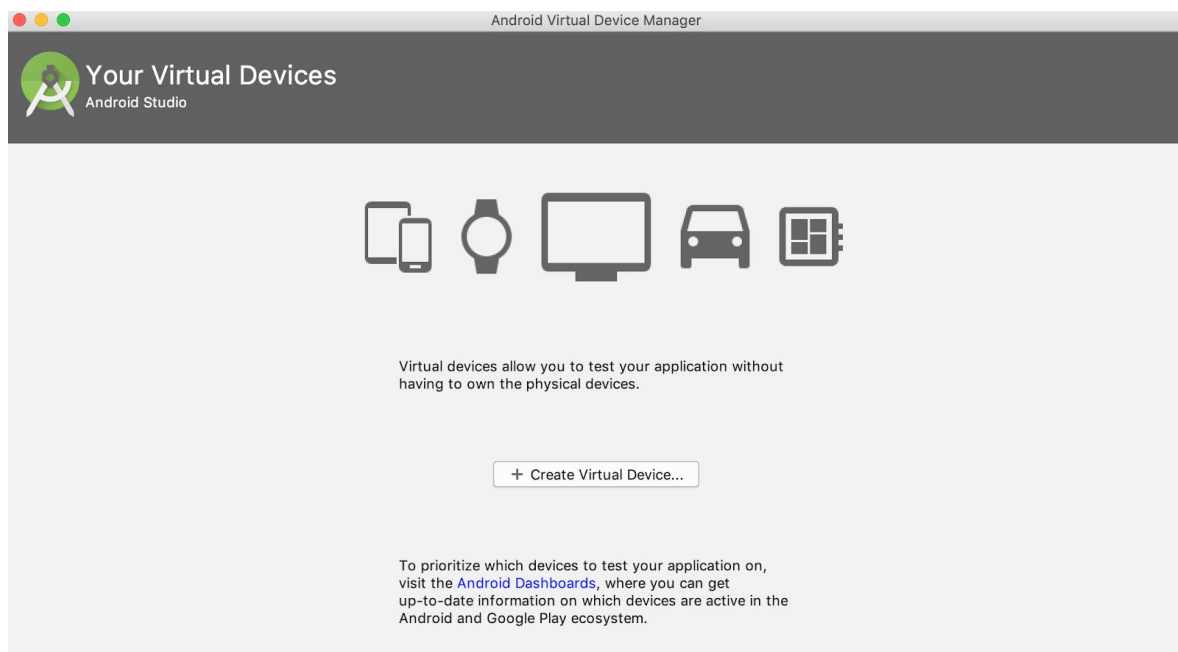


- Click on the build symbol as shown to 'Make Project'
- Wait for the code to successfully build

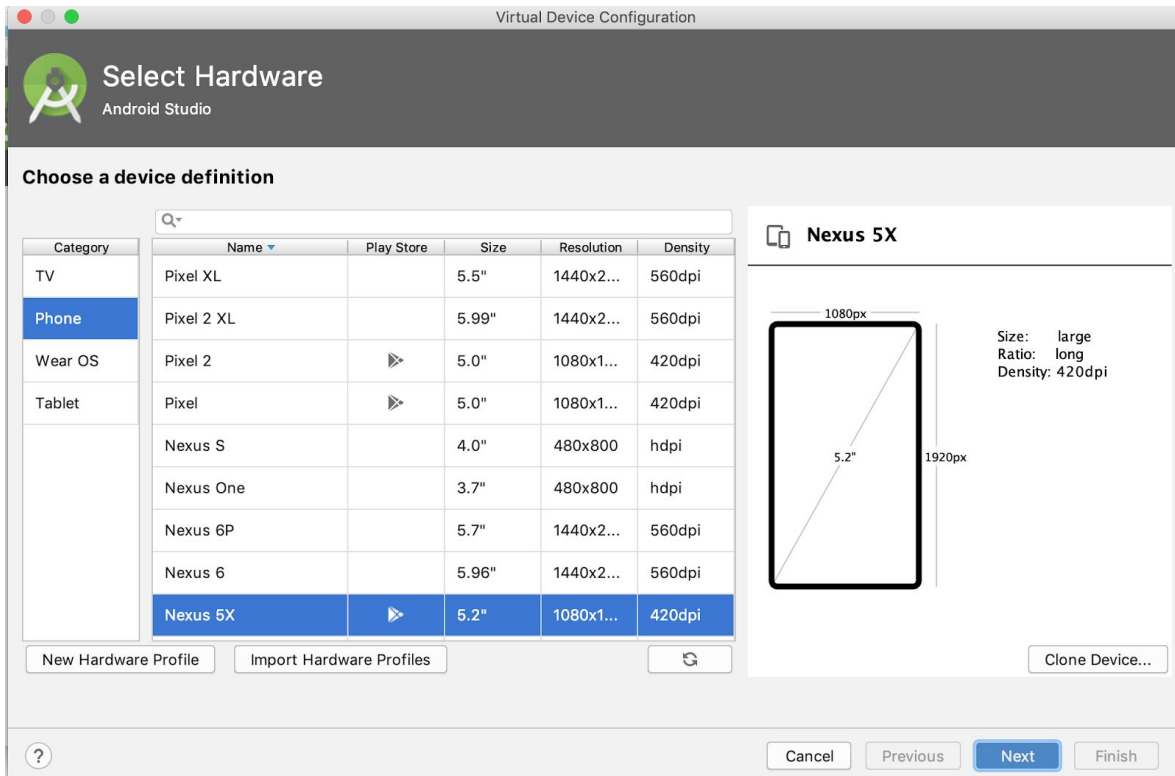
## Setting up the Emulator



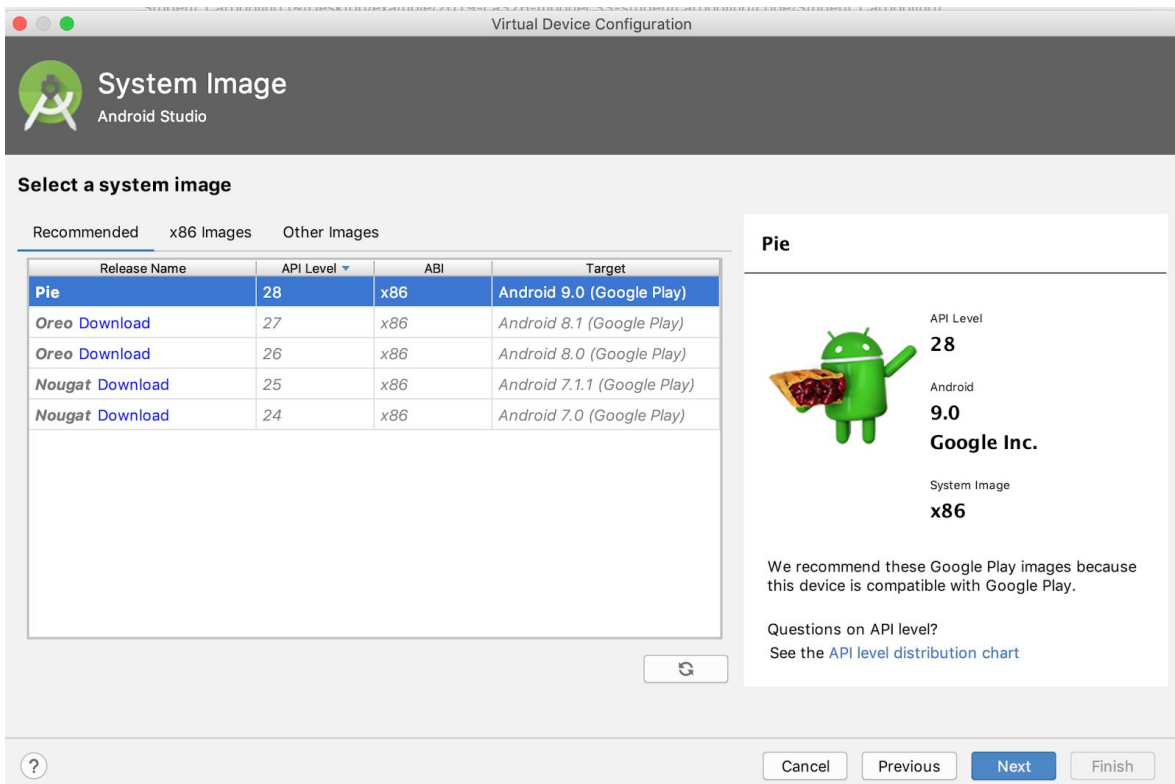
- Click on the above icon as shown



- Select 'Create Virtual Device'

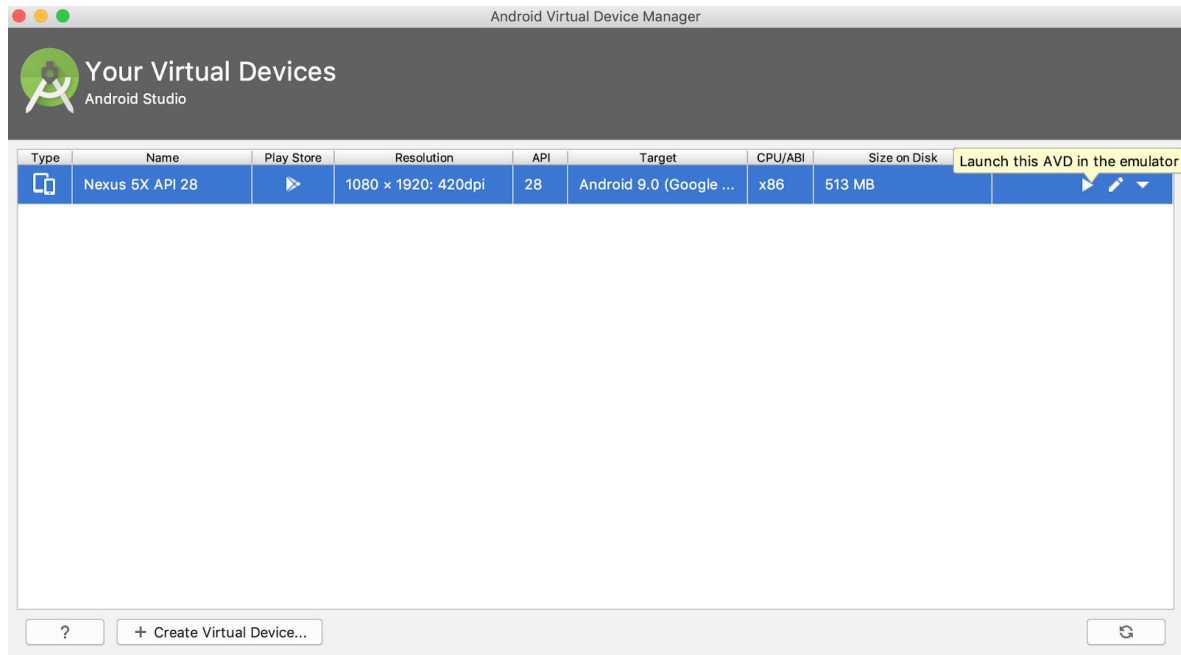


- Choose **Nexus 5X** as your device

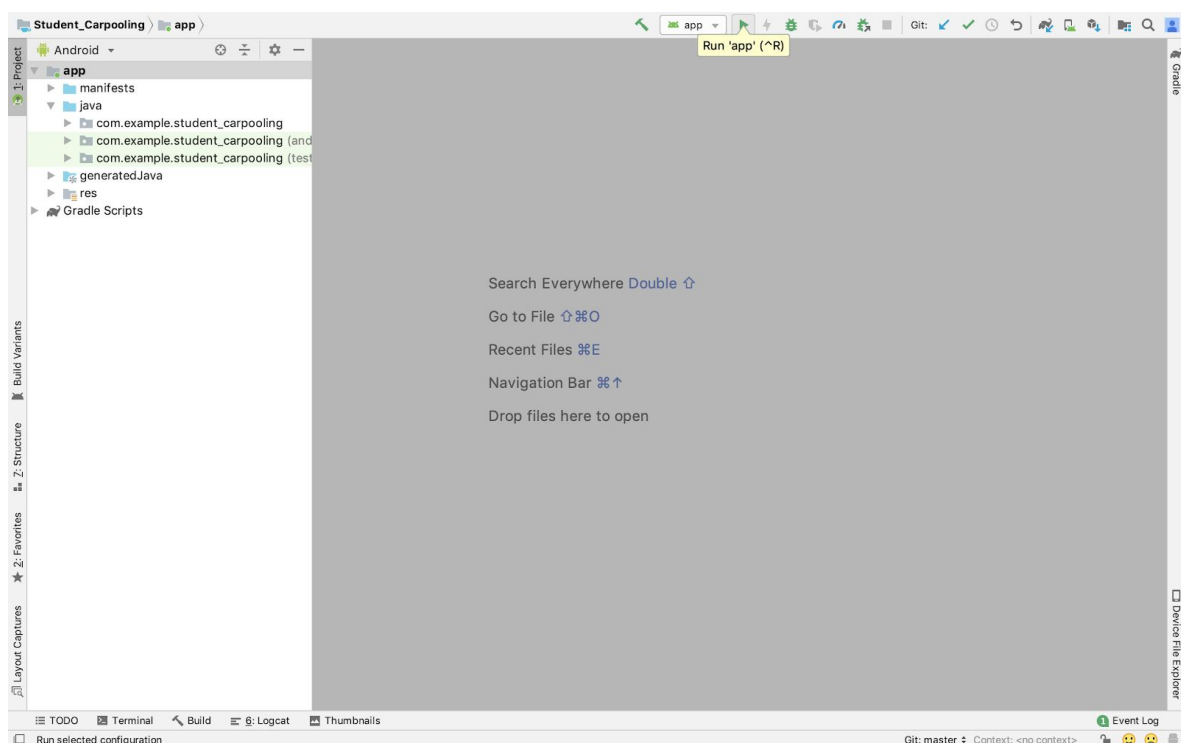


- Select API Level 28

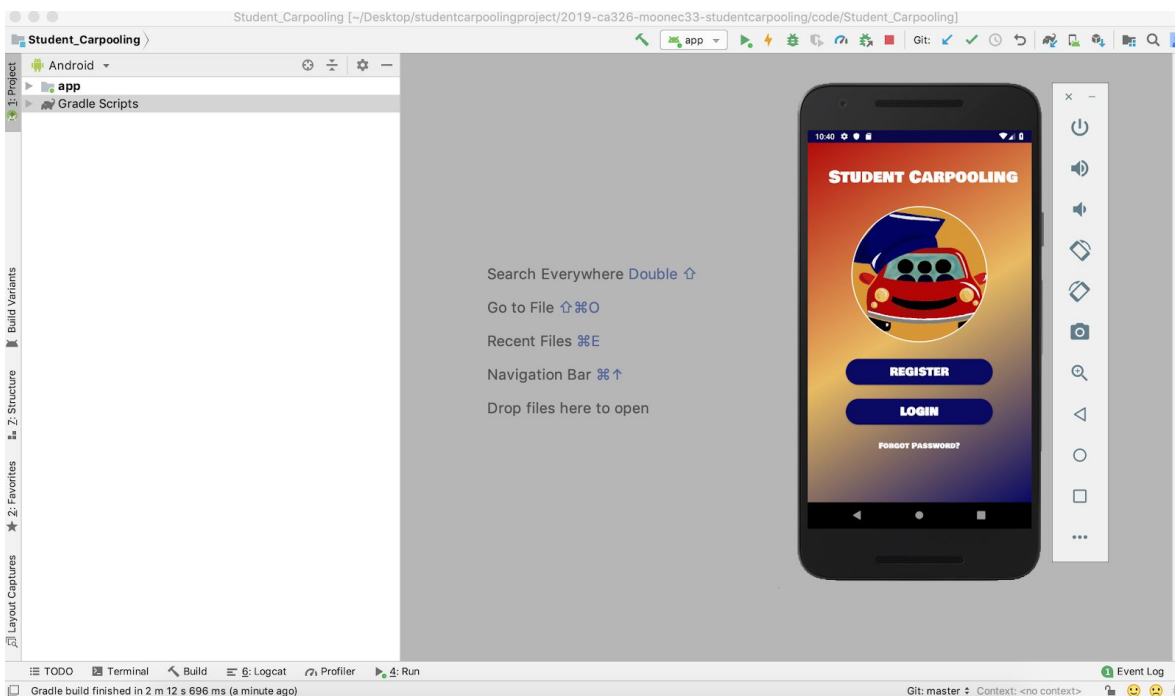
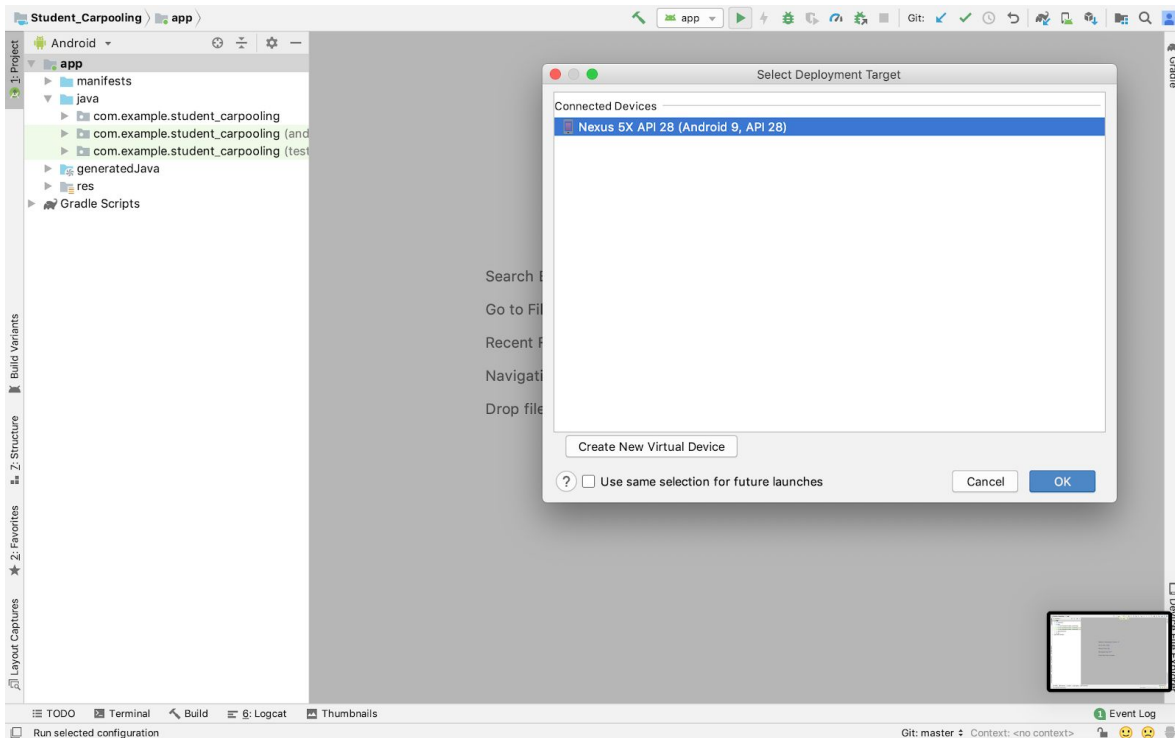
- Click next, then click finish
- Now, reopen Android Virtual Device Manager and highlight the one you wish to run and click on the play button which says 'Launch this AVD in the emulator'



- Once that has completed, click 'Run app' as shown below



- Select your newly created device, and press **OK**



- The device and the application will now be shown on screen.

