

Midhat Hasan Raghieb Abul Farah

Developing a Mobile Application for Ride Sharing Service

Helsinki Metropolia University of Applied Sciences

Information Technology

Thesis

May 2018

Author(s) Title	Midhat Hasan Raghil Abul Farah Developing a Mobile Application for Ride Sharing Service
Number of Pages Date	55 pages 11 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Patrick Ausderau, Principal Lecturer
<p>Motor vehicle owners, on average travelling, have around 2-3 seats empty and utilising them in exchange for money and time is where the ridesharing industry thrives. It would also be beneficial by reducing problems such as traffic, pollution and could create safer communities and improve the current transportation network industry.</p> <p>The primary goal is to understand how ridesharing works and build a mobile application prototype that helps passengers to find a driver from their current location to their destination.</p> <p>This thesis describes the development process of the ridesharing application starting by analysing similar application available, then using the Scrum methodology and brainstorming sessions getting a better understanding on how it works and gathering the requirements and specifications. Passengers can request a ride with its duration and fare and also receive notification once a driver accepts it. The application provides the shortest route for the driver to pick and drop passengers to their destination.</p> <p>The prototype has been developed and tested successfully on two operating systems, Android and iOS. It would, however, be beneficial for the application to include a payment and feedback system. Further studies about matching algorithm could also help to improve it.</p>	
Keywords	RideSharing, Mobile Development, React-native, Expo, Firebase, Google-Maps API

Contents

1	Introduction	1
2	Project Specification	2
2.1	Current state analysis and specification	2
2.2	Materials and Methods	3
3	Theoretical Background	5
3.1	Ridesharing	5
3.2	Technologies	7
3.2.1	React-Native	8
3.2.2	React-Native packages	8
3.2.3	Expo	9
3.2.4	Firebase	9
3.2.5	Google Maps	10
4	Implementation	11
4.1	Bootstrap	11
4.2	Screens	13
4.3	Navigation	19
4.4	Authentication	21
4.4.1	Registration	21
4.4.2	Log in	24
4.4.3	Log out	25
4.5	MapView	25
4.6	Passenger	29
4.6.1	Ride request	29
4.6.2	Ride cancellation	33
4.6.3	Ride confirmation	34
4.7	Driver	35
4.7.1	Ride response	35
4.7.2	Ride routes	37
5	Results	43
5.1	Passengers ride request	43
5.2	Passengers ride cancellation	44

5.3	Passengers ride confirmation	45
5.4	Drivers ride response	46
5.5	Drivers ride routes	47
6	Discussion	50
6.1	Database	50
6.2	User Interface state	50
6.3	Authentication	51
6.4	Passenger	51
6.5	Driver	53
7	Conclusion	54
	References	55

1 Introduction

In today's era, the majority of the people living in large cities have access to public transportation which is not always optimal and affording a personalised motor vehicle can be challenging. On the other hand, Forbes magazine estimates that by the year 2020, the vehicle manufacturing industry will assemble more cars which also includes 10 million self-driving cars [1]. In times of a developing economy, the future dedicates towards a sharing economy which is a peer-to-peer (P2P) based online-platform service that allowing the access and utilisation to goods and services supported by the community [2].

On average travelling, motor vehicle owners have around 2-3 seats empty and utilising them in exchange for money and time is where the ridesharing industry thrives. It not only benefits the vehicle owner but also reduces problems such as traffic congestion, environmental pollution and creating safer communities by improving the transportation network industry [3]. Ridesharing is the topic in which the general public has recently increased its interest and is searching for convenient ways of utilising its services in complement of public transportation.

The primary goal of this thesis is to understand how ridesharing services work and build a mobile application prototype upon the knowledge gathered. The application should be easy to use and reach the quality level necessary to be published on the app stores. The concept of Ridesharing is to allow passengers to request for a ride and drivers to accept passengers ride request. To fulfil these requirements, the ride requests and responses are collected, processed and presented to the end user on a map. The application also needs to be cross-platforms available, such as Android and iPhone Operating System (iOS), for reaching a maximum of users.

2 Project Specification

2.1 Current state analysis and specification

Numerous ridesharing related applications such as Uber [4] and Lyft [5] were studied to get a better understanding of how ridesharing works and their services allowing passengers to share a ride with other passengers for a lower cost [6]. Open source project such as icare [7] and carpooling [8] existed, however developing and implementing the cross-platform application seemed beneficial in comparison to them. In addition, there is the interest in learning how to develop such application from scratch.

The specification for building the application were gathered and the following features below were required to build an efficient mobile app prototype:

- Cross-platform mobile application
- Authentication for different user roles
- Navigation between various screens
- MapView for passengers and drivers
- Passenger functionalities
- Driver functionalities

Based on the analysed features above user stories were created that are short, simple descriptions of the features written from the perspective of an end user. The Connextra user story template in figure 1 was used to create the user stories.

```
As a <role> I can <capability>, so that <receive benefit>
```

Figure 1: Connextra user story template (Reprinted from MISHKIN BERTEIG [9])

The following user stories below describes how various roles behave in the application:

- As a user, I can sign up with a driver role, so that I can accept passengers ride requests
- As a user, I can sign up with a passenger role, so that I can request for a ride
- As a user, I can log in with a driver account, so that I can accept passengers ride requests.
- As a user, I can log in with a passenger account, so that I can request for a ride
- As a driver, I can view my current location and all passengers ride request as markers on the map, so that I can pick nearby passengers
- As a driver, I can accept passengers ride requests so that I can make money.
- As a driver, I can view direction routes before picking up passengers from their origin, so that I can navigate and pick them up efficiently
- As a driver, I can view direction routes after picking up passengers to their destination, so that I can navigate efficiently to right places
- As a driver, I can receive payment when successfully dropping off a passenger to their destination, so that I get reward for offering a ride.
- As a passenger, I can view other passengers ride request from my current location, so that I can know who to share the ride with.
- As a passenger, I can request for a ride entering my location and destination addresses, so that I can get a ride
- As a passenger, I can view the journey fare and duration before making a ride request, so that I can decide if I request it.
- As a passenger, I can view the ride request as a marker on the map, so that I can get feedback that the ride request has been made.
- As a passenger, I can cancel the ride request, so that I can enter a new ride request or decide not to request the ride.
- As a passenger, I can make payment when reaching the destination, so that I can remunerate the driver for the service

2.2 Materials and Methods

The thesis followed the Scrum development methodology with weekly sprints to make the development fit the short time frame of the project. Wall Post-it notes were also used on Kanban board to ensure the physical presence of the priorities and proper levels of engagement with tasks. The implementations of the Project were performed on Balsamiq for mocking, Visual Studio Code Editor for coding and Git for version control.

A brainstorming was carried out to generate ideas for the application. The tools used were pen and paper, whiteboard, Post-its and a note-taking app Wunderlist. The Pinterest website was used to be inspired and create new ideas, and each session was documented and would typically last an hour. Following prototyping was carried out using different mediums such as A5-papers, mock tool Balsamiq and whiteboard to sketch the prototype of the application.

Before implementing the prototype, several works were studied and gathered from Google Scholars, Medium articles and YouTube Lectures. The studies provided the necessary information of how ridesharing works, the design of the system and the best practices required when developing a cross-platform mobile application. The Medium articles and YouTube lectures seemed relevant to this project, and new frameworks and tools were learned to help speed up the process.

3 Theoretical Background

3.1 Ridesharing

Ridesharing is not a brand-new concept, and it was discovered between 1914 -1918 known as the Jitney craze that allowed individuals to figure out an innovative and profitable way to use their automobiles. In 1908, Henry Ford mass-produced the Model-T automobile, which was relatively affordable to an average successful person. However, by 1915 it reached a rapid downfall as there were concerns over the safety and liability of the passengers. In 1942, the U.S government needed to figure out a solution to save rubbers, and the most reasonable option was arranging ridesharing to workplaces. In 1973, the oil crisis caused an impact on the market price of gasoline [10;11]. By 1990s, Kowshik predicted that ridesharing would exist with a better dynamic system which includes a better matching technique [12].

Today's ridesharing revolution was made possible due to the technological advances such as:

1. **Global Positioning System(GPS)** navigation used for creating the best driving route, navigation assistance and arrangement for ridesharing.
2. **Mobile phones** for allowing the travellers to efficiently request a ride with itineraries from their current location
3. **Social network** for establishing trust and accountability between the drivers and passengers creating a safer community
4. An **electronic payment system** for allowing customers to make payments to the driver as a reward for their service.

RideSharing is a means of transportation that allows multiple travellers to journey to the same location [13]. It is a convenient way for travellers to commute and beneficial for the environment such as reducing traffic congestion, fuel costs, time costs and the stress of driving.

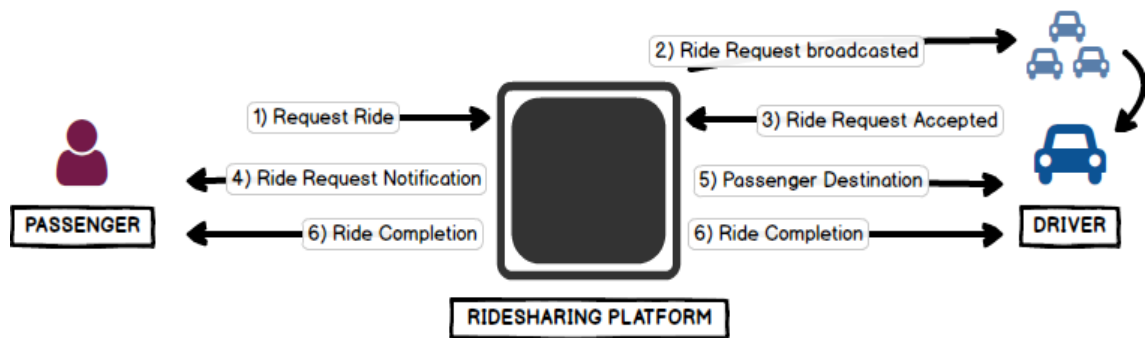


Figure 2: Ridesharing workflow between passenger and driver

The illustration presented in figure 2 describes the operation of how ridesharing platform works between a driver and a passenger. Each of the following operations below provides the detail explanation:

1. Passenger requests for a ride providing its origin and destination address.
2. Nearby drivers are alerted of the passenger's ride request. The ridesharing platform selects the driver based on the distance to the passenger and rating. The driver is provided with 10-15 seconds to accept the passenger ride request, or else the next driver in the queue will be assigned.
3. The driver accepts the passenger ride request and receives the passenger's origin.
4. The passenger receives a notification that the ride request is accepted and also gets the driver's details and the estimation time of arrival.
5. When the driver arrives at passenger's origin, the driver receives passenger's destination address and starts the journey.
6. Once the journey completes, the ridesharing platform automatically charges the passenger and pays the driver.

The ridesharing platforms, such as Uber and Lyft exists providing an app that allows drivers and passengers to connect for a ride journey at an agreed price. The passenger would use the app to request a ride at a particular time and location, and the app in return provides an instruction including the journey fare, the location of the driver and the estimation time of arrival. The application also keeps the personal information of the passenger safe without disclosing it to the driver. The GPS on the app provides the best optimal route for the driver to pick up the passenger and for the passenger to request a ride. It included benefits for the passengers such as the rating system for the quality of the journey, an efficient payment system such as paying using a credit card and a reward system.

The number of active drivers of the ridesharing platform Uber rose from a base of zero in mid-2012 to 160,000 by the end of 2014. By 2017 the total number of drivers have passed 1.5 million [14] of which the percentage of drivers: 14% being women, 19% aged under 30, 25% over age 50 and 71% have living dependents [15]. However, the ridesharing platforms provide services in exchange for a commission of the fares drivers earn ranging between 0% to 25% as advertised. Despite the fact of offering low ride fares for its customers, it does not benefit much from the drivers. Christian Perea provided an infographic displaying how the ridesharing platforms benefited from the decreased costs based on the random sample collection of 37 actual UBERX trips. Cities such as Chicago collected a higher commission from the driver about 54% for a minimum fare of \$4.20 of which the driver gets \$2.25 whereas in Houston collected 45% commission for a minimum fare of \$5.00 of which the driver receives \$2.29. [16]

Nevertheless, the study conducted by Massachusetts Institute of Technology (MIT), it explained how ridesharing impacted the economy with benefiting both the drivers and passengers, reducing traffic congestion and decreasing the public health hazard. The study shows congestion reduction by a factor of 3 for helping the same number of passengers compared to current ridesharing platforms serving a single passenger. The drivers would have shorter work shifts by the reduction in the taken trips for the same amount of fare and passengers would save time by waiting less [17]. Ian Hatha-way et al. stated how ridesharing has emerged as a critical part of the monetary system in phases of employment in which a majority of the employees are either part-timers or self-employed, especially in large metropolitan regions [18]. Finally, the valuations of leading companies such as Uber at \$68B, Didi Chuxing at \$34, Lyft at \$5.5B, Ola at \$3.5B, Grab at \$3B and Careem at \$1B determines that it has an enormous market potential all around the world [19].

3.2 Technologies

This section provides the details of the selected tools and libraries, which ensures that users have the best possible experience and developers are productive when developing and maintaining the code.

3.2.1 React-Native

React Native [20] is a framework for building native mobile applications using JavaScript and React. React [21] is a JavaScript library for building user interfaces. On March 2015 React Native was released as an open source project on GitHub and announced during the Facebooks F8 conference [22]. React Native utilises JavaScript and applies the same design principles used in React library, allowing to write rich declarative user interface mobile components.

Applications written with React Native are the real mobile apps which use the same fundamental UI building blocks of an Android or an iOS app. There is no time wasted on the compilation as the app reloads instantly with a change in state. The Hot Reloading option, helps in retaining the old application state while the new code is applied. Inserting native code written in Objective-C, Java or Swift for specific optimisation of the application can be included easily [23]. Moreover, React Native utilises a JavaScriptCore for code execution. JavaScriptCore is an optimising virtual machine which renders JavaScript into native components [24].

3.2.2 React-Native packages

React-native-elements [25] is an open source library that provides cross-platform user interface toolkit such as Button, Card and much more which works on Android and iOS. The library is entirely built using JavaScript and is easily supported using Expo. Moreover, the components can be customizable easily.

React Navigation [26] is an open source library used for routing and navigation of React-native apps. It provides an easy to use built-in navigators and navigation components which work seamlessly across platforms such as Android and iOS giving it platform specific look and feel with smooth animations and gestures. Moreover, it is highly customizable using JavaScript. Finally, it is extensible at every layer providing the ability to write custom navigators.

3.2.3 Expo

Expo is a set of tools, libraries and services that help building native mobile application such as Android and iOS project using JavaScript. The application written in Expo are React-Native apps which contain the Expo Software Development Kit (SDK). The SDK is native, and the library provides access to the device's system functionality such as camera, contacts, local storage. That means that the use of XCode or Android Studio or writing any native code is required. Since the application is built entirely in JavaScript, it can be ported and run on any native environment containing the Expo SDK. Moreover, it also provides User Interface (UI) components which handle a variety of use-cases that are not available in the React Native core such as the icon, blur view. Finally, the Expo SDK provides access to services which are quite challenging to manage by majority app such as managing Assets, Push notification and building native binaries that are ready to be deployed to the app store [27].

3.2.4 Firebase

Firebase [28] is a cloud service for mobile and web application development platform that provides the necessary tools needed to develop a high-quality application, grow user base, monetise on the business and focus on the users. It also provides SDK for all major client platforms such as Android, iOS, Web, C++, and Unity.

The Firebase Realtime Database [29] is used to store and sync data with Non-Relation Structured Query Language (NoSQL) cloud database. The data stored is synchronised in Realtime across all devices and is also available when the application is offline. Implementation of the database is effortless as pushing a JavaScript Object Notation (JSON) representation of the information to the server, without the need of setting up a backend Structured Query Language (SQL) structure. The JSON data pushed are sorted by a unique ID which is automatically generated by Firebase.

The Firebase Authentication [30] provides a drop-in authentication solution with best practices and ready-made library for authenticating users to the application, maximising the sign-in and sign-up conversion. It supports all sorts of authentication such as standard authentication (email and password), phone number and also Identity providers such as Google, Facebook, and Twitter. The Firebase Authentication provides easy integration with other Firebase services such as Realtime Database and Cloud Func-

tions and leverages industry standards like Open Authorization (OAuth) 2.0 and OpenID Connect. Finally, the Firebase console aids in configuring the Firebase setup for application and hosts it on a container at Google Cloud Provider cloud infrastructure. Its free account currently allows creating maximum of five projects, and further projects would require a project quota increase.

3.2.5 Google Maps

Google Maps [31] is a mapping service developed by Google that provides everything required to build location-based apps. The Google Maps Application Programming Interface (API) [32] provides numerous integration API services for platforms such as Android, iOS and web services. The following Google Maps API services are enabled: Google Places API Web Service, Google Maps Directions API and Google Maps Distance Matrix API.

Google Places API [33] provides contextual information about user current location, valuable information about local businesses and point of interest and an autocomplete search feature for a type-ahead location-based prediction that are represented either in Extensible Markup Language (XML) or JSON.

Google Maps Directions API [34] provides directions for modes of transport such as driving, walking, cycling or transit and waypoints route up to 23 locations and an estimation travel time based on historical and current traffic conditions that are represented either in XML or JSON.

Google Maps Distance Matrix API [35] provides the travel distance and time for a matrix of origins and destination based on the given start and end points that are represented either in XML or JSON.

4 Implementation

4.1 Bootstrap

In this subsection, will be discussing about bootstrapping the application using the Exponent Development Environment (XDE), Firebase and Google Maps API.

The Expo XDE tool provided by Expo is used for setting up a react-native project and the listing 1 provides the required dependencies which are installed using a dependency manager such as Node Package Manager (Npm).

```
{
  "main": "node_modules/expo/AppEntry.js",
  "private": true,
  "dependencies": {
    "expo": "^25.0.0",
    "firebase": "^4.11.0",
    "react": "16.2.0",
    "react-native": "https://github.com/expo/react-native/archive/sdk-25.0.0.tar.gz",
    "react-native-elements": "^0.19.0",
    "react-native-google-places-autocomplete": "^1.3.6",
    "react-native-modal": "^5.3.0",
    "react-navigation": "^1.5.2"
  },
  "devDependencies": {
    "babel-eslint": "^8.2.2",
    "eslint": "^4.18.2",
    "eslint-config-airbnb": "^16.1.0",
    "eslint-plugin-flowtype": "^2.46.1",
    "eslint-plugin-import": "^2.9.0",
    "eslint-plugin-jsx-a11y": "^6.0.3",
    "eslint-plugin-react": "^7.7.0",
    "flow-bin": "^0.67.1"
  }
}
```

Listing 1: Project main and development dependencies required for the ridesharing application

Firebase is configured using the Firebase console, which aids in creating a firebase project and also view different firebase services that are enabled. Upon successful creation of the firebase application, three options are provided to choose from which are iOS, Android, and Web app. Typically, an adapter such as React-native-Firebase [36] is used to integrate Firebase into react-native efficiently. However starting from version 3.1 Firebase provides a JavaScript SDK, which has almost full support for React-native and including it is easy.

This application utilises the Firebase JavaScript SDK, and the configuration code from the Firebase console [37] is copied and used in a Firebase singleton file which uses the firebase as a dependency installed from the required dependencies in listing 1. The singleton is a software design pattern that ensures that a single instance of the object is created and can be accessed concurrently by individual components of the app [38]. The Firebase singleton file in the listing 2, initialises the application with firebase configuration and exports instances of firebase, database, and auth.

```
import firebase from 'firebase';

const config = {
  apiKey: 'AIzaSyAVAGPigfhikSdqFcbu3j0_j4TfCxW16yg',
  authDomain: 'ride-sharing-app-1516150146817.firebaseio.com',
  databaseURL: 'https://ride-sharing-app-1516150146817.firebaseio.com',
  projectId: 'ride-sharing-app-1516150146817',
  storageBucket: '',
  messagingSenderId: '163416353579'
};

firebase.initializeApp(config);

export default firebase;

export const database = firebase.database();
export const auth = firebase.auth();
export const googleAuthProvider = new firebase.auth.GoogleAuthProvider();
```

Listing 2: Firebase singleton file exposing database, auth and google auth provider for the ridesharing application

The Firebase Realtime Database Rules in listing 3 using a JavaScript-like syntax, is used to configure users read and write access rights operations to the database. This is achieved by checking if the user id obtained using Firebase Authentication matches with the user id stored on the database. Database Rules can also be used to define data validation rules and database indexes to support ordering and querying of data.[39]


```

{
  "rules": {
    "drivers": {
      "$uid": {
        ".read": "$uid === auth.uid",
        ".write": "$uid === auth.uid"
      }
    },
    "passengers": {
      "$uid": {
        ".read": "$uid === auth.uid",
        ".write": "$uid === auth.uid"
      }
    }
  }
}

```

Listing 3: Database security rule for passengers and drivers

Finally, the Google Maps is configured using the Google API Console [40], which aids with creating a Google Maps project, providing with an API KEY which is an identification for the developer and its usage of the service when calling the API and a place to enable numerous Google Map services. The following Google Map services are enabled such as Google Places API Web Service, Google Maps Directions API and Google Maps Distance Matrix API for the ridesharing application.

4.2 Screens

Screens are the building blocks of the application which presents the features and functionalities separated to their specific task. The React-native packages such as React-Native elements is a UI toolkit that provides custom premade components, and the Expo library offers the MapView component.

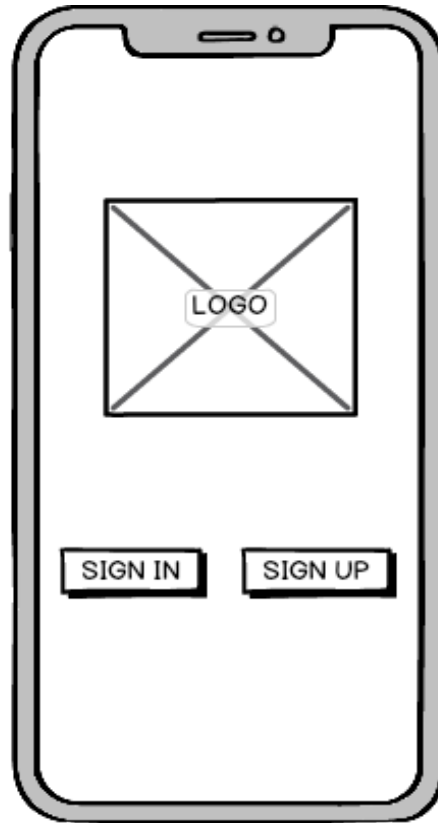
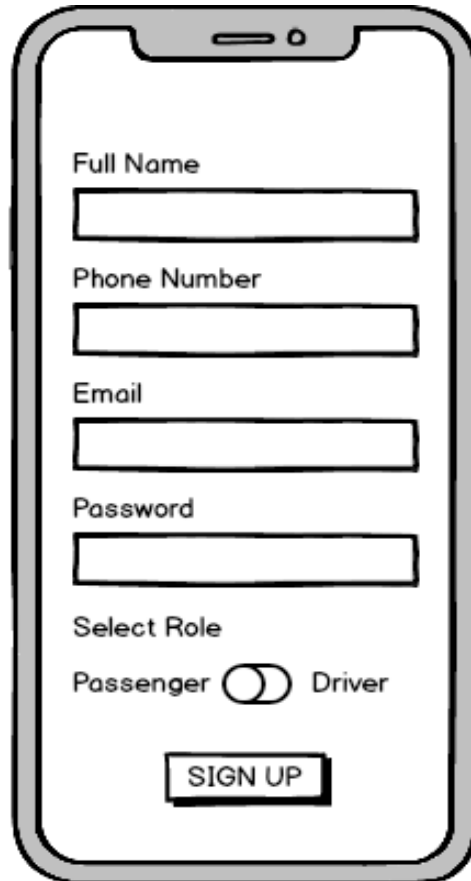


Figure 3: The Welcome Screen displaying an image logo and two buttons

The Welcome Screen in figure 3 will be the initial view that loads when the application starts. It consists of an image logo and two buttons for Signing in and Signing Up. The SignIn and SignUp button on press will navigate to the SignIn or SignUp Screen respectively.



The image shows a wireframe of a mobile application's sign-up screen. It features a rounded rectangular frame representing the phone. Inside, the following elements are arranged vertically from top to bottom: a label 'Full Name' above a rectangular input field; a label 'Phone Number' above a rectangular input field; a label 'Email' above a rectangular input field; a label 'Password' above a rectangular input field; a label 'Select Role' above a toggle switch. The toggle switch consists of two circles, with the left one filled and the right one empty, positioned between the labels 'Passenger' and 'Driver'. At the bottom center is a rectangular button with the text 'SIGN UP' inside.

Figure 4: The SignUp Screen displaying four input fields, a toggle button and a button

The SignUp Screen in figure 4 is displayed when pressing the SignUp button on the Welcome Screen from figure 3. It consists of four input fields with labels, allowing users to fill in details such as full name, phone number, email, password and an option to select a role using the toggle button. By default, the passenger role is selected and toggling changes to driver role. The SignUp button on the SignUp Screen when pressed will successfully create a user account redirect to the appropriate Home Screen based on the selected role.

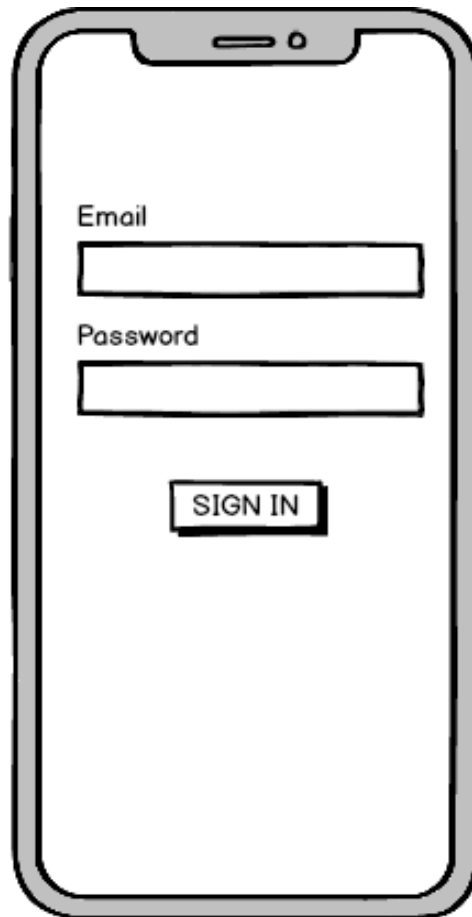


Figure 5: The SignIn Screen displaying two input fields and a button

The SignIn Screen in figure 5 is displayed when pressing the SignIn button on the Welcome Screen from figure 3. It consists of two input fields with labels allowing users to enter their authentication details such as email and password and a button to sign-in into the application. The SignIn button on the SignIn Screen when pressed will redirect to the Home Screen based on their selected role when registering from figure 4.

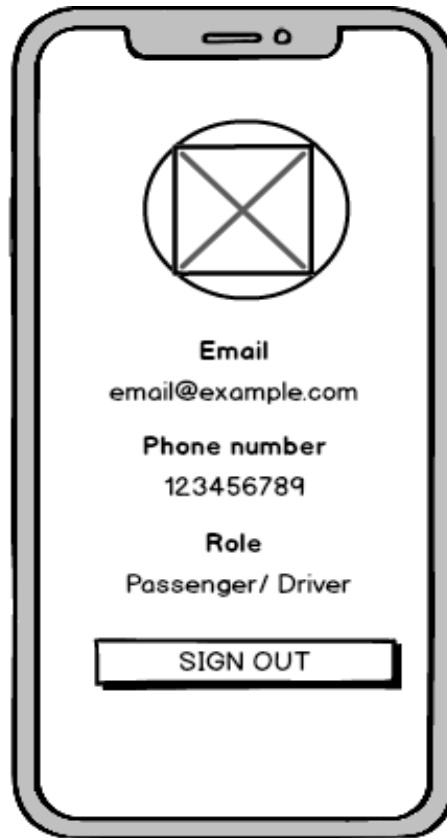


Figure 6: The Profile Screen displaying an image logo, three input fields and a Button

The Profile Screen in figure 6 is displayed when the user is authenticated successfully into the application either from the SignUp Screen in figure 4 or the SignIn Screen in figure 5. It consists of an image logo of the user, three input fields displaying user's personal information such as email, phone number, current role and a sign out button to log out from the application. The SignOut button on the Profile Screen when pressed will redirect to the Welcome Screen in figure 3.

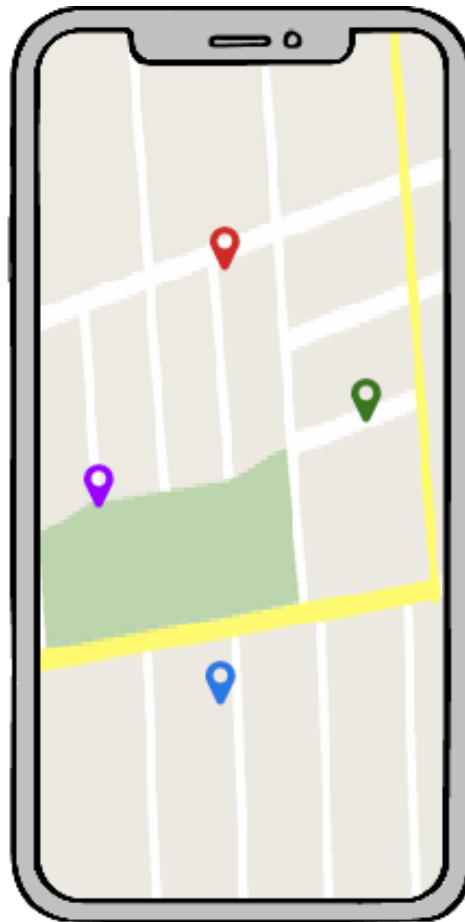


Figure 7: The Home Screen displaying different markers on the MapView based on user's roles

The Home Screen in figure 7 is displayed when the user is authenticated successfully into the application either from the SignIn Screen in figure 5 or the SignUp Screen in figure 4. It will consist of a MapView and MapView coordinates. The MapView will display the MapView coordinates as markers based on their selected roles during the registration in figure 4. The MapView coordinate makers are made identifiable using different images instead of the default icon on users. The car marker image will identify users with driver role, the passenger marker image will identify users with the passenger role and the default marker icon will locate current users requesting a ride on the MapView.

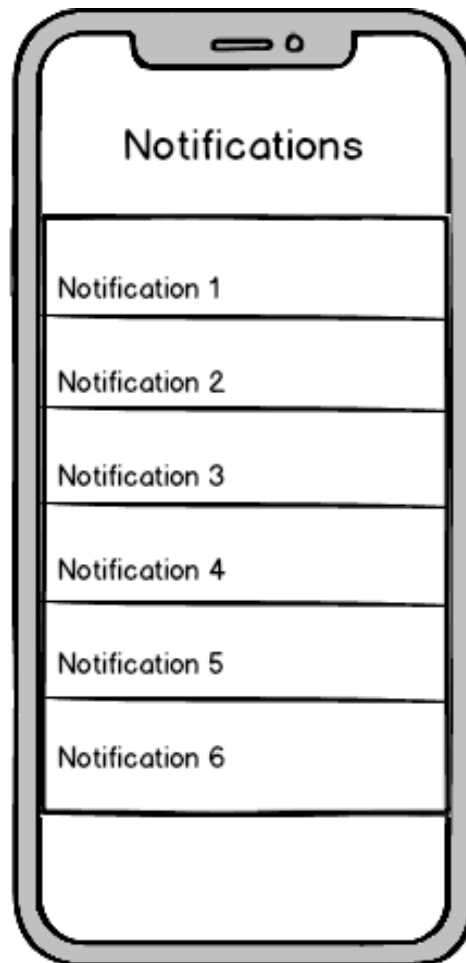


Figure 8 The Notification Screen displaying listview of notifications

The Notification Screen in figure 8 is displayed when the user's ride request is either being accepted or cancelled. It consists of a listview containing the notification details of user's ride request. It is also a useful way for notifying users about their ride status when the user has not enabled the push notification services.

4.3 Navigation

Navigation is a vital part of the application that allows users to switch between screens. The React Navigation library provides essentials components such as Tab and Stack navigations required for the application. The screens depicted in figures 2 to 7 will be separated into two different navigation states based on authentication which is the signed in and signed out state.

The Signed-Out state will be the composition of the following screens that are Welcome Screen, SignIn Screen and SignUp Screen onto a StackNavigator when users have not authenticated to the application. The StackNavigator is stack-based [41] navigation which is conceptually similar to how web browser handle navigation, in which the browser pushes and pops items from the navigation stack as the user interacts with it and results into the viewing of different screens. The addition of StackNavigator to the application in listing 4 is for the signed-out state of screens of unauthenticated users.

```
export const SignedOut = StackNavigator({
  Main: { screen: Main, navigationOptions: { ... } },
  SignIn: { screen: SignIn, navigationOptions: { ... } },
  SignUp: { screen: SignUp, navigationOptions: { ... } }
});
```

Listing 4: SignedOut navigation state using StackNavigator

The Signed-In state will be the composition of the following screens that are Profile Screen, Home Screen and Notifications Screen onto a TabNavigator when users have authenticated to the application. The TabNavigator is tab-based [42] navigation, which is one of the most common styles of navigation in the mobile application that separates different screens into tabs and results into the viewing a specific screen when pressing the particular tab button. This navigation applies to authenticated users of the app, and the listing 5 explains adding it to the application.

```
export const SignedIn = TabNavigator({
  Profile: { screen: Profile, navigationOptions: { .... } },
  Home: { screen: Home, navigationOptions: { ..... } },
  Notifications: { screen: Notifications, navigationOptions: { ..... } },
},
{ initialRouteName: 'Home' },
{ tabBarOptions: { style: { ... } } });
```

Listing 5: SignedIn navigation state using TabNavigator

When the user authenticates to the application based on the ordering of the screens, the TabNavigator displays the first screen as the initial screen. However, selecting which screen to display first can be set using the `initialRouteName` property provided. There is also an option for adding custom icons to the tabs using the `tabBarOp-`

tions property. The icons can either be created by creating custom icons or using a premade icons library provided by Expo Vector Icons that includes various vector icons such as FontAwesome, Material Icons and Entypo.

4.4 Authentication

Authentication creates identification for the users of the application. The database security rule for passengers and drivers configured in listing 3 ensures that correct passengers can request a ride and the right drivers can accept the ride request. It also provides users with a personalised profile page including ride histories, discussed in section 4.6 and 4.7. For simplicity of the application, the Email and password based authentication provided by the Firebase Auth is used and can be enabled easily from the Firebase console under the sign-in method tab. Authenticating successfully to the application, stores the user details separately based on the user's chosen role using Firebase Realtime database. Nevertheless, the appropriate screen will be navigated based on successfully updating of the user details. The following process takes place within the authentication flow that are Registration, Login and Logout.

4.4.1 Registration

The users of the application enter the required fields, which are `fullName`, `email`, `phoneNumber` and the selected `role` which are passenger or driver role into the form presented on the Sign-Up screen from figure 4. The Firebase Auth handles the user's email address and password in listing 6, and on successful handling, it creates a new user in the Authentication table.

```
auth.createUserWithEmailAndPassword(email, password)
```

Listing 6: Firebase Auth creating a user with email and password in the Authentication table

On successful registration, Firebase returns a promise, which is an object that may produce a single value over time in the future. The value can either be a resolved value or an unresolved value due to a network error for example. The Firebase promise object includes user's metadata, `email`, `emailVerified`, `metadata` and the authorisation token as `uid` in figure 9.

```

{
  .....,
  email: "test7@test.com"
  emailVerified: false,
  metadata: {a:..., b: ..., lastSignInTime: ...}
  w:"ride-sharing-app-1516150146817.firebaseio.com",
  uid:"f1VRHDWKEgS89UnSmo10sN8Fi4C3"
}

```

Figure 9: Response output from firebase when registration is successful

The Firebase auth instance of the `currentUser` property contains the `updateProfile` property which is used for updating basic profile information such as the user's display name and a profile photo URL and for this application case the display name is updated using the user object in listing 7.

```

auth.currentUser.updateProfile({ displayName: fullName });

```

Listing 7: Firebase Auth is updating current users profile display name

When the user's display name is updated successfully, the user's details entered during the registration process from figure 4 are stored based on the roles selected in the using the Firebase Realtime Database. The user's details are stored under its `currentUser` id of either the `drivers` or `passenger` database references.

```

database
  .ref('drivers')
  .child(userId)
  .set({ fullName, email, phoneNumber, role });

```

Listing 8: Storing user details under driver's role with current user id

Selecting the `driver` role stores the user details under the `currentUser` id of the `driver` database reference in listing 8 and selecting the `passenger` role stores the user details under the `currentUser` of the `passenger` database reference in listing 9.

```

database
  .ref('passengers')
  .child(userId)
  .set({ fullName, email, phoneNumber, role });

```

Listing 9: Storing user details under passenger's role with current user id

The authorisation token from the Firebase response object in figure 9 helps in identifying the user and will be used for persisting. With persistence, the user would not require authenticating when using the application next time. Persisting the user is achieved using the `AsyncStorage` which is provided by the React-native library.

```

AsyncStorage.setItem('userData', JSON.stringify(authData));

```

Listing 10: Storing firebase response with Async storage

The `AsyncStorage` sets the firebase response to a key called `userData` in listing 10. When the authorisation token is successfully set, it will navigate to the `SignedIn` state in listing 11 using the `navigate` property of React Navigation, that provides navigating to the desired screen which is the Home Screen from listing 5.

```

this.props.navigation.navigate('SignedIn');

```

Listing 11: Navigation to Main Screen upon successful registration or log in

When an unsuccessful registration occurs, Firebase throws a validation error that is caught and updated to an error state in listing 12. For example, when a user enters a similar email address stored in the Authentication table, Firebase rejects creating the user and returns a validation error.

```

onSignUp(fullName, phoneNumber, email, password, role)
  .catch(error => this.setState({ error }));

```

Listing 12: Catching Firebase validation error for sign up and setting it to the error state

The catch block handles the Firebase validation error and updates the error state. The error state then renders the error message to the user above the `signUp` button on the `sign-up` screen in figure 4.

4.4.2 Log in

The users of the application will enter the required fields, which are the `email` address and `password` into the form presented on the Sign-In screen from figure 5. The Firebase Auth handles the user's `email` address and `password` in listing 13 and checks if it is an existing user with the similar Authentication table used during the registration process.

```
auth.signInWithEmailAndPassword(email, password)
```

Listing 13: Handling user's email and password with firebase auth

On successful login, Firebase returns a Promise data object from figure 9 which is persisted using the `setItem` property provided by `AsyncStorage` with a key called `userData` in listing 14.

```
AsyncStorage.setItem('userData', JSON.stringify(authData));
```

Listing 14: Storing firebase response data to `userData` with Async storage

Successfully setting the authorisation token navigates to the Signed-in state from listing 11 and when an unsuccessful login occurs, Firebase will throw a validation error that is caught and set to an error state in listing 15. For example, when a user enters an invalid email address or an incorrect password for a valid email address stored in the Authentication table, Firebase rejects logging in the user and returns a validation error.

```
onSignIn(email, password)
  .catch(error => this.setState({ error}));
```

Listing 15: Catching Firebase validation error for sign in and setting it to the error state

The catch block handles the validation error and sets it to an error state. The error state then renders the error message to the user above the `SignIn` button on the `Sign-In` screen from figure 5.

4.4.3 Log out

The users can log out successfully from the application using the sign out button on the Profile screen from figure 6. The Firebase Auth handles the logging out process using the `signOut` method on line 1 in listing 16.

```
1  auth.signOut()  
2    .then(() => AsyncStorage.removeItem('userData'))  
3    .then(() => this.props.navigation.navigate('SignedOut'))  
4    .catch(err => throw err.message);
```

Listing 16: The logging out process

Successfully logging out returns a Firebase promise of void which does not include any Firebase response data. The `userData` key is removed using the `removeItem` property of `AsyncStorage` on line 2 in listing 16 ensuring the removal of user persistence, this causes the user to re-authenticate next time when using the application. Finally, the removal of the authorisation token navigates to the `Signed-out` state on line 3 of listing 16. The `navigate` property of React navigation navigates to the desired screen, which is the Welcome Screen from figure 3.

4.5 MapView

When the passenger successfully authenticates to the application, the `MapView` component is displayed. This `MapView` is a part of the `react-native-maps` library built by Airbnb that uses Apple maps or Google maps on iOS and Google maps on Android. The Expo library provides the `Map` component that can be easily imported into the project and does not require any setup. However, when building a standalone application some custom deployment configuration is required for Android but not iOS. The `MapView` component is a part of the Home Screen in figure 7.

```

const driverRef = database.ref("drivers");
const currentUID = auth.currentUser.uid;

driverRef.once("value", snapshot => {
  if (snapshot.hasChild(currentUID)) {
    this.setState({
      role: {
        ...this.state.role,
        driver: true
      }
    });
  }
});

```

Listing 17: Setting driver role if user id exists in driver

The Home Screen from figure 7 contains the initial state of two roles which are passenger and driver. The `componentWillMount` lifecycle method calls an asynchronous listener attached to the Firebase Realtime Database references to `passengers` and `drivers`. The `once` property listens to the database references once, and then the `hasChild` property is used on the snapshot value to check if the `currentUser` id matches the database references. If the `drivers` reference contains the `currentUser` id as child reference then the driver role is updated in listing 17 and if the `passenger` reference contains the `currentUser` id as child reference then the passenger role is updated in listing 18.

```

const passengerRef = database.ref("passengers");
const currentUID = auth.currentUser.uid;

passengerRef.once("value", snapshot => {
  if (snapshot.hasChild(currentUID)) {
    this.setState({
      role: {
        ...this.state.role,
        passenger: true
      }
    });
  }
});

```

Listing 18: Setting passenger role if user id exists in passenger

When the Home Screen from figure 7 mounts initially it loads a list of passenger's ride requests on the map. The loading is achieved using the `componentWillMount` method which is available on the Home Screen lifecycle methods. The lifecycle method contains an asynchronous listener attached to the Firebase Realtime Database reference `ride-requests` in listing 19.

```
const rideRequestRef = database.ref('ride-requests');

rideRequestRef.on('value', snapshot => {
  const passengers = snapshot.val() !== null ? Object.values(snapshot.val()) : [];
})
```

Listing 19: Listening to passenger's ride request from Firebase Realtime Database

The `ride-requests` reference is initially triggered once for the state of data and triggered again when the data changes. This data is retrieved using the `on` method on the `ride-request` reference and is assigned to a passenger's variable. If the passenger's variable in listing 19 exists, the state of `passengers` is updated and rendered in listing 20 as Map coordinates on the `MapView`.

```
this.state.passengers.map(marker => <MapView.Marker/>)
```

Listing 20: Rendering passenger data as a marker on the Map

The `key` property in listing 21, is required for mapping over a state that is rendered. It helps React to track changes of the element's id such that it requires rendering or not thus an `index` will be provided. The `image` property is required for displaying the passenger icon and adding of custom passenger icons to the application. Since there will be multiple passenger's ride request marker on the map, they are separated into three different sets of icons such as the passenger's own ride request icon, other passenger's ride requests icon and passenger ride request accepted icon when the ride request of the passenger has been accepted by the driver.

```

this.state.passengers.map((marker, index) => (
  <MapView.Marker
    key={index}
    image={ICON}
    coordinate={{
      latitude: Number(marker.origin.coordinates.lat),
      longitude: Number(marker.origin.coordinates.lng)
    }}
    onPress={() => this.showPassengerDetails(marker)}
  />)
)

```

Listing 21: Mapping passenger's markers coordinates to MapView Marker

The `coordinate` property requires exact marker position which are the `latitude` and `longitude` of the passenger's origin and destination addresses. The `onPress` property triggers an event that displays a modal window with the current passenger details.

```

<MapView.Marker
  coordinate={this.state.location.coords}
/>

```

Listing 22: MapView marker rendering driver's location coordinates

The driver marker is displayed when the user authenticates with the driver role and grants the permission is requested by the app to access the user's current location using the `Permission` module from the Expo module. Successfully granting allows the application to access user's current location using the `getCurrentPositionAsync` property on `Location` module from the Expo module and updating the location state. The `MapView` marker `coordinates` property receives the location coordinates and renders as `MapView Marker` coordinate in listing 22.

On unsuccessful granting of permission, the location result state updates with permission denied text, and the user will not be able to locate the marker on the `MapView`. Last but not least, to avoid conflict in identifying the driver icon on the map, a condition is used to render different icons such as the yellow car marker for the current driver and random colour car marker for other drivers on the `MapView`.

4.6 Passenger

The Passenger role has the ability to create and cancel a ride request. In this section, the passenger's functionalities are explained based on the chosen technologies discussed in subsection 3.5.

4.6.1 Ride request

The ride requests created by the passenger will ensure that both the passenger and the driver has the correct ride request information. Passengers creating a successful ride request requires the following values which are the origin address, destination address, journey fare and duration. A payload object will be created containing the values, and then the Firebase Realtime database inserts the payload object under the `currentUser` id of the `ride-request` reference.

The ride request process is broken down into three parts:

1. Address search
2. Ride duration and fare
3. Updating database reference

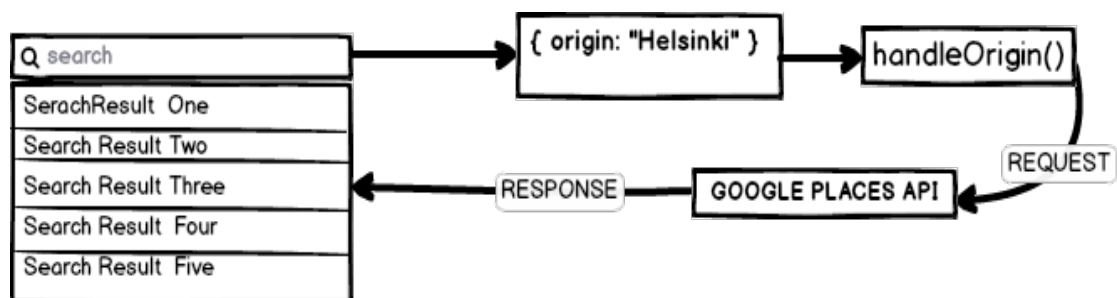


Figure 10: Search box searching for an address using google maps API

The correct address is obtained when inputting an address on the input fields in figure 10. The user types into the input search box, each keystroke updates the initial state of users input. The updating of the initial state of user input is achieved using callbacks with the method `handleChange`, which is bound to the input search boxes. Each us-

er's keystroke calls `setState`, which is an asynchronous function for updating state on the react component from its initial state within the handling method in listing 23.

```
handleChange = (event) => this.setState({[event.target]: event.target.value});
```

Listing 23: Updating state of users input of address

The user input state when updating requires debouncing by one millisecond. Debouncing is a way to limit the rate at which the function runs. When the debouncing and updating the addresses state completes, an HTTP `GET` request is sent to the Google Maps Places API using the `fetch` API. The `fetch` API provides an interface for fetching resources such as network resources and is similar to performing `XMLHttpRequest`. The body of the request in listing 24 will contain the updated address state as input parameters and the Google Maps API keys obtained from subsection 4.1.

```
fetch(`https://maps.googleapis.com/maps/api/place/autocomplete/json?input=Helsinki&types=geocode&key=YOUR_API_KEY` )
```

Listing 24: Fetching Origin address from google places API

Once the HTTP request succeeds, an HTTP response is received which contains a list of address predictions matching the input parameters requested in figure 11. The HTTP response will update the addresses state and will be rendered into a list view.

```
{
  predictions : [{
    description: "Helsinki, Finland",
    id: "a88c998a95685a43779840919259d41788eb8210",
    ...},
  ],
  {
    description: "Budapest, Helsinki Way, Hungary",
    id: "236ec4be9e5140821bba3e5cbdec498cf41e16a",
    ...},
  }, {...}, {...}, {...}
},
status: "OK"
}
```

Figure 11: Response out from google autocomplete API

The autocomplete component library called `react-native-google-places-autocomplete` is used, which will encapsulate the functionalities when are searching for an address, displaying address prediction on the listview, debouncing of the search result and handling of XHR cancellation in listing 25.

```
<GooglePlacesAutocomplete
  placeholder="Enter origin"
  minLength={2}
  autoFocus={false}
  returnKeyType={'default'}
  fetchDetails={true}
  listViewDisplayed="auto"
  query={query}
  onPress={this.handleOrigin}
  styles={styles.origin}
  nearbyPlacesAPI="GoogleReverseGeocoding"
  currentLocation={false}
/>
```

Listing 25: The origin search box component using `GooglePlacesAutoComplete`

The passenger will have the ability to determine the journey price and duration before creating a ride request, and this will ensure that the passenger receives correct amount passenger for payment and the duration time the journey will take. With the help of Google Distance Matrix API, it aids in calculation of the ride duration and fare. An HTTP GET request is sent to Google Distance Matrix API with the updated state addresses as input parameters in listing 26.

```
const {origin, destination} = this.state;

let DISTANCEMATRIX = `https://maps.googleapis.com/maps/api/distancematrix/json?units=imperial
&origins=${origin.address}&destinations=${destination.address}&mode="DRIVING"&key=API_KEY`;

fetch(DISTANCEMATRIX)
```

Listing 26: Sending a request for calculating distance

Once the HTTP request succeeds, an HTTP response is received, it will contain the `destination_addresses`, `origin_addresses`, `rows` and `status` matching the input parameters requested.

```

{
  rows: [{
    elements: [{
      distance: {text: "13.3 mi", value: 21433},
      duration: {text: "24 mins", value: 1431},
      status: "OK"
    }]
  }]
}

```

Figure 12: The rows elements response by Google Maps Distance Matrix API showing the duration and distance of the origin and destination entered

The distance value from the HTTP response available on the `elements` property from figure 12 is used to calculate the journey fare with formula in listing 29 which requires a base fare, time rate, distance rate, surge, distance and time.

```

function calculateFare(baseFare, timeRate, distanceRate, surge, time, distance){
  const distanceInKm = distance * 0.001;
  const timeInMin = time * 0.0166667;
  const pricePerKm = timeRate * timeInMin;
  const pricePerMinute = distanceRate * distanceInKm;
  const totalFare = (baseFare + pricePerKm + pricePerMinute) * surge;

  return Math.round(totalFare);
};

```

Listing 27: The formula for calculating ride fare

Once the ride fare is calculated, the duration value from figure 12 and using the calculated fare updates the component state with rides fare and duration that is rendered and shown when the passenger has entered the origin and destination addresses.

```

database
  .ref("ride-requests")
  .child(currentUser)
  .set(null)
  .then(() =>
    database
      .ref("ride-history")
      .child(currentUser)
      .push(payload2)
  );

```

Listing 28: Pushing the ride request payload to ride-request and ride-history firebase database references

Finally, in order to send the request successfully to the database references which are `ride-requests` and `ride-history` the following are required in the payload which are the `currentUser` id, `displayName` obtained from the Firebase auth function from listing 2, the ride request itineraries such as the `origin` and `destination` addresses, journey fare, duration and status are required which are available from the react component state, automatic timestamp `ISOString` generated with the Date API in component state, and the status `pending` in listing 28.

4.6.2 Ride cancellation

The ride cancellation requested by the passenger will ensure that the passenger does not require a ride, and the driver does not have to accept the passenger's ride request. Passengers cancelling a ride request requires the following values which are the `currentUser` id and timestamp. A payload object will be created containing the values, and then the Firebase Realtime database will check if the passenger's ride request contained the `accepted_by` property and based on that appropriate payload object will be inserted under the `currentUser` id of the `ride-cancellation` reference. The ride cancellation process is broken down into two parts which are ride cancellation payloads and updating of database references.

The ride cancellation payload requires the `currentUser` id, which can be obtained from listing 2. The first payload will contain the passenger data, the status `cancelled`, the timestamp of the cancelled journey and a `cancellation fee` in listing 29. This payload is executed if the passenger's ride request contains an `accepted_by` property, this property exists only if the driver has accepted the passenger's ride request.

```
const payload1 = {
  ...this.state.passengerData,
  status: 'cancelled',
  cancelledAt: timeStamp,
  fareCancellation: '€5'
};
```

Listing 29: Payload one contains passenger Data, status, cancellation time and a fare cancellation fee

The second payload will contain the passenger data, the status cancelled and the timestamp of the cancelled journey in listing 30. This payload is executed by default when the passenger's ride request does not contain an `accepted_by` property, as this property exists only if the driver has accepted the passenger's ride request.

```
const payload2 = {
  ...this.state.passengerData,
  status: 'cancelled',
  cancelledAt: timeStamp
};
```

Listing 30 Payload two contains passenger Data, status and cancellation time

The ride cancellation payloads from the listing 29 and 30 are updated to the `ride-cancellation` and `ride-history` Firebase database references in sequence in listing 31.

```
database
  .ref("ride-cancellation")
  .push(payload)
  .then(() =>
    database
      .ref("ride-history")
      .child(currentUser)
      .push(payload)
  );
```

Listing 31: Pushing payload `ride-cancellation` and `ride-history` Firebase database references

Once both `ride-cancellation` and `ride-history` Firebase database reference are updated, the `currentUser` id of the `ride-request` reference will be set to 'null' which causes removes the coordinate from the `MapView`.

4.6.3 Ride confirmation

The ride request confirmation will be received by the passenger as a notification when their ride has been accepted by the driver, that can be viewed on the Notification screen from figure 8.

```
const notificationsRef = databse.ref("notifications");
const currentUID = auth.currentUser.uid;

notificationsRef.on("value", snapshot => {
  if (snapshot.hasChild(currentUID)) {
    this.setState({
      notificationsAvailable: Object.values(snapshot.val())
    })
  }
})
```

Listing 32: Passengers ride request notifications

An asynchronous listener is attached to the `ride-notification` Firebase database reference in listing 32 which checks if there are notification value for the `currentUser` id exists and the state `notificationAvailable` is updated with notification value which is the loaded as a list of notification on the Notification Screen from figure 8.

4.7 Driver

The Driver role has the ability to respond to ride request by filling the driver seats with passenger's ride request and drive towards to ride request route. In this section, the driver's functionalities are explained based on the chosen technologies discussed in subsection 3.2.

4.7.1 Ride response

The ride response created by the driver will ensure that the passenger ride request has been accepted and the driver has accepted the correct ride request. Passenger will be notified which driver has accepted their ride request by checking the driver name on the `accepted_by` field, viewing the latest notification on the notification screen and displaying the colour change of their ride request icon to green.

The ride response process is broken down into three parts:

1. Adding the passenger to current ride
2. Updating the passenger ride information
3. Sending a notification of the ride status

When the driver authenticates successfully to the Home Screen from figure 7, from its current location if there are any nearby ride request he will be able to view them on the MapView as passenger markers. Upon clicking on them, the driver displays a modal screen with the Passenger details along with an accept button. The accept button when clicked allows the driver can fill the number of available seats with accepted passengers. The number of available seats of the driver's car depends on the number of acceptance made by the driver. The passengers accepted data are available under the `drivers` Firebase database reference child `accepted` in listing 33.

```
database
  .ref('drivers')
  .child(currentUserID)
  .child('accepted')
  .push(this.state.passengerData)
```

Listing 33: Passengers added under the drivers accepted reference

Upon successful addition of the passenger, the driver notifies the passenger by updating the `accepted_by` and `status` fields in listing 34. The `accepted_by` field will include the driver's payload which is driver's `user id`, `displayName`, `email`, `phone number` and the `current timestamp` and the `status` as `confirmed`.

```
const { uid } = this.props.marker;

database
  .ref('ride-requests')
  .child(uid)
  .update({ accepted_by: driver, status: 'confirmed' })
```

Listing 34: Updating passenger status from pending to confirmed and accepted by with driver payload

The passenger's ride request icon will have a colour change which occurs when the icon state updates from `pending` to `confirmed` in listing 35.


```

this.setState({
  passengerData: {
    ...this.state.passengerData,
    status: 'confirmed',
    accepted_by: displayName
  }
});

```

Listing 35: Updating the current state of passenger data with status and accepted_by values

Upon successful updating of the passenger ride request information, the Notification screen from figure 8, is also updated which displays the status passenger's ride request on its listview. The Notification update is achieved using the same payload information which is the passenger data which the driver has accepted onto the `notifications` Firebase reference of the passenger's user id.

4.7.2 Ride routes

The ride routes ensure that the driver heads to the correct direction of the passenger's current ride request. Drivers can view two ride routes which are the Pick-Up route and the Drop Off route. The driver's routes are displayed when it is current location, and the passenger's ride request pickup and drop off locations are available. The ride routes process is broken down into two parts: Pick Up and Drop Off

As discussed in the subsection 4.5, the driver's current location is obtained initially when the `MapView` component loads and using the `getCurrentPositionAsync` property on `Location` module provided by Expo that updates the driver's location state on the `MapView` on listing 36. When the driver departs from its current location, the new updated location is obtained using the `watchPositionAsync` property on `Location` module provided by Expo to detect the driver's current location. The options used are `maximumAge` of a one-millisecond, `timeout` of two-milliseconds and enabling the accuracy level on `enableHighAccuracy`, which updates the driver's location internal state and the driver marker on the `MapView`.

```

this.watchID = Location.watchPositionAsync(
  {
    enableHighAccuracy: true,
    timeout: 20000,
    maximumAge: 10000
  },
  location => this.setState({ location })
);

```

Listing 36: Updating driver's current location using watchPositionAsync

The driver will obtain a list of passenger's locations from the accepted references under its `currentUser` id of the drivers Firebase database reference. Each passenger from the list includes an `origin` and `destination` key which contains the formatted address in the `address` key and the `latitude` and `longitude` coordinates in the `coordinates` key. The passenger's `origin` and `destination` values are updated to the `passengerLocations` `origin` and `destination` state in listing 37.

```

database
  .ref("drivers")
  .child(user.uid)
  .child("accepted")
  .on("value", snapshot => {
    const vals = Object.values(snapshot.val());
    const origin = vals.map(val => val.origin);
    const destination = vals.map(val => val.destination);
    this.setState({
      passengerLocations: {
        origin: origin,
        destination: destination
      }
    });
  });
});

```

Listing 37: Updating the passenger's location from the driver's accepted firebase reference

Upon receiving the driver's current location and passenger's origin addresses successfully, the passenger's origin addresses are sorted by the distance from the driver's current location using the `sort-by-distance` library in listing 38.

```

const sorted = sortByDistance(origin, passengerOrigin, opts);

```

Listing 38: Sorting passenger origin from driver's origin by distance

The `origin` is the driver's current location coordinates, `passengerOrigin` is the list of passenger's origin location coordinates and `opts` is the options property which receives the `lat` and `lng` properties. Upon successful sorting of the location coordinates in listing 38, the last location coordinate from the sorted result is assigned to a `destination` variable, and the remaining sorted result to a `waypoints` variable. The `destination` variable is then assigned and updated to the `pickUpOrigin` state and the `waypoints` variable assigned and updated to the `pickUpWaypoints` state in listing 39.

```
const waypoints = sorted.slice(0, -1);
const destination = sorted[sorted.length - 1];

this.props.view.setState({
  pickUpWaypoints: waypoints,
  pickUpOrigin: destination
});
```

Listing 39: Collecting waypoint and origin from the sorted result and setting new pickup waypoints and origin states

The `Direction` component uses these values `drivers`, `current location`, `passenger's waypoints` and `destination` to draw routes between coordinates using the 'react-native-maps-directions' library and Google Maps Direction API in listing 40.

```
<MapViewDirections
  origin={this.state.driverLocation}
  waypoints={this.state.pickUpWaypoints}
  destination={this.state.pickUpOrigin}
  apiKey={`YOUR_API_KEY`}
  strokeWidth={2}
  strokeColor="black"
  mode="driving"
/>
```

Listing 40: Displaying pick up routes from the driver location

The `origin` is the start routing location from, `destination` is the from the starting location to and the `waypoints` are an array of location coordinates between the origin and destination and `mode` is the transportation mode. When the driver's location coordinates

coordinates are similar to the passenger location coordinates, the driver will apply the `confirm` button presented in the passenger details component. This confirmation removes the passenger ride request from the map by setting the passenger's ride request data to null on the `accepted` reference and moving the passenger's ride request data under `confirmed` reference of the `currentUser` id of the `drivers` Firebase database reference in listing 41. This ensures that no overlapping of the driver marker with the passenger marker occurs and correct direction is rendered between the driver and passenger.

```
database
  .ref("drivers")
  .child(driverUID)
  .child("confirmed")
  .push(this.state.passengerData)
  .then(() =>
    database
      .ref("drivers")
      .child(user.uid)
      .child("accepted")
      .child(passengerUID)
      .set(null)
  );
```

Listing 41: Passenger's ride confirmation and removal from Firebase database reference

Successfully picking up all the passengers and confirming their ride requests, the passenger's destination location coordinates will be generated as markers on the `MapView` of the Home Screen from figure 7. Using the driver's current location and the passenger's destination addresses, they will be sorted by the distance from the driver's current location and stored in a `const` variable called `sorted` using the `sort-by-distance` library in listing 42.

```
const sorted = sortByDistance(origin, passengerDestination, opts)
```

Listing 42: Sorting passenger destinations from driver's origin by distance

The `origin` is the driver's current location coordinates, `passengerDestination` is the list of passenger's destination location coordinates and `opts` is the options property that accepts a `lat` and `lng` property. Upon successful sorting of the location coordinates in listing 68, the last location coordinate from the sorted result is assigned to a `destination` variable, and the remaining sorted result to a `waypoints` variable.

The `destination` variable is then assigned and updated to the `dropOffDestination` state and the `waypoints` variable assigned and updated to the `dropOffWaypoints` state in the listing 43.

```
const waypoints = sorted.slice(0, -1);
const destination = sorted[sorted.length - 1];

this.props.view.setState({
  dropOffWaypoints: waypoints,
  dropOffDestination: destination
});
```

Listing 43: Sorting passenger destinations from driver's origin by distance

Using the available values such as the driver's current location, passenger's waypoints and destination values are passed onto the `Direction` component which uses the `react-native-maps-directions` library to draw routes between two coordinates using the Google Maps Directions API in the listing 44.

```
<MapViewDirections
  origin={this.state.driverLocation}
  waypoints={this.state.dropOffWaypoints}
  destination={this.state.dropOffDestination}
  apiKey={`YOUR_API_KEY`}
  strokeWidth={2}
  strokeColor="black"
  mode="driving"
/>
```

Listing 44: Displaying drop-off routes from the driver location

When the driver reaches the passenger's destination, the passenger makes an electronic payment to the driver. Upon receiving successful payment from the passenger, the driver presses the `complete` button located in the passenger details component.

```
database
  .ref("ride-completions")
  .push(this.state.passengerData)
  .then(() =>
    database
      .ref("drivers")
      .child(driverUID)
      .child("confirmed")
      .child(passengerUID)
      .set(null)
  );
```

Listing 45: Passenger's ride completion and removal from Firebase database reference

This removes the passenger's ride request from the map setting the passenger's ride request data to null on the confirmed reference and adds the passenger's ride request data under ride-completion reference of the currentUser id of the drivers Firebase database reference in listing 45. This ensures the passenger journey has completed and the driver receives the payment.

5 Results

5.1 Passengers ride request

The passenger when authenticating successfully to the application displays **Screen A** of figure 13, which contains a MapView with a floating hand button.



Figure 13: The process of sending a ride request

When the floating hand button is clicked, a ride request modal is displayed on **Screen B** containing two search input fields with labels as *Origin* and *Destination* and two buttons with labels as *Close* and *Request*. The request button is disabled since it requires the passenger to enter the origin and destination addresses in the input field. When the passenger enters the required fields, a response with ride duration and fare is displayed on **Screen C**. If the passenger agrees with the response duration time and destination, the request is forwarded with clicking the request button on **Screen D** which generates a ride request query entered to the database under the `ride-request` database reference and closes the modal showing the passenger ride request origin coordinates on the MapView on **Screen E**.

5.2 Passengers ride cancellation

The passenger notices the ride requests as red icon on the MapView on **Screen A** in figure 14.

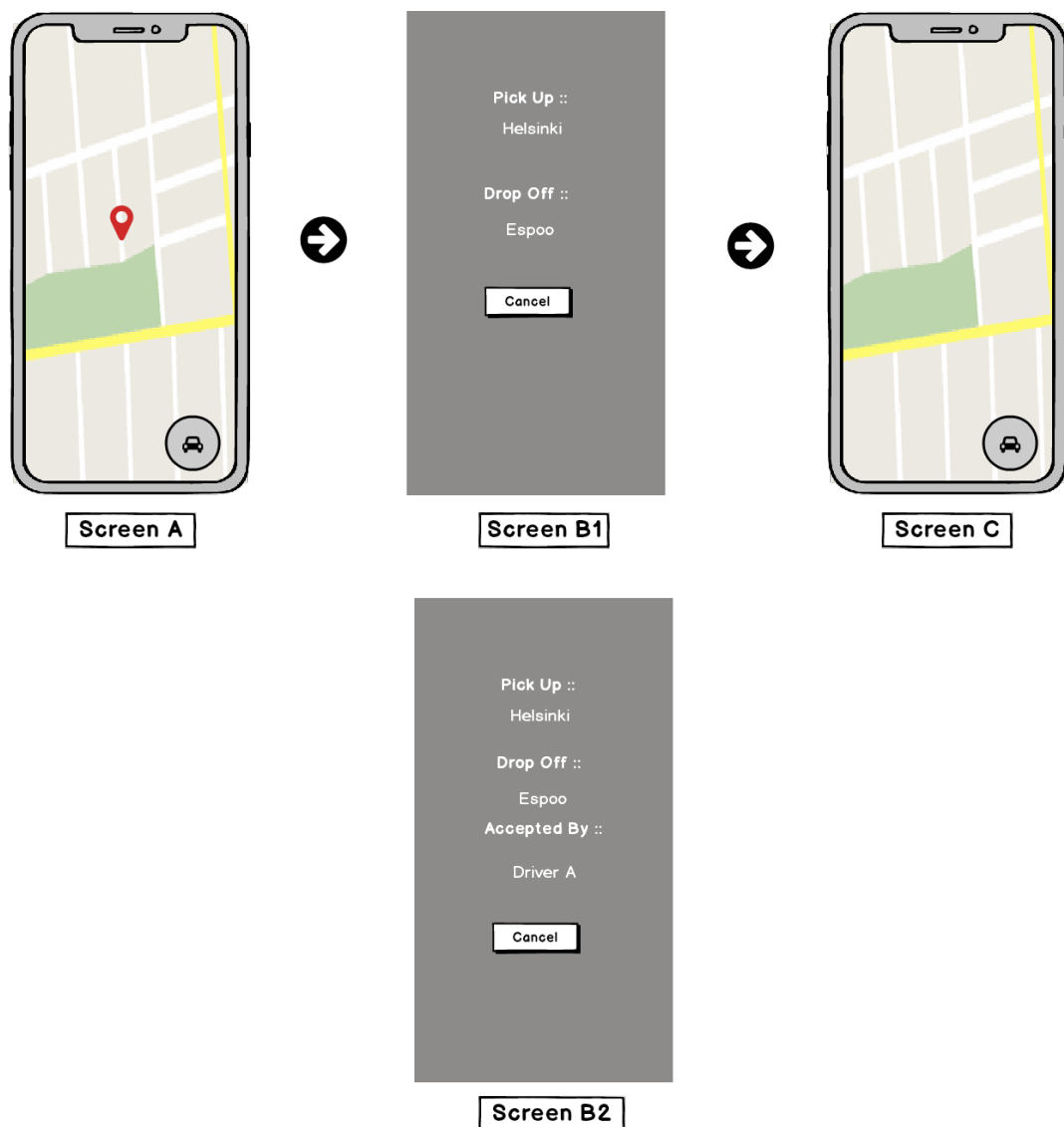
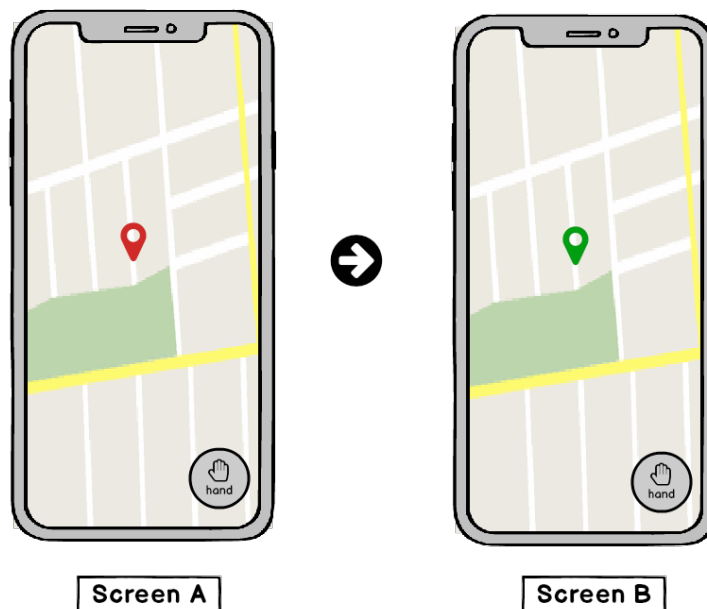


Figure 14: The process of cancelling a ride.

The passenger coordinate pin when clicked opens the modal window attached onto the icon on **Screen B1**, which displays the current ride details and a cancel button for cancelling the current ride request. If the ride request has already been accepted it will display the `accepted_by` property on **Screen B2** included with the ride details and cancelling the ride request will cause the passenger to pay a ride fare fine. If the ride request has not been accepted by any driver, then the passenger can cancel the ride request without paying a ride fare fine. When the user clicks the cancel button, the current ride request data that is being cancelled is inserted into the `ride-cancellation` and `ride-history` references and finally the `currentUser` id of the `ride-request` is set to `null` which removes the ride request and renders the new set of coordinates on the MapView on **Screen C**.

5.3 Passengers ride confirmation

The passenger creating the ride request successfully displays a passenger icon as red colour on the MapView on **Screen A** in figure 15. When the passenger ride request is accepted by the driver, the passenger icon turns from red to green on **Screen B**.



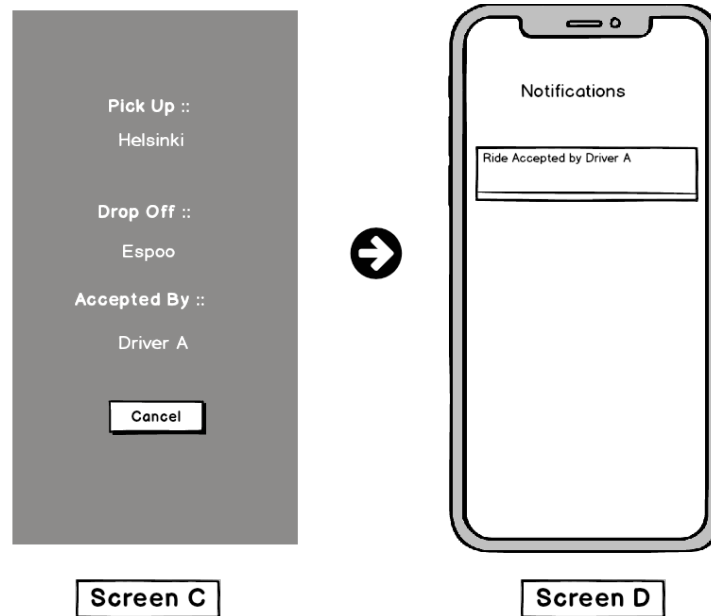
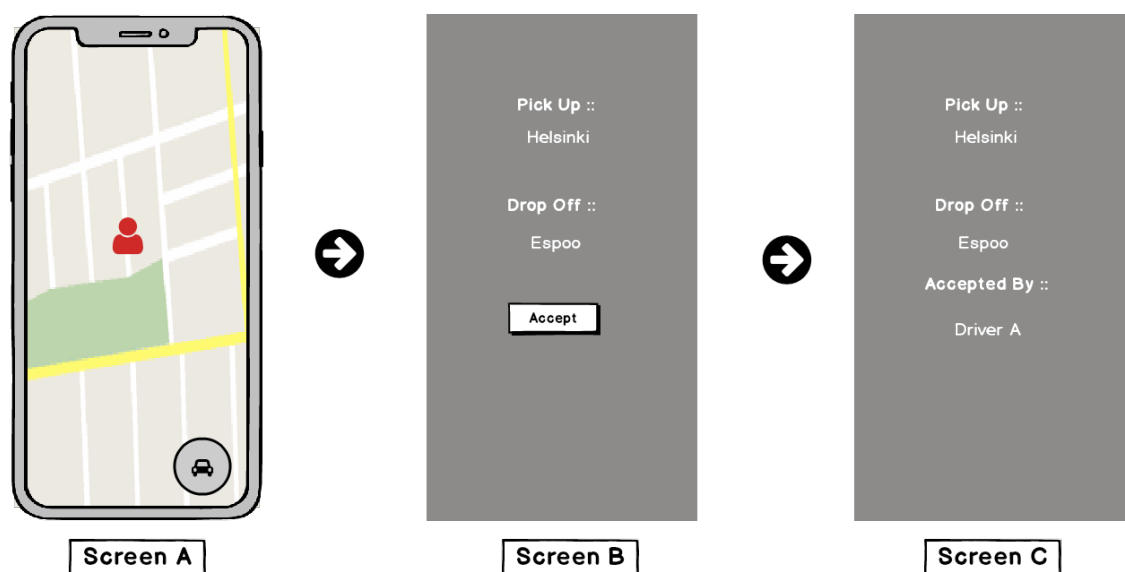


Figure 15: The process of ride notification

The passenger when clicks the green marker on **Screen B**, a modal attached to the icon opens containing the ride details including the driver name on **Screen C** and also notices a ride request notification in the notifications list on **Screen D**.

5.4 Drivers ride response

The driver notices the passengers ride request icon from its current location on the MapView on **Screen A** of figure 16 and responds to the passenger icon by clicking it.



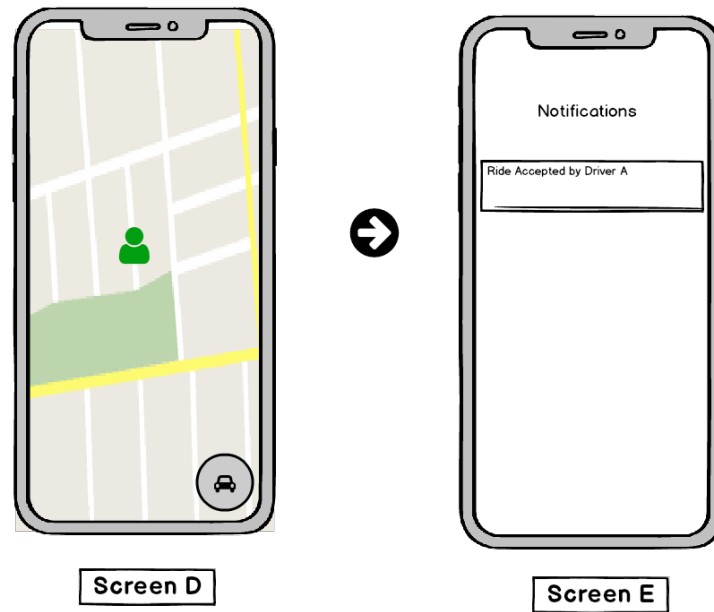
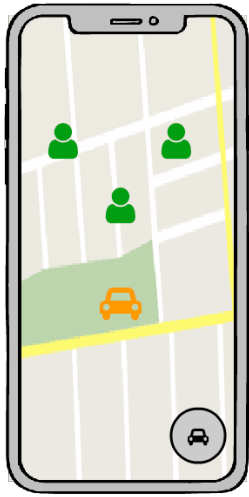


Figure 16: The process of responding to a ride request

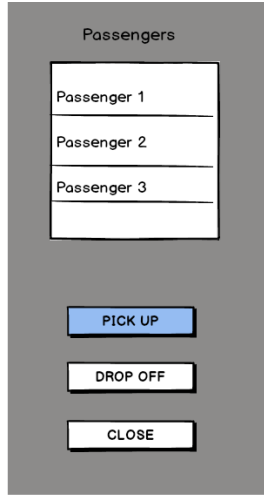
The passenger icon when clicked opens a popup window attached with the icon on **Screen B** showing the passenger's ride details and an accept button for accepting the ride request. The accept button when clicked, displays the driver's name on the `accepted_by` field in **Screen C**. The modal window is closed when clicked anywhere within **Screen C**, the passenger icon turning from red to green in **Screen D** which displays that the ride has been accepted and finally the ride request details are notified to the passenger in **Screen E**.

5.5 Drivers ride routes

The driving routes which are the pickup and drop off routes are explained with six Screens A to B in figure 17. The **Screen A** is displayed when the user has authenticated with the driver role and has accepted passenger's ride requests.



Screen A



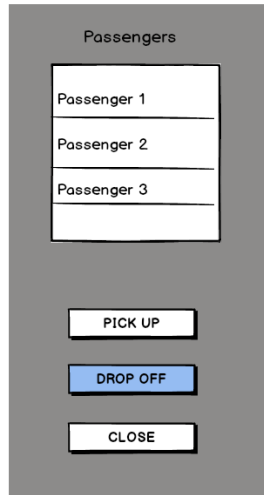
Screen B



Screen C



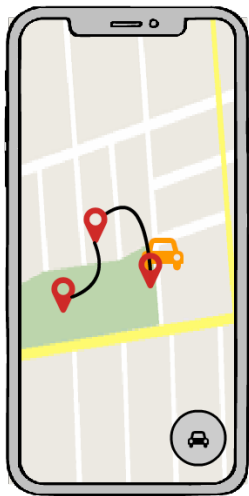
Screen D



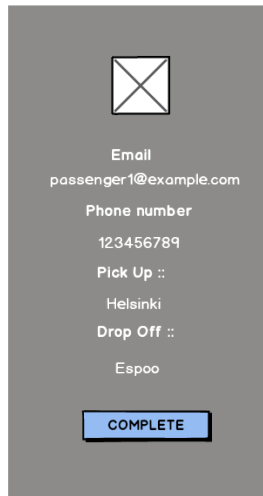
Screen E



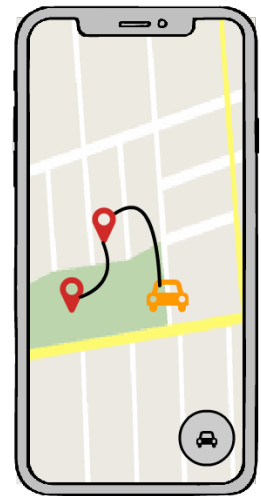
Screen F



Screen G



Screen H



Screen I

Figure 17: The process of generating ride routes

When the driver clicks the car icon button located on the bottom right end corner, it displays the modal window with containing the list of passengers on the ride and three buttons which are PICK-UP, DROP-OFF and CLOSE on **Screen B**. The PICK-UP button when clicked generates the driving routes from the driver's location to the passenger's origin location on **Screen C**. The passenger icons are automatically removed when the driver reaches the passenger's origin location and taps twice on the icon. This removes the passenger icon automatically and are used when the driver's location is similar to the passenger location on **Screen D**. When the driver again clicks again onto the car icon in the bottom right corner, the DROP-OFF button from modal window on **Screen E** generates new markers of passenger's destinations location and the driving routes from the driver's location on the MapView on **Screen F**. When the driver successfully drops the passengers to its destination on **Screen G**, the driver clicks on the passenger's marker which opens a modal window showing passengers details and a COMPLETE button on **Screen H**. When the passenger has made payment to the driver, the driver clicks on the COMPLETE button which removes the passenger destination icon from the MapView on **Screen I** and finally it is inserted to the `ride-completion` reference.

6 Discussion

6.1 Database

The Firebase Realtime Database used for this project is sufficient for the application use cases. However, developing further could be quite challenging to solve due to its limitations [43] such as:

- Data stored as one large JSON tree
- Limited deep querying capabilities for sorting and filtering
- Sharding data required for scaling
- High pricing charged for bandwidth and storage

During the time of writing, Firebase released a new flagship database called the Firebase Cloud Firestore in beta. It solves all the limitation described above, and its features include a better data model, which stores data in documents organised in collections, faster compound deep querying capability for sorting and filtering, automatic scaling without the need to shard data and lower pricing charged for bandwidth and storage [44]. Moreover, it can be extended with Firebase Cloud Functions [45], that can handle event trigger changes made to the database.

6.2 User Interface state

The Reacts component state [46] used for this project was not an efficient solution for the application use cases. There were numerous bugs encountered during development and fixing often required misusing the `setState` method and the lifecycle methods of the React component. Moreover, multiple application components needed to access the Firebase Realtime Database for handling logic and rendering of data. It will be beneficial if a single state tree known as a store, were used with Firebase database model, of which the user interface can listen and render the data needed. State management libraries such as Redux [47] allows managing the user interface state with a predictable state container and can be integrated with React efficiently. Furthermore, it's minimalistic API, strong ecosystem and rich developer tool such as Redux devtool [48] provide a great developer experience.

6.3 Authentication

Although the Authentication described in subsection 4.4 supports the primary use cases of the application, few use cases would improve the current implementation of the system such as user email verification, resetting of password and authenticating with different authentication providers.

The users can register with any [email name] @ [any domain], and since the registered email does not involve an actual account process, which is a challenge to identify if the users are real. The Email verification of user accounts can help with identifying the users. The Firebase Auth provides email verification for users and enabling will allow the application to send an email containing a validation link to the address. Thus, when users register on the application can receive a validation link and be verified by confirming it. However, if the user chooses not to confirm his verification, necessary action can be taken to prevent the user from signing in such as blocking the account.

Moreover, the users are not able to reset the password when password is forgotten. The Firebase Auth provides password reset method that allows user to reset password and also allow custom email templating using the Firebase console. It also provides the possibility of resetting the password and redirecting within the application.

Furthermore, majority of the users would prefer simple authentication with well-known auth providers such as Facebook, Google Auth and Phone number, that can be easier to authenticate and also be a returning user of the application. The user can also skip the registration process as the authentication identity providers would present the users details to the application automatically.

6.4 Passenger

The current features and functionalities of the passengers were sufficient to the passenger requirements discussed in subsection 2.1 user stories. However, features such as passenger profile management, real-time tracking of driver's location, communication with the driver, a dynamic matching algorithm with other passengers and drivers, and a feedback system would improve the passengers ridesharing experience.

The Profile screen from figure 6 currently displays the passenger's details and its ride histories. The current implementation of the application does not allow profile management such as updating passenger's details, profile picture and switching role. The Firebase Cloud Storage [49] can be used to store and update the passenger's profile picture as it uses the Google Cloud Storage bucket. It allows uploading of rich content such as images, video and documents and provides an API for creating image references that can work along with the database. Furthermore, the updating of passenger's details, profile picture and switching role can be achieved using the update method available on the Firebase Realtime Database API.

The real-time tracking of the driver location assures the passengers about the driver being nearby its current location and the estimation time of arrival. It allows the passenger to locate the correct car without being lost and can be solved using the background geolocation feature. Unfortunately, the background geolocation [50] is in progress and hopefully will be available the upcoming year with easy integration to the application.

The communication between the drivers and passengers is a necessity, as it allows the driver and passenger to communicate in real-time in regard to their ride request. For example, if the driver is unable to locate the passenger despite the correct location on the map or the passenger not being able to recognise the driver's vehicle and also helps in situations where the driver cannot arrive on the estimated time of arrival due to traffic. It can be solved using a messaging and voice over IP (VoIP) services such as WhatsApp and Viber provide an excellent medium for users to communicate in real-time with sending text messages, voice calls, documents and user location. However, integrating the messaging services isn't possible, and passengers wouldn't be comfortable sharing personal information with the driver thus there are two solutions. The first solution is to create a real-time chat service using the Firebase Realtime Database, which stores the messages between the driver and the passenger and displays them concurrently in real-time. The second solution is to use a service such as Send Bird, which provides a messaging SDK and Chat API that can be integrated easily into the application in comparison to the creation of a real-time chat service [51].

The passengers can make a ride request but needs to wait for nearby drivers until the request has been accepted. The application would be helpful if it provided a ride-matching for passengers with other passengers located within the same area and with

drivers nearby. It currently calculates the driver routes manually based on the passenger's origin and destination coordinates and displays the driving routes on a map view. Masoud and Jayakrishnan [52] study on real-time algorithm solves the peer-to-peer ride-matching problem, which can be further studied and integrated into the application.

The passenger when reaching its destination and completing the payment transaction cannot rate or provide feedback for the service currently. The feedback system is an integral part of the application that helps in improvement of the system and contribute to a safer community. It can be solved using the Firebase Realtime database, which stores the passengers and driver's feedbacks under a feedback database reference.

6.5 Driver

The current features and functions of the drivers are sufficient to driver's requirements discussed in subsection 2.1 user stories. However, features such as a payment system, better driving route and ride cancellation would improve the driver's ridesharing experience.

The driver when successfully dropping the passenger to its destination receives a payment either by cash or using an electronic payment system. However, it would be useful if the payment was exchanged between the passenger and driver within the application automatically when the journey completes using a payment service such as Stripe, Apple Pay and Android Pay. The Payment SDK [53] available on the Expo Library, cannot be easily integrated as it currently supports react-native and can be included through ExpoKit [54] which requires linking of Android and iOS dependencies.

Moreover, the driving routes to the origin and destination of passenger location are manually calculated and generated upon picking up the passenger. A dynamic driving routes created each time when accepting the passenger's ride request would be useful for the driver. Furthermore, when the driver accepts a passenger's ride request cannot cancel the journey. For example, if the driver has an accident or stuck in traffic could safely cancel the ride request allowing the next driver nearby to pick up the passenger. The addition of cancellation feature to the application should work similar to the ride acceptance in subsection 4.7.1, where the driver notifies the passenger about the cancellation of the journey on the map and notification screen.

7 Conclusion

Majority people in developed countries are looking for numerous ways of utilising existing resources being their car, where an average car is ridden by one to two persons. Thus, the primary aim to create a mobile application prototype of ridesharing using the modern tools and technologies was successful. The brainstorming and prototyping session helped in gathering the main elements needs of the application before it developed.

The mobile app developed using Expo is at its early stage and does not support all native features yet. The mobile app prototypes were quickly made using the Expo XDE. This allowed me to focus on using JavaScript without the need to handle any native code. The Expo client was used to test the application rapidly without the need to deploy to the app-store. The Firebase services such as the Realtime database and Firebase Authentication helped in handling the application logic and authentication.

The mobile app offers numerous functionalities both for the driver and the passenger. Passengers can send a ride request and view them on the map. Drivers can accept the passenger's ride request on the map. The application notifies the passenger about their ride-request acceptance in three places: by changing its icon colour and adding the driver's name in the attached pop-up window and updating the ride status on the notification screen. When the driver successfully accepts all the passengers, the pickup and drop off buttons generates the driving routes to their origin and destination locations respectively. Finally, the exchange of payment is made between the passenger and the driver when arriving at the passenger's destination successfully.

Further improvements such as using a state management library, database model, authentication features, profile management, Realtime tracking of driver's location, communication between the drivers and passenger, a matching algorithm, feedback and a payment system would make it a better ridesharing application.

References

- 1 10 million self-driving cars will hit the road by 2020 here's how to profit [Internet]. Forbes.com. 2018 [cited 21 March 2018]. Available from: <https://www.forbes.com/sites/oliviargarret/2017/03/03/10-million-self-driving-cars-will-hit-the-road-by-2020-heres-how-to-profit/#623fcc7c7e50>
- 2 Radcliffe B. Sharing Economy [Internet]. Investopedia. 2018 [cited 3 May 2018]. Available from: <https://www.investopedia.com/terms/s/sharing-economy.asp>
- 3 The Environmental Benefits of Ridesharing - The Official HyreCar Ridesharing Blog [Internet]. The Official HyreCar Ridesharing Blog. 2018 [cited 8 May 2018]. Available from: <https://hyrecar.com/blog/environmental-benefits-ridesharing/>
- 4 Making career moves? Sign up to be an Uber Driver or get a ride to the airport | Uber [Internet]. Uber.com. 2018 [cited 21 March 2018]. Available from: <https://www.uber.com/>
- 5 Lyft. A Ride in Minutes [Internet]. Lyft. 2018 [cited 21 March 2018]. Available from: <https://www.lyft.com/>
- 6 Evans A, Butler D, Butler D. Lyft Line Vs. UberPool: Which Budget Rideshare Option Wins? [Internet]. Ridester.com. 2018 [cited 3 May 2018]. Available from: <https://www.ridester.com/lyft-line-vs-uberpool-differences/>
- 7 diowa/icare [Internet]. GitHub. 2018 [cited 3 May 2018]. Available from: <https://github.com/diowa/icare>
- 8 hsgr/carpooling [Internet]. GitHub. 2018 [cited 3 May 2018]. Available from: <https://github.com/hsgr/carpooling>
- 9 Berteig M. User Stories and Story Splitting - Agile Advice [Internet]. Agile Advice. 2018 [cited 3 May 2018]. Available from: <http://www.agileadvice.com/2014/03/06/referenceinformation/user-stories-and-story-splitting/>
- 10 MIT "Realtime" Rideshare Research Rideshare History & Statistics [Internet]. Ridesharechoices.scripts.mit.edu.2018 [cited 21 March 2018]. Available from: <http://ridesharechoices.scripts.mit.edu/home/histstats/>
- 11 Chan N, Shaheen S. Ridesharing in North America: Past, Present, and Future. Transport Reviews. 2012;32(1):93-112.
- 12 Kowshik R, Gard J, Loo J, Jovanis P, Kitamura R. Development of User Needs and Functional Requirements for a Realtime Ridesharing System. INSTITUTE

OF TRANSPORTATION STUDIES UNIVERSITY OF CALIFORNIA, BERKELEY. 1993;(UCB-ITS-PWP-93-22).

- 13 Carpool | Rideshare [Internet]. Rideshare.org. 2018 [cited 6 February 2018]. Available from: <https://rideshare.org/modes/carpool/>
- 14 Hall J, Krueger A. An Analysis of the Labour Market for Uber's Driver-Partners in the United States. ILR Review. 2017;001979391771722.
- 15 In the Driver's Seat: A Closer Look at the Uber Partner Experience | Uber Newsroom [Internet]. Uber Newsroom. 2018 [cited 22 March 2018]. Available from: <https://www.uber.com/newsroom/in-the-drivers-seat-understanding-the-uber-partner-experience/>
- 16 Perea C. What's the Real Commission That Uber Takes from Its Drivers? [Infographic] [Internet]. The Rideshare Guy Blog and Podcast. 2018 [cited 21 March 2018]. Available from: <https://therideshareguy.com/whats-the-real-commission-that-uber-takes-from-its-drivers-infographic/>
- 17 CSAIL A. How ridesharing can improve traffic, save money, and help the environment [Internet]. MIT News. 2018 [cited 22 March 2018]. Available from: <http://news.mit.edu/2016/how-ridesharing-can-improve-traffic-save-money-and-help-environment-0104>
- 18 Muro I. Tracking the gig economy: New numbers [Internet]. Brookings. 2018 [cited 21 March 2018]. Available from: <https://www.brookings.edu/research/tracking-the-gig-economy-new-numbers/>
- 19 Valuations of leading ride-hailing companies (April 2017) [Internet]. Atlas. 2018 [cited 22 March 2018]. Available from: <https://www.theatlas.com/charts/HJX3QMVTI>
- 20 React Native: A framework for building native apps using React [Internet]. Facebook.github.io. 2018 [cited 18 April 2018]. Available from: <https://facebook.github.io/react-native/>
- 21 React - A JavaScript library for building user interfaces [Internet]. Reactjs.org. 2018 [cited 18 April 2018]. Available from: <https://reactjs.org/>
- 22 F8 2015 - React Native and Relay - Bringing Modern Web Techniques to Mobile [Internet]. Facebook Code. 2018 [cited 18 April 2018]. Available from: <https://Code.facebook.com/videos/931163756933706/f8-2015-react-native-and-relay-bringing-modern-web-techniques-to-mobile/>
- 23 Introducing Hot Reloading • React Native [Internet]. Facebook.github.io. 2018 [cited 18 April 2018]. Available from: <http://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html>

- 24 JavaScriptCore – WebKit [Internet]. Trac.webkit.org. 2018 [cited 18 April 2018]. Available from: <https://trac.webkit.org/wiki/JavaScriptCore>
- 25 React Native Elements • Cross Platform React Native UI Toolkit [Internet]. React-native-training.github.io. 2018 [cited 18 April 2018]. Available from: <https://react-native-training.github.io/react-native-elements/>
- 26 React Navigation • Routing and navigation for your React Native apps [Internet]. Reactnavigation.org. 2018 [cited 18 April 2018]. Available from: <https://reactnavigation.org/>
- 27 Expo [Internet]. Expo. 2018 [cited 18 April 2018]. Available from: <https://expo.io/>
- 28 Firebase. (2018). Firebase [online] [cited 18 Mar. 2018]. Available from: <https://firebase.google.com/>
- 29 Firebase. (2018). Firebase Realtime Database | Firebase. [online] [cited 18 Mar. 2018] Available at: <https://firebase.google.com/docs/database/>
- 30 Firebase. (2018). Firebase Authentication | Firebase. [online] [cited 18 Mar. 2018] Available at: <https://firebase.google.com/docs/authentication>
- 31 Google Maps [Internet]. Google.com. 2018 [cited 11 Mar 2018]. Available from: <https://www.google.com/maps>
- 32 Google Maps APIs | Google Developers [Internet]. Google Developers. 2018 [cited 18 April 2018]. Available from: <https://developers.google.com/maps/>
- 33 Google Places API | Google Developers [Internet]. Google Developers. 2018 [cited 18 April 2018]. Available from: <https://google-developers.appspot.com/places/>
- 34 Google Maps Directions API | Google Developers [Internet]. Google Developers. 2018 [cited 18 April 2018]. Available from: <https://google-developers.appspot.com/maps/documentation/directions/>
- 35 Get Started | Google Maps Distance Matrix API | Google Developers [Internet]. Google Developers. 2018 [cited 18 April 2018]. Available from: <https://google-developers.appspot.com/maps/documentation/distance-matrix/start>
- 36 invertase/react-native-firebase [Internet]. GitHub. 2018 [cited 26 March 2018]. Available from: <https://github.com/invertase/react-native-firebase>
- 37 Firebase Console [Internet]. Console.firebase.google.com. 2018 [cited 11 May 2018]. Available from: <https://console.firebase.google.com/u/0/>

- 38 Osmani A. Learning JavaScript Design Patterns [Internet]. Addyosmani.com. 2018 [cited 18 April 2018]. Available from: <https://addyosmani.com/resources/essentialjsdesignpatterns/book/#singletonpatternjavascript>
- 39 Understand Firebase Realtime Database Rules | Firebase Realtime Database | Firebase [Internet]. Firebase. 2018 [cited 10 May 2018]. Available from: <https://firebase.google.com/docs/database/security/>
- 40 Google Cloud Platform [Internet]. Console.developers.google.com. 2018 [cited 26 March 2018]. Available from: <https://console.developers.google.com/>
- 41 Data Structures/Stacks and Queues - Wikibooks, open books for an open world [Internet]. En.wikibooks.org. 2018 [cited 9 April 2018]. Available from: https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues#Stacks
- 42 US20130222272A1 - Touch-sensitive navigation in a tab-based application interface - Google Patents [Internet]. Patents.google.com. 2018 [cited 9 April 2018]. Available from: <https://patents.google.com/patent/US20130222272A1/en>
- 43 Choose a Database: Cloud Firestore or Realtime Database | Firebase [Internet]. Firebase. 2018 [cited 11 April 2018]. Available from: <https://firebase.google.com/docs/firestore/rtdb-vs-firestore>
- 44 Cloud Firestore | Firebase [Internet]. Firebase. 2018 [cited 19 April 2018]. Available from: <https://firebase.google.com/docs/firestore/>
- 45 Cloud Functions for Firebase | Firebase [Internet]. Firebase. 2018 [cited 19 April 2018]. Available from: <https://firebase.google.com/docs/functions/>
- 46 Component State - React [Internet]. Reactjs.org. 2018 [cited 10 May 2018]. Available from: <https://reactjs.org/docs/faq-state.html>
- 47 Introduction | Redux [Internet]. Redux.js.org. 2018 [cited 19 April 2018]. Available from: <https://redux.js.org/>
- 48 reduxjs/redux-devtools [Internet]. GitHub. 2018 [cited 11 May 2018]. Available from: <https://github.com/reduxjs/redux-devtools>
- 49 Cloud Storage | Firebase [Internet]. Firebase. 2018 [cited 19 April 2018]. Available from: <https://firebase.google.com/docs/storage/>
- 50 Background location tracking [Internet]. Expo.canny.io. 2018 [cited 19 April 2018]. Available from: <https://expo.canny.io/feature-requests/p/background-location-tracking>

- 51 Messaging SDK and Chat API for Mobile Apps and Websites | SendBird [Internet]. SendBird. 2018 [cited 8 May 2018]. Available from: <https://sendbird.com/>
- 52 Masoud N, Jayakrishnan R. A Realtime algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system. Transportation Research Part B: Methodological. 2017; 106:218-236.
- 53 Payments - Expo Documentation [Internet]. Docs.expo.io. 2018 [cited 19 April 2018]. Available from: <https://docs.expo.io/versions/latest/sdk/payment>
- 54 Detaching to ExpoKit - Expo Documentation [Internet]. Docs.expo.io. 2018 [cited 19 April 2018]. Available from: <https://docs.expo.io/versions/v27.0.0/expokit/detach>