SahaYatra App

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Jayaprakash A
(111501010)

Under the guidance of

Dr. Albert Sunny

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**SahaYatra App**" is a bonafide work of **Jayaprakash A** (**Roll No. 111501010**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr. Albert Sunny

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Dedication

I dedicate this thesis to my mother **Padmaja A** and my father **Nagendra Kumar A** (Executive Director, Balasore Alloys) who have been a constant support for me all throughout my education. I'm extremely grateful to my mother without whose constant encouragement and motivation, I would not have been able to complete my education.

Acknowledgement

I would like to humbly acknowledge with sincere gratitude, my guide, **Dr. Albert Sunny**, (Assistant Professor, IIT Palakkad) for giving me an opportunity to work under his guidance. Under his guidance, I gained a lot of knowledge in the subject. We had several invaluable discussions which helped me expand my thinking horizons. He was truly a source of inspiration. His personal involvement and vast knowledge, motivated me and brought out the best in me.

Special thanks to my friend **A Hemanth Kumar** (Student, IIT Palakkad) who helped me in testing the app by providing his mobile phone and other help whenever needed. It is due to his help that I could test the app across multiple devices.

I would like to thank my brother **Sona Praneeth Akula** (Software Engineer, Oracle Pvt Ltd.), who has helped me by suggesting necessary technologies that has helped me immensely. If not for his continuous feedback on the app's performance and UI design, the app would not have come out so beautifully.

We would also like to mention that we have taken some image resources from Ola developer website for sketches which require a lot of time and expertise.

Contents

List of Figures

List of Tables

Abstract

In today's revolutionary world, apps are everything. Right from small tasks like sending messages to some highly complex things e-commerce we rely completely on apps. That is the level of our bonding with apps. Various services are provided over the internet using apps. The revenue generated by the global mobile app industry has skyrocketed. [1]

One such application with very high importance and high revenue is that of ride share applications. Revenue in the Ride Sharing segment amounts to US$371m in 2018 [2]. Rideshare platforms use a smart phone app which connects drivers with passengers in the area. The driver logs into the app to set their status to online, which indicates that they are available to accept a ride.

The landscape of private transportation changed dramatically when ride share company **Uber** first launched their services. They offered a premium black car service as an alternative to the traditional taxi ride.

In this project, we have tried to create a ride share application that dispatches rides based on demand. Let us consider the following scenario, at a very remote location with high cost of rides. Suppose there are two riders A and B who wish to travel from say, point X to point Y. Now B requests for a ride but is ready to wait for half an hour more too. A requests for a ride 10 minutes later. The current applications allocate two different drivers for them, leading to more cost. This can be cut down, if B waits for A, then both are benefited as the price they pay is now halved. Our project tries to handle such issues.

A ride later feature has also been included which allows the passengers to book a ride

for later purposes.

Last but not the least, we have implemented feature that keeps tracking of driver in case of absence/poor internet connectivity at the driver end.

Chapter 1

Introduction

The fast paced world that we are living in, everything is done by finger's touch. Smart phone and internet is the only thing that one needs to have. With this the entire world seems to shrink into our palms. Apps have made our lives far more easier than we can ever think off.

Long gone are the days when elders used to say "Don't ride with/give rides to strangers". Ride share applications are on a roll these days. It is a rare sight of people living in urban areas not using these services.

1.1 Motivation

The present day ride service platforms are quite advanced enough. Demand based ride allocation, live tracking of driver for customer benefit after ride allocation. But is everything fine? The answer is a big **No!**. We have features that allows us to have track of live location of driver after ride allocation. But what if the driver is in a location of poor internet connectivity? In such situations the rider would not be able to see driver. He/she might feel driver has cancelled the ride which is not the case.

This feature of tracking of driver without use of internet in case of poor internet con-

nection is the main objective of the project. But where do we start from? We need a ride sharing application for the same. Why create it the traditional way? So, we tried to roll out a demand based ride dispatch app with unique fare pricing and driver allocation scheme.

1.2 Problem definition

Several attempts on developing ride service platforms have been made. Some are paid versions while some are open sourced. But all these projects do not have enough flexibility, such that they could be tweaked to build the project that could match our requirements. So, a new attempt to create a new ride share platform that can have lot of flexibility needs to be developed. This platform must include all the essential features of ride service application.

This project deals with several sub problems as mentioned below:

1. **Driver and rider profiles:** Maintaining details of customers in database after proper authentication and verification of user. Store details about the driver, vehicle number, license number, vehicle type and seating capacity.

2. **Input start and end location for ride:** When the customer wishes for a ride, he/she needs to provide start and end location. Having proper controls in the application, so that it is done efficiently and without much complexity

3. **Driver allocation:** After placing a ride, a driver must be allocated to the person. The driver must not be some random driver or the program must not do it statically. This must be done in real time.

4. **Ride sharing:** People who do not wish to travel alone, should be able to share their rides with other passengers(unknown). In such cases, the fare of the journey gets

scaled down accordingly. The selection of co-passengers after start of ride must be based on the journey of the existing passengers.

5. **Ride later:** People should be allowed to pre-book a ride for a later time. The allocated driver must be made ensure that he/she is not allotted any ride at that time.

6. **Driver tracking after acceptance of ride:** The customer must be able to see the driver's current location and vice versa. So that both of them are aware of the exact location of each other.

7. **Conclude ride and fare estimation:** After completion of ride, the details must be stored in database. Also the customer and driver should be given details about the fare price.

Last but not the least the most important task is to develop an aesthetically clean and beautiful user Interface for the application.

1.3 Challenges

The project does not look as easy as described. There are many challenges to it. Several important decision needs to be made in driver allocation and fare price calculation. We will now explain the challenges in brief.

1. **Communication betwen fragments:** Selecting a driver for a ride is not an easy task. The drivers who are more close to start location must be preferred more. But there must be a limit to the distance between his/her location and start point. Also, the server cannot be forced to stop in search in driver for particular customer. These tasks need to be done parallel to other server tasks.

2. **Driver allocation:** Selecting a driver for a ride is not an easy task. The drivers who are more close to start location must be preferred more. But there must be a limit

to the distance between his/her location and start point. Also, the server cannot be forced to stop in search in driver for particular customer. These tasks need to be done parallel to other server tasks.

3. **Driver location tracking:** Sharing the location of driver continuously to rider is a quite difficult task. We need to have real time communication between the driver and rider. At the same time this should not cause load on the server

4. **Driver location estimation:** Sharing the location of driver continuously to rider is a quite difficult task. We need to have real time communication between the driver and rider. At the same time this should not cause load on the server

5. **Ride later:** Sharing the location of driver continuously to rider is a quite difficult task. We need to have real time communication between the driver and rider. At the same time this should not cause load on the server

1.4 Organization of the report

The rest of the report is organized in the following manner:

- **Chapter** 2 deals with the server and app setup like which architecture was used for server and app. It also contains details regarding various technologies and libraries used.

- **Chapter** 3 presents a detailed explanation of the approach we have developed to solve the problem detailed in section 1.2

- **Chapter** 4 deals with various results we have obtained using our application and analyse the results of our experiments. We will also discuss about the experimental setup used for testing the application on larger scale.

- Finally, in **Chapter** 5, we provide a brief summary on the work done as a part of **BTP** and some possible extensions which can be done for the project in future.

Chapter 2

Setup and technologies

This project uses several technologies and libraries / packages for the server and the android application. In this chapter, we will describe about various technologies, libraries and packages used for the server and the application.

2.1 Android App (Frontend)

For this project we have decided to use Android as it is one of the most popular mobile operating system among users in India. Android smart-phones being cheap nowadays, most of the people can easily afford them. We have decided to support the app for android versions greater than 5.0 (API 21) as now $\approx 89\%$ users owning an android phone have this OS as per the distribution chart [3].

2.1.1 Setup

As a part of this project, we have developed two android applications. One for the customer and other for driver partners. We have designed the application in an MVVM (**M**odel-**V**iew-**V**iew**M**odel) style architecture. This architecture separates out backend logic from the view components logic. Such a separation allows for better performance as compared to having both UI and backend logic together. In MVVM, the data required for UI is

fetched from View Models which reduces our worry for data update from server/cache. The ViewModel handles the network requests, database and cache read / writes without bothering the UI, thereby imporving performance.

2.1.2 Libraries

This application requires loading of maps for showing the location of customers / drivers. In order to display maps on the android app, we use `Google Maps API` [4]. For using this API, we need to create an account on google and create a project on google console. On successful creation of project, an API key is generated which should be used for using the Maps API. We can also restrict the usage of this API key to prevent misuse of the key in case. In order to determine the live location of either customer or driver, we have used `google play services location library`. This service provides us exact location of the user based on information fused from GPS and network providers

We have used `Volley` [5] for performing network related requests from app to server. Volley handles network requests in a background thread without freezing the UI. It has also option for bundling requests in a queue and executing them in FIFO order. But volley does not allow sending of body in HTTP requests `GET` and `DELETE`. So we have used a modified version of HttpStack and used that to comfortably send `GET` and `DELETE` requests from application.

Also, for the android app, we have used the latest `androidx` [6] libraries instead of the default support libraries which is supposed to become standard in future android releases. We have also used the latest Material components library for android instead of those which come with the support library. In order to design the layouts for android app we have mostly used `ConstraintLayout` as it proved to be efficient for rendering components as opposed to using RelativeLayouts and LinearLayouts [7].

We have also used a library called `CardStackView` [8] for displaying rides to driver as a stack of cards. This library displays a set of cards in stacked order, so that only one card

is visible at a time. This behaviour is much like cards in Tinder.

2.2 Server (Backend)

For our android application, we need a server for handling requests sent from application with appropriate responses. For this project, we have decided to use **ReST**ful (**Re**presentational **S**tate **T**ransfer) web services based HTTP server.

2.2.1 Setup

We have developed HTTP servers with ReST based architecture for handling requests from the application. ReST based architecture have proved to be advantageous for its simplicity and ease of use from user's end.

We have divided the whole server functionality in multiple microservices where each microservice is dedicated for specific functionality. Currently, we have designed the following microservices

- Authentication
- Profile
- Catalog
- Location updates
- Share auto
- Mini cab

We have used `POSTMAN` application for verification and testing of ReST API endpoints. We have also written using `chai` and `mocha` for automated verification of API endpoints.

2.2.2 Libraries

For the http server, we have decided to use the popular `ExpressJS` [9] framework which is a framework based on NodeJS for building web-applications using principles and approaches of NodeJS. `ExpressJS` is useful for handling `ReST API` CRUD requests. We use the `body-parser` library to parse input requests as JSON objects which can be used for performing calculations at the server side. We use the `method-override` library for performing `DELETE` requests from web-application side.

We are using `MySQL database` with `KnexJS` [10] SQL builder library. Using KnexJS eases the process of writing SQL queries through functions which map to query types.

We are using `axios` [11] npm package for communication among microservices.

We are using `Winston` [**?**] logging service for handling logs. Logs are very useful for debugging requests and analyze what has happened wrong at user's end.

We are using `nodemailer` [12] library configured with Gmail API for sending mails to users related to login codes, ride information etc.,

We are using `Redis` [13] for caching route based requests for faster response to certain queries like location updates.

2.2.3 Microservices

Microservices architecture has its own advantages and disadvantages. One of the major advantage being separation of functionality enabling easier scaling of service as opposed to monolithic architecture. Also, in this architecture style if any service fails, the whole backend does not affected. Only that service which has failed gets affected keeping the rest of other services up and running. The failed server for the service can be handled by other server as decided by load balancer. One of the possible disadvantages of this architecture is communication between services which can increase response time on some requests. Also, some failures caused due to integration of services can be a huge trouble.

Authentication

This service handles authentication related requests like generation of tokens, revoking of tokens and verifying tokens. Tokens are a good way for authentication requests.

We are using `jwt tokens` [14] (JSON web tokens) in association with `passportjs` [15] for token related functiond which are used for securing ReST APIS on the server side.

Profile

This service handles profile related requests like registration, login and updates. Both driver and customer have their own ReST API URLs for handling requests. We are using authentication service when updating details in profile.

We are using `bcrypt` npm library for encrypting passwords (to be stored in the database) and verifying passwords during login.

Share auto

This service handles all operations related to the share auto. This includes declaring availability of driver, taking customer request, allocation of rides, cancelling rides, starting and finishing rides

Mini cab

This service handles all operations related to the mini cab. This includes all operations available in share auto with a different implementation necessary for mini cab. This service additionally handles requests for booking rides at a later date and time.

Catalog

This service makes requests to other vehicle related services like share auto, mini cab and retrieves the information based on request.

Location updates

We are using `redis` [13] with `ioredis` [16] npm package for calculating location updates in case driver is unable to send them and customer is requesting from server.

Redis serves as a high performance cache which reduces our dependence on fetching data from database. Storing items which do not frequently change can be stored in redis cache.

2.3 API/Frameworks

In order to establish communication between various parties we are using a framework called `Realtime Framework` [17]. Using this framework, we can establish communication between two users in a publisher-subscriber mode. Such a model helps in instantaneously publishing and receiving messages using channels.

We are using `Open Route Service` [18] for determining routes between source and destination via way-points. This service also helps in determining the distance, estimated time etc for travelling along the route.

Chapter 3

Server

Designed server as a ReST based service. Each response for the request has the following format.

```
{
    // JWT token to be used for Secure APIs
    authorization: "Bearer <token-id>"
}
```

Listing 1: Response format for header for Secure APIs

```
{
    // A number indicating status of the operation
    status_code: "",
    // Message associated with the status_code
    message: "",
    // Detailed explanation for the success/failure of the request
    reason: "",
    // Additional information returned
    additional_info: {

    }
}
```

Listing 2: Response format for requests

Microservices are a software emgineering technique a service-oriented architecture (SOA) architectural style that organizes an application as a collection of small individual services. The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more efficient. It improves the parallability of the server as it is decomposed into smaller services. Microservices-based architectures enable continuous delivery and deployment. [19]

3.1 Authentication service

Security for any app is highly necessary. We tried to secure our apps by using JSON web tokens. The figure shown below clearly depicts the working of it. We have added authentication as a middleware. So for any endpoint first the json web token has to be

authenticated and then the rest functioning would be taken care.

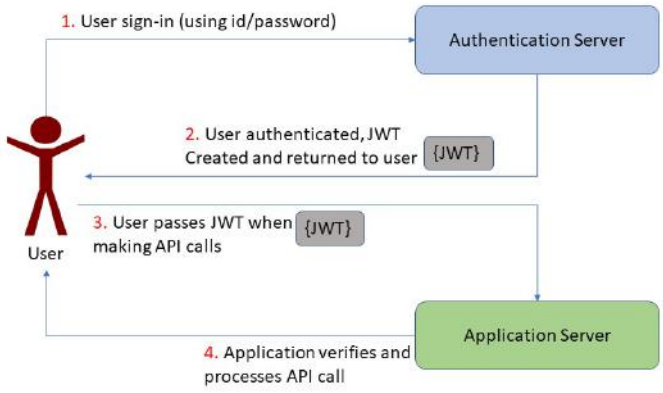


Fig. 3.1: Authentication process

3.2 Profile service

The profile microservice mainly takes care of both customer and driver personal records and details. This service handles the register, account verify, authenticate and login functionality differently based on the type of the user. The list of APIs handled by the profile service are listed below:

Method	Endpoint(/customer)	Description
POST	**/login**	Used to login into device for registered users
POST	**/register**	Used to register for first time users
POST	**/authenticate**	Authorises the users who are trying to log in
POST	**/verify**	Authorises the users who are trying to register

Table 3.1: Profile service customer endpoint

Method	Endpoint(/driver)	Description
POST	**/login**	Used to login into device for registered users
POST	**/register**	Used to register for first time users
POST	**/authenticate**	Authorises the users who are trying to log in
POST	**/verify**	Authorises the users who are trying to register

Table 3.2: Profile service driver endpoint

It manages four databases namely `dev_temp_driver`, `dev_temp_customer`, `dev_perm_driver` and `dev_perm_customer`. The temporary databases are primarly for all those users who have created account, but not yet verified. While the permanent databases store all the information of the verified users. This division of database enables us to maintain cleaner tables that do not contain unnecessary details.

3.3 Catalog service

The catalog microservice as its name suggests is used to list all possibilities. It is used for retrieving information about the drivers existing nearby from all the vehicle services. It is also used to get information about all the rides taken in the past(also from all vehicle services). The list of APIs handled by the catalog service are listed below:

Method	**Endpoint(/customer)**	**Description**
GET	**/rides**	Retrieve information about all the rides undertaken
GET	**/catalog**	Retrieve information about the nearby available rides

Table 3.3: Catalog service customer endpoint

Method	**Endpoint(/driver)**	**Description**
GET	**/rides**	Retrieve information about all the rides undertaken
GET	**/catalog**	Retrieve information about the nearby available rides

Table 3.4: Catalog service driver endpoint

This service uses the vehicle databases for retrieval of information. They are `dev_share_auto` and `dev_mini_cab`. Based on the result obtained from both the databases and merges accordingly.

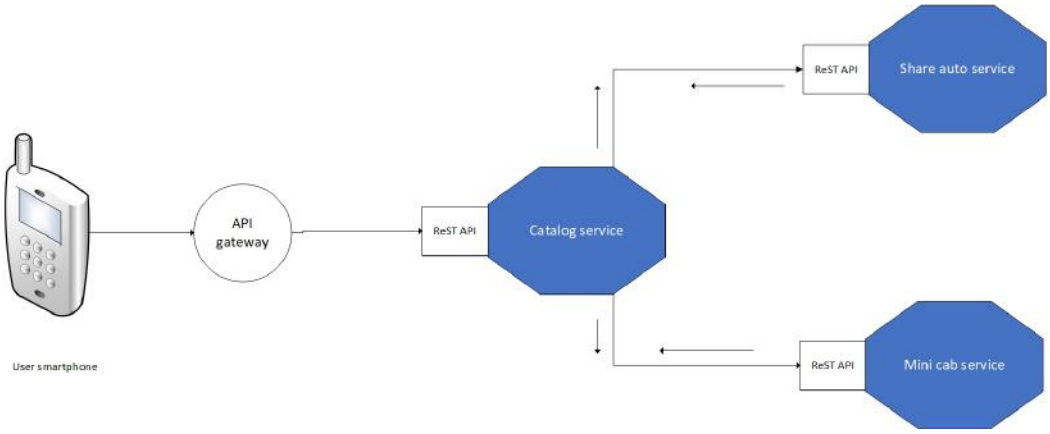


Fig. 3.2: Catalog process

3.4 Location updates service

The location updates service is used for handling the location update request for customer. We are collecting certain information from driver during rides like speed at a location and storing them in the database. This information is useful when driver is unable to communicate his location to customer and then customer requests his approximate location from the service. So, once the ride starts, all information related to the ride like ride_id, route, latest location, current average speed along the route are stored in `redis` cache. If a customer requests location of driver, we calculate the estimated location of driver using information stored in redis cache. For calculating position, we need speed and time. Time is retrieved based on interval at which customer is requesting for updates. If a speed entry is present in the database at any nearby location of the driver's last available location, we use that value or else we use the average speed along the route as retrieved from open route map service / redis cache. Using speed and time, we can calculate the distance which the driver would have travelled (in case he's travelling at the same average speed and has not stopped mid way) in the time interval and estimate the position on map.

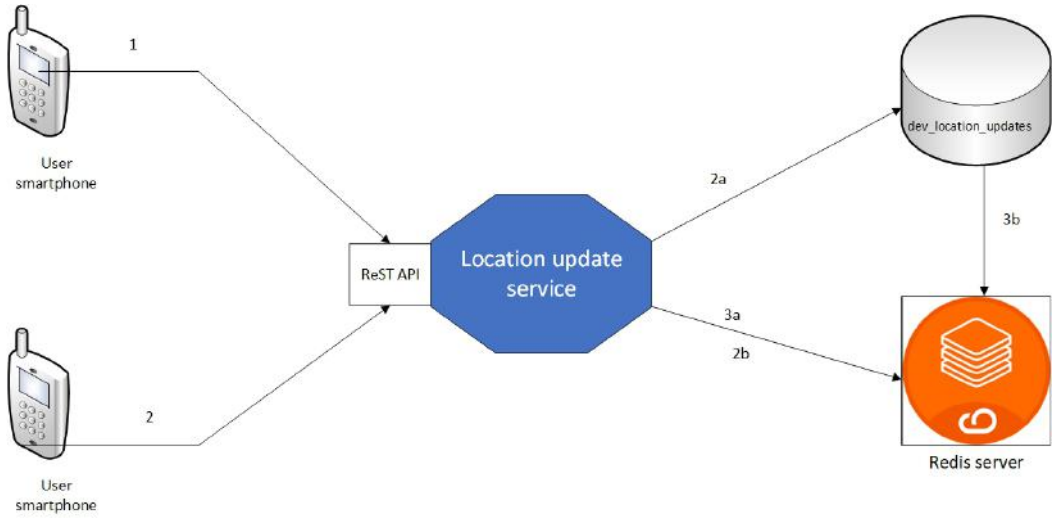


Fig. 3.3: Location update process

3.5 Vehicle service

A lot of vehicle types can be used for ride application. Each user has a particular interest of choice of the vehicle he/she rides in. Also the drivers also have different types of vehicles. In this project we have mainly two types of vehicles: share auto and mini cab

3.5.1 Share auto service

Share auto service is the main backbone of the entire project. It handles all the functionality starting from placing a ride to ride allocation. This service handles the post ride tasks that include collecting feedback. It handles many APIs incoming from both customer and driver app. They are listed below

Method	Endpoint(/customer)	Description
POST	**/assigned-ride**	Retrieve information about assigned ride(if any)
POST	**/cancel-ride-with-feedback**	If ride cancelled by customer collect feedback
POST	**/catalog**	Retrieve information about vehicle services available nearby
POST	**/feedback**	After the ride is finished collect feedback about the ride
POST	**/request-rides**	Places a request for ride
POST	**/request-rides-later**	Places a request for ride to be taken later
DELETE	**/request-rides**	Retrieve information about all the rides undertaken
POST	**/rides**	Retrieve information about all the rides undertaken

Table 3.5: Customer end points description

Method	Endpoint(/driver)	Description
POST	**/accept-ride**	Accept customer ride request and add information to rides table
POST	**/accept-ride-later**	Accept customer ride request and sends an e-mail to the customer about the journey
POST	**/cancel-ride-with-feedback**	If ride cancelled by driver collect feedback
POST	**/feedback**	After ride is finished collect feedback about the ride
POST	**/finish-ride**	Update information about ride after the ride finishes
POST	**/look-for-rides**	Search for any rides available nearby
POST	**/reject-ride**	If the driver is not interested in the ride request then reject
POST	**/ride-available**	The drivers makes himself available for rides
DELETE	**/ride-available**	The drivers makes himself unavailable for rides
POST	**/start-ride**	Update information about ride after ride starts
GET	**/rides**	Retrieve information about all the rides undertaken

Table 3.6: Driver end points description

All the information required for the functionality of this microservice is stored in `dev_share_auto` database.

3.5.2 Minicab service

Mini cab service is an important aspect of the entire project. It handles all the functionality starting from placing a ride to ride allocation. This service handles the post ride tasks that include collecting feedback. It handles many APIs incoming from both customer and driver app. It also has the feature of taking rides later along with instantaneous rides. They are listed above.

All the information required for the functionality of this microservice is stored in `dev_mini_cab` database.

Chapter 4

Android application

In this chapter, we will explore in detail the working of the android application. As discussed in Chapter 2, we have developed two applications. One for customer and other for driver. Separation of applications for customer and driver helps in developing features faster.

4.1 Saving session information

In order to prevent login of user into app whenever the app is started, we save the login information of the user in shared preferences file. So, whenever the user opens the app, we check if there is any session information file on device. If any such file exists we straightaway take the user to main activity, else we take him to the login activity.

4.2 Fragmentised application

A Fragment represents a part of UI in a FragmentActivity. One can combine multiple fragments to create a multi-panel UI and can reuse the same fragment in several other activities. Fragment is a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities). Both the apps(driver and

customer) include a lot of fragments. Some of the fragments used in customer application are

Name of fragment	**Purpose**
AssignedRide	Displays the details of assigned driver
CancelRideFeedback	Provides a list of possible reasons for cancelling the ride
ConfirmRequestRideLater	Used to set details of the ride to be taken later like number of seats, date and time
ConfirmRequestRideNow	Used to set details of ride to be taken immediately
Dashboard	Profile related things like update password can be viewed and modified
FinishRideCustomerFeedback	After the ride is finished displays the fare and also takes feedback about the ride
Maps	Loads the map to be used for identfying the route and location of driver and various other purposes
PickupDrop	Used to display the address of the pickup and drop locations based on the pickup and drop pins
Profile	
RideRequest	Used to change the availability or non availability of driver for rides
SelectDateTime	Used to select date and time of the ride to be taken later
SelectNumberOfSeats	Used to select number of seats of the ride
YourRides	Displays all the rides undertaken in a recycler view

Table 4.1: Description of fragments used in customer app

Fragments used in the customer application are

Name of fragment	Purpose
AvailableRidesInfoToDriver	Displays the available customer requests in a card stack view
CancelRideFeedback	Provides a list of possible reasons for cancelling the ride
ConfirmedRides	Displays all the rides currently undertaken in a viewpager
Dashboard	Profile related things like update password can be viewed and modified
FinishRideDriverFeedback	After the ride is finished displays the fare and also takes feedback about the ride
Maps	Loads the map to be used for identfying the route and location of customers and various other purposes
Profile	
TakeRides	Used to change the availability or non availability of driver for rides
YourRides	Displays all the rides undertaken in a recycler view

Table 4.2: Description of fragments used in driver app

4.3 Viewmodels

As described in the earlier chapters, we have used view models for communicating data between various fragments. The lifecycle of the viewmodel helps us to pass information from a previously existing fragment to fragment recently created. The various view models used are:

Name of viewmodel	Purpose
PickupDrop	Used to store information about the last available location of the user
RestAPIResponse	Used to observe the response for ReST APIs which allows for parallel execution
ConfirmedRideInfo	Stores information about the assigned ride request

Table 4.3: View models used in customer app

Name of viewmodel	Purpose
PickupDrop	Used to store information about the last available location of the user
RestAPIResponse	Used to observe the response for ReST APIs which allows for parallel execution
ConfirmedRideInfo	Stores information about the all the alloted rides

Table 4.4: View models used in driver app

4.4 Services

4.4.1 Google maps service

The most important element of a ride booking application is maps. One needs to fix both pickup and drop locations. Also the driver needs to know the route to pickup and drop points of customer. For all these purposes we need maps service.

We have used Google maps service(developer version) for this purpose. It has many advantages. We can add customised markers to the map layout. It allows us to draw routes on the map that can used to guide the driver to appropriate destination. The Google Maps service uses HTTPS encyption that makes the app even more secure.

4.4.2 Live Location updates service

Another key feature is decoding the live location of the user on the go. It is essential to find out customers/drivers nearby for better prospects. So the information about the current location of the user is highly essential. We have used Google Mobile Service(GMS) location service.

It provides the best recent location presently available. If a location is not available, which is quite rare, null is returned. The best accuracy available while respecting the location permissions will be returned. This information is stored in viewmodels on location changed listener and is used by various other fragments.

4.4.3 Address decoding service

Pickup and drop location are necesary for any ride. But using the location in LatLng format does not make any sense to the user. The users clearly cannot understand the location when the above format is used.

To make things easier, we have used the address decoder service. This service makes use of **geocoding** technique. Geocoding is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coordinates (like latitude 37.423021 and longitude -122.083739), which you can use to place markers on a map, or position the map. So now the users have an idea of the location being marked on the map as the address of latest location is displayed continuously.

4.4.4 Network requests (Volley)

The main functional and processing unit of the entire project is the server. As described in previous chapter, it manages all the functionality of the project. The apps requests the server for various information. These are done by the sending API requests.

Volley is HTTP library that we have used in this project for sending ReST API requests. Volley has a lot of added benefits. It allows multiple concurrent connections, parallel requests and can prioritize the request. We have primarily used GET, POST and DELETE methos for various ReST API end points.

However we had faced a small issue while using GET and DELETE methods. By default volley prevents the data a part of params from being sent. So we had override some of the methods in volley library.

4.5 Notifications

A notification is a message that are displayed outside your app's UI to provide the user with reminders, communication from other people. We are displaying some notifications

related to various activities. It is highly recommended to display notifications.

Some of the notifications shown in customer app are:

Notification type	Purpose
LOOKING_FOR_DRIVERS	Notification displayed when driver is looking for nearby customers
DRIVER_ASSIGNED	Displays a notification with few details of the customer
RIDE_CANCELLED_BY_CUSTOMER	Displays the reason why customer cancelled the ride after allocation
RIDE_CANCELLED_BY_DRIVER	Displays the reason why driver cancelled the ride after allocation
START_RIDE	Notification displayed after the ride is started
JOINED_RIDE	The name of co-passenger is displayed when new customer joins the ride
FINISH_RIDE	Notification displayed after the ride is finished

Table 4.5: The purpose of various notifications displayed in customer app

Some of the notifications shown in driver app are:

Notification type	Purpose
LOOKING_FOR_CUSTOMERS	Notification displayed when driver is looking for nearby customers
CUSTOMER_ASSIGNED	Displays a notification with few details of the customer
RIDE_CANCELLED_BY_CUSTOMER	Displays the reason why customer cancelled the ride after allocation
RIDE_CANCELLED_BY_DRIVER	Displays the reason why driver cancelled the ride after allocation
START_RIDE	Notification displayed after the ride is started
FINISH_RIDE	Notification displayed after the ride is finished

Table 4.6: The purpose of various notifications displayed in driver app

4.6 Channel communication

Depending solely on the server for communicating information across users is highly unrecommended. It increases the load on the server. So, we have tried to reduce the load on server by using other means of communication between the apps.

Once a ride has been assigned the end parties now no longer communicate through server. We have used a realtime framework that helps in communicating information across devices. All the users have their own channels on which they receive messages. These messages are further decoded to get relevant information.

The various types of messages being sent across the apps are

Message Type	Purpose
LOCATION_UPDATE	The latest location of the user is sent to the user at the other end of the channel
RIDE_CANCELLED_BY_CUSTOMER	The status of ride is sent along with the reason why customer cancelled ride to driver
RIDE_CANCELLED_BY_DRIVER	The status of ride is sent along with the reason why customer cancelled ride to driver
START_RIDE	The status of the ride is communicated to the customer
JOINED_RIDE	The name of co-passenger is displayed when new customer joins the ride
FINISH_RIDE	The status of the ride is communicated to the customer along with the fare

Table 4.7: The various types of messages being sent across the apps

4.7 TimerTasks

A timer task is a event that is executed once, twice or made to run repeatedly until it is stopped. The timer task can be made to run at sequential intervals unless the timeout reaches or the desired event occurs.

There are many scenarios in the application where timer tasks has been used. Location updates are sent continuously until the ride ends. This is done with the help of a timer task. Look for rides endpoint by the driver is run in a timer task. This timer task runs until driver gets a non null response, ie, driver gets a non empty list of customer rides available nearby.

Similarly in the customer app too, after the ride is placed we use a timer task. It repeatedly checks if the customer has been allocated a driver until the timeout. On timeout the customer ride request is invalidated and customer needs to request for rides again.

In other important feature of the app, tracking of driver in case of non-existance of internet connectivity at the driver's end also timer task has been quite useful. The user expects a location update at a fixed regular interval. So, incase we do not receive location update after significant time, we treat it as internet connectivity failure and switch to location estimator explained later.

4.8 Customer app

4.8.1 Login / Registration / Authentication

The app starts with a blank screen where users either registers themselves or logs in to their respective accounts. Login is followed by authentication where user is asked to enter the code snet to him via e-mail. This is actually done for preventing misuse of an account by others. Similarly, after register users need to verify their account too.

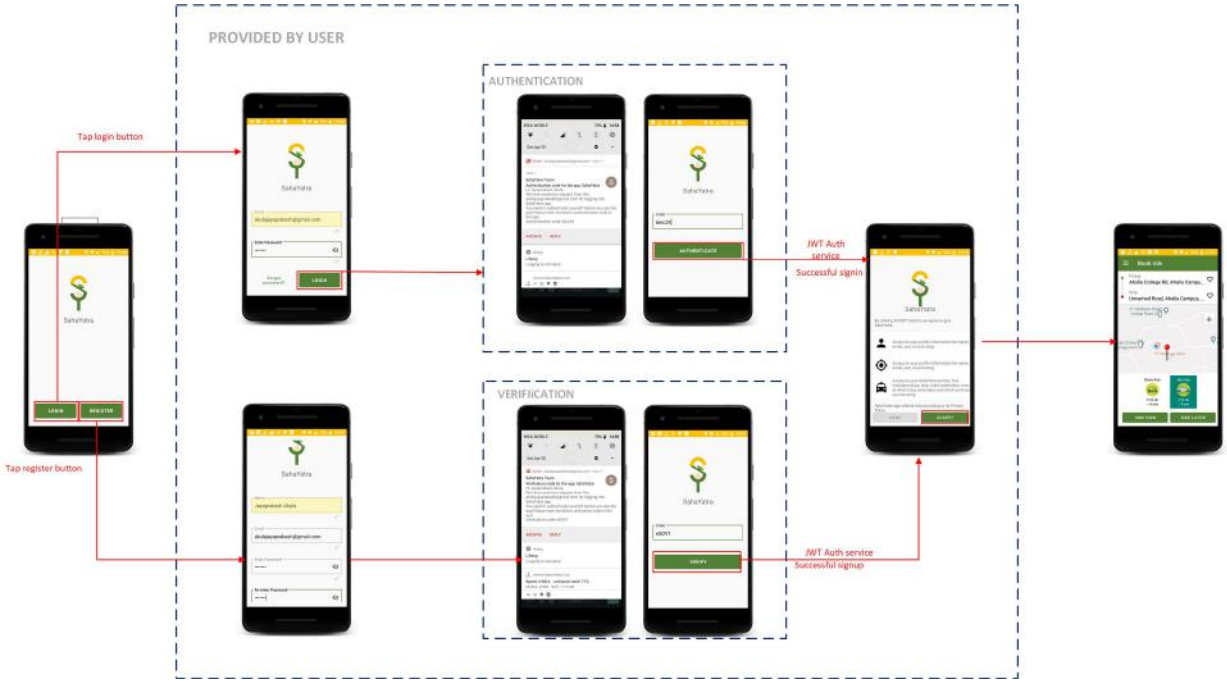
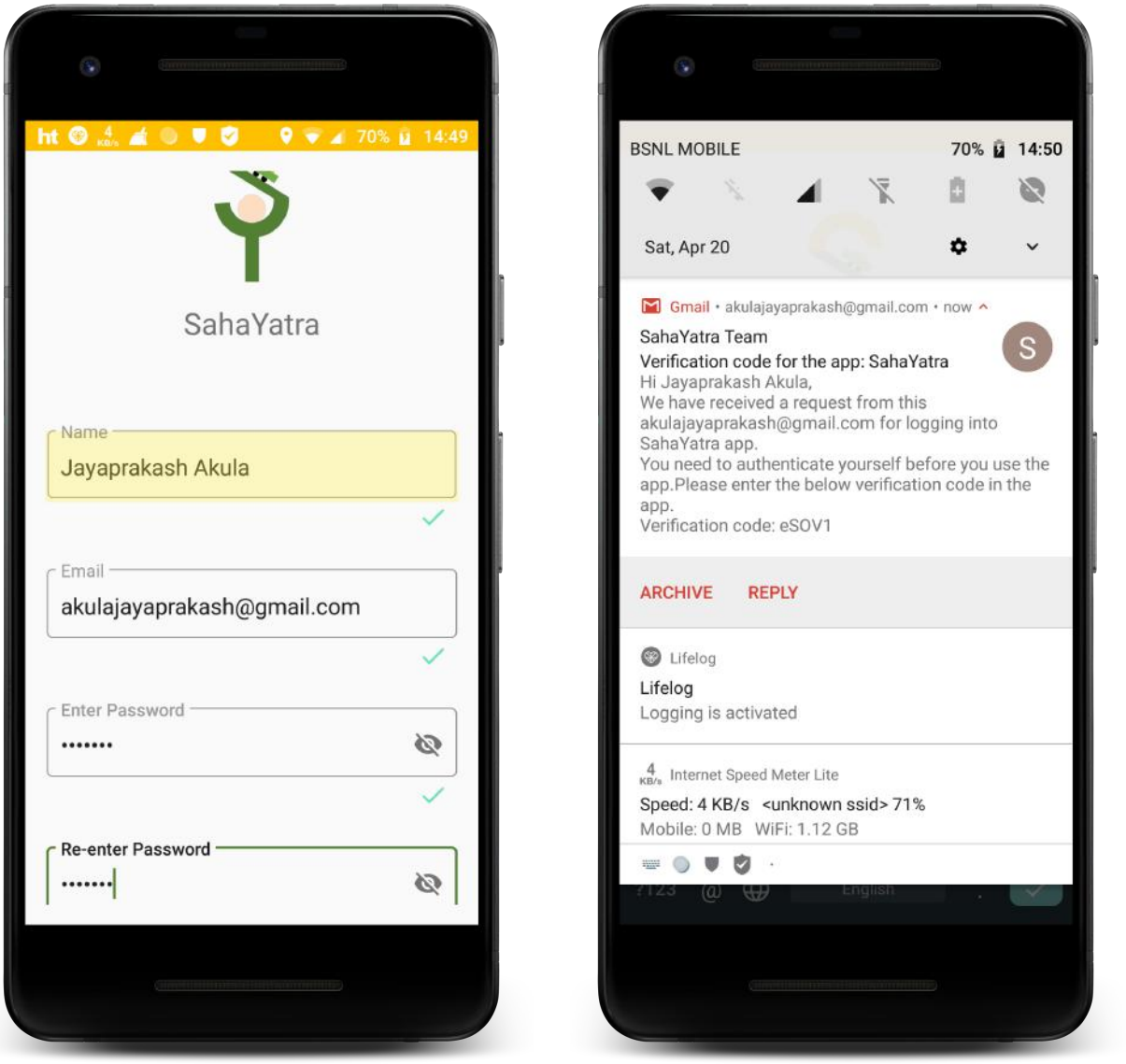
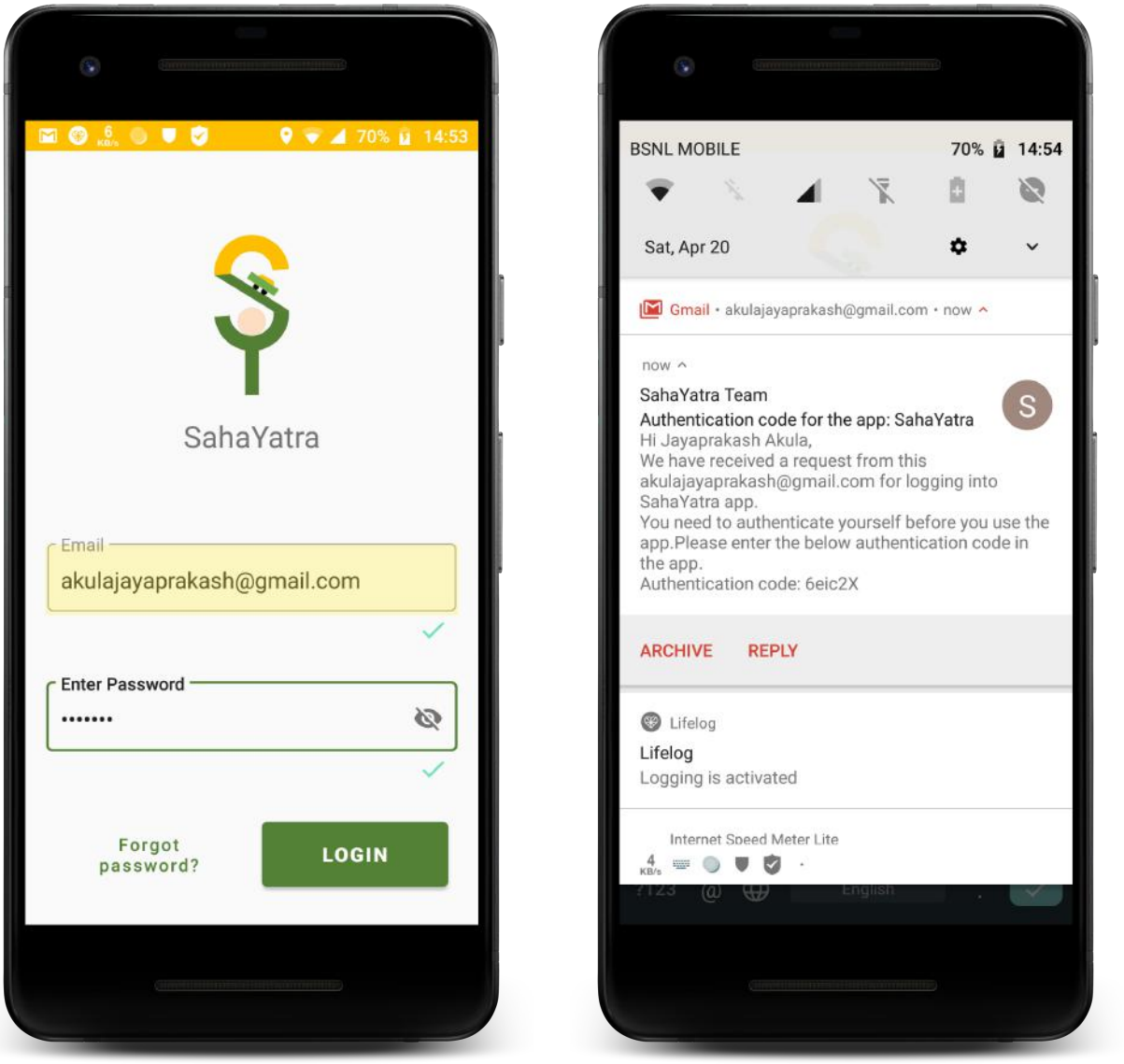


Fig. 4.1: Authentication/signup workflow



(a) Register screen **(b) Verification screen**

Fig. 4.2: Register & verification user screen.



(a) Login screen **(b) Authenticate screen**

Fig. 4.3: Login & authenticate user screen.

4.8.2 Ride now/ Ride later

The customer after registering himself/herself is now able to book rides for himself. This begins with marking pins for pickup and drop location. After they are set a list of all vehicular services available nearby are shown. Based on the mode of the ride ie, ride now or ride later different fragments open up confirming the ride booking. The customer could then select number of seats or the date and time (in case of ride later). Finally he/she

presses the request ride button that starts the required processing at the backend.

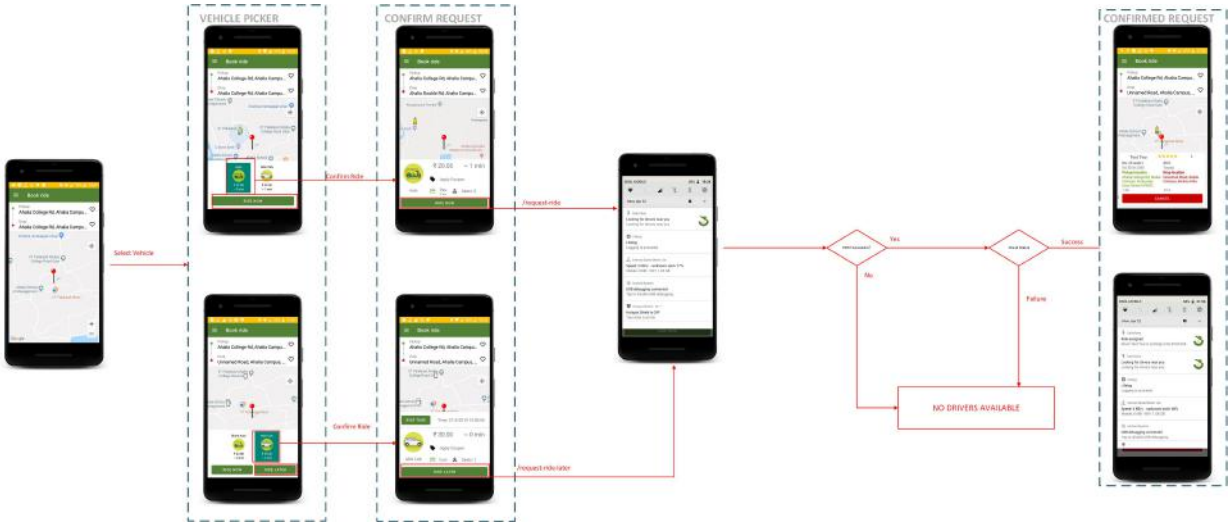


Fig. 4.4: Request ride initiation workflow

4.8.3 Cancellation

The customer could cancel the ride due to some reasons which he is bound to mention in the cancel rides feedback fragment.

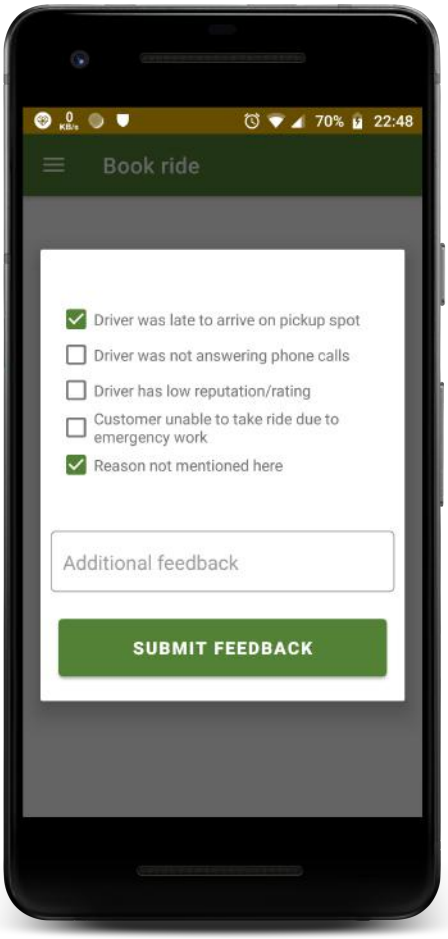
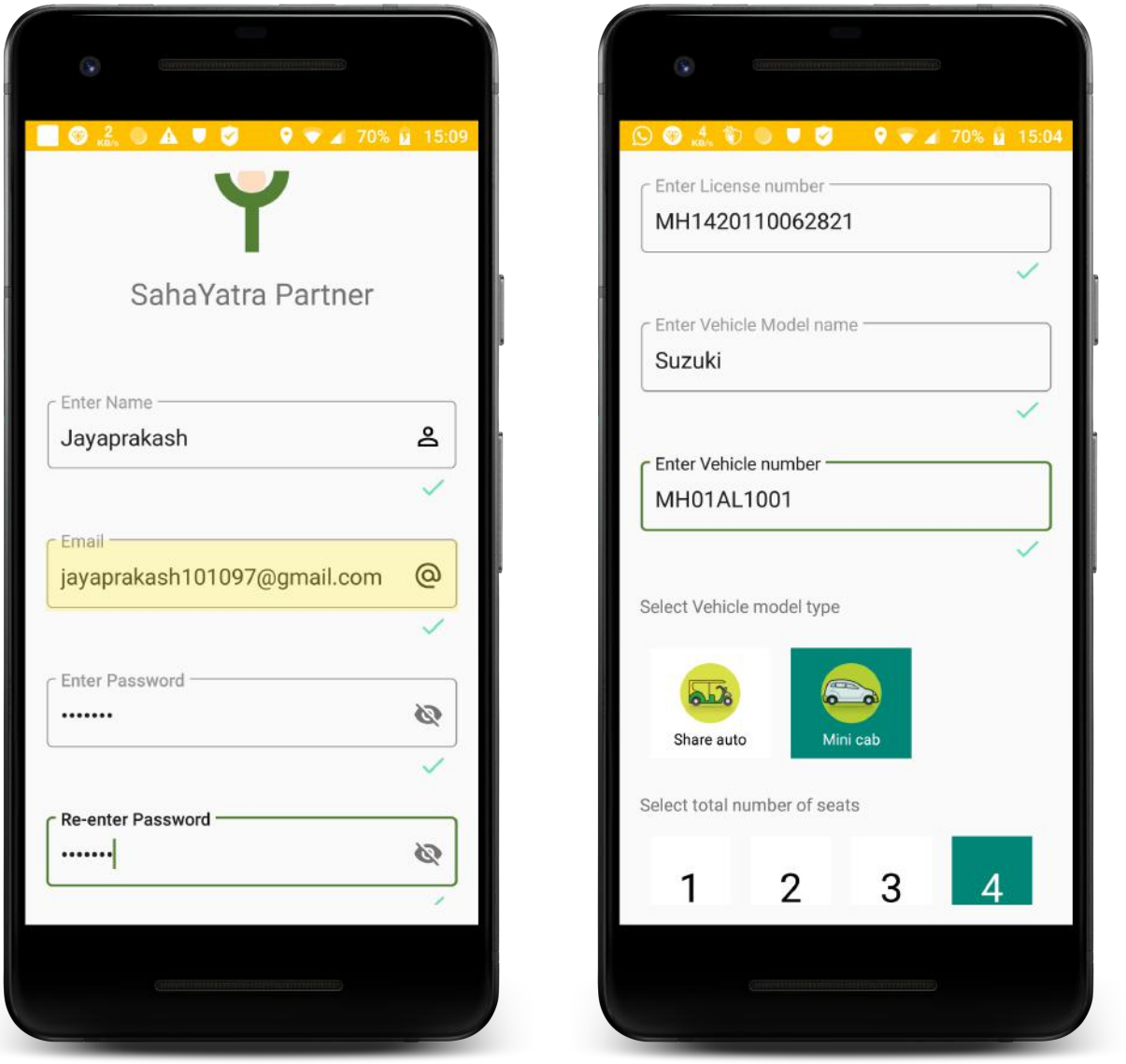


Fig. 4.5: Customer cancel ride

4.9 Driver app

4.9.1 Login / Registration / Authentication

The app starts with a blank screen where users either registers themselves or logs in to their respective accounts. Login is followed by authentication where user is asked to enter the code snet to him via e-mail. This is actually done for preventing misuse of an account by others. Similarly, after register users need to verify their account too.



(a) Register screen **(b) Register screen**

Fig. 4.6: Register user screen

4.9.2 Available for rides

The driver has a switch that enables him to toggle between ready to take rides and busy. When the switch is enabled, request is sent updating availability status of driver.

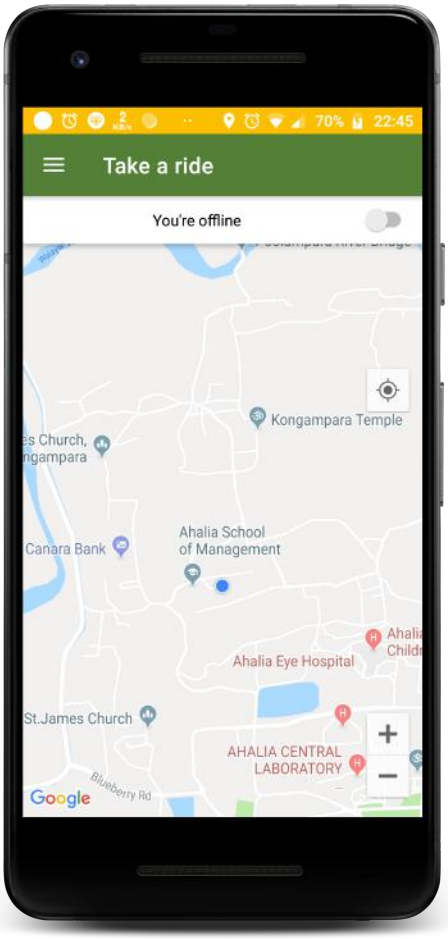
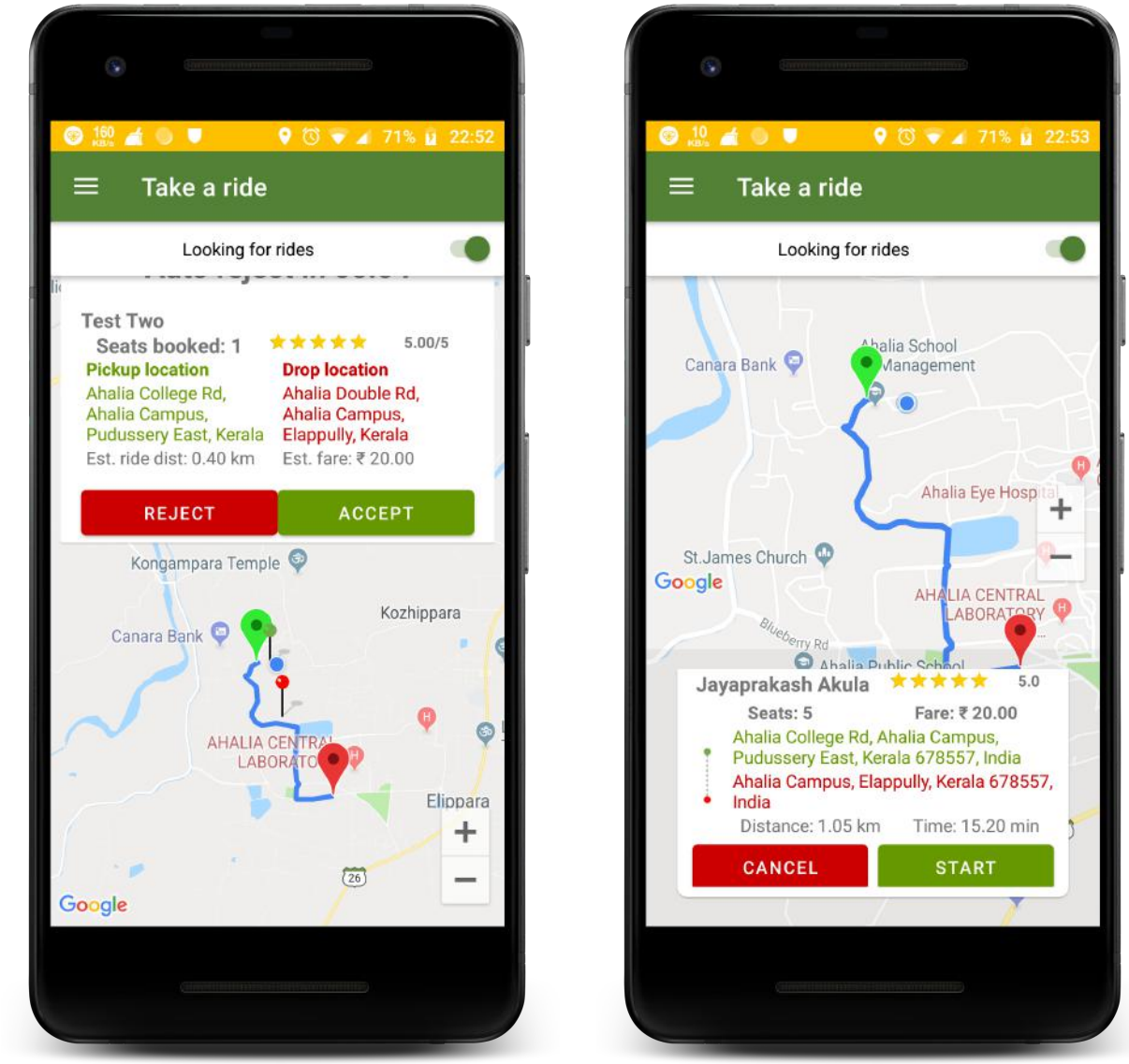


Fig. 4.7: Driver available for rides

4.9.3 Ride selection/rejection

A list of customers is displayed in the drivers app in a card stack view. A few requests would be received, meanwhile blocking these requests. So, that no two drivers can accept same ride. However to prevent excessive blocking, the rides are rejected based on countdown timer. He/she can accept a ride by swiping right or pressing accept button and reject by left swipe and vice versa. When a particular ride gets accepted requests are made that create entry for accepted ride and the rejected rides are again unblocked.



(a) Driver rides available **(b) Ride accepted**

Fig. 4.8: Driver ride selection

4.9.4 Cancellation

The driver could cancel the ride due to some reasons which he is bound to mention in the cancel rides feedback fragment.

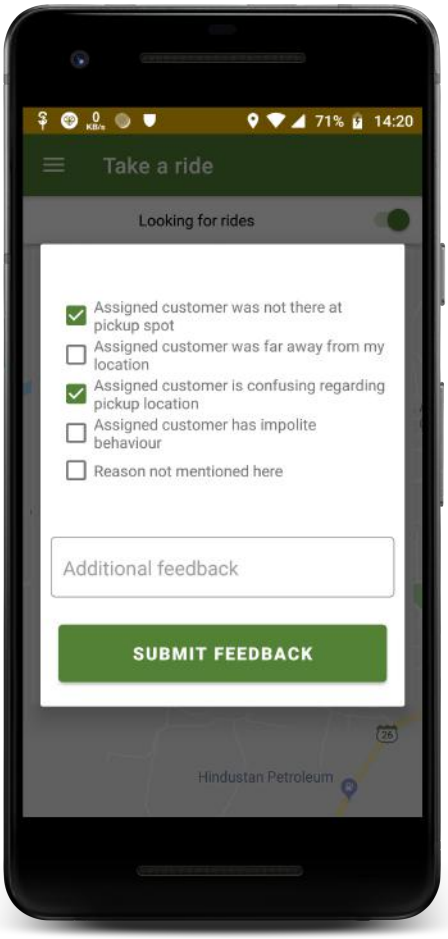


Fig. 4.9: Driver ride cancelled

4.10 Ride progress

When a particular ride starts and the driver is still in search of other rides as his/her vehicle as some empty seats left, then rides would be further requested with updated available number of seats. We then receive a list of ridesm from which certain rides are filtered as they might require the driver to deviate a lot from the main route. Only rides that lie in vicinity of driver's location and whose endpoints either lie in the previously existing estimated route or previously existing rides estimated route endpoint lie on the thier route are taken into consideration. He/she can take rides as long as there are empty seats still available. All the undergoing rides are displayed in the viewpager below.

The driver can start ride only if the customer is in a 100 m radius, until then the button

is disabled. Similarly the finish ride button also gets activated in the vicinity of the drop location.

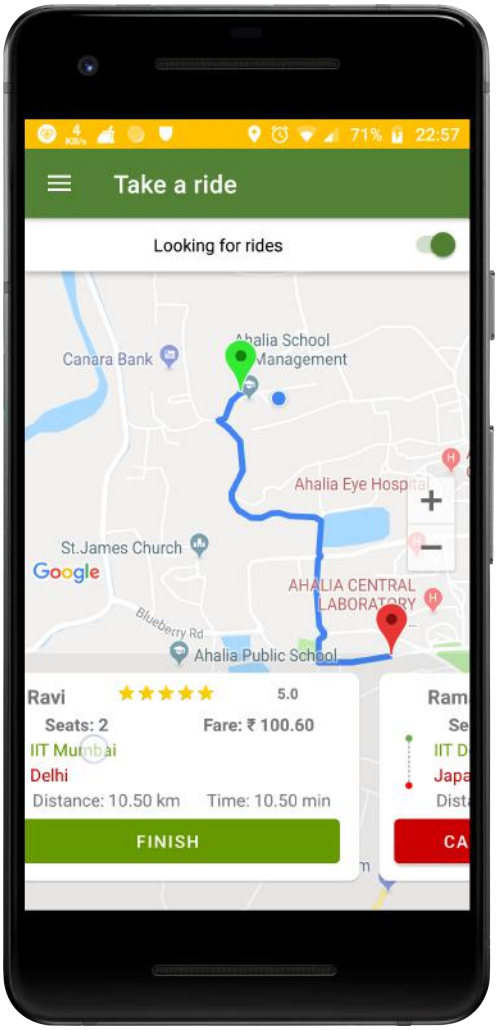


Fig. 4.10: Driver ride progress

4.11 Fare calculation

The application calculates the fare for the ride by taking into consideration many details. The ride pricing per km is divided into to two parts: Base fare and Additional fare. The base fare is the fare for the initial 1.5km of the ride. It is the least amount the customer has to pay even if he/she had travelled less than 1.5km. The additional fare is the per km charge for the remaining distance(total distance - 1.5km).

The base fare and additional fare are also dynamic. They change according to time ie, early morning, morning, afternoon, evening, night and late night. The fare increases in the times when generally lesser number of drivers would be available, ie early morning and late night. The fares also depend on the fact if it is weekend or weekday. The prices are usually less on weekdays compared to weekends.

In case the ride is being shared, the entire fare would not have to be paid by all the customers neither does the total amount of money received by driver remains the same. So the fare when the ride is shared by `n` customers is $\frac{n}{n+1} *$ Fare if travelled alone

4.12 Completion of ride

The ride completes when the driver presses the finish button on his/her app. The driver is under no obligation by the customer to travel further, no matter what the customer does. To prevent misuse of the finish ride button. The button activates only when the driver is located in a 100m radius of the drop location.

This is followed by customer and driver giving their feedback about the ride. The ride fare is calculated by the app in the method described earlier. Currently the app supports only cash based payment.

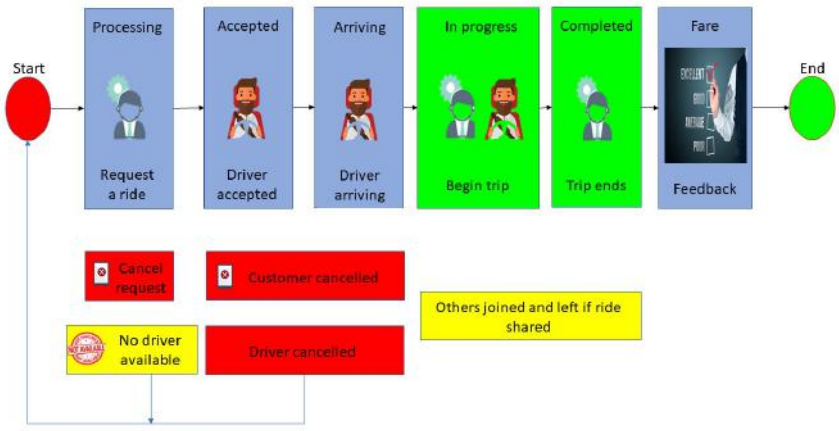


Fig. 4.11: Android app workflow

Chapter 5

Conclusion & Future Work

5.1 Conclusion

In this project we have a developed a robust ride sharing application that is equally suitable to both rural and urban landscapes. The app is highly flexible and the riders get to choose their choice of vehicles based on a estimate. The ride allocation algorithm is highly optimised and the best possible matching is done. All the possible scenarios of errors have been handled. Though the app may have its own limitations in the number of vehicular services, but they can be easily added.

5.2 Future work

Currently the process of selecting the pickup and drop location is based on the pickup and drop marker. It can be extended to choosing location on the basis of address. This process is actually called geocoding. **Geocoding** is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coordinates (like latitude 37.423021 and longitude -122.083739), which we can use to place markers on a map, or position the map.

The app at the time of submission of project features cash as the only mode of payment.

It can be extended to include various other cashless transfers too.

The locationn estimation can be improvised by using better techniques. Currently it just focuses on driver's past experiences at those particular location. It can be generalised and several other factores like traffic data could also be used.

At the current state of application does not do any sort of caching the data. Some of the requests can be cached to improve the performance of the app. For ex, your rides endpoint can be cached. If any updates are present, then we can update them.

As desribed earlier microservice makes horizontal scaling easier. In case of heavy traffic on the server, we can create new instances of those services. To maintain proper load on all these new servers, we need to have a load balancer in place to distribute load among these services.

The project currently has two vehicular services mini cab and share auto. We can create many other vehicular services lke prime, sedan, auto and some other rental services too.

The UI can be improved by reading about the latest technological developments in android section that could probably improve the performance of the app. The server needs more testing. Only certain services have been tested completely using

References

[1] C. D. HQ, "What is Rideshare? How Ridesharing Companies Like Uber Work ," https://commercialdriverhq.com/what-is-rideshare/, [Online; accessed 10-April-2019].

[2] Statistica, "Ride Hailing," https://www.statista.com/outlook/368/119/ride-sharing/india, [Online; accessed 10-April-2019].

[3] Google, "Distribution dashboard," https://developer.android.com/about/dashboards/, 2018, [Online; Last accessed 19-November-2018].

[4] Android, "Maps SDK for android," https://developers.google.com/maps/documentation/android-sdk/intro, 2018, [Online; Last accessed 19-November-2018].

[5] J. Davidson, "Volley," https://developer.android.com/training/volley/index.html, 2019, [Online; Last accessed 08-April-2019].

[6] A. Developers, "AndroidX Overview," https://developer.android.com/jetpack/androidx/, 2018, [Online; Last accessed 19-November-2018].

[7] T. Hagikura, "Understanding the performance benefits of ConstraintLayout," https://android-developers.googleblog.com/2017/08/understanding-performance-benefits-of.html, 2018, [Online; Last accessed 19-November-2018].

[8] Y. Kaido, "CardStackView," https://github.com/yuyakaido/CardStackView, 2019, [Online; Last accessed 08-April-2019].

[9] ExpressJS, "ExpressJS Documentation," https://expressjs.com/en/starter/hello-world.html, 2018, [Online; Last accessed 19-November-2018].

[10] T. Griesser, "KnexJS docs," https://knexjs.org/, 2018, [Online; Last accessed 19-November-2018].

[11] N. U. R. N. Emily Morehouse, Matt Zabriskie, "Axios Node package," https://github.com/axios/, 2019, [Online; Last accessed 08-April-2019].

[12] A. Reinman, "Nodemailer Documentation," https://nodemailer.com/about/, 2018, [Online; Last accessed 19-November-2018].

[13] "Redis," https://redis.io/, 2019, [Online; Last accessed 08-April-2019].

[14] C. by Auth0 Developers, "JSON Web tokens," https://jwt.io/, 2019, [Online; Last accessed 08-April-2019].

[15] J. Hanson, "Passport JS," http://www.passportjs.org/, 2019, [Online; Last accessed 08-April-2019].

[16] "IO Redis Node package," https://redis.io/, 2019, [Online; Last accessed 08-April-2019].

[17] (2018) Realtime framework documentation. https://framework.realtime.co/messaging/#android. [Online; Last accessed 19-November-2018].

[18] OpenRouteService, "API interactive examples," https://openrouteservice.org/dev/#/api-docs/directions, 2018, [Online; Last accessed 19-November-2018].

[19] "Microservices," https://en.wikipedia.org/wiki/Microservices, 2019, [Online; Last accessed 17 April 2019].