UNITE ON WHEELS

A Car Pooling Platform

Software Requirements Specification

0.1

30-05-2020

Adarsh Baghel
Ruchir Mehta
Namani Sreeharsh
Shaikh Ubaid

Prepared for
CS 258 Software Engineering
Spring 2019

Revision History

Date	Description	Author	Comments
20-2-2020	Version-1.0.0	A-R-H-U	UI design Implementation
15-3-2020	Version-1.2.0	A-R-H-U	UI Improvements
20-5-2020	Version-2.0.0	A-R-H-U	Backend Implementation
29-5-2020	Version-2.2.0	A-R-H-U	Combining UI and Backend

Document Approval

The following Software Requirements Specification has been accepted and approved by the following:

Signature	Printed Name	Title	Date

Table of Contents

1. Introduction

This section gives a scope description and overview of everything included in this SRS document. Also, the purpose for this document is described and a list of abbreviations and definitions is provided.
Since inappropriate planning of the cities, there has been a big problem of traffic in most cities of India. People spend much of their time in traffic every day. In Addition to this many vehicles in traffic makes rapid oil consumption, there has been an uprising problem of air pollution. Oil supplies are very limited all over the world and oil prices are extremely expensive in our country. Therefore, most of the people have to take buses and since the number of public transportation vehicles are not sufficient, they travel under uncomfortable conditions. There are some attempts to solve these problems, however, they focus only on intercity transportation. We came up with an effective solution as www.uniteonwheels.com .Our project will be used for both intercity and urban transportations all over India. As a result, our system will be designed to solve these problems and deficiencies of other systems .

1.1 Purpose

The purpose of this document is to give a detailed description of the requirements for the "Car Pooling (UniteOnWheels)" (CP-IITI) software. It will illustrate the purpose and complete declaration for the development of the system. It will also explain system constraints, interface and interactions with other external applications. This document is primarily intended to be proposed to a customer for its approval and a reference for developing the first version of the system for the development team.

1.2 Scope

The "*Car Pooling for IITI*" is a GPS-based mobile application which helps people to find the closest vehicle based on the user's current position and other specifications like time of departure, price, contact information of passengers travelling and number of seats available. This information will act as the basis for the search results displayed to the user. The application should be free to download from either a mobile phone application store or similar services. Furthermore, the software needs both Internet and GPS connection to fetch and display results. All system information is maintained in a database, which is located on firebase(an online db service). The application uses the mobile phone's GPS navigator. By using the GPS-Navigator, users can view available vehicles on a map and can navigate towards them. The application also has the capability of representing both summary and detailed information about the available vehicles which will take the passenger to the required destination .

This application will bring about a big revolution in sharing vehicles thus reducing pollution and traffic in cities. This will be operated from both the passenger intended to travel and the passenger already travelling in a vehicle and willing to share their ride.

1.3 Definitions, Acronyms, and Abbreviations

TERMS	***DEFINITION***
CLUSTER	*Group of people travelling in the same automobile.*
CLUSTER ADMIN	*The person who acts as a gateway for communicating about the whereabouts of the cluster with the prospective passengers.*
CLUSTER MEMBERS	*All members of the cluster including cluster admin.*
FAVOURITE MODE	*Mode which shows the positions(locations) of the connected friends only. (future work)*
PUBLIC MODE	*Mode which enables the users to track the routes of all the available clusters (inclusive of our favourite people).*
TOGGLE BUTTON	*Mode which toggles visibility of the cluster's route to preferred people and anonymously to the public.*
REQUEST	*the request made by new members joining the cluster. Which is approved by cluster admin.*

1.4 References

1) *IEEE STD 1233-1998, IEEE Guide for Developing System Requirements Specifications*
2) *IEEE STD 830-1998, IEEE Recommended Practice for Software Requirements Specifications*
3) *https://senior.ceng.metu.edu.tr/2014/such/documents/SRS.pdf*
4) *https://pub.dev/packages (Flutter packages)*

1.5 Overview

(1) In the rest of the SRS we have discussed the user and system, constraints and requirements. Further we will discuss the limitations of the application. Getting started documentation has also been mentioned in this SRS.

(2) SRS is organised in a way firstly describing the features offered by our application moving to the user, system constraints and some general constraints.

2. General Description

2.1 Product Perspective

We have disseminated this section into Product Functions, User Characteristics, Product Perspective and some General constraints.

2.2 Product Functions

***#Sign Up**: Users need to sign up to use the app. The users should have a username and phone no. After filling their name, phone and correct OTP information, they register into the system.*

***#Sign In**: If a user is signed up, s/he can sign in the system by filling phone no in the phone input box, and completing the otp procedure.*

***#Sign Out**: A user may need to sign out of the system. He/She can do it by clicking the sign out button which is placed on the User profile page.*

***#Creating clusters to the database**: Our application also needs user's amicable co-ordination. If users are ready to accompany some people then, this may also become a potential cluster and adding this to the database will improve the reachability of the app. So, we enable users of the app with the best possible UI to add their current location, their departure time, departure point, arrival point. This is also integrated with personal chat with the passengers of the cluster for knowing the conditions of the traffic and also the feasibility of their inclusion in their cluster i.e, if there exists heavy traffic in the route planned before then, by personal chat existing passengers may request him/her to choose another cluster over them or begin a new cluster.*

***#Accepting passengers preferences**: This functionality allows passengers to enter their departure point, arrival point, mode of transport(car,autos,cabs,busses), date and time of departure.There will be certain tolerance limit set by the user itself(for this we will have to set lower tolerance limit(before preferred time) and upper tolerance limit(after preferred time) for extracting recommendations) . User also can select the clusters which he/she wants to join i.e., if he/she wants to join a favourite*

***#Creating a join Request to a cluster:** This functionality give the users to join any cluster of their choice, this request is sent to the cluster admin, where cluster admin can approve the request and hence the request maker can join the cluster.*

#Approving a Join Request: *This functionality allows cluster admins to verify and approve the join request of a passenger. upon approval the passenger is able to join the cluster.*

#Personalised suggestions*: Our CarPooling Application analyses the database frequently and gives personalised suggestions regarding best departing time(based on time tolerances)(this is different from preferred time as it may happen that a big cluster may begin their journey at some time just before or after the preferred time, so our app may request us to depart at the nearest time possible to the preferred time) , travelling time, mode of transport(car,autos,cabs,busses) and nearest pickup locations and all other locations of pickup(there will be a certain criterion (distance)for suggesting a place as the best point for boarding the vehicle which is certainly walkable from the current position of the user)(this is different from our point of departure as sometimes, it may happen that a big cluster may depart from a point near the preferred pickup point , so our app may suggest them to depart from the changed point) (this functionality also encompasses route to that point)so that number of clusters is minimised at any point of time, which is beneficial to all the users, as per head cost is significantly reduced.*

#Send Message*: The users can communicate with each other by sending messages via whatsapp button.*

 # Block User: *When a user receives a disturbing message, s/he can block the user who send that message.*

#See cluster's route*: This functionality enables the user to see a particular cluster's current route and this is preceded by a set of all available clusters ready to accommodate additional passengers.*

#Disable additional inclusions: *This functionality disables the cluster to allow additional passengers. Passengers can do this if they feel that they can't accommodate additional passengers and it stops covering their details.*

#Feedback from the customers: *This functionality enables the customer to give their feedback/suggestions regarding the service provided by the app and it lets the developers know what functionalities are to be added from time to time on a regular basis . Users rate the application by stars ranging from 1 to 5 inclusive of floating numbers ending with .5 .*

2.3 User Characteristics

Users of this application belong to the community of IIT INDORE in the preliminary stage of the app. Users need to enter the details and know the details at any point of time , mentioned in the section of 2.2. They need to go to the city or some other tourist spots present in the vicinity of IIT INDORE . For that, they need to know where and when the maximum number of passengers are using a particular automobile and a list of groups of people who are ready to accommodate more

people. People needing to use the pooling service will only use the application. They may also be ready to leave from the place

2.4 General Constraints

- **Flutter: version** 1.0
- **Firebase: version** `7.14.5`
- **Delivery Date:** 14th Feb 2020
- **Budget Constraints:** 10,000$

2.5 Assumptions and Dependencies

- *The device should be a "SmartPhone" , not a Feature Phone. However we also support creating and joining requests on the web.*
- *Minimum RAM(Random Access Memory): 1 GB*
- *Minimum Memory: 100 mb*
- *Android Version: 6.0 MarshMallow*
- *Location Service: The device must have a GPS(Global positioning system) receiver hardware installed in it.*
- *Internet Connection: An active internet connection is a must to use this application.*
- *users have a latest version of whatsapp installed on their phone.*

3. Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

This software product is developed for drivers and hitchhikers. Products will be deployed to an application and all users of the system will access the system through the application based interface which includes multiple pages according to the system functionality, for instance, for login functionality there will be a login page. To access the system, every user has a unique username and password. In addition, there will be a database which stores and manipulates all the data about the users. App will only be the interface for all the user data which is stored by database and the execution of provided functionalities. After the sign up, user information will be transferred to the database. In the sign up process, Google sign in will be used to authenticate users. After that point, users can register through the web interface. After logging in, users will be able to log out whenever they want.

3.1.2 Hardware Interfaces

The system runs on a mobile device, using android OS. So there is no such hardware interface, it will be managed by the in-built OS.

3.1.3 Software Interfaces

The system has Google Map API as a subsystem. Google Map subsystem has their own app based interface which is a map consisting of roads and locations in a desired area and users can easily interact with this system.

3.1.4 Communications Interfaces

In communication between driver and hitchhiker, For communication between users and drivers, a chatting portal will be used, the system shall support messaging functionality and users will be able to send and receive messages through the remote mobile devices.

3.2 Functional Requirements

3.2.1. Sign In

Use Case ID	UC2
Actor(s)	User
Description	User Log In
Preconditions	The user shall be able to sign in to the system.
Post conditions	Users will be able to use the system.
Precedence	Mandatory
Normal flow of event	1. The user opens the app and enters his phone no to the system. 2. User presses the login button. 3. User enters his or her OTP received by phone .
Alternative Flow(s)	Flow 1: 1. If the user enters the wrong OTP information, the warning message for example ”Wrong OTP information” will be shown to the user. Flow 2: 2. If the user enters his or her OTP correctly, the user will be redirected to the application relevant page of the system.

3.2.2. Sign Out

Use Case ID	UC3
Actor(s)	User
Description	User Log Out
Preconditions	The user shall be able to log out into the system.
Post conditions	Users will be able to leave the system.
Precedence	Not mandatory
Normal flow of event	1. User presses the log out button. 2. User leaves the system. 3. The app's login page will be loaded.

3.2.3. Create Cluster.

Use Case ID	UC4
Actor(s)	User
Description	Users shall be able to add clusters.
Preconditions	The user shall be able to sign in to the system.
Post conditions	Users shall retrieve transportation requests from the other users.
Precedence	Mandatory
Normal flow of event	1. Users shall enter her or his profile page. 2. Users shall press the Create button on the homepage. 3. The Create cluster page will be loaded. 4. Users enter departure time, available seats, start location, endLocation, his phoneNo, cost incurred.

	5. User draws a route on the map panel.

3.2.4. Delete Cluster.

Use Case ID	UC5
Actor(s)	User
Description	Users shall be able to delete cluster.
Preconditions	The user shall add a transportation route before.
Post conditions	Users cannot see the route which is deleted by the user.
Precedence	No mandatory
Normal flow of event	1. User shall presses my transportations button. 2. My transportations page will be loaded. 3. User selects the route he or she wants to delete. 4. Delete button is clicked. 5. The user deletes the route.

3.2.5. List Clusters.

Use Case ID	UC6
Actor(s)	User
Description	Users shall be able to List clusters.
Preconditions	The user shall see be on the home page..
Post conditions	Users will be able to create a join request to the admin who owns the cluster.
Precedence	No mandatory
Normal flow of event	1. User presses the Join button on the home page.

3.2.6. Create a Join request to a Cluster.

Use Case ID	UC7
Actor(s)	User
Description	Users shall be able to create a join request.
Preconditions	The user shall click a cluster from the list of the clusters.
Post conditions	Users will be able to contact the admin who owns the cluster.
Precedence	No mandatory
Normal flow of event	2. User presses the send join request button on the journey details page.

3.2.7. Search Transportation Route

Use Case ID	UC8
Actor(s)	User
Description	Users shall be able to search the route.
Preconditions	The user shall sign in to the system.
Post conditions	Users will be able to select a route from the available route list.
Precedence	No mandatory
Normal flow of event	1. User fills “from” input field. 2. User fills the “to” input field. 3. User presses the search button.
Alternative Flow(s)	Flow 1: 1. User forgets to fill “from” or “to” input field. 2. The related warning message is shown to the user to fill the input fields properly.

	1. User fills the input fields properly. 2. The available routes will be listed.

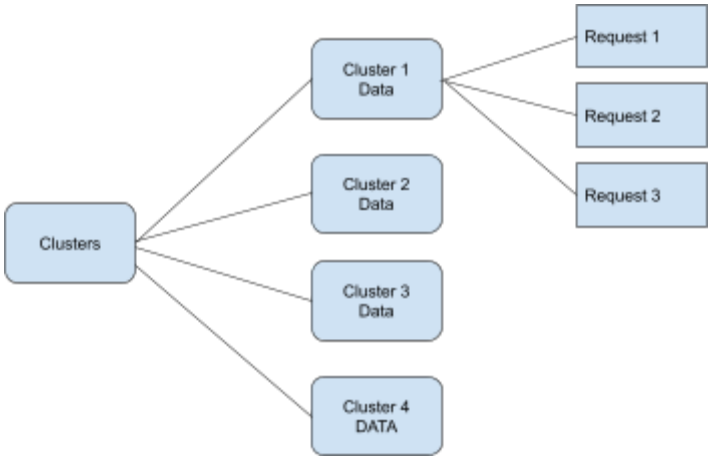
3.2.8. Send Message

Use Case ID	UC9
Actor(s)	User
Description	Users shall be able to send messages through the system via whatsapp.
Preconditions	The user shall sign in to the system.
Post conditions	Users will be able to communicate with each other.
Precedence	No mandatory
Normal flow of event	1. User enters the profile page of the user who is intended to be communicated. 2. User presses the send message button. 3. The message page will be loaded. 4. User types the content of the message. 5. User presses the send button to send the message content. 6. The message content will be stored and viewed in the message panel.

3.2.9. Rate User

Use Case ID	UC11
Actor(s)	User
Description	Users shall be able to rate the driver through the system.
Preconditions	The transportation route shall be completed with driver and hitchhiker.
Post conditions	The driver's rating will be updated.
Precedence	No mandatory
Normal flow of event	1. After the transportation, the hitchhiker login to the system. 2. Popup window is open.
Alternative Flow(s)	Flow 1: 1. Hitchhiker clicks the star icon to rate the driver's related attitude. Flow 2: 1. Hitchhiker clicks close icon. 2. The popup window will be closed.

3.3 Classes / Objects



3.3.1 Cluster

```
class Cluster {
  String clusterID;  //Identifies each cluster uniquely
  String adminName;  //Name of the host of a cluster
  String initialLocation; //Starting position of the cluster
  String finalLocation; //Ending location of the cluster
  String phoneNo; // Contact phone number of host of cluster
  String cost;
  int noOfPassengers;//Number of passengers currently in the cluster
  String carNo;
  String carType; //Explains model or give idea of type of car
  int leavingTime; //time when cluster leaves initial location
  String get date =>//Converts the date stored in integer format
DateTime.fromMillisecondsSinceEpoch(leavingTime)
      .toIso8601String()
      .substring(0, 10);
  String adminUserID; //Identifies host uniquely

  LatLng startPoint; //Coordinates of initial location
  LatLng endPoint;//Coordinates of final location
  Map<String, Request> requests = {};
```

```
  int get pWatingRequest { //Number of requests waiting to be
accepted
int count = 0;
    requests.forEach((key, value) {
      if (!value.isAccepted) count++;
    });
    return count;
  }

  int get pApprovedRequest {//Keep note of accepted requests
    int count = 0;
    requests.forEach((key, value) {
      if (value.isAccepted) count++;
    });
    return count;
  }

  DateTime get pLeavingTime => //Converts the date stored in integer
format
DateTime.fromMillisecondsSinceEpoch(leavingTime);

  Cluster(
      {this.adminName,
      this.initialLocation,
      this.finalLocation,
      this.phoneNo,
      this.cost,
      this.noOfPassengers,
      this.carNo,
      this.carType,
      this.leavingTime,
      this.adminUserID});

  Cluster.fromMap(Map data) //It is a constructer which gets a map
of data from firebase in JSON format and sets the data in an
instance of Cluster class
{    this.adminName = data["adminName"] ?? "";
```

```
    this.initialLocation = data["initialLocation"] ?? "";
    this.finalLocation = data["finalLocation"] ?? "";
    this.phoneNo = data["phoneNo"] ?? "";
    this.cost = data["cost"] ?? "";
    this.noOfPassengers = data["noOfPassengers"] ?? 1;
    this.carNo = data["carNo"] ?? "";
    this.carType = data["carType"] ?? "";
    this.leavingTime = data["leavingTime"] ?? 0;
    this.adminUserID = data["adminUserID"] ?? "";
    (data["requests"] ?? {}).forEach((key, value) {
      this.requests.addAll({key: Request.fromMap(value)});
    });
  }

  Map<String, dynamic> toMap() {//This function return the JSON form
which is understandable to firebase from the Cluster at which it is
called.
    return {
      "adminName": adminName,
      "initialLocation": initialLocation,
      "finalLocation": finalLocation,
      "phoneNo": phoneNo,
      "cost": cost,
      "noOfPassengers": noOfPassengers,
      "carNo": carNo,
      "carType": carType,
      "leavingTime": leavingTime,
      "date": date,
      "adminUserID": adminUserID,
      "startPoint": startPoint.toJson(),
      "endPoint": endPoint.toJson(),
    };
  }
}
```

3.3.2 Current User

```
class CurrentUser //This class makes an instance of the Current User who
has signed in
{
  String get uid => user.uid; //User Id which uniquely identifies the
current user on his/her device
  String get phoneNo => user.phoneNumber;
  String get userName => user.displayName; //Name of currently signed in
user
  String get userEmail => user.email;
  double lat;//Identifies latitude of current user
  double lng;//Identifies longitude of current user
  String password; //Maintains password set by current user
  FirebaseUser user;
  // collection ref. to update user in firestore
  final CollectionReference userData =
      Firestore.instance.collection('UserData');

  Future<FirebaseUser> getCurrentUser() async
  {//gets the current user information from the stored database
    user = await FirebaseAuth.instance.currentUser();
    return user;
  }

  Future<void> setCurrentData() async {}

  Map toMap() {//This function return the JSON form which is
understandable to firebase from the CurrentUser at which it is called.
    return {
      "userName": userName,
      "password": password,
      "uid": uid,
      "phoneNo": phoneNo,
      "lat": lat,
      "lng": lng,
    };
```

```
  }
 //This function stores the information of current user in the local
database of device
  Future<void> storeUserInMemory(FirebaseUser user) async {
    SharedPreferences sharedPreferences = await
SharedPreferences.getInstance();
    sharedPreferences.setString("username", user.displayName);
    sharedPreferences.setString('email', user.email);
    sharedPreferences.setString('phone', user.phoneNumber);
    sharedPreferences.setString('uid', user.uid);
  }

  Future updateUserData(String uid, String username, String email,
      String password, String phoneNo) async {
    return await userData.document(uid).setData({
      'username': username,
      'email': email,
      'uid': uid,
      'password': password,
      'phoneNo': phoneNo,
    });
  }
 //This function logs the current user out of the running session
  Future<void> logOut(BuildContext context) async {
    await FirebaseAuth.instance.signOut();
    Navigator.of(context).pushReplacementNamed("/init");
//Replace the current route of the navigator by pushing the route named
and then disposing the previous route once the new route has finished
animating in.
  }
}
```

3.3.2 Request

```
class Request {
  String phoneNo;
  String requestUserID; //Identifies each requesting user uniquely
  String requestUserName; //Cluster request user name
```

```
  bool isAccepted; //true if the request is accepted
  int requestTime; //time when the request is made

  Request(
      {this.requestUserID,
      this.isAccepted,
      this.phoneNo,
      this.requestUserName});

  Request.fromMap(Map data) {//Constructor which takes the JSON format of
data from firebase and sets the data in an instance of the Request class,
and returns that instance.
    this.requestUserID = data["requestUserID"] ?? "";
    this.isAccepted = data["isAccepted"] ?? false;
    this.phoneNo = data["phoneNo"] ?? "";
    this.requestUserName = data["requestUserName"] ?? "";
    this.requestTime =
data["requestTime"]??DateTime.now().millisecondsSinceEpoch;
  }

  Map<String, dynamic> toMap() {//Function which returns the JSON format
of the data of the request object call the function
    return {
      "adminUserID": requestUserID,
      "requestUserName": requestUserName,
      "isAccepted": isAccepted,
      "phoneNo": phoneNo,
      "requestTime": requestTime,
    };
  }
}
```

3.3.2 CarPoolingProvider Class

```
class CarPoolingProvider with ChangeNotifier {
  //This is a central distribution unit (CDU) which fetches data from the
database and provide it to member functions of other classes in the system

  //VARIABLES ------------------------
  CurrentUser currentUser = CurrentUser();//Current user initiated
  /*
  *CLUSTERS key: unique cluster ID
  *this helps in accessing clusters more effectively*/
  Map<String, Cluster> globalClustersMap = {};
//This contains all the clusters which are present in the database
  Map<String, Cluster> myClustersHistoryMap = {};
//This attribute hold the history or past interacted data
  Map<String, Cluster> myRequestHistoryMap = {};
//
  //
  //INIT -----------------------------
  CarPoolingProvider() {
    currentUser = CurrentUser();
    currentUser.getCurrentUser();
    loadGlobalClusterData(force: true);
    loadMyClustersHistoryData(force: true);
  }

  //
  //LOADERS --------------------------

  Future<String> loadGlobalClusterData({bool force = false}) async {
//This function initiates all the clusters in globalClusterMap which are
present in the database
    if (force || globalClustersMap.length == 0) {
      await Firestore.instance
          .collection("clusters")
          .getDocuments()
          .then((value) {
```

```
        value.documents.forEach((element) {
          globalClustersMap.addAll({
            element.documentID: Cluster.fromMap(element.data),
          });
          globalClustersMap[element.documentID].clusterID =
element.documentID;

          print("Data Loaded from firebase");
          notifyListeners();
        });
      });
    }
    return "done";
  }

  Future<String> loadMyClustersHistoryData({bool force = false}) async {
    if (force || myClustersHistoryMap.length == 0) {
      currentUser.user = await currentUser.getCurrentUser();
      await Firestore.instance
          .collection("clusters")
          .where("adminUserID", isEqualTo: currentUser.uid)
          .getDocuments()
          .then((value) {
        value.documents.forEach((element) {
          myClustersHistoryMap.addAll({
            element.documentID: Cluster.fromMap(element.data),
          });
          myClustersHistoryMap[element.documentID].clusterID =
              element.documentID;
          notifyListeners();
        });
      });
      fillRequestData();
      print("loaded my clusters from firebase");
    }
    return "done";
  }
```

```
  Future<String> loadRequestData({String clusterId, bool force = false})
async {
    if (force || myClustersHistoryMap[clusterId] != null)
      await Firestore.instance
          .collection("clusters")
          .document(clusterId)
          .get()
          .then((value) {
        myClustersHistoryMap.addAll({
          value.documentID: Cluster.fromMap(value.data),
        });
        notifyListeners();
      });
    fillRequestData();
    print("Data Loaded from firebase");
    return "done";
  }

  Future<String> fillRequestData() async {
    if (myClustersHistoryMap.length != 0) {
      globalClustersMap.forEach((key, element) {
        if (globalClustersMap[element.clusterID].requests[currentUser.uid]
!=
            null) {
          myRequestHistoryMap.addAll(
              {element.clusterID: globalClustersMap[element.clusterID]});
        }
      });
    }
    print("Data Loaded from firebase");
    return "done";
  }

  Future<String> createClusterData(Cluster cluster) async {
    cluster.adminName = currentUser.userName;
    cluster.adminUserID = currentUser.uid;
```

```
    cluster.phoneNo = currentUser.phoneNo;
    DocumentReference docRef = await Firestore.instance
        .collection("clusters")
        .add(cluster.toMap())
        .catchError(onError);
    myClustersHistoryMap[docRef.documentID] = cluster;
    globalClustersMap[docRef.documentID] = cluster;
    notifyListeners();
    print("Data uploaded to firebase");
    return "done";
  }

  Future<String> createClusterJoinRequest({@required String clusterID})
async {
    if (clusterID == null || clusterID == "") {
      Fluttertoast.showToast(msg: "Cluster Id is not correct");
      return "not Done";
    }
    Request request = Request.fromMap({});
    request.isAccepted = false;
    request.phoneNo = currentUser.phoneNo;
    request.requestUserID = currentUser.uid;
    request.requestUserName = currentUser.userName.toString();
    request.requestTime = DateTime.now().millisecondsSinceEpoch;
    await Firestore.instance
        .collection("clusters")
        .document(clusterID)
        .setData({
      "requests": {currentUser.uid: request.toMap()}
    }, merge: true).catchError(onError);
    notifyListeners();
    print("Data uploaded to firebase");
    return "done";
  }

  Future<String> acceptUserRequest(
```

```
      {@required String clusterID, @required String requestUserId}) async
{
    await Firestore.instance
        .collection("clusters")
        .document(clusterID)
        .setData({
      "requests": {
        requestUserId: {
          "isAccepted": true,
        }
      }
    }, merge: true).catchError(onError);
    notifyListeners();
    Fluttertoast.showToast(msg: "Request Made", webShowClose: true);
    print("Data uploaded to firebase");
    return "done";
  }

  void onError(dynamic err) {
    Fluttertoast.showToast(
        msg: err.toString(),
        timeInSecForIosWeb: 2,
        toastLength: Toast.LENGTH_LONG,
        webShowClose: true);
  }
}
```

3.4 Non-Functional Requirements

Non-functional requirements may exist for the following attributes. Often these requirements must be achieved at a system-wide level rather than at a unit level. State the requirements in the following sections in measurable terms (e.g., 95% of transactions shall be processed in less than a second, system downtime may not exceed 1 minute per day, > 30 day MTBF value, etc).

3.4.1 Performance: Location of members travelling in a vehicle should be updated in the database without any delay and this live location should also be reflected in other user's applications.

3.4.2 Reliability: Our software ensures location accurately upto 5-mts.

3.4.3 Availability : Our software is made to ensure 24/7 service**.**

3.4.4 Security: One can join a ride in both the Favorite mode and Public mode where when the person is in Favorite mode, he can join rides of his friends only while when he is in Public mode he can join all those rides where the passengers are willing to accept an anonymous ride.

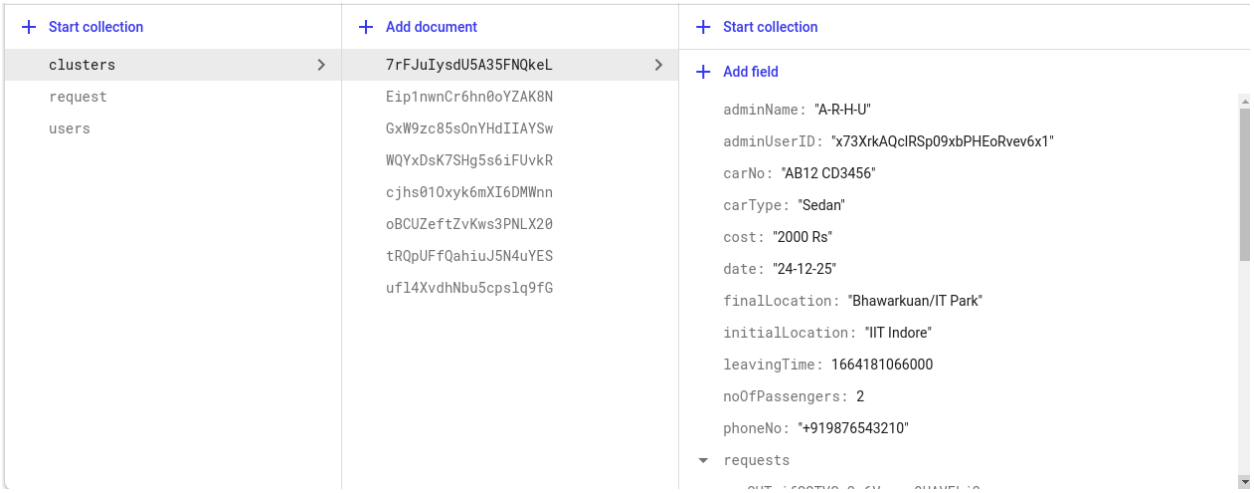
3.4.5 Maintainability: The application code is refactored, separated into different modules (modularity) so that it is easier to understand and debug. This ensures that the code is less complex and developer friendly.

3.4.6 Portability: Our application is easy to download from the Google PlayStore and easy to transfer between users in form of an application bundle known as apk file which is easily available on our website www.uniteonwheels.in.

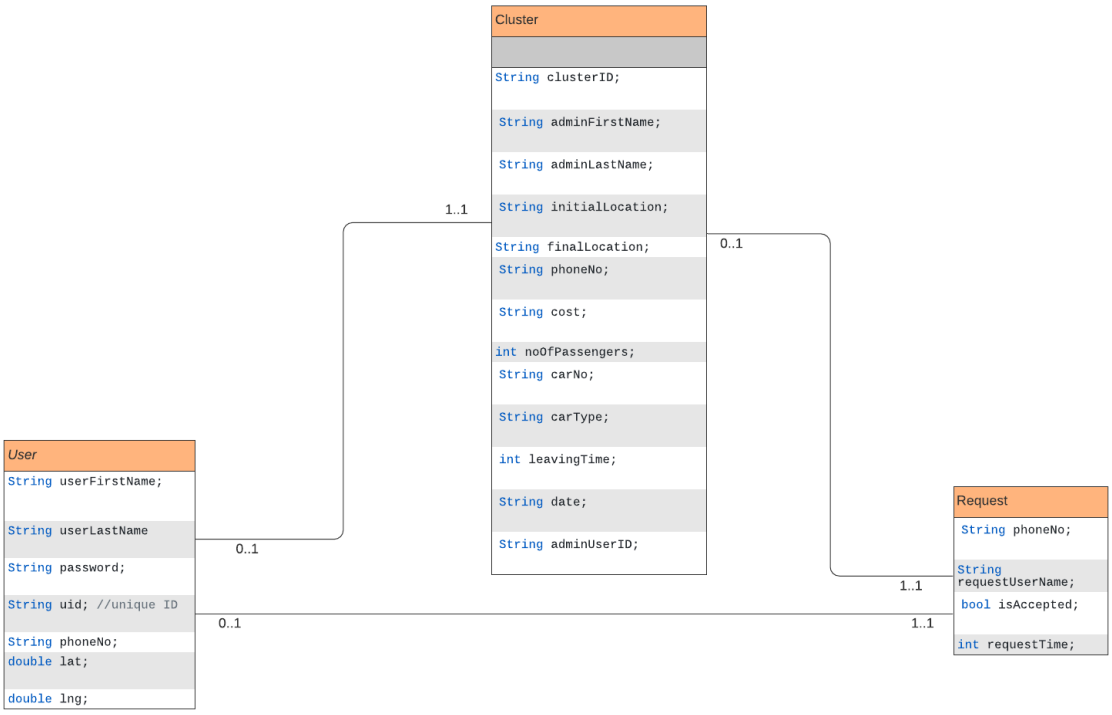
3.6 Logical Database Requirements

A database will be used to store current location of the passengers travelling and that location will be accessed by the passengers who are intended to travel in the vehicles. For this purpose we have used the firebase database as our online database. Users can create, update and delete clusters. as well as can join, approve and delete requests.

3.6.1 Database schema on Cloud storage



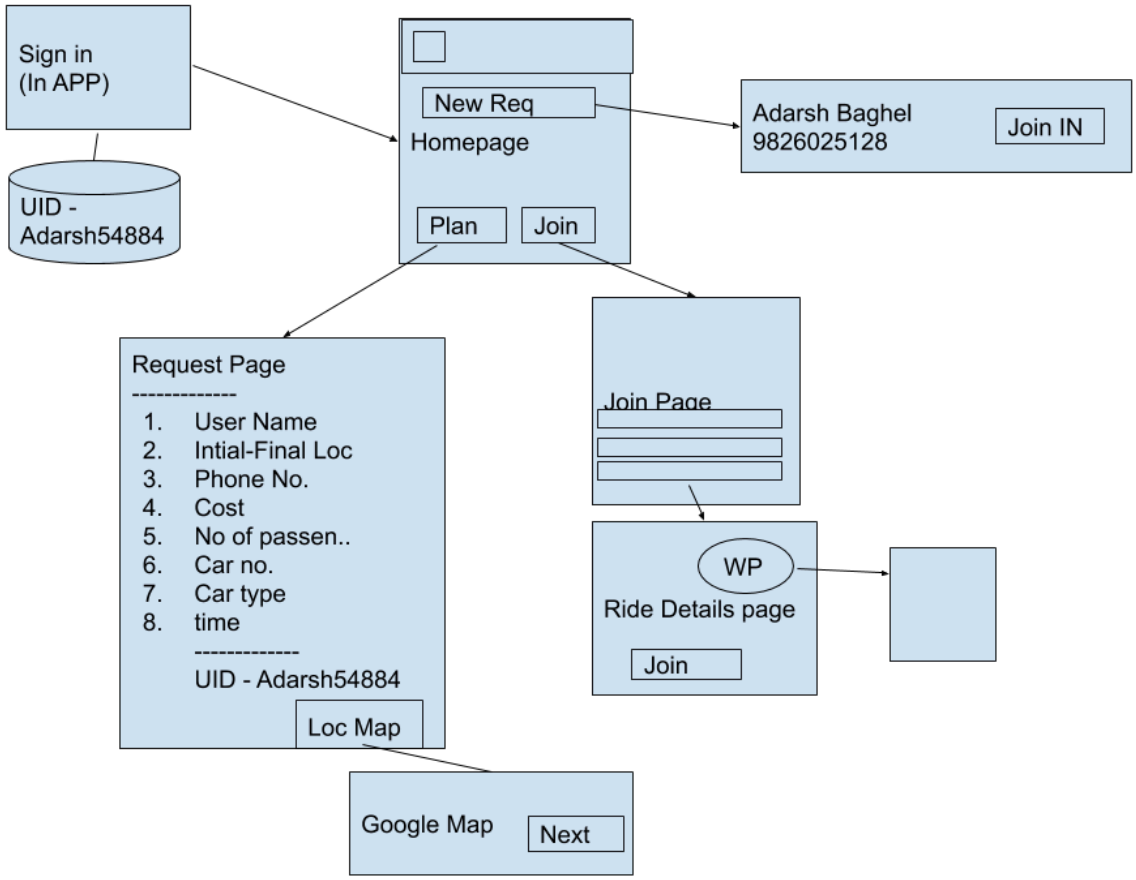
3.6.2 Data Formats and Constraints



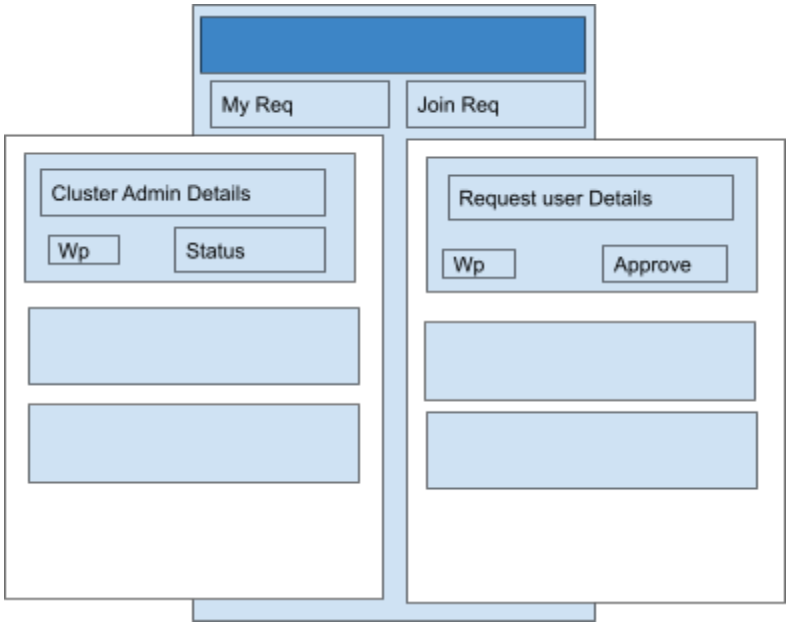
3.8 Design Layouts

The design templates which were used for designing the application. These went through continuous updation over iterations.

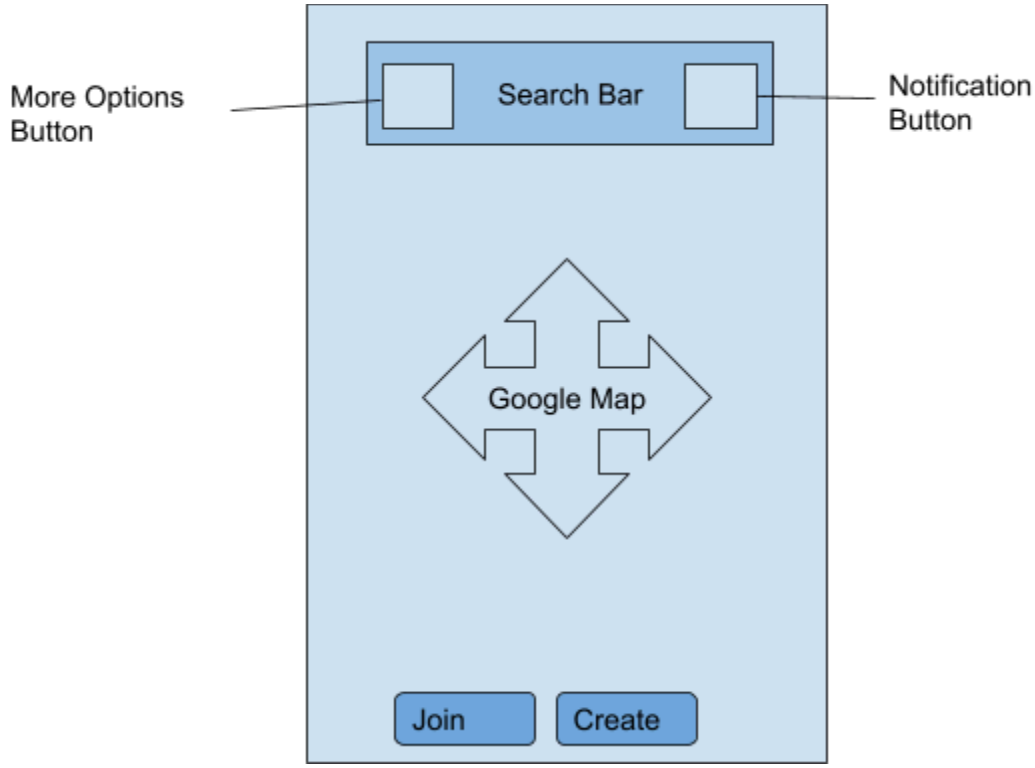
3.8.1 Application Design Structure



3.8.2 Notification Page Layout.

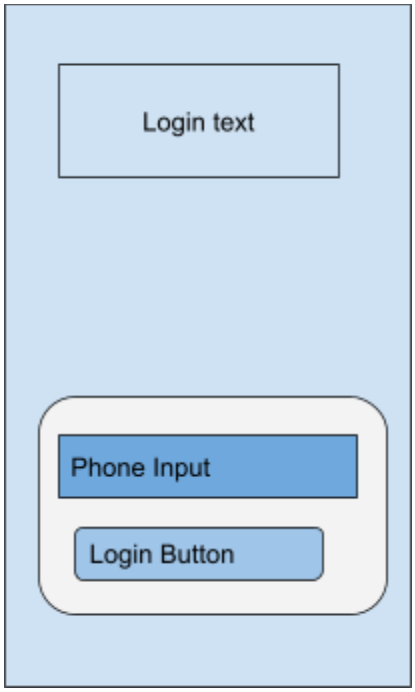


3.8.3 Home Page Layout.



This page helps users to navigate to different pages, to provide join Create and approve cluster and their requests.

3.8.4 Login Page



If they do not have an account, they can register the system by clicking the “Sign Up” button. In this page, users have to fill the form which is about their name, username, phone number and otp.

4. Conclusion

In this document, the functional and other requirements of the system are described. Furthermore, the needs of the user are stated through the document. However, all requirements are not defined and some of the requirements need to be clarified in this document. To sum up, this document is the primary document which upon all of the subsequent design, implementation, test and validation processes will be based.

5. Supporting Information

At the beginning of the document, table of contents and list of figures are included.