POLITECNICO
MILANO 1863

Computer Science and Engineering
Project of Software Engineering 2

PowerEnJoy

Design Document

Authors:

Alberto Zeni 813977

Manuel Parenti 876085

Reference Professor: Mottola Luca

Academic Year 2016–2017

Table of Contents

1 Introduction

1.1 Purpose

This document contains the architectural and design choices taken for the PowerEnJoy system. It's written to explore the details of the system we want to create and describes a more technical view of it.
In this document are presented:

- An overview of the architecture
- An increasingly more detailed analysis of the main components of the system
- The external interfaces of those components
- The runtime behavior in the most common scenarios
- Design patterns, architectural styles and some other decisions over the design

1.2 Scope

PowerEnJoy is a car sharing service based on a mobile and web application. It provides its users with the tools and functionalities needed to pick up a car in a parking zone near them and ride to their destination. The system should be able to register new users with their personal information, such as name, surname, address, email and payment information. Once a user is registered to the system he should be able to reserve a car using the web or the mobile app, the position of the nearest car available is calculated by the system using the user position (provided by the gps or inserted manually by the user).

PowerEnJoy enables its customers to drive ecologic cars at their need, either they want to be eco-friendly or just want to reach a place and they don't have an available transport.

The system includes different extra functionalities, such as the possibility to share the rides with other people with a reward, the control over the cars' location and sensors' data to manage exceptional and non-exceptional user behavior during the course of the rides.

The system also provides the users with the best possible service by having some operators that can move the cars from unsafe areas and plug the cars in special parking areas, so that the availability of available cars at any moment is satisfying.

1.3 Definitions, Acronyms, Abbreviations

USER: A person who is registered on PowerEnJoy, he can register, find information about his account, search for a car and use the car sharing service using a web application via a browser or using a mobile application. Users are provided with an account ID and a password which they need to use to access the service. Sometimes in the document they are referred to as Customers.

SAFE AREA: Public parking areas defined by the PowerEnJoy company, these won't cause any police officer to write a ticket for the car or even remove the car.

SPECIAL PARKING AREA: These areas are defined by the PowerEnJoy company and they are provided with electric plugs for charging the battery of the cars.

RESERVATION: This term refers to the possibility of reserving a car for a specific user. He can do this with the web application or the mobile app, after he finds a car he wants to pick up. The cars will be reserved for that user for an hour, and after the time expires if the user doesn't pick up the car he is charged of a small fee. Users can reserve a car for up to an hour before they need to drive.

DISPATCHER: A person or a system that will manage the parking and recharging of the cars. The dispatcher will send operators on field to do the operations required in order to overcome the eventual bad behavior of some users.

GPS: The Global Positioning System is a global navigation satellite system that provides geolocation and time information to a GPS receiver in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

SENSOR: A sensor is a device that converts real world data (Analog) into data that a computer can understand.

1.4 Reference Documents

- RASDPowerEnJoy.pdf
- Assignments AA 2016-2017.pdf

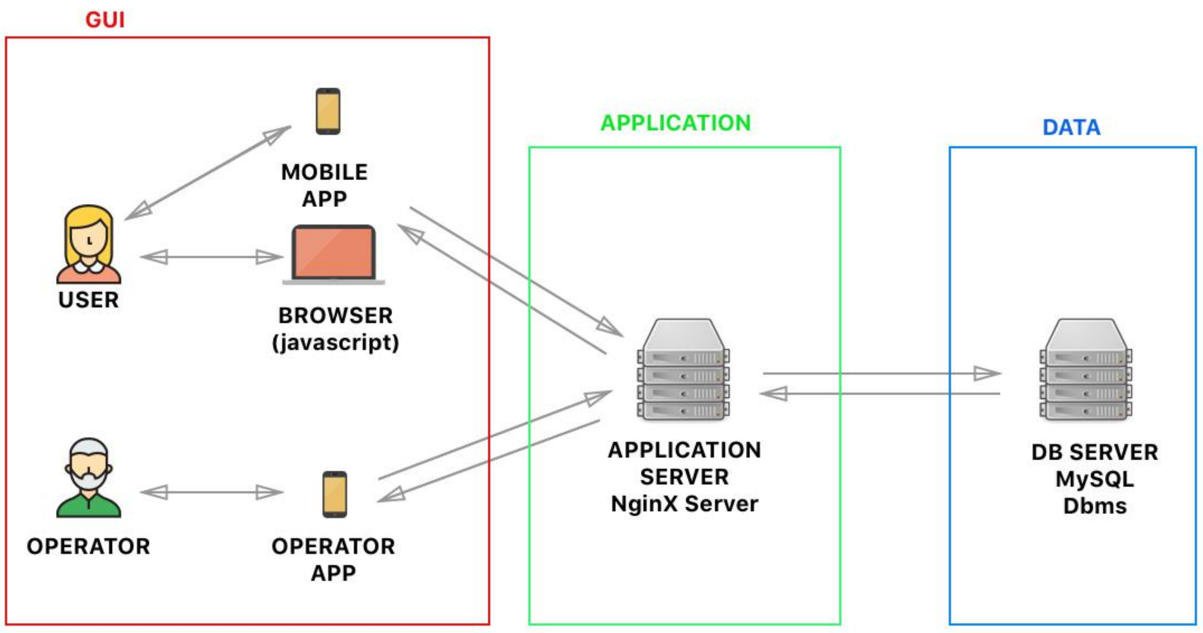
1.5 Document Structure

- **Introduction:** this section describes what is the purpose and the structure of the design document, it also describes some introductory concepts.
- **Architecture Design:** this section describes our architectural decisions about the system, for clarity purposes it is divided in sub-sections:
    - A. Overview: this section describes the division in tiers of our application and also the general architecture.
    - B. High Level Components and their interaction: this section explains the overall application, and how its elements interact, through a top down analysis approach
    - C. Component View: this section covers the element of the system in a more detailed way, as indicated in the top down analysis.
    - D. Deployment View: this section explains how the components are distributed in the tiers in order to make the system work correctly.
    - E. Runtime: this section shows the course of various tasks of our system through some diagrams.
    - F. Component Interfaces: this section shows the interfaces between the elements of the system.
    - G. Architectural styles and patterns: this section explains the architectural and pattern choices we have done for our system.
- **Algorithm Design:** this section reports the most particular algorithms that our system needs in order to provide the services that we already described. Some pseudo-code is used to simplify the implementation in order to focus only on the relevant parts of the algorithm.
- **User Interface Design:** this section describes the user experience with UX diagrams
- **Requirements Traceability:** this section explains how the decisions made in the RASD document are linked to the design elements in this document.

2 Architectural Design

2.1 Overview

PowerEnJoy has a three-tier architecture described as follows

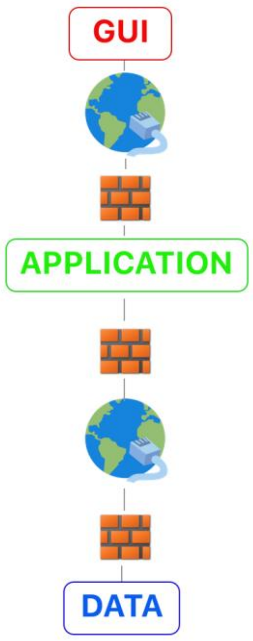


General Architecture

The client GUI interacts with the application server through some APIs, information is provided via the UI and then elaborated by the application server. Some sensible data are stored in the DB SERVER such as payment information and personal info about the user.

The proposed architecture is a three-tier architecture because our intent is to have a good level of modularity. This way the system's performances can be improved by making some changes in just one of the tiers. Also, the scalability, security and reliability can be potentially high at a reasonable cost.

The development of the whole system can be distributed in different specialized teams so that we can obtain a better result in less time.



Tier Representation

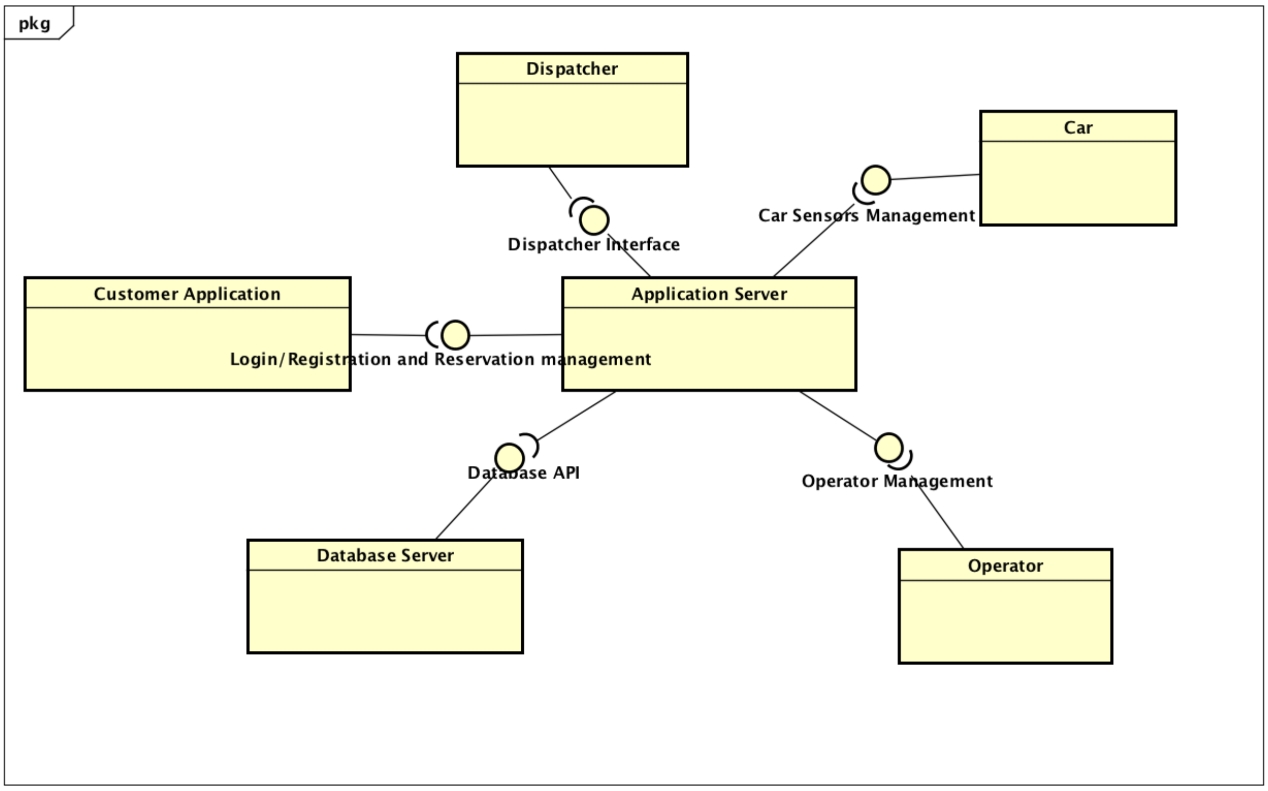
Security is really important since this system manages sensible information, so we need to also guarantee that our system prevents unauthorized access to it, as well as data integrity.

Aside from the best practices for having secure data, such as the use of complex passwords, anomaly behavior detection and encrypted communications we want to have a strong doorway to discourage and prevent breaches in the network and servers.

The most secure configuration for the firewall is the De-Militarized Zone, we need that to control the network's internal traffic.

2.2 High level components and their interaction

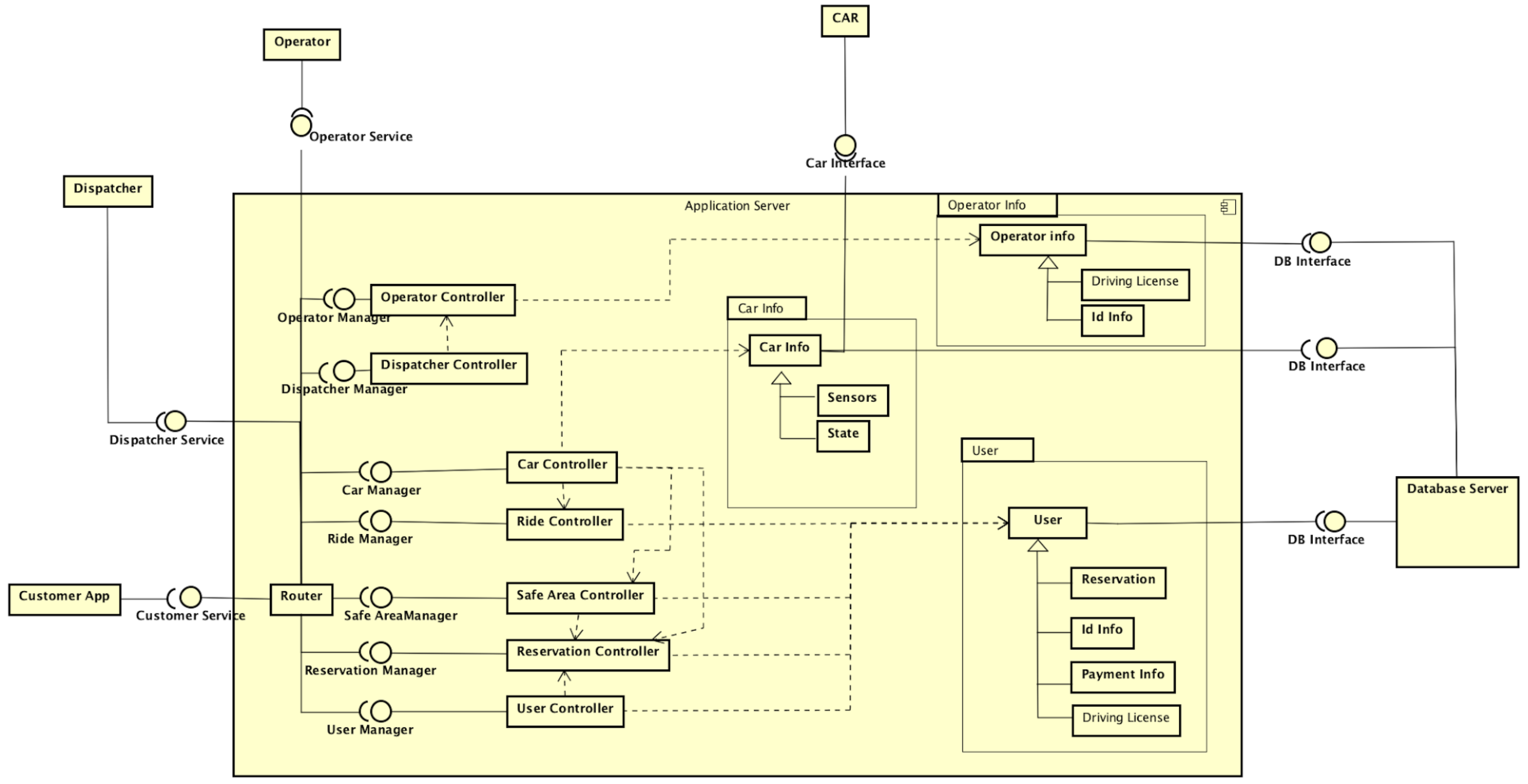
The high-level architecture is composed of five different elements. The main element is the application server. This element handles all the client requests coming through the mobile or web app as well as interacting with the DBMS to check data, receiving data coming from the cars in the system, communicate with the dispatcher and with the operators as well. The communication with the customers is made in a synchronous way, the client starts the communication opening the app (web or mobile) and logging in, after that he waits the answer of the server which has to communicate with the DBMS in order to check his data. When the user successfully logs into the system he could set his own preferences and then send a reservation request to the server, then the server checks the car availability in the area selected by the user. If there are cars available the server sends a confirmation to the user and then the communication becomes asynchronous, now the server has to wait the user to unlock the car or, in the case the user doesn't unlock the car in an hour, the server itself sends to the user a notification which tells that his reservation has been canceled because he waited too much time. If no cars are available the server asks to the customer if he wants to wait for a car and then in case of agreement of the customer the communication becomes asynchronous because the server sends a notification to the user only if a car becomes available, obviously if the user refuses to wait the communication ends. The Cars are connected to the internet, in this way they can send information to the application server continuously, so the communication between cars and system is synchronous. The communication between server and dispatcher/operator is asynchronous because the server communicates with them only in case of necessity.


pkg
Dispatcher
Car
Car Sensors Management
Dispatcher Interface
Customer Application
Application Server
Login/Registration and Reservation management
Database API
Operator Management
Database Server
Operator

2.3 Component view

2.3.1 Application Server Component

The main element of the architecture, as described above, is the application server. Here is proposed its component view:



Operator Controller: manages the interaction between the server and the operator app, it also checks that the info of the operator is correct.

Dispatcher Controller: manages the interaction between the server and the dispatcher.

User Controller: manages the interaction between the server and the customer, it also checks that the info of the user is correct.

Car Controller: manages car information, which comes from the car and the database.

Safe Area Controller: checks the information regarding the safe areas, including the special ones with charging stations.

Ride Controller: manages the information of the current ride.

Login Controller: checks the information of that the user entered with the ones stored in the Database.

Reservation Controller: checks the possible options for the reservation, it manages the requests using data coming from other sub-elements in order to do its tasks.

Router: manages to receive and send information between the server and all the various applications of our system.

User: our representation of the user in the DB.

Car Info: our representation of the user in the DB.

Operator Info: our representation of the operator in the DB.

CAR: the car sensors which communicate with our server.

Operator: the operator application.

Dispatcher: the application that interacts with the system/person which does the role of dispatcher.

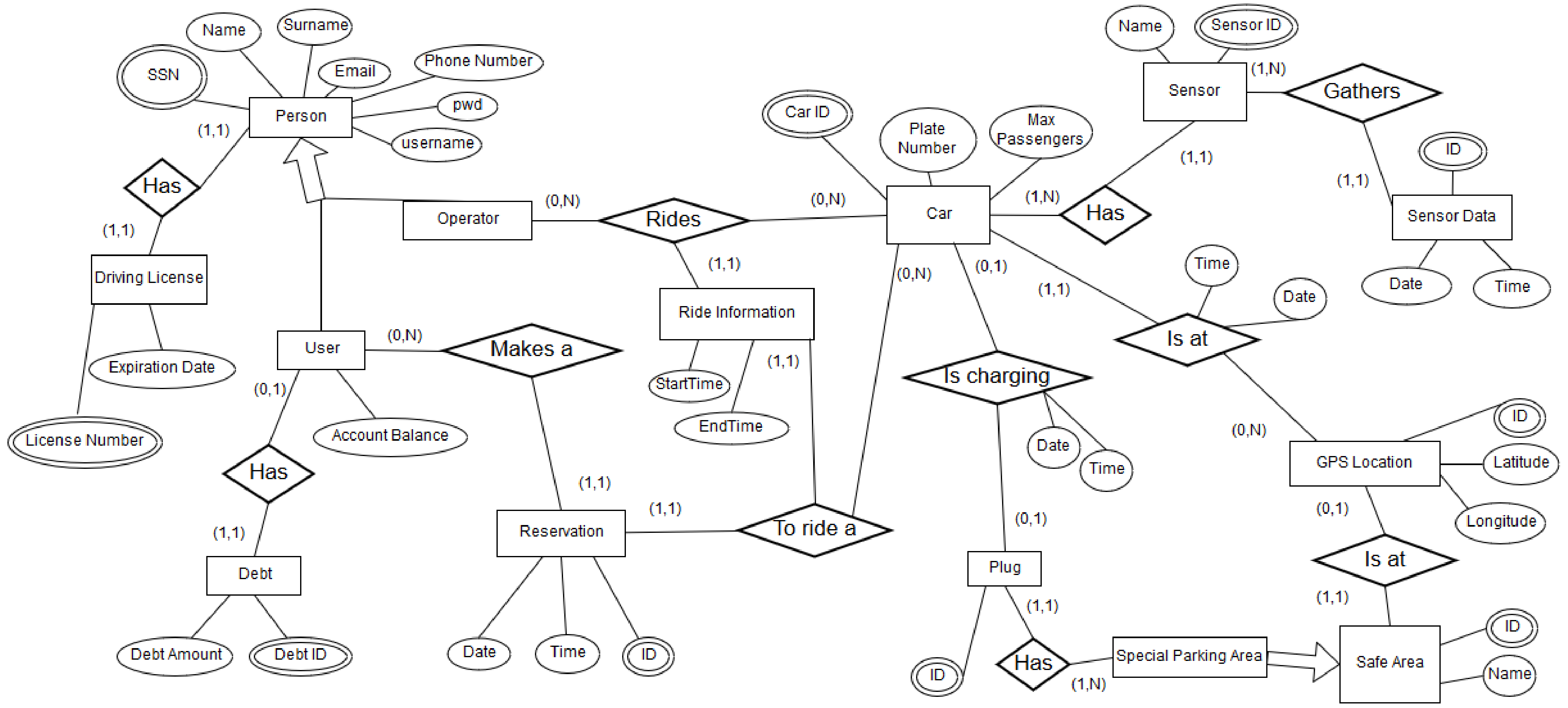
Customer: our main application used by the customers of PowerEnJoy.

DB: the database used to store data persistently.

The router manages the communication with each application of the system and so all the information coming from customers and various operators of the system pass through it. When a request arrives, the server sends it to the appropriate element of the server which manages it.

2.3.2 Database component

The database component contains the database server, so it will have a DBMS and the application server will reach the data through the DBMS APIs. Here we want to model how the management of the system's data will be saved. The following is a basic ER diagram, it is the starting point for the development of the database, since a logical schema can be derived and from there a relational database can be simply created.

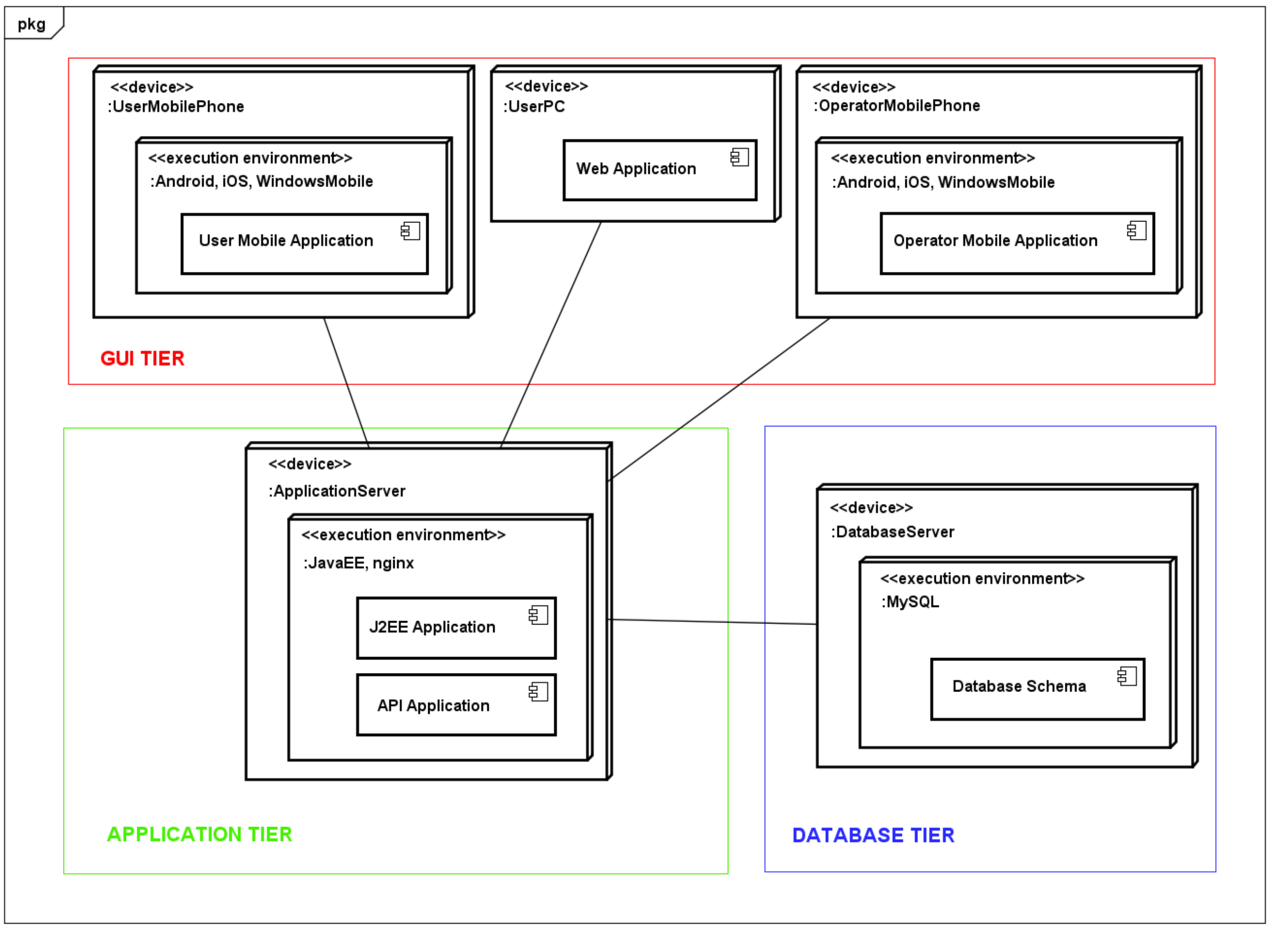


For the translation into a logical schema there might be some conflicts to solve:

The creation of cross tables and foreign keys is quite trivial so we won’t discuss it, but there are a few generalization problems to solve.

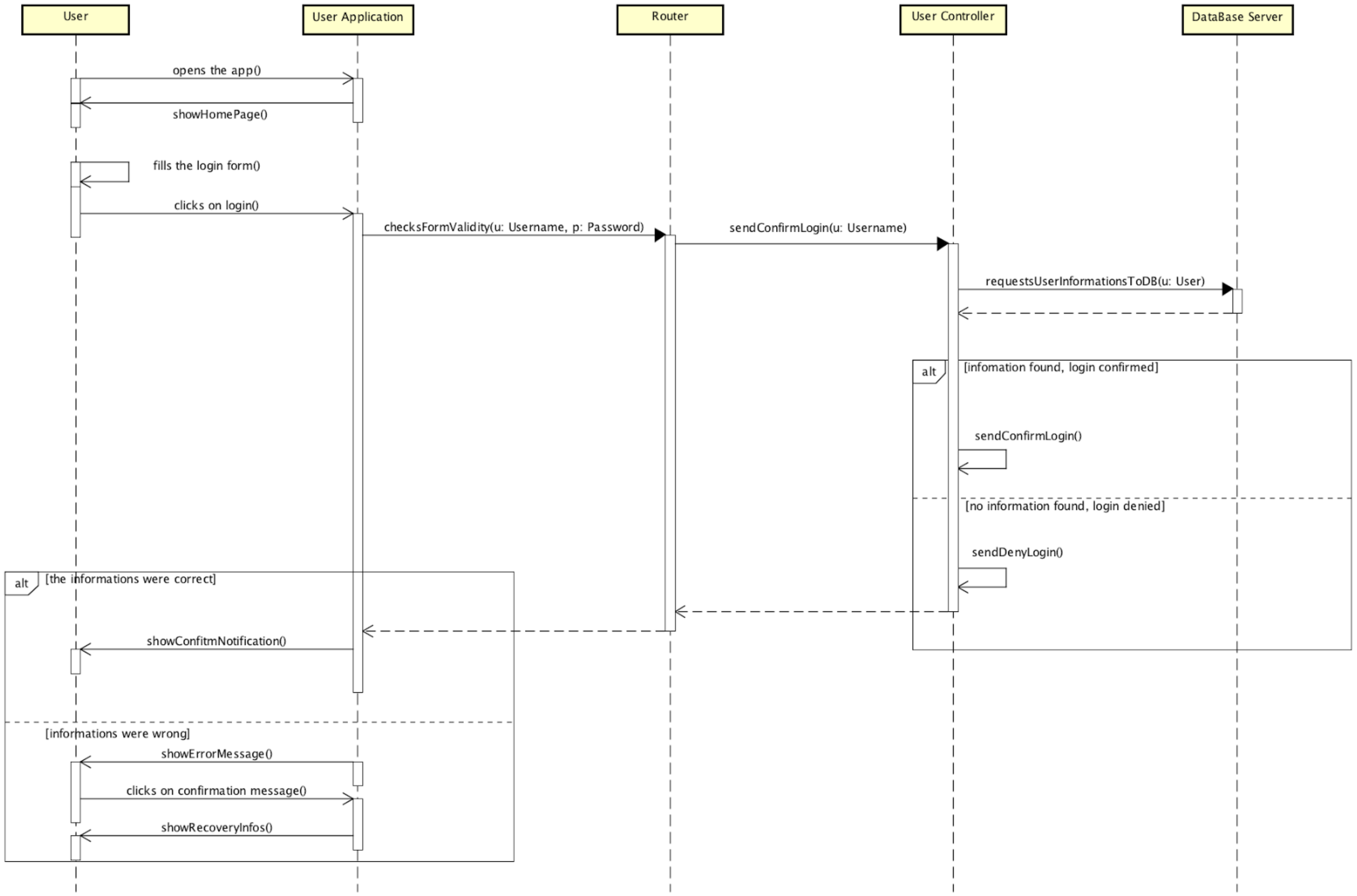
- Person can be split in two different entities: User and Operator, because they have different relations with other entities and even different attributes. Note that no driving license can be left without an owner, so this constraint must be kept.
- Safe Areas and Special Parking Areas have different relations too, so they can be considered as two separate entities. Note that special parking areas have a gps location, so the relation “Is at” must be duplicated.

2.4 Deployment view



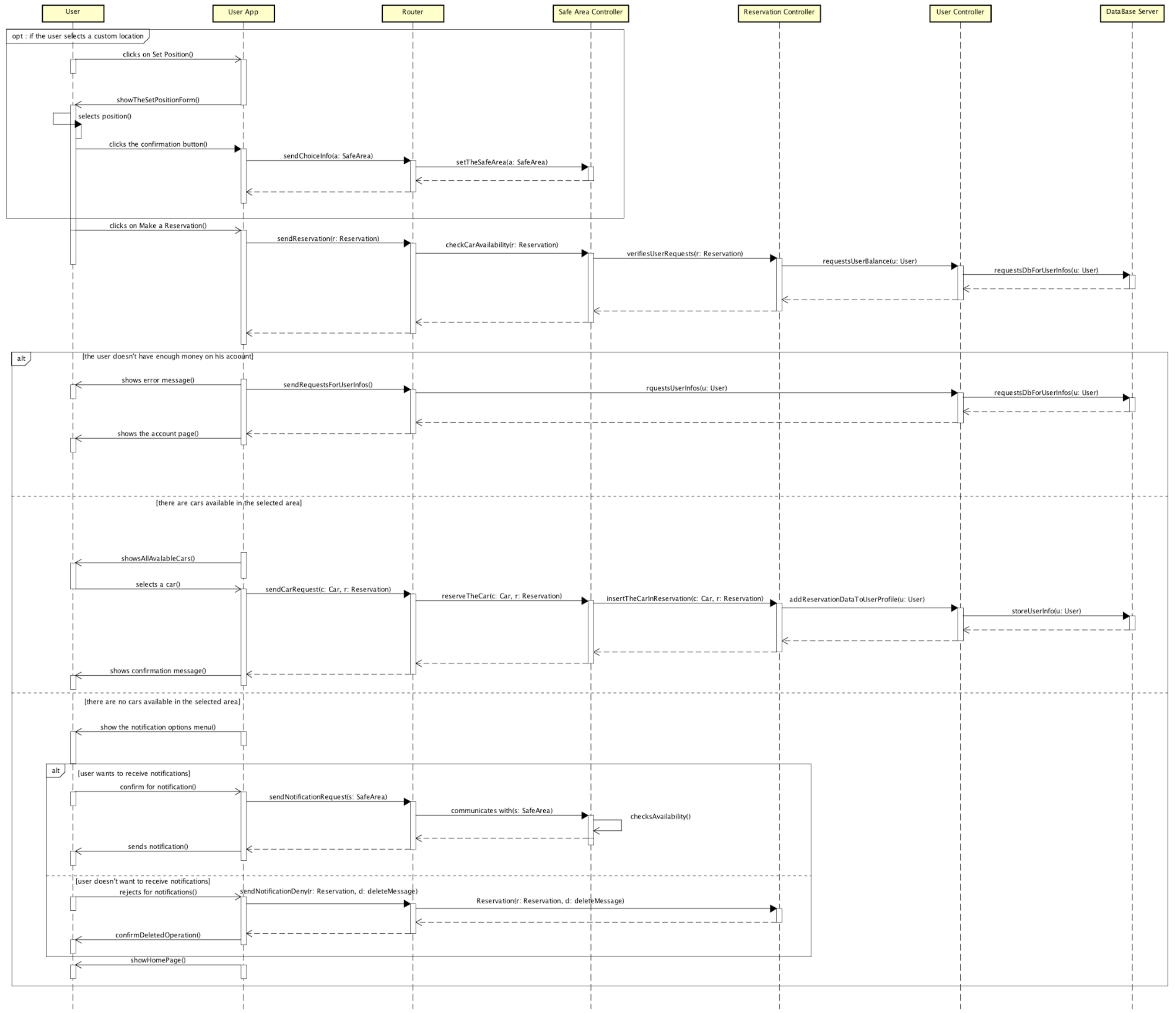
2.5 Runtime view

2.5.1 Login Sequence Diagram


User
User Application
Router
User Controller
DataBase Server
opens the app()
showHomePage()
fills the login form()
clicks on login()
checksFormValidity(u: Username, p: Password)
sendConfirmLogin(u: Username)
requestsUserInformationsToDB(u: User)
alt
[infomation found, login confirmed]
sendConfirmLogin()
[no information found, login denied]
sendDenyLogin()
alt
[the informations were correct]
showConfitmNotification()
[informations were wrong]
showErrorMessage()
clicks on confirmation message()
showRecoveryInfos()

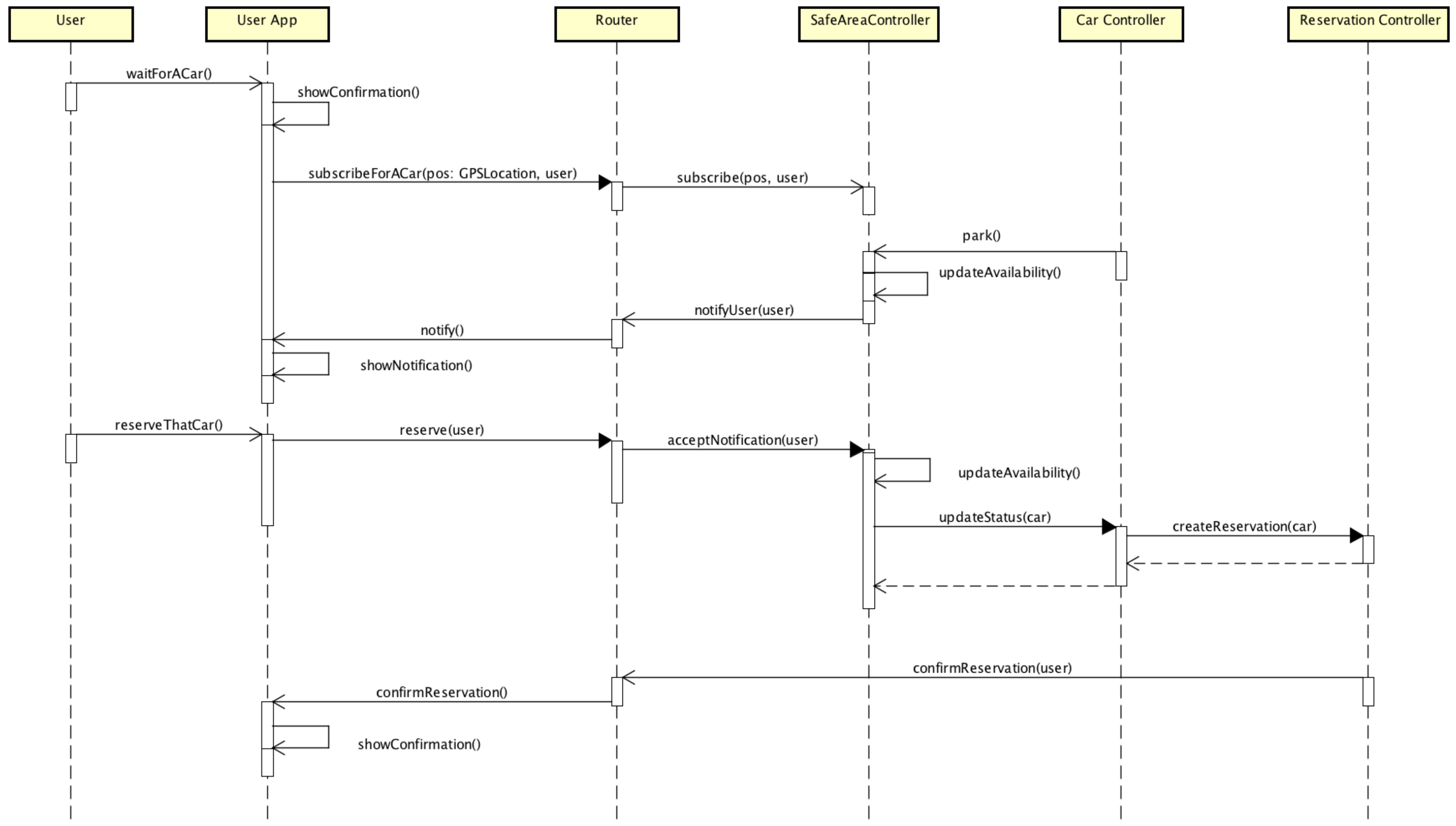
This sequence diagram describes the various interactions between the system components for the Login operation. At first the user opens the app, once it’s opened the app shows to the user the login form which the user must compile in order to start using the service. Once fully filled the user should press the Login button to confirm that all the required data has been inserted, after this operation the apps sends this information to the router of the application server which redirects the info to the correct controller, in this case the user one, which communicates to the database to check the given information. If the information inserted are correct the controller generates a confirm message which is passed back to the user app that opens the account page of the customer, if the information were wrong the controller generates another message, to communicate that something went wrong, which also is passed back to the application that shows an error message and some recovery options to the customer (in case he/she forgot the password).

2.5.2 Make a Reservation Sequence Diagram


User
User App
Router
Safe Area Controller
Reservation Controller
User Controller
DataBase Server
opt : if the user selects a custom location
clicks on Set Position()
showTheSetPositionForm()
selects position()
clicks the confirmation button()
sendChoiceInfo(a: SafeArea)
setTheSafeArea(a: SafeArea)
clicks on Make a Reservation()
sendReservation(r: Reservation)
checkCarAvailability(r: Reservation)
verifiesUserRequests(r: Reservation)
requestsUserBalance(u: User)
requestsDbForUserInfos(u: User)
alt
[the user doesn't have enough money on his acoount]
shows error message()
sendRequestsForUserInfos()
rquestsUserInfos(u: User)
requestsDbForUserInfos(u: User)
shows the account page()
[there are cars available in the selected area]
showsAllAvalableCars()
selects a car()
sendCarRequest(c: Car, r: Reservation)
reserveTheCar(c: Car, r: Reservation)
insertTheCarInReservation(c: Car, r: Reservation)
addReservationDataToUserProfile(u: User)
storeUserInfo(u: User)
shows confirmation message()
[there are no cars available in the selected area]
show the notification options menu()
alt
[user wants to receive notifications]
confirm for notification()
sendNotificationRequest(s: SafeArea)
communicates with(s: SafeArea)
checksAvailability()
sends notification()
[user doesn't want to receive notifications]
rejects for notifications()
sendNotificationDeny(r: Reservation, d: deleteMessage)
Reservation(r: Reservation, d: deleteMessage)
confirmDeletedOperation()
showHomePage()

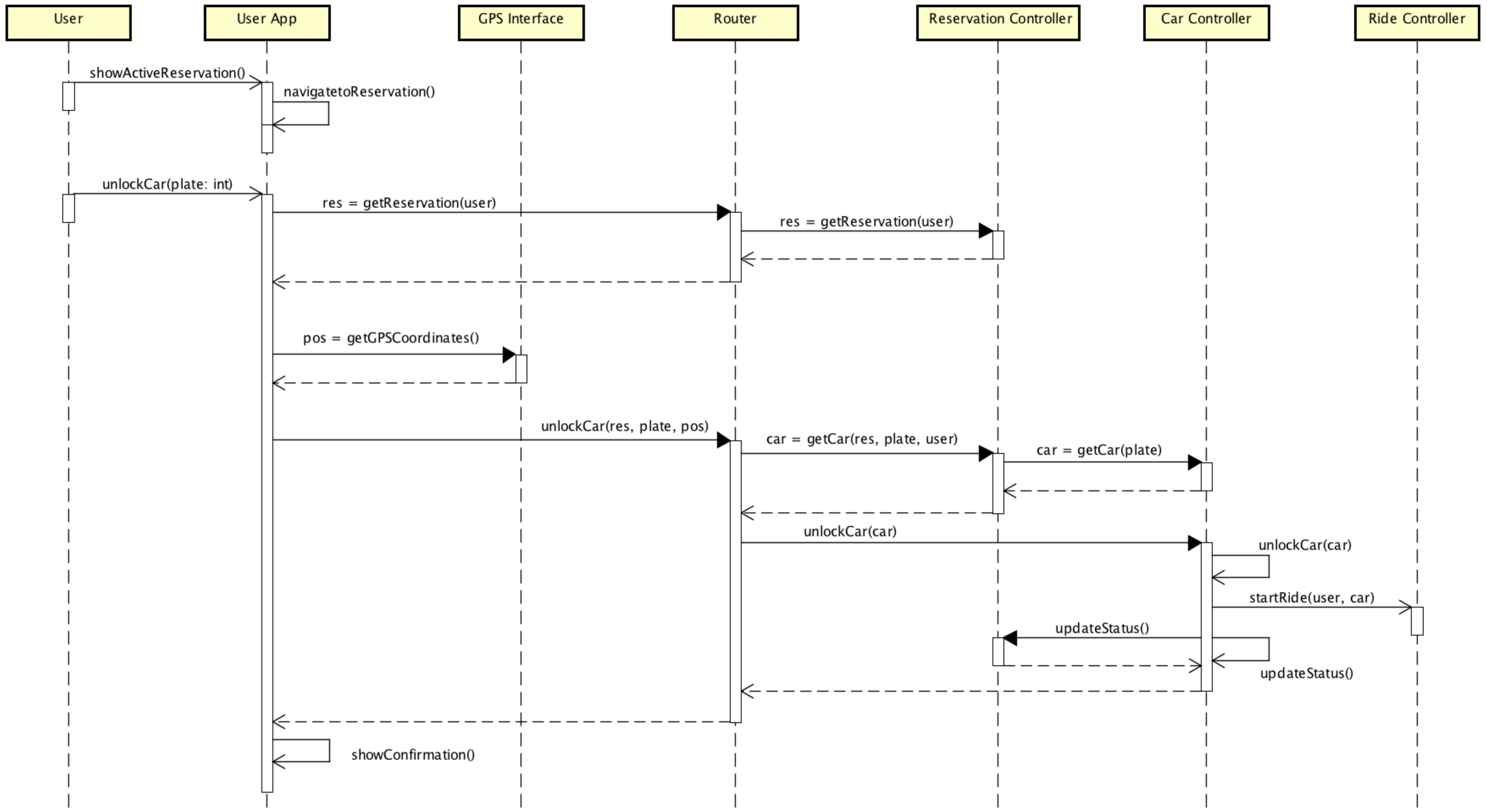
This sequence diagram describes the operations that occurs when a user requests to make a reservation. Once the users reach the home page of the application, the app gives to him an option to set a custom location, if he/she wants the proper button will be pressed, in this case a proper form is showed (app interface) in which the user can select his favorite safe area, after this operation the app communicates with the router in order to send the request to the safe area controller which is now set with the custom safe area. Then the following operations are the same in both the cases that the position has been customized or not. So, the user pushes the make a reservation button, the apps sends the request to the server which then communicates with all the indicated controllers in order to grant to the user the possibility of making a reservation, all the result about the analysis of the controllers are then sent back to the app. From now on some different things can occur. The first case is the one in which the customer has not enough money to make a reservation, the app then shows the problem to the user and then requests detailed information about the user via a communication through the router. This first case has the priority, if this happens the user isn’t able to do anything else, he must recharge the account balance in order to reserve a car and use the service again. The second case describes the situation in which some cars are available and so the reservation can be done. The app shows an interface to the customer in which the available cars are indicated, then when the user made his choice the app communicates with the router, which sends the various information to all the controllers which changes the car state to free to reserved and store all the information regarding the reservation in the DB. The last case describes the situation in which no cars are available in the selected safe area at the moment the reservation is requested, when this happens the system gives a choice to the user, if he wants to wait some time and receive a notification if a car becomes available or if he wants to delete the pending reservation. If the user selects the first option the app sends the application server the decision, then the router communicates with the safe area controller that checks periodically the car availability (through some methods that we’ll cover on the next diagram) and then notifies the user (even if too much time is passed and no cars are available). If the user selects the second option, the app sends his decision to the router which communicates with the reservation controller that deletes the pending reservation and then sends a notification of the cancellation to the user app via the router.

2.5.3 New car available notification



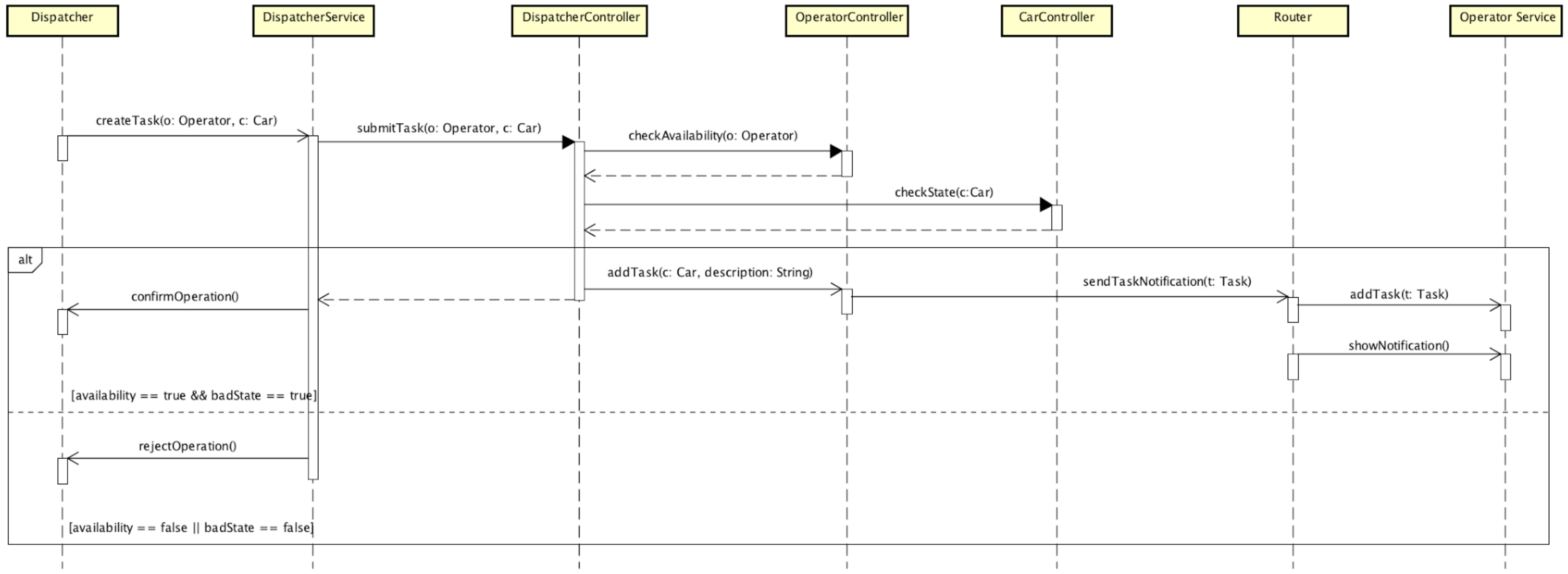
This sequence diagram describes the process that is behind the notification request of the customer. When the request is done, the application sends the information about the user’s preferences to the router which then communicates with the proper safe area controller to stay in “alert mode” in the case some car becomes available. When a car is parked car controller communicates with the safe area controller, if the car is able to do another ride (it has enough battery and it is parked in the right place) the controller notifies the user of the car availability and then the user is able to reserve that car in a process similar to the one described above.

2.5.4 Unlock a car



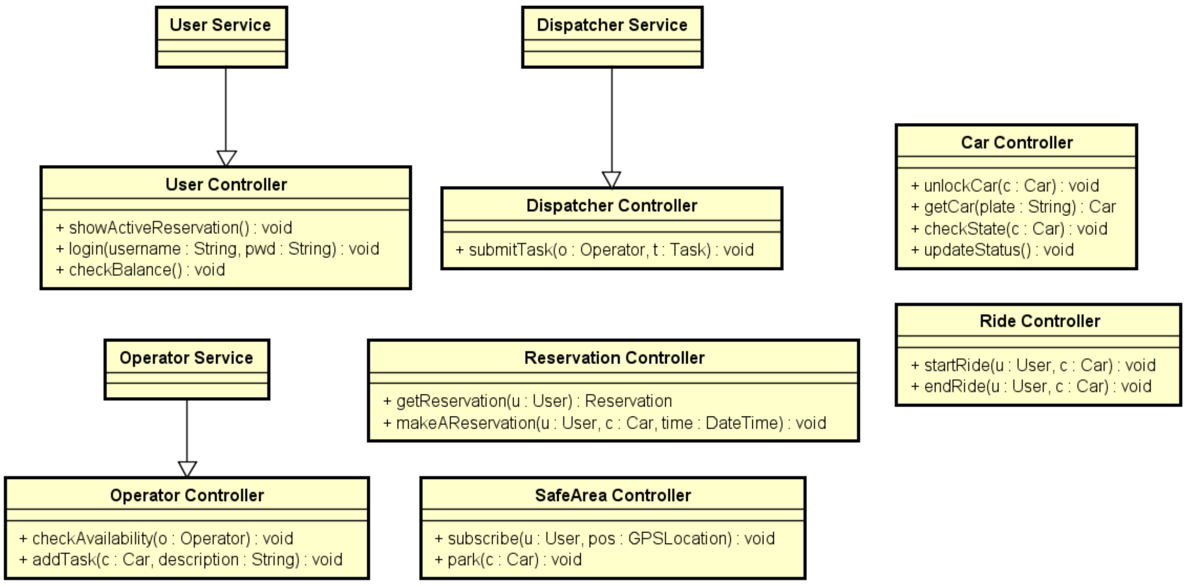
This sequence diagram describes the car unlocking process. After the reservation, the app shows to the user the active reservation, and an unlock interface. The user, in order to unlock the car, must insert the car plate number of the reserved car, which was given to him after the reservation, into the application, then the app checks the user position with the GPS and then sends all the information (position and number plate) to the router, which then communicates with the reservation controller in order to unlock the car. After the unlock has been completed, the car and reservation state will be updated with the new information and the ride controller enters in dialogue with the other controllers in order to start the ride, when all this operations have been completed, the car controller communicates to the router and the customer app receives a notification.

2.5.5 Dispatcher creates a new task for the operator


Dispatcher
DispatcherService
DispatcherController
OperatorController
CarController
Router
Operator Service
createTask(o: Operator, c: Car)
submitTask(o: Operator, c: Car)
checkAvailability(o: Operator)
checkState(c:Car)
alt
addTask(c: Car, description: String)
sendTaskNotification(t: Task)
addTask(t: Task)
confirmOperation()
showNotification()
[availability == true && badState == true]
rejectOperation()
[availability == false || badState == false]

This sequence diagram describes the action that a dispatcher can perform. The dispatcher has access to the information of the cars and the operators so that a task can be created to move or put in charge a car parked in a specific location, and this task must be performed by an operator. The dispatcher chooses from a list of operators using a particular criterion (that might be “choose the nearest operator” or “choose the operator that worked the least this day”) and lets the system notify that operator through his application. Then the operator can see the notification and accept it, so that the task created for him is pushed in his list of tasks, that can also be seen on the mobile application.

2.6 Component interfaces



User Service
User Controller
+ showActiveReservation() : void
+ login(username : String, pwd : String) : void
+ checkBalance() : void
Dispatcher Service
Dispatcher Controller
+ submitTask(o : Operator, t : Task) : void
Car Controller
+ unlockCar(c : Car) : void
+ getCar(plate : String) : Car
+ checkState(c : Car) : void
+ updateStatus() : void
Ride Controller
+ startRide(u : User, c : Car) : void
+ endRide(u : User, c : Car) : void
Operator Service
Reservation Controller
+ getReservation(u : User) : Reservation
+ makeAReservation(u : User, c : Car, time : DateTime) : void
Operator Controller
+ checkAvailability(o : Operator) : void
+ addTask(c : Car, description : String) : void
SafeArea Controller
+ subscribe(u : User, pos : GPSLocation) : void
+ park(c : Car) : void

2.7 Selected architectural styles and patterns

2.7.1 3-Tier Architecture

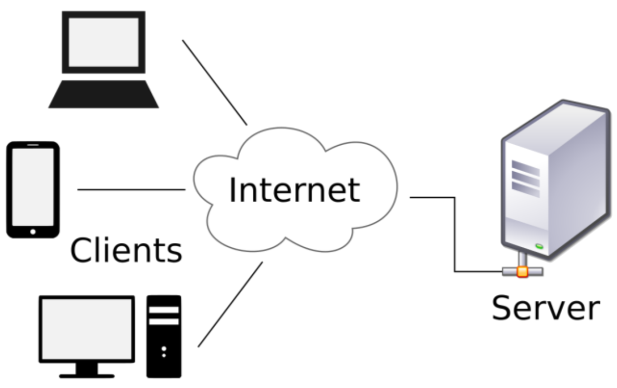
The architectural style that we have chosen for our application is a standard 3-tier architecture. The tiers are the following:

1. GUI Tier: Thin clients. They display information and make the different services and functionalities reachable from the client. Clients can be smartphones, laptops, PCs and they communicate with the other tiers through the internet.
2. Application Tier: It controls application functionality, manages requests coming from the clients and sends results, data and notifications to them. It retrieves data from the Data Tier.
3. Database Tier: Houses database servers where information is stored and retrieved. Data in this tier is kept independent of application servers.

2.7.2 Design patterns

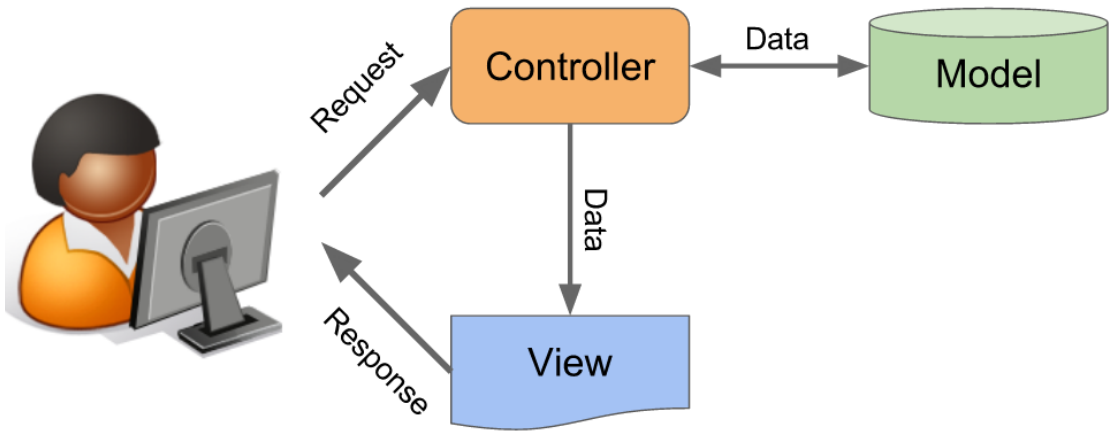
Client Server

We designed our application to follow a client-server approach and this is at the basis of the whole system's structure because the application should be distributed, reachable from a large number of different devices and it needs to provide the service to all of them. We designed our clients to be as thin as possible, letting the tasks regarding the management of requests and data to the servers, while the clients should only visualize results and provide the users with the possibility of exploiting all the functionalities they are allowed to use. Other reasons that led us to a client-server approach are: high maintainability and scalability, security and the synchronization of all the data involved in the PowerEnJoy system.



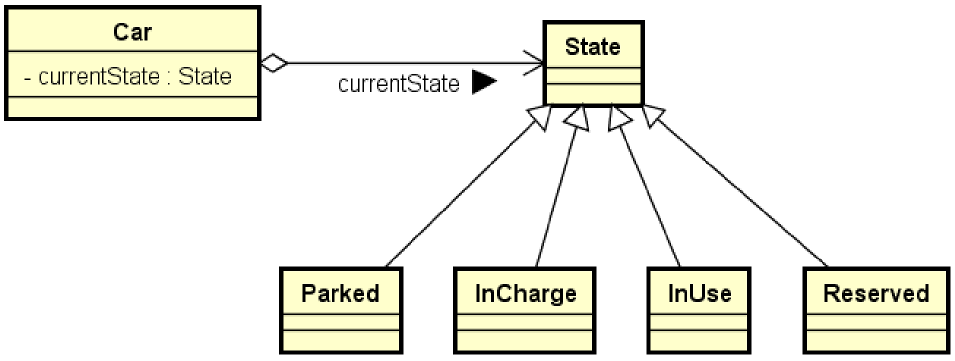
Model-View-Controller

A model-view-controller pattern has been followed since the beginning of the design, dividing the tasks made by the different components to increase modularity and to adapt to the recognized best practices. The MVC pattern fits very well our choice to use a 3-tier architectural style, since the view might be thought as the interface and what the user can see, the controller can be seen as the manager of data and the requests coming from users and all the devices composing our system and the model can be seen as the representation of the system's entities.



State pattern

To implement the state of the cars, already shown in the state chart diagrams, we use a state design pattern. This choice has been made because cars in different states have to be controlled in different ways by the system and have different behaviors. Also, cars change lots of states during their lifecycle.



3 Algorithm design

```
float enstablishCostOfTheRide(Ride ride){
        float discount=0;
        float ticket=0;
        float rideCost=ride.getCost();
        if(ride.getPassengersAtStart()>=2){
                discount+=rideCost/10;
        }
        if(ride.getBatteryPercentage()>=50){
                discount+=rideCost/5;
        }
        if(ride.getCar().carIsParkedInASpecialArea()&&ride.getCar.isPlugged()){
                discount+=(rideCost/100)*30;
        }
        if((ride.getCar().getBattery()<20)||ride.carDistanceFromSpecialArea(ride.getCar())>=3){
                ticket=(rideCost/100)*30;
        }
        rideCost=rideCost-discount+ticket;
        return rideCost;
}

float carDistanceFromSpecialArea(Car car){ //nearestneighborsearch kdtree
        Coordinate carPosition=car.getCoordinates();
        Kdtree tree= new Tree();
        ArrayList<Coordinate> specialSafeAreaPositions=new ArrayList<Coordinate>();
        ArrayList<Node> allNodes= new ArrayList<Node>();
        for(specialSafeArea sp: getSpecialSafeAreas()){
                specialSafeAreaPositions.add(sp.getCoordinates());
        }
        specialSafeAreaPositions.add(carPosition);
        tree=new buildKdtree(tree.root, carPosition);//the tree is built recoursively considering each
coordinate as a node and dividing
          //the nodes in multiple areas splitting the whole area in half alernatibly on x and y
         //and each time the depth of the node is updated
        return getDistanceFromNearestParkingArea(tree, carPosition);
}
```

```
Kdtree buildKdtree(Node nodeOfTheTree, ArrayList<Coordinate> Coordinates){//each node has
coordinates, depth, coordinates,

                                                    //left and right node
          int numOfSafeAreas= coordinates.length;
          int median = numOfSafeAreas>>1;
          if (median>1){
                    sortAlternativelyTheArray(Coordinates);
                    buildKdtree(node.atLeftNode(upgradeDepth),
splitTheNodeswithTheMedianAtHeight(Coordinates));
                    buildKdtree(node.atRightNode(upgradeDepth),
splitTheNodeswithTheMedianAtWidth(Coordinates));
          }
    node.Coordinate = Coordinates[median];
}
//the node can now be found with the depth
float getDistanceFromNearestParkingArea(Kdtree tree, Coordinate positionToSearch){
          Node nodeToFind = tree.searchInTreeForNode(positionToSearch);
          Node nearNode = tree.findTheNearestNodeUsingAnotherNode(nodeToFind);
          return distance(nodeToFind, nearNode);
  }
 }
}
```

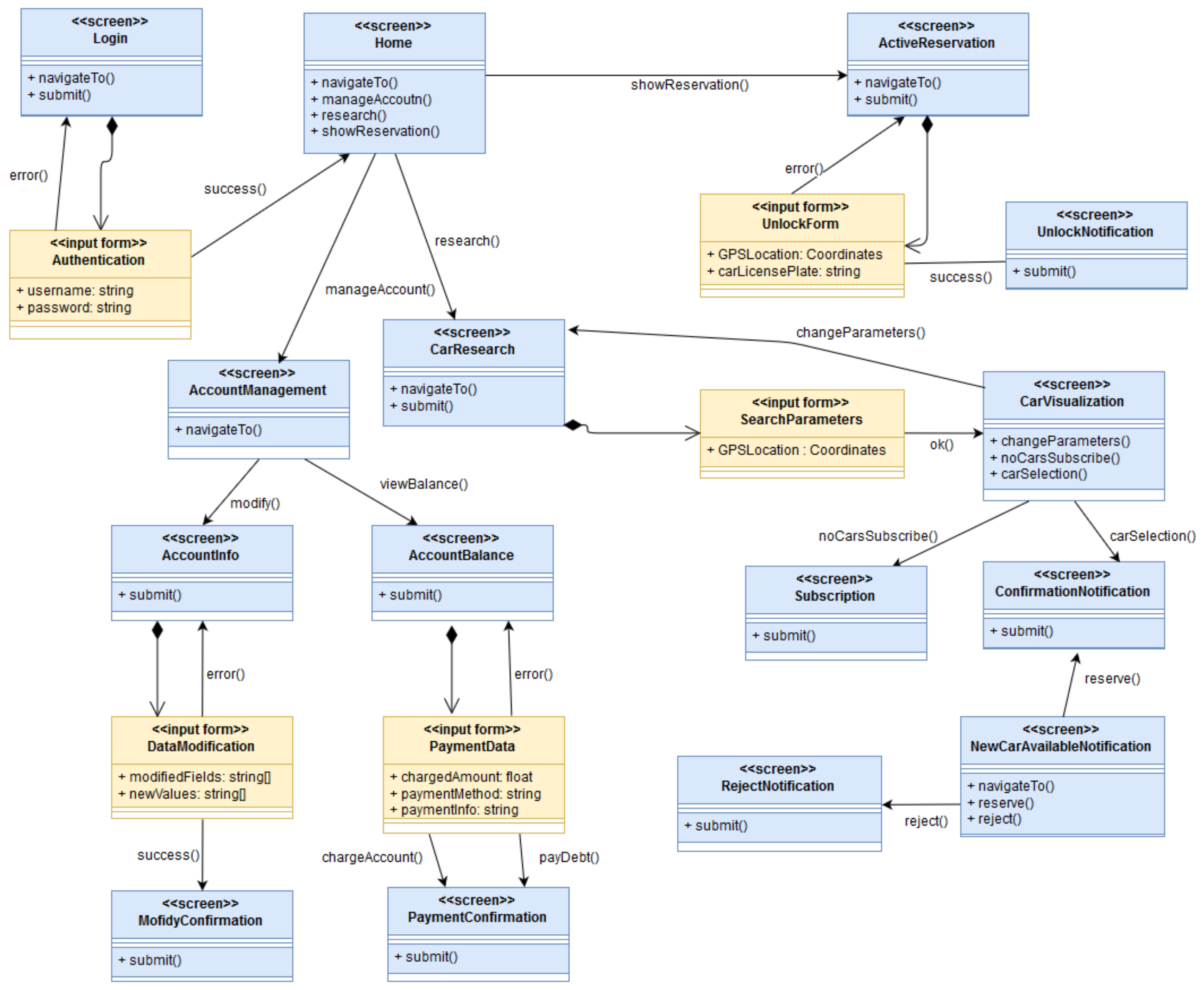
4 User interface design

4.1 Interface mockups

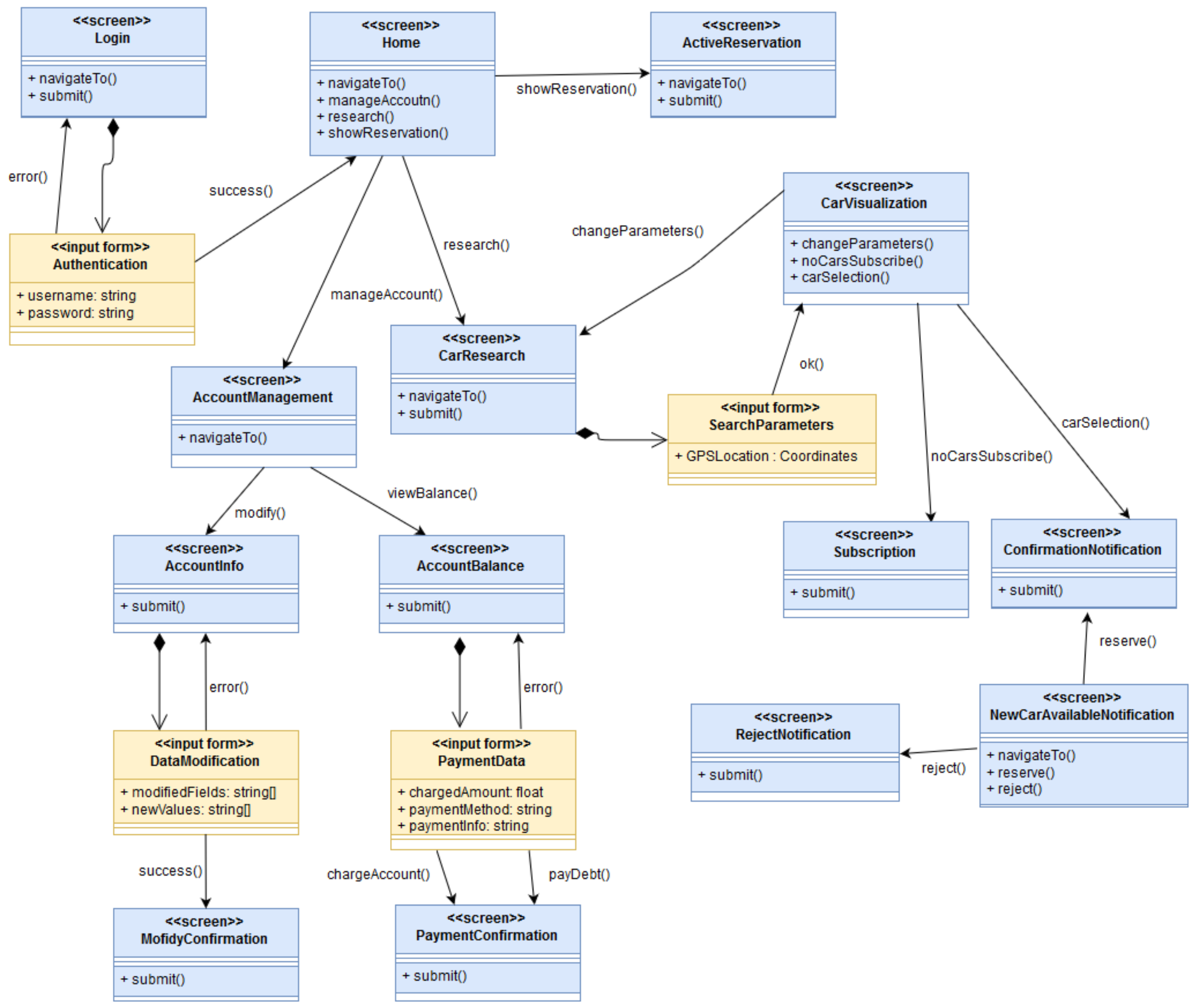
Mockups for the interfaces are already in the RASD in paragraph 3.2.

4.2 UX Diagrams

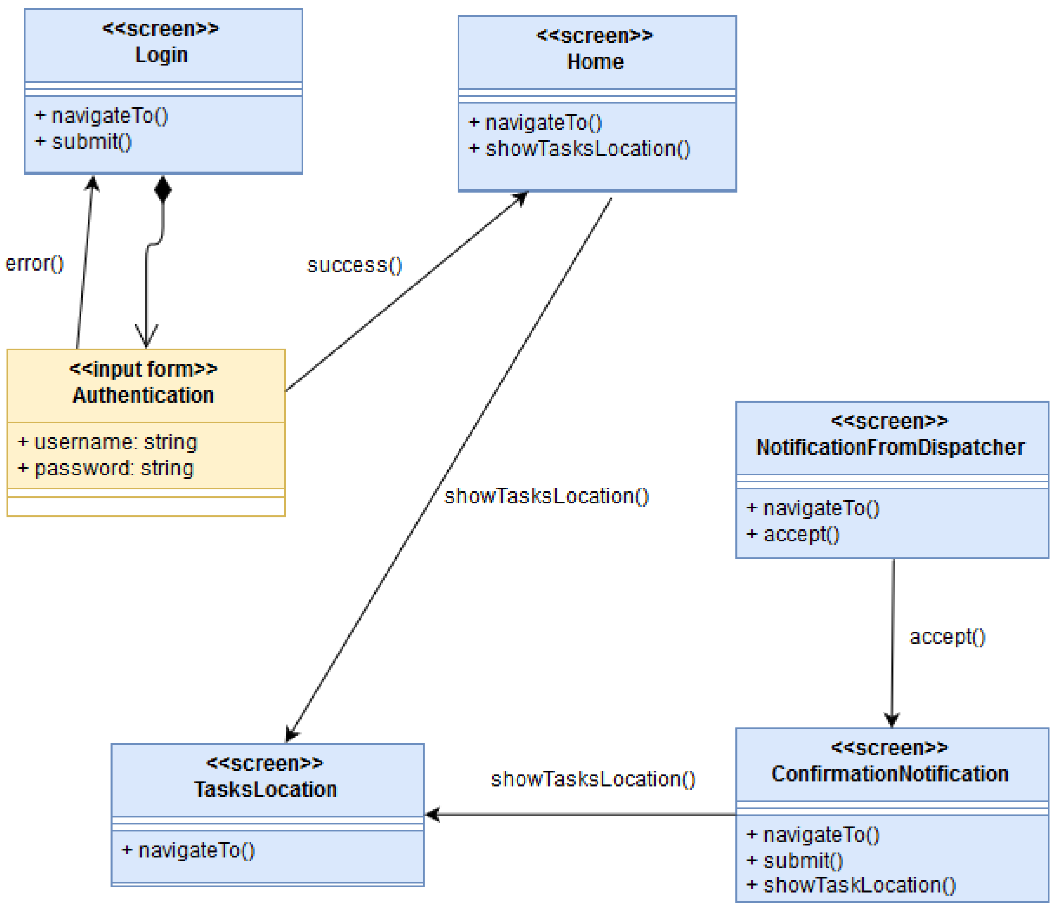
UX user mobile:



UX user browser:



UX operator mobile:



5 Requirements traceability

In this section, we write which components take care of the requirements needed to fulfill the goals. To see which are the requirements of every goal please refer to chapter 3 of the RASD.

- **<G1> Allow a person to register into the service**
  - User Application
  - Router
  - User Controller

- **<G2> Allow user to log into the application**
  - User Application
  - Router
  - User Controller

- **<G3> Reservation of a car for up to one hour before a user picks it up**
  - User Application
  - Router
  - User Controller
  - Reservation Controller
  - Safe Area Controller
  - Car Controller

- **<G4> Find a car near their current location or near a specified location**
  - User Application
  - Router
  - Safe Area Controller
  - Car Controller

- **<G5> Pick up a car from a safe area and reach another safe area**
  - User Application
  - Router
  - User Controller
  - Reservation Controller
  - Safe Area Controller
  - Car Controller
  - Ride Controller

- **<G6> Drive their car if the amount of money they have on their account is enough**
  - User Controller
  - Ride Controller

- **<G7> Pay for their rides with the money they put on their PowerEnJoy account**
    - User Controller
    - Ride Controller

- **<G8> Receive discounts based on their good behavior**
    - User Controller
    - Ride Controller
- **<G9> The dispatcher knows the position of all the cars**
    - Car Controller
    - Dispatcher Service
    - Router
    - Dispatcher Controller

- **<G10> The dispatcher knows the battery percentage of every car**
    - Car Controller
    - Dispatcher Service
    - Router
    - Dispatcher Controller

- **<G11> The dispatcher can send operators to move or charge cars when needed**
    - Car Controller
    - Dispatcher Service
    - Router
    - Dispatcher Controller
    - Operator Controller
    - Operator Service

6 Additional Information

6.1 Hours of work

Time spent to write this document:

Manuel Parenti : 22 hours

Alberto Zeni : 22 hours

6.2 Used Tools

The tools that supported us are:

MS Office Word 2016: to write and redact this document;

Astah: to create UML diagrams;

Draw.io : to create the ER and UX diagrams;