Design Document- PowerEnJoy

POLITECNICO
MILANO 1863

Version 1.0

- Bagna Francesco Matteo (mat. 878556)
- Barletta Carmen (mat. 877129)

Commit date: 11/12/16

Summary

1: Introduction

1.1: Purpose

The purpose of the DD (Design Document) is to give more details on the PowerEnJoy project, along with views on the components that will be designed in order to reach the goals of the system. This document, addressed to developers of the system, will identify, and as such make simpler to develop, the high level architecture, the main components of the system, some algorithms and design choices for the project.

1.2: Scope

In this document we included the diagrams that we created for the design of the system: High level view diagrams, which show the basic devices and components of the system; Component view diagrams, which show a deeper view on the components that will be designed for the system; Deployment view diagrams that show the devices on which the parts of our systems will be installed; Runtime view diagrams the workflows and message exchanging between the components for some of the use cases already presented; Component interfaces offered by the components of the system; the Architectural styles and decisions made on design and protocols; A view on the most important algorithms for the system; Some of the user interfaces for the system; Finally, the components that make possible the fulfillment of the goals of the project. For a deeper view on the general scope of the project, see the RASD document, section 1.2.

1.3: Acronyms, abbreviations and definitions

For other Acronyms, abbreviations and definitions see the RASD document.

- RASD: Requirement Analysis and Specifications Document.
- DD: Design Document.
- Administrator: The person who administrates the system.
- HTTP: HyperText Transfer Protocol.
- TCP: Transmission Control Protocol.
- IP: Internet Protocol.
- GEB: Green e-Box.
- NFC: Near Field Communication.
- DBMS: DataBase Management System.
- JASON: JavaScript Object Notation.
- REST: Representational State Transfer.
- RESTFul: REST without session.

1.4: References

References used in this document are:

- The RASD document of our group.
- Assignments AA 2016-2017.pdf

- Sample Design Deliverable Discussed on Nov. 2.pdf
- Pervasive Data Management in the Green Move System: a Progress Report
- Green Move: A Platform for Highly Configurable, Heterogeneous Electric Vehicle Sharing.
- A Flexible Architecture for Managing Vehicle Sharing Systems.

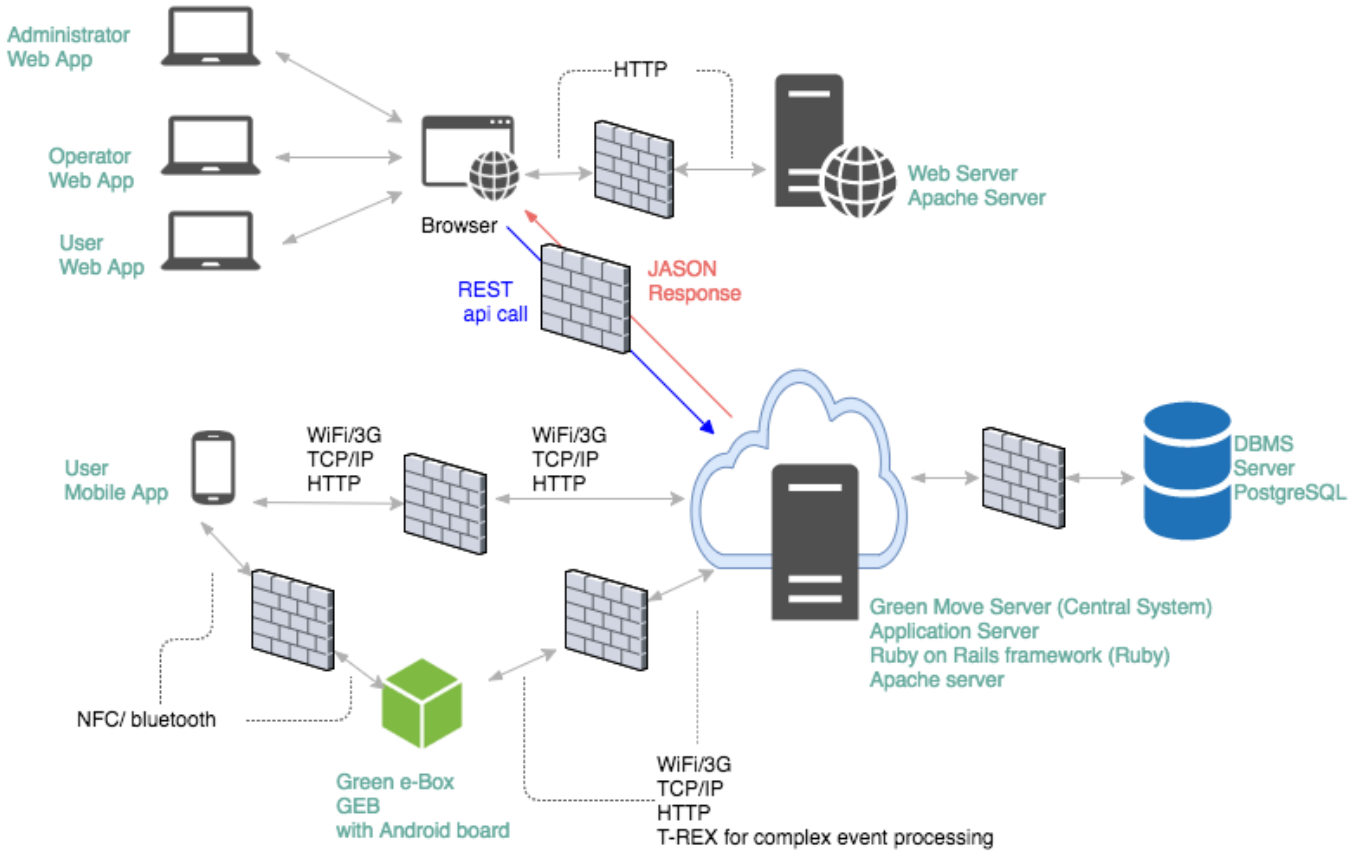
1.5: Document Structure

The DD document will contain a sequence of chapters that will explain the choices that we made during the project.

- **1. Introduction:**
  Introduces the general aspects and the scope of the project.
- **2. Architectural Design:**
  Describes the general design of the system. Contains the high level views, the component view, the Deployment view, some Runtime views, the Component Interfaces and other architectural decisions.
- **3. Algorithm Design:**
  Contains the main algorithms of the system, in our case the reservation and ride management.
- **4. User Interface Design:**
  Contains some of the user interfaces for the project.
- **5. Requirements Traceability:**
  Contains a specification of the components that fulfill the goals described in the RASD.
- **6. Effort Spent:**
  The number of hours spent for doing this document.
- **7: References:**
  Programs used to make this document.

2: Architectural design

2.1: Overview



Hardware infrastructures.

Server of the Central System:

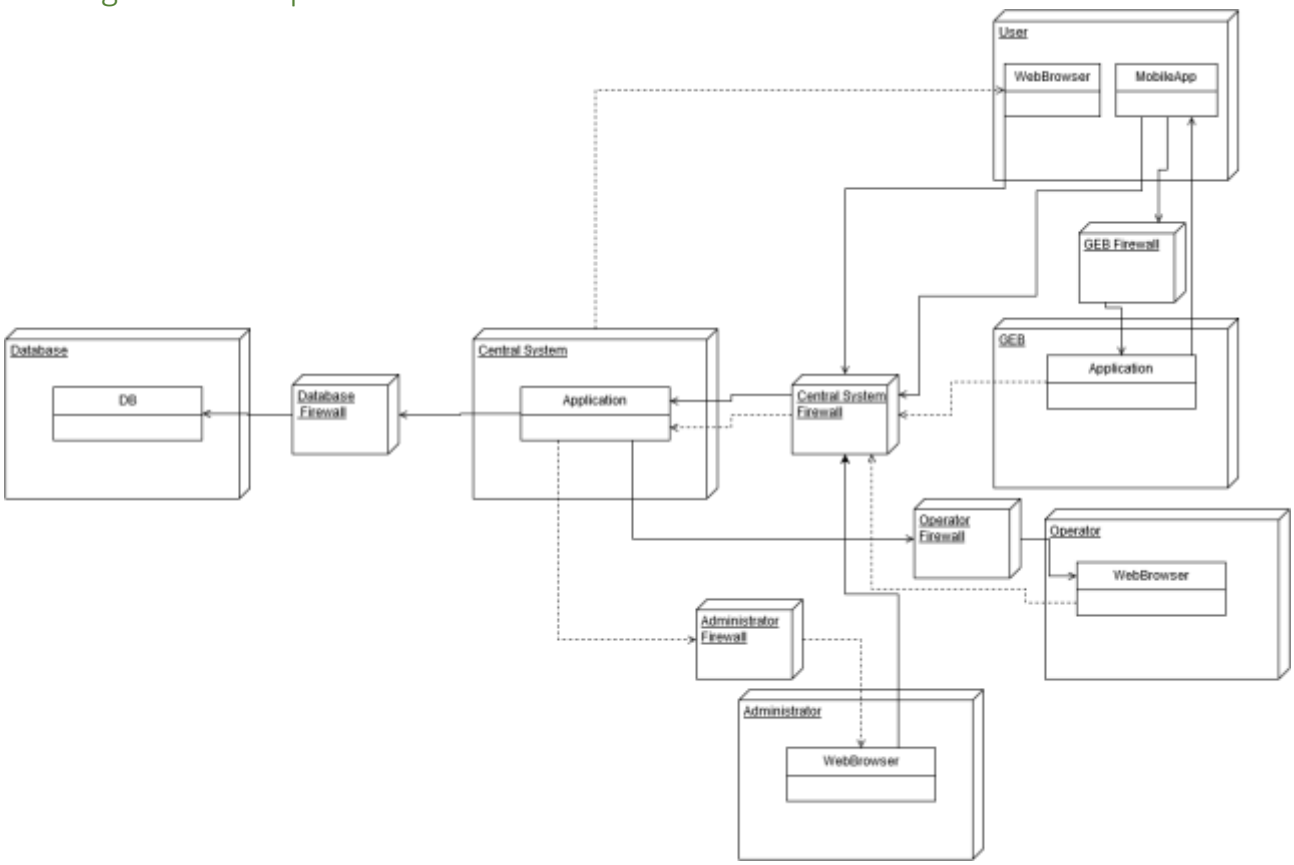
The size and number of these servers can largely vary depending on the number of users accessing the server, thus can't be determined upfront.
For this reason, a cloud architecture has been chosen as the base for building the PowerEnJoy hardware infrastructure.
In the Database we only store the active reservations in order to prevent a denial of service attack, in which numerous reservations are made and then cancelled or no ride is done.
Ruby on Rails framework is an open-source web-app framework that has yet been used by the developers of the GEB system because of its wide variety of interpreters and the easy implementation of design pattern. Ruby on Rails uses Ruby code. JRuby is an implementation of Ruby VM running on a Java VM, in which Ruby Threads are mapped in Java Threads and Java libraries are integrated. For these reasons the already available Java client library for T-Rex could

be used without implementing it from the beginning and all these motivations led GEB developers (and also us) to the decision of using this technology.

Green e-Box:

Each vehicle is equipped with a Green e-Box, which allows each car to interact with the Central System: it is composed of an embedded board and an android board. The purpose of the GEB is to acquire the vehicle signals from sensors and to handle the connection both with the Central System and with the User's smartphone. The GEB is directly connected to the permanent 12V line of the vehicle, in order to have a constant monitoring of cars. Through his smartphone the user is able to retrieve electronic keys needed to open and close the car and to use it. This is done with key encryption in order to assure security.

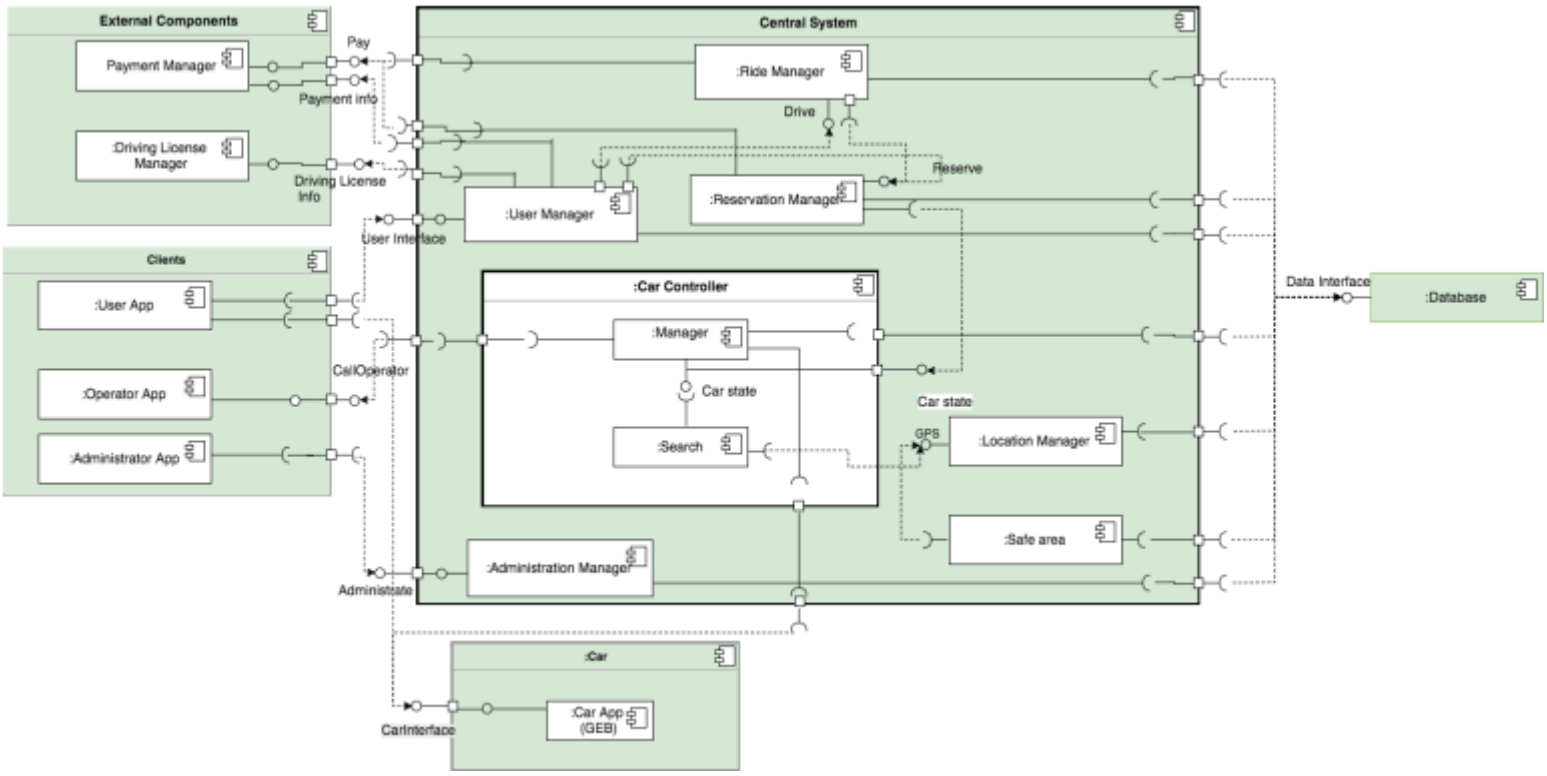
2.2: High level components and their interaction



The high level components of our architecture are mainly six. The most important is the Central System, a singleton that can receive requests from the Users, which can be done from a Web Browser or from a Mobile App. The Browser only lets the user register himself, while the Mobile App lets the user make reservations. This communication is synchronous because the user must wait for the central to acknowledge its requests before continuing the operations. The Central system can then send asynchronous messages to the user, for example sending e-mails with the

details of a reservation. The Central system can receive data from the GEB system installed on the cars, which contain the car's status and other info on the Ride the car is doing. The GEB system can also send to the system requests for recharging, which will be the passed synchronously to an Operator. At the end of the recharging task the car automatically sends a finish notification to the central system.
An Administrator can send requests to the Central System in order to manage and see the data of the system. This communication is done synchronously, because the Administrator must wait for the Central System to process his request. The Central system can also push notifications asynchronously to the Administrator. The Central system can extract synchronously information from the Database.
The GEB system can also directly interact with the User, like when the User wants to unlock the car or when notifications are sent from the GEB to the Mobile App of the User.
All the entering communications are protected by firewalls. The user firewalls are not displayed, because we aren't sure there are any. If there are some, also those connections are protected.

2.3: Component View

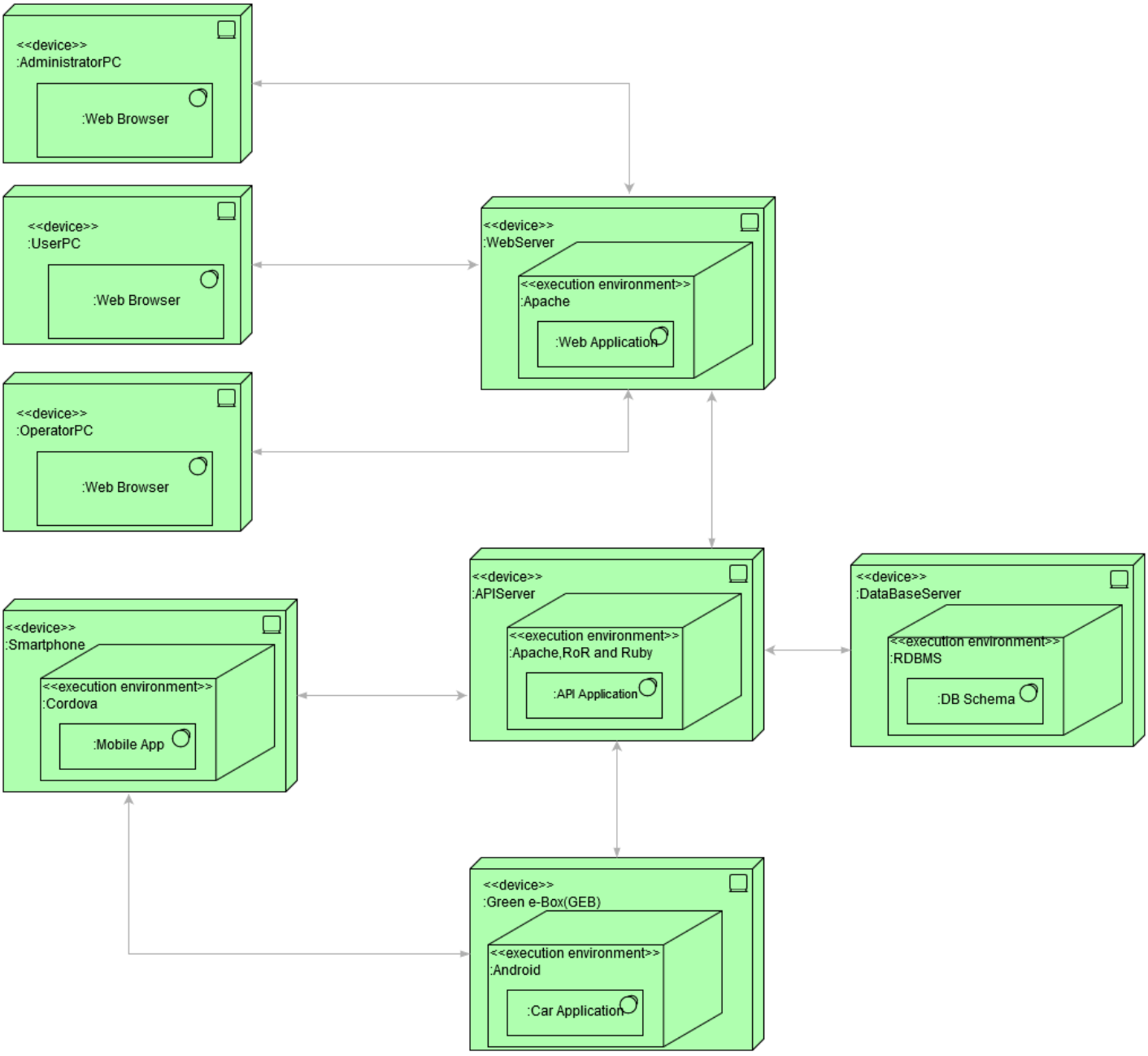


These are the main components of our system:

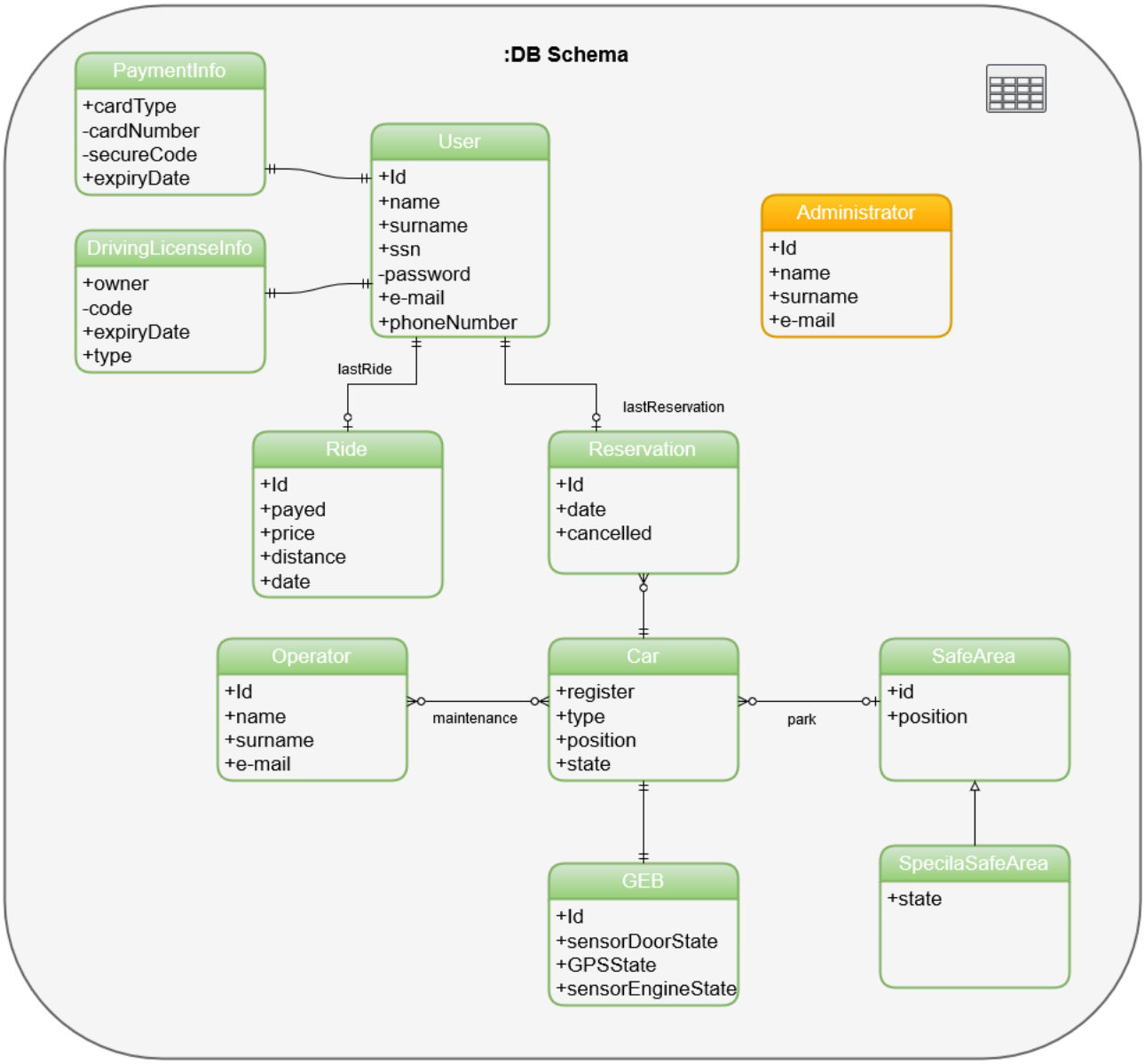
- Payment Manager: The external component that manages payment and the procedures to verify the payment information of a user.
- Driving License Manager: The external component that manages the procedures to verify the driving license of a user.
- User app: The user application is a client used by the user in order to access the system.
- Operator app: The operator application lets the operator receive and respond to recharging requests.
- Administrator app: The application used by the administrator to monitor all the system.
- User Manager: Manages all the user-related information, passing them to the other components of the system.
- Administration Manager: Lets the administrator access to the data and other components of the system.
- Car App(GEB): The GEB System in the car, tells the statuses of the car to the Car Manager

- Ride manager: Manages all the concurrent rides and the driving operations. A new Ride is instantiated when a user turns the engine of a car on.
- Reservation Manager: Manages all the operations of a reservation and is used as a bridge from the Car manager to the Ride Manager.
- Car Manager: Manages all the car statuses and the information with the other parts of the system.
- Car Search: Manages the search for cars during a reservation and the searching for safe areas during a ride.
- Location Manager: Manages the GPS system and the other location-related information.
- Safe Area: Manages the safe areas and special safe areas.
- Database: The DB of the system.

2.4: Deployment View

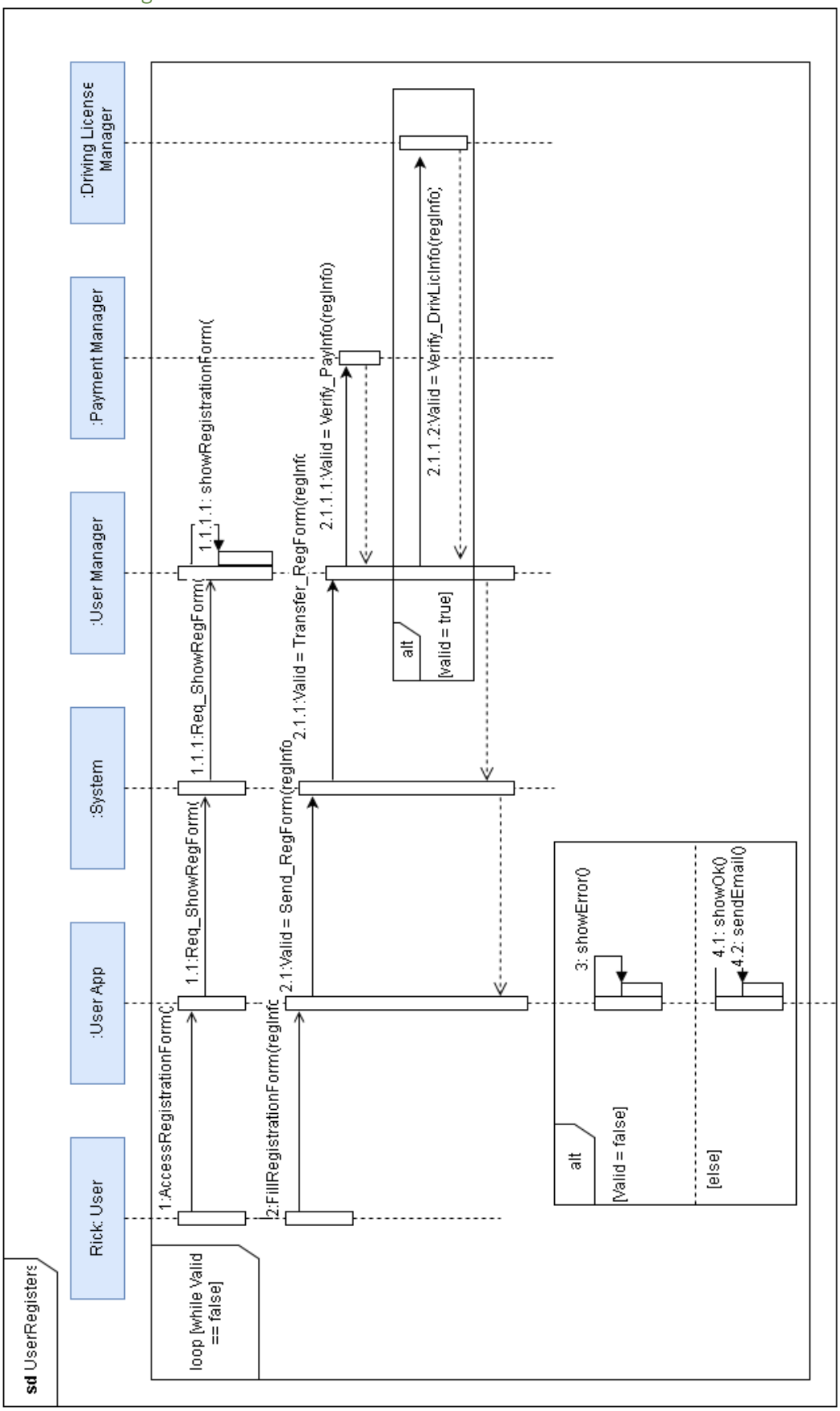


Focus on the Database internal structure.



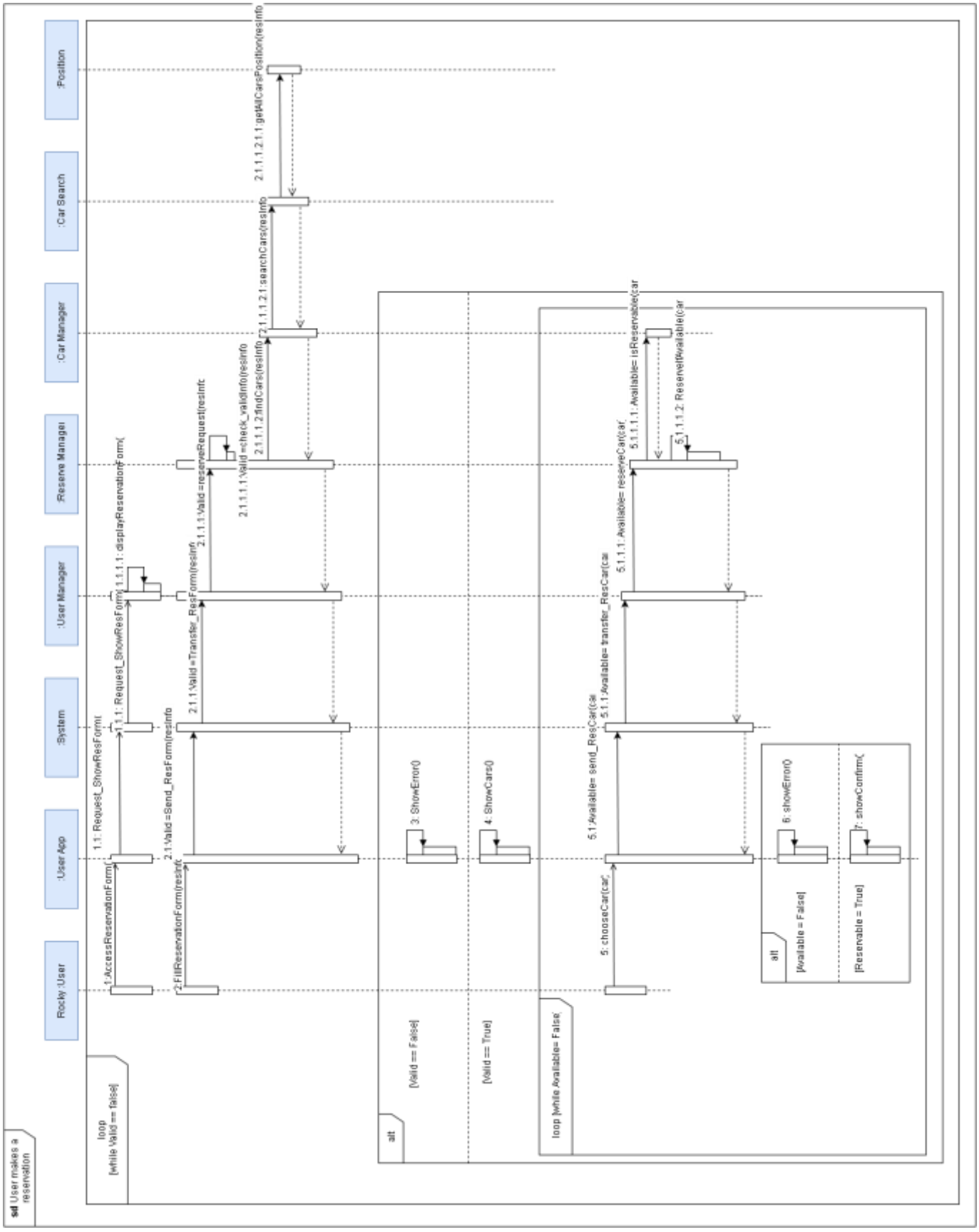
2.5: Runtime View

2.5.1: User registers



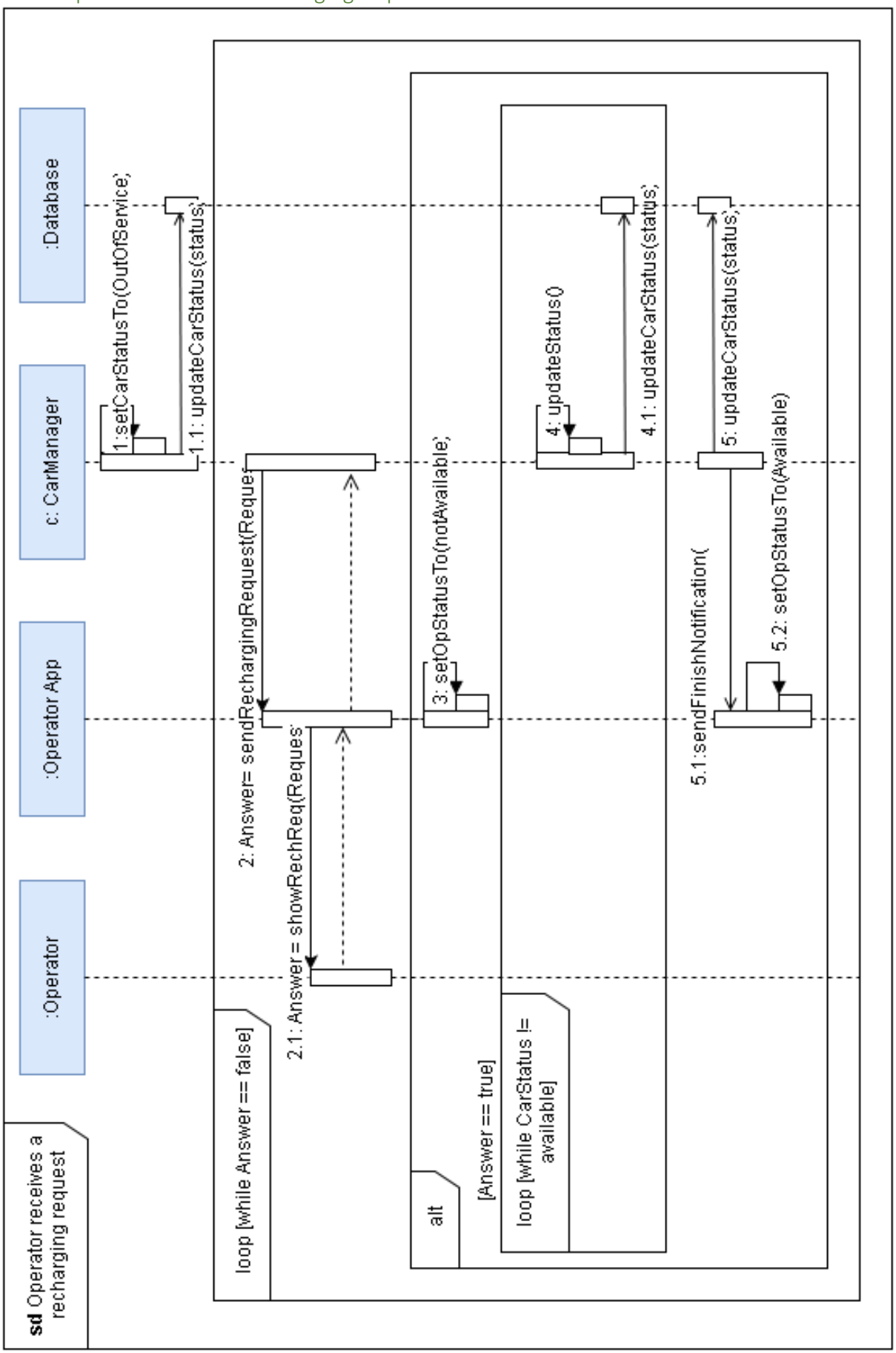
In this diagram we can see how an unregistered user (Rick) registers himself to the system. After having seen the registration form, the user has to fill the form, and all the information is sent through the User App to the System and then to the User Manager. Then the payment information is verified using the Payment Manager and, if the information is correct, also the Driving License is verified. Then the response is sent to the User App, that communicates the result of the operation.

2.5.2: User makes a reservation



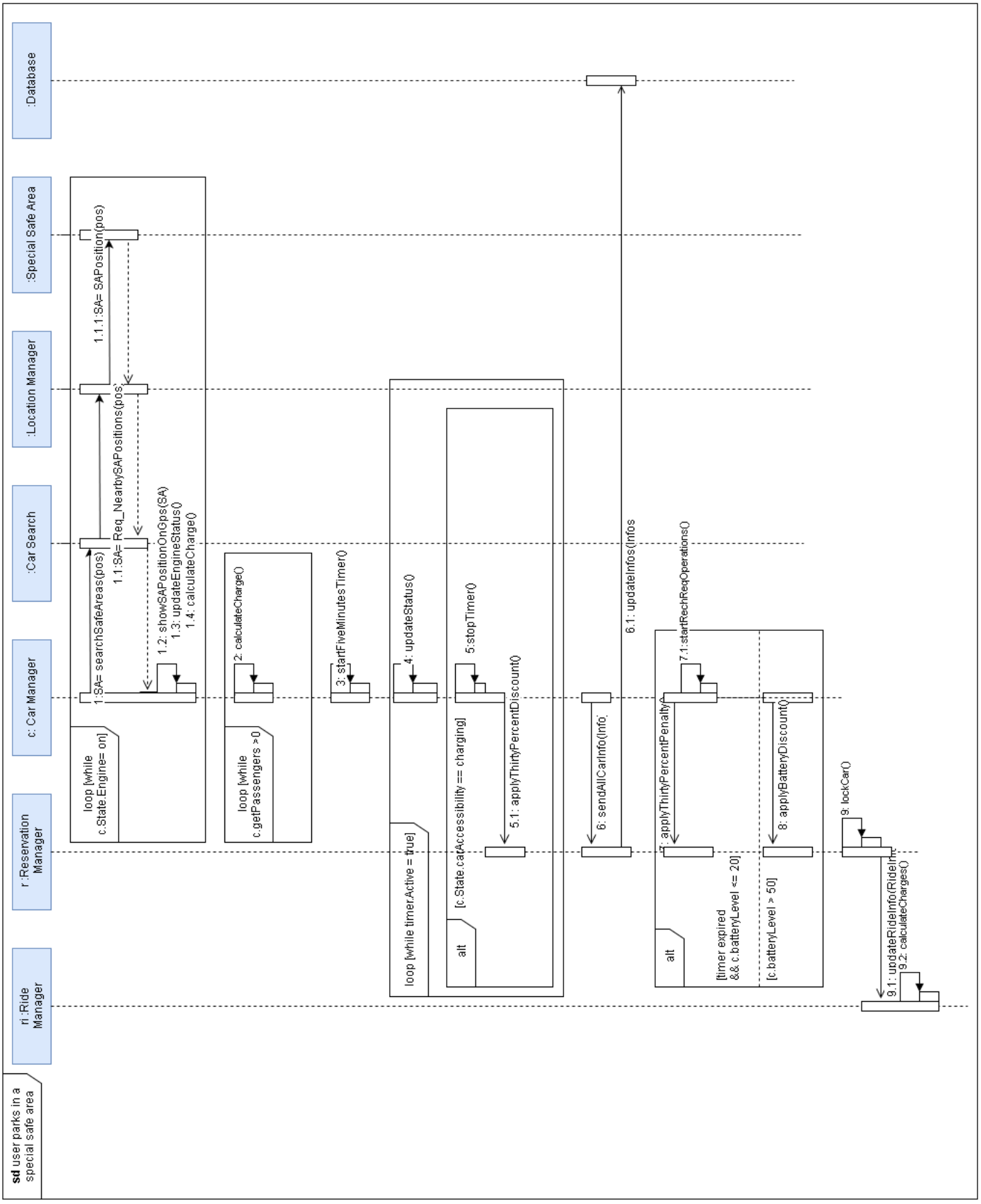
In this diagram we can see how a logged in user makes a reservation. After having seen the reservation form, the user fills it (with an address from where to do a search or with his GPS position) and the form is sent to the Reservation Manager. The information is controlled to be valid, then using the Car Search component all the available cars in the proximity are reached and a response containing the available cars (if any) is sent to the User App. If the information was valid, or if there were no available cars, an error message is shown. Otherwise, the user must choose a car by trying to reserve it. The request is sent to the Reservation Manager, that controls another time that the car chosen by the user is in fact available (in order to avoid double reservations), and reserves the car if reservable. Then a notification of the success/fail of the operation is sent to the user.

2.5.3: Operator receives a recharging request



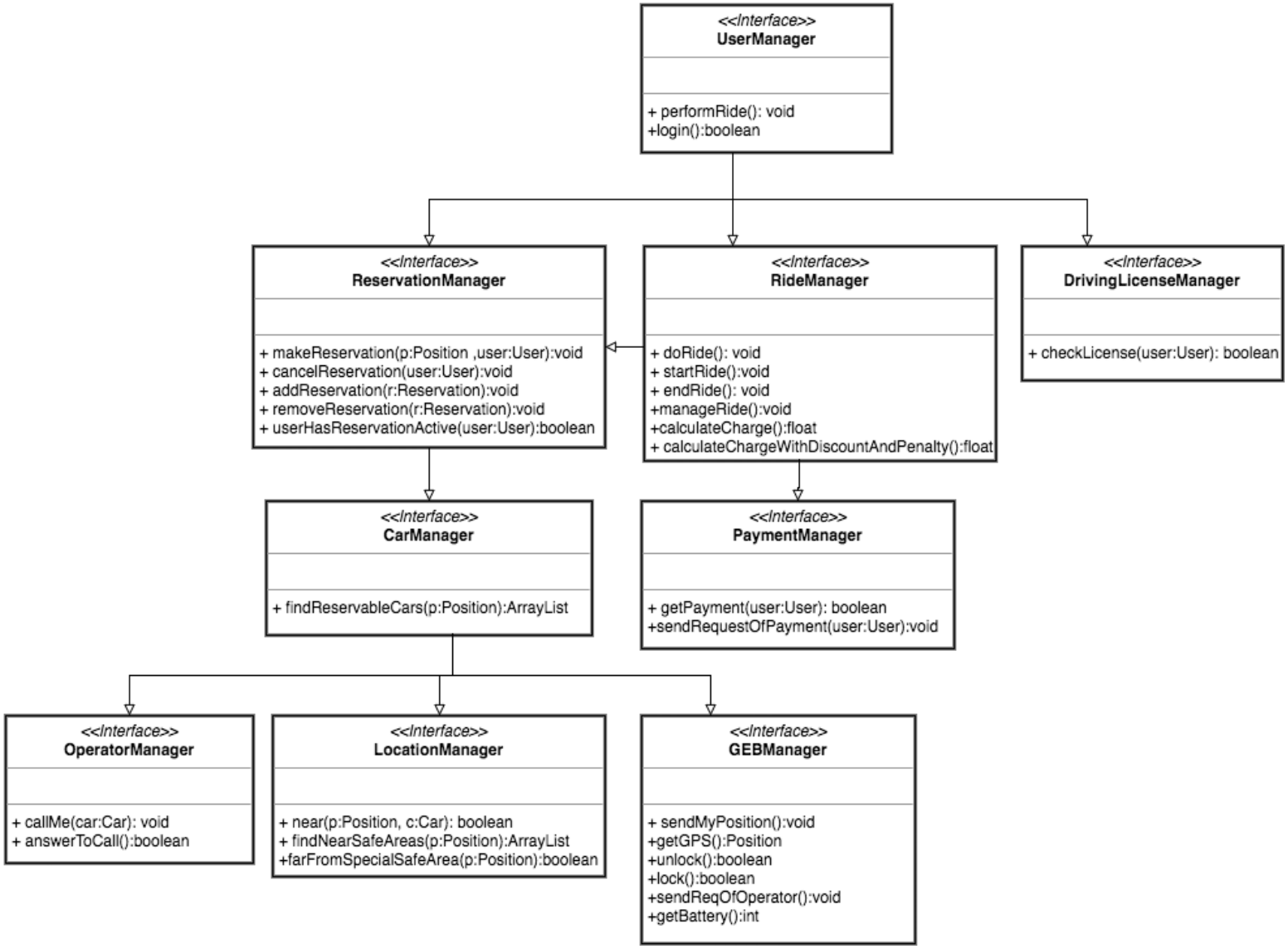
In this diagram we can see how a recharging request is managed. First, the car is set to be not available and its status is sent to the Database. Then the Recharging request is sent directly to the Operator App. The operator must choose its answer. If the Answer is False, another Operator is chosen and another request is sent. If the Answer is True, the Operator App sets the Operator to Not Available, while the Car Manager starts to monitor the car's status and to update it if there are any changes (because a charging car can be reserved like any other). When the car becomes Available, the Car Manager sends a notification to the Operator App to communicate the finish of the procedure, so that the Operator can be called again.

2.5.4: User parks in a special safe area.



In this diagram we can see the last operations done on a ride, In the specific case in which the user parks in a special safe area. All the status updates are taken from the Car App (GEB) component and sent to the Car Manager component in the System (Even if it isn't shown in the diagram for the sake of simplicity). While the car is on, the car searches for safe areas nearby through the Car Search Component, the Location Manager Component and then the Safe Areas Component. It then shows them on its interface (GPS), while calculating charge and updating the car's statuses. Then the car continues charging money until all the people are gone from the car. Then a five minutes' timer is started. If the user puts the car on charge before the end, he receives a discount on his ride. Then all the car's info is sent to the Reservation Manager, which sends them to the database. Then it is calculated the penalty or discount for the charge of the battery. At last, the car is locked, the info of the ride is updated on the Ride Manager and the final charge is calculated.

2.6: Component Interfaces



2.7: Selected architectural styles and patterns

2.7.1: Overall Architecture

We have chosen a three-tiers architecture for our system.

- 1: Database, the Data Access Layer, directly mapped to the Database tier.
- 2: Application Logic, mapped physically both on the Central system and on the car with the GEB system, represents the BLL: Business Logic Layer.
- 3: Thin Client, with a simple user interface mapped both on computers and on user's smartphones.

2.7.2: Protocols

T-Rex: Used by the Central System to communicate with the GEB. It uses the TCP/IP protocol for request-responses to obtain a resilient channel with clients. T-Rex Server included in the Central System exchanges the following types of packets with the car' GEB:

- Ping Packet: to check the connection of clients
- Publication Packet: event packet that contains information
- Subscription and Unsubscription Packet

T-Rex adopts a publish-subscribe paradigm. For this reason, the T-Rex Server opens a TCP socket port and starts listening for event subscriptions and publications.
We have chosen to use T-Rex because the GEB system is directly implemented with this technology, and it's simpler to keep using this one than to adapt other protocols.

HTTP: Used for connection with the internet.

RESTFul API with JSON: used by the clients to interact with the BLL. It is supported by HTTP for the client's authentication needed by the API.

2.7.3: Design Patterns

MVC: Model-View-Controller has been used in our application. Our application server will use Ruby on Rails Framework, because the GEB system supports this Framework and it's easier to use it than to adapt a new Framework. For the web interface we will use the AngularJS Framework.

Façade: We use this design pattern to implement the User Manager component, so that the user can access to a single interface rather than to many interfaces from other components.

Observer: Used by the Car manager to keep updated in the database the states of the different sensors of the car. The Database is an observer of the Car Manager, and the Car Manager is also an observer of the GEB system, which notifies the changes of the car's sensors.

2.8: Other design decisions

Because we needed a maps service for the GPS system and for the search for cars and safe areas, we chose to integrate Google Maps API in our system.

3: Algorithm design

Here are the main algorithms for our system. They display how a reservation can be made, cancelled, how a ride is managed and ended. It is done in a pseudo-Java code.

```
//Methods belonging to the User Manager Class
public class User{
    public Reservation lastReservation;
    Ride lastRide;
    private boolean blocked;

public void performRide(){
    this.lastRide=new Ride(lastReservation);
    this.lastReservation.getCar().unlock();
    this.lastRide.doRide();
}
}
//Methods belonging to Reservation Manager Class
public class ReservationMgr(){
//This set of Reservations is used to keep track of the not cancelled Reservations,
// of the Reservation with a not yet expired timer and  of Reservations which ended Ride
// has a pending payment
    public static ArrayList<Reservation> acriveReservations= new ArrayList<Reservation>();


    public void makeReservation(Position p,User user){
    boolean valid;
//Control if there is yet one reservation for the user or if there is a pending payment
// for his last Reservation and so the user is blocked
//(In both cases he can't make a new Reservation)
     if(userHasReservationaActive(user)){
            if(user.getBlocked())
                Gui.sendMessageOfPendingPayment(user);
            else
                Gui.sendMessageOfExistingReservation(user);
            valid=true;
    }else
            valid=false;

    while(!valid){
        ArrayList cars= CarMgr.findReserveableCars(p);
        if(!cars.isEmpty()){
            Car car=user.chooseCar(cars);
            car.setAccessibilityToReserved();
            user.lastReservation= new Reservation(user,car);
            addReservation(user.lastReservation);
            valid=true;}
        else
            Gui.sendErrorMessage(user);
            }
}
```

```
47  public static void cancelReservation(User user){
48      if(user.lastReservaion.timer.isAlive()){
49          user.lastReservation.setCancelled(true);
50          removeReservation(user.lastReservation);
51          user.lastReservaion.timer.interrupt();}
52  }
53
54      public static void addReservation(Reservation r){
55          acriveReservations.add(r);
56      }
57      public static void removeReservation(Reservation r){
58          acriveReservation.remove(r);
59      }
60      public static boolean userHasReservationActive(User user){
61          for(Reservation r: activeReservations)
62              if(r.getUser().equals(user))
63                  return true;
64          return false;
65
66      }
67
68
69  }
70  //Methods belonging to Reservation  Class
71  public class Reservation{
72      private User user;
73      private Car car;
74      private boolean cancelled=false;
75      public Thread timer= new Thread();
76      private int penalty;
77
78  public Reservation(User u, Car c){
79      this.user=u;
80      this.car=c;
81      reservationTimer();
82  }
```

```
83
84  //If User cancels Reservation (cancelReservation(User user)),
85  //the timer thread is interrupted and if at line 95 is executed
86  //If car's engine is turned on, the timer thread is interrupted and if at line 98
87  //is executed
88  //If none of the two previous events occur, the timer expires,
89  //the user receives a penalty and if he pays it,the Reservation is removed
90  //form the set of activeReservations
91  public void connectionTimer(){
92      try{
93          timer.sleep(3600000);
94      }catch(InterruptedException e){
95          if(cancelled){
96              car.setAccessibilityToAvailable();
97              return;}
98          if(car.getState().getEngine().equals("on"))
99              return;
100     }   penalty=1;
101         car.setAccessibilityToAvailable();
102         user.sendRequestOfPayment(penalty);
103         if(Payment.getPayment(user))
104             ReservationMgr.removeReservation(this);
105         else user.setBlocked(true); //Pending payment
106 }
107 }
108 //Methods belonging to Ride Manager Class
109 public class Ride{
110     Reservation reservation;
111     State carState;
112     Car car;
113     float price;
114     int passengers;
115     boolean dicountPassengers;
116     boolean discountBattery;
117     boolean discountPluggedIn;
118     boolean penalty;
119     boolean paid;
120     public Ride(Reservation res){
121         this.reservation=res;
122         this.car=reservation.getCar();
123         this.carState=reservation.getCar().getState();
124         this.price=0;
125         this.passengers=0;
126         this.discountPassengers=false;
127         this.discountBattery=false;
128         this.discountPluggedIn=false;
129         this.penalty=false;
130         this.paid=false;
131     }
```

```
    public void doRide(){
    startRide();
    endRide();
    }

    public void startRide(){
        while(reservation.timer.isAlive()&& carState.getEngine().equals("off")){
                try{
                    Thread.sleep(200);}
                catch(InterruptedException e){
                    Thread.currentThread().interrupt();
                }
            }
        if(carState.getEngine().equals("on")){
            reservation.timer.interrupt();
            carState.setDoor("locked");
            this.passengers=carState().getPassengers();
            manageRide();}
    }

    public void manageRide(){
        while(carState.getEngine().equals("on")){
              car.getGPS();
              Gui.showNearSafeAreas(reservation.getUser());
              this.price=calculateCharge();
              Gui.showCharge(reservation.getUser(),this.price);
                if(car.getBatteryLevel()<=10)
                    Gui.sendLowBatteryMessage(reservation.getUser())
                if(car.getPassengers()>=3)
                    discountPassengers=true;
              }
    }
```

```
//After the User exits the car and there are no more passengers,
//we give him 5 min of time to put car in charging.
    public void endRide(){
        while(car.getPassengers()>0){
                this.price=calculateCharge();
                Gui.showCharge(reservation.getUser(),this.price);}
        if(car.inSafeArea(car.getGPS())
            carState.setDoor("locked");
        else if(car.inSpecialSafeArea(car.getGPS())){
                    try{
                    Thread.sleep(300000); //5 min timer
                    }catch(InterruptedException e){
                    Thread.currentThread().interrupt();
                    if(carState.getAccessibility().equals("Charging")){
                        discountPluggedIn=true;
                        carState.setDoor("locked");}
                }
            }
        if((car.getBatteryLevel<=20&&!carState.getAccessibility().equals("Charging"))||
             LocationMgr.farFromSpecialSafeArea(car.getGPS())){
                penalty=true;
                carState.setDoor("locked");
                car.setAccessibilityToOutOfService();
                Operator.sendRequestOfMaintenance()}
        if(car.getBatteryLevel()>=50){
            discountBattery=true;
            if(!carState.getAccessibility().equals("Charging")) //if car hasn't been put
                car.setAccessibilityToAvailable();}             // in charging


        this.price=calculateChargeWithDiscountAndPenalty();
        Payment.sendRequestOfPayment(reservation.getUser());
        if(Payment.getPayment(reservation.getUser())){
            paid=true;
            ReservationMgr.removeReservation(reservation);}
        else reservation.getUser().setBlocked(true); //Pending payment
}
```

```
201
202
203 }
204 //Method belonging to Car Manager
205 public class CarMgr{
206     public static ArrayList<Car> availableCars=new ArrayList<Car>();
207     public static ArrayList<Car> chargingCars=new ArrayList<Car>();
208     public static ArrayList<Car> reservedCars=new ArrayList<Car>();
209     public static ArrayList<Car> outOfServiceCars=new ArrayList<Car>();
210
211     public static ArrayList findReservableCars(Position p){
212             ArrayList<Car> reserveableCar=new ArrayList<Car>();
213             for(Car c: availableCars)
214                 if(LocationMgr.near(p,c))
215                     reserveableCar.add(c);
216             for(Car c: chargingCars)
217                 if(LocationMgr.near(p,c))
218                     reserveableCar.add(c);
219             return  reserveableCar;
220     }
221 //All setters belonging to Car class which modify the stateAccessibility
222 //(like car.setAccessibilityToAvailable())
223 // of a Car will switch CarAccessibility and also push/pop the Car
224 // in/from the corresponding CarMgr set.
225
226
227 }
```

4: User interface design

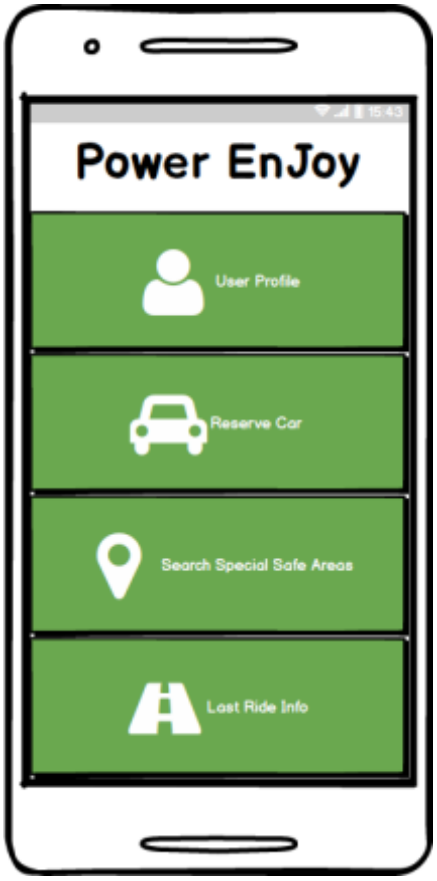
4.1: Mockups

The mobile app will have almost this kind of interface, to interact with the user and let him use the Power EnJoy car sharing service. These mockups have been made, following the “good practices” for designing user interfaces: like depth of navigation when interacting with the system and quick access the main functionalities.

When the user will start the app, the system will show the following login layout:

User Interface 1 Login layout

As the user’s credentials are validated and verified, he will see a main menu option as It’s shown by the following image:

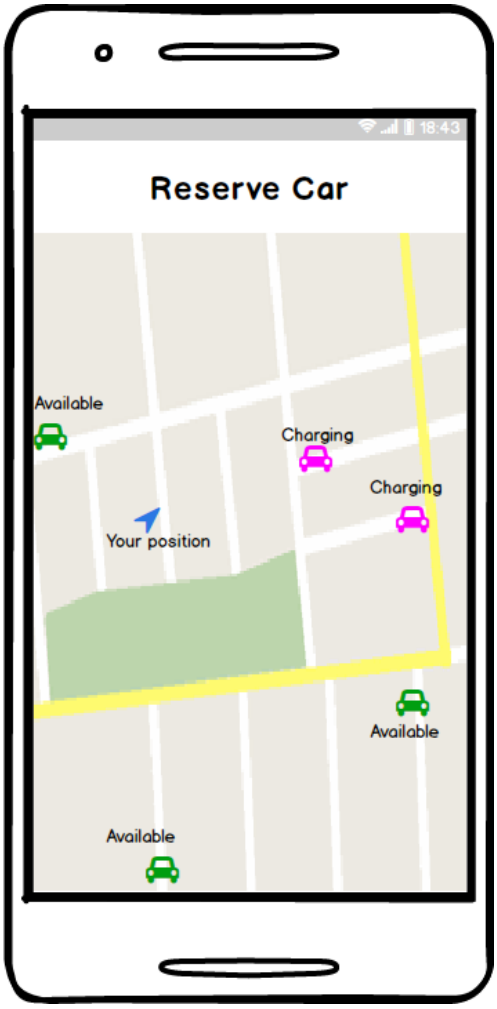


User Interface 2 Full interface main menu

The user profile shows, credentials and payment information, while through the “last Ride Info” he can access to a summary of his last ride if he has already done it and then also manage the pending payment (is he has one) of the ride.
He will also able to search for a special safe area and make a reservation with the “Reserve Car” option, which shows a search bar where the user can enter a specific address of interest for the car’s reservation, as it’s displayed in the following layout:

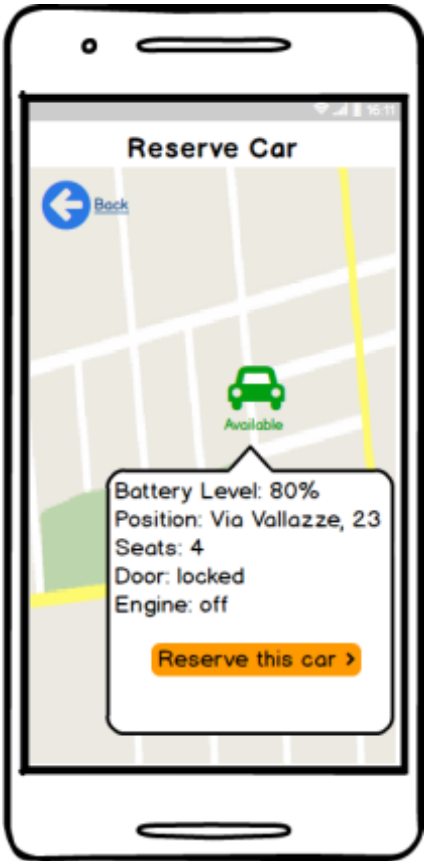
User Interface 3 Reservation Page

After having searched for an address or having given his current position, a map will the nearby cars and their state will be displayed, like in the following layout:



User Interface 4 Showing car nearby

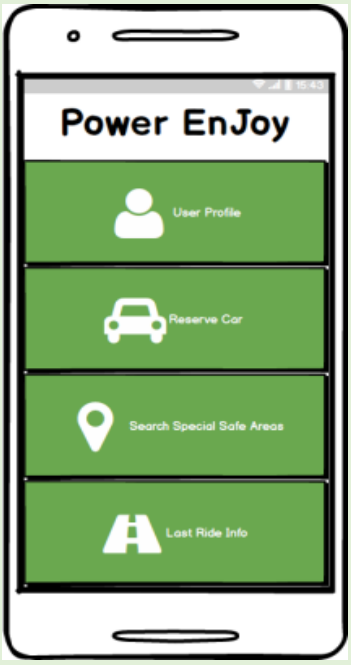
When the user will tap on one of these car, he will see more details about the car's state and if It's available he will have the opportunity to reserve it, as the following image shows:



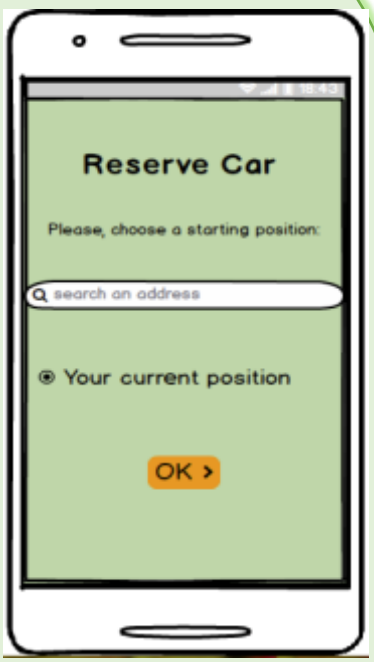
User Interface 5 Car's state

In general, the system will have a typical navigation style, fast and easy to use.

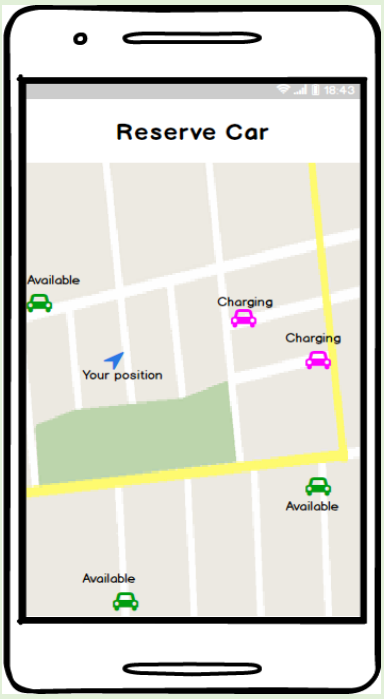
After the login, the user accesses to the main options' menu



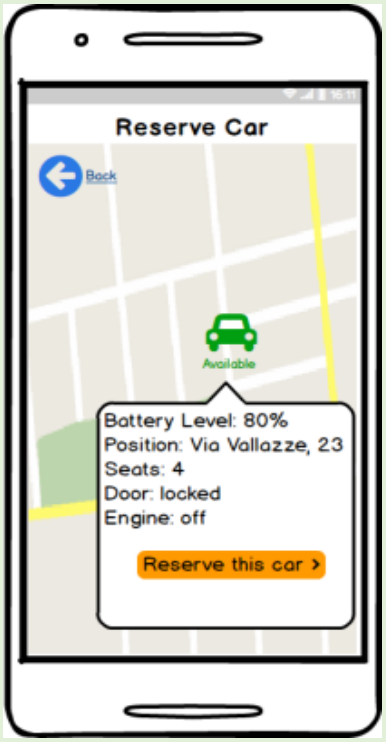
User chooses a starting position



The Reserve Car option shows the map with nearby available or charging cars



The user can see the car's state, by tapping on it and then chose to make a reservation



5: Requirements traceability

The components presented in this document were designed in order to fulfill the goals and requirements already presented in the RASD document. Here is a list of all the goals for this project and the components that were designed in order to fulfill them. All these Goals are aided by the Database component.

- [G1] Identify a user in a unique way.
  - User App on the client side and User Manager on the server side.
  - Payment Manager and Driving License Manager in order to verify the information given by the user.
- [G2] For every user easy and fast to find available cars in the proximity
  - The Car Manager and Car Search components, to begin the search for cars.
  - The Location Manager component, that accesses to the Database in order to retrieve the position of cars.
- [G3] Allow users to reserve a single car for up to one hour in advance to the rental
  - The Reservation Manager, to manage reservation operations.
  - The Car Manager, Car Search and Location Manager components, as said in [G2].
- [G4] Penalize users who reserve a car without taking it in the expected time.
  - The Reservation Manager, which has a timer that gives a penalty if the car isn't picked up within one hour from the reservation.
- [G5] Allow user to open, enter and use the car.
  - The Car App (GEB) component, which exchanges messages with the User App to open the car and with the Car Manager component to manage the car's statuses.
  - The Safe Area component along with the Location Manager component, which show constantly the position of the car and of Safe and Special Safe areas.
  - The Ride Manager component, which stores all the information about the usage of the car.
- [G6] Allow user to be constantly aware of the cost of his ride.
  - The Car App(GEB) component, which constantly calculates and shows on its interface on the car the cost of the ride.
- [G7] Make easy, for every user, to geolocate all the parking safe areas and all the available special safe areas.
  - The Safe Area component along with the Location Manager component.
- [G8] Incentivize the virtuous behavior of the users
  - The Car Manager component calculates penalties and discounts as already explained in the RASD document.
- [G9] Provide a way to use the cars in a continuative way
  - The Operator App component, which immediately asks for a recharging request if a car has too little battery charge or if it is too far from a Special Safe Area.
  - The Car Manager and Reservation Manager components, which, with the system of penalties/discounts and the one-hour timer allow the cars to not be all always not available.
- [G10] Maintain an equal distribution of the cars among the areas
  - This goal is mainly fulfilled by having a large number of cars, but the Safe Area component, along with the Location Manager component, help in this by constantly displaying the nearest Safe areas and, in case of special safe areas, the number of free parks

5.1: Traceability Matrix

Goal	Functional Requirements	Use Cases	Components
G1	FR1, FR2, FR3, FR4	UC1, UC2	User App, User Manager, Payment Manager, Driv.Lic.Manager
G2	FR6	UC3	Car Manager, Car Search, Location Manager
G3	FR7, FR8, FR9	UC3	Reservation Manager, Car Manager, Car Search, Loc. Manager
G4	FR10, FR12	UC4	Reservation Manager
G5	FR11, FR14, FR15, FR18	UC4, UC8	Car App(GEB), Car Manager, User App, Safe Area, Loc. Manager, Ride Manager
G6	FR17	UC4, UC8	Car App(GEB)
G7	FR13, FR19, FR20	UC8	Safe Area, Location Manager
G8	FR5, FR15, FR16, FR21, FR22, FR23, FR24, FR25	UC5, UC6, UC7, UC8	Car Manager
G9	FR15, FR16, FR24	UC8	Operator App, Car Manager, Reservation Manager
G10	FR13, FR25	UC5, UC8	Safe Area, Location Manager

6: Effort spent

6.1: Hours of work

6.1.1: Barletta Carmen

28/11 → 3h
29/11 → 2h
30/11 → 2h
1/12 → 2h
2/12 → 1h30m
3/12 → 3h
4/12 → 1h
5/12 → 4h30m
6/12 → 2h

6.1.2: Bagna Francesco Matteo

28/11 → 3h
29/11 → 2h
30/11 → 2h
2/12 → 1h30m
3/12 → 3h30m
4/12→ 4h
5/12 → 4h30m
6/12 → 2h

7: References

During the development of the DD document were used the following programs:

- Microsoft Office Word
- Balsamiq Mockups 3 (for the graphical interface prototypes)
- Draw.io (for diagrams)