POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
M. Sc. in Computer Science and Engineering
Dipartimento di Elettronica, Informatica e Bioingegneria

Power EnJoy

Software Engineering 2 - Project

DD

Design Document

Version 1.1

Authors:
Andrea BATTISTELLO Matr: 873795
William DI LUIGI Matr: 864165

Academic Year 2016 - 2017

1. Introduction

1.1. Purpose

The purpose of Design Document is to provide a representation of the architecture and design choices made for Power EnJoy system. This document also specifies the interfaces, components and systems to be developed and how they interact.

Architectural choices are the most important choices to make because they are the first to be made and because they have the largest impact on the choices that follow.

This document is written for project managers, developers, testers and for the Quality assurance team. It can be used as a structural overview to help maintenance and further development.

1.2. Scope

Duckburg is a very large city, with a lot of people. Over the years, the number of people with a driving license has increased. This caused a lot of traffic, boosted air pollution, and made it really hard to find a parking spot.

The purpose of Power EnJoy is to provide an alternative to this. We want to create a commuting service that is highly available and reliable. In fact, the reason why people drive instead of using public transport is usually related to the fact that they don't want to wait for a bus / train (low availability) and they don't want to risk not getting home if there's no public transport available (low reliability) because of a strike, vehicle malfunctions, or just because it's late in the night.

This system will allow users to rent a car for a period of time. PE will let anyone see the available cars (by showing in a map those cars that are closer to the user's location). Moreover, PE will let activated users actually book those cars, and will charge them according to the duration and usage (discounts will apply in specific cases). The system will also have an administrative interface, which will not be accessible by standard users.

1.3. Definitions, acronyms, abbreviations

1.3.1. Definitions

- **Payment information** refers to either a bank account or a credit card number.
- **Device** indicates either a mobile phone / tablet running PEM or a browser that is using PWA.
- **Safe area** refers to any legal parking spot in the city, where any driver can leave the car without interfering with the traffic.
- **PE Safe area** is a safe area that can be used exclusively by PE cars.

- **Web server** is a system that receives HTTP requests and provides web-based contents.
- **Application server** is a system that contains all the business logic. Exposes some interfaces that are used by the web server, mobile applications and cars. It uses containers to balance the requests load.
- **Component** is a piece of software that encapsulates some functionalities and can be reused
- **Container** is a piece of software that manages the components and offers an abstraction for parallelism, transactions and many others.
- **Entity** is a class that maps directly into the database. An example is given by JPA.

1.3.2. Acronyms

- **PE** Power EnJoy system
- **PEM** Power EnJoy Mobile application
- **PEW** Power EnJoy Web application
- **PEA** Power EnJoy Administrator application
- **PEC** Power EnJoy Car application
- **SMS** Short Message Service
- **GPS** Global Positioning System
- **RASD** Requirement analysis and specification document
- **DD** Design document
- **UI** User interface
- **UX** User experience design
- **MVC** Model view controller
- **JDBC** Java DataBase Connectivity
- **JPA** Java Persistence API
- **JEE** Java Enterprise Edition
- **EJB** Enterprise Java Bean

1.3.3. Abbreviations

- **[Gn]** n-th goal
- **[Dn]** n-th domain assumption
- **[Rn.m]** m-th requirement related to goal [Gn]

1.4. Reference documents

- [1] IEEE Software Engineering Standards Committee, “29148-2011 - Systems and software engineering — Life cycle processes — Requirements engineering”, 2011.
- [2] Payment Card Industry (PCI) Data Security Standard, v3.2, PCI Security Standards Council, LLC.
- [3] Response Times: The 3 Important Limits, JAKOB NIELSEN, 1993.
- [4] Software Engineering 2 course slides.
- [5] The assignment of *Power EnJoy*
- [6] RASD (Requirement specification and Specification document) of *PowerEnjoy*

1.5. Document structure

The document is composed of five sections and an appendix.

- The **introduction** section, this one, is intended to find the goal of a design document, clarifies the definitions and acronyms used throughout the document and gives a general idea of the main functionalities of the system to be developed.
- The second section is called **architectural design** and will focus on the architectural choices made for this system, the description of the identified components and how they interact either by specifying the interface and by giving a dynamic view of the interaction with the use of sequence diagrams.
  This section is the core of the document and will extensively use UML diagrams to formalize and define as much as possible the chosen architecture and the reasons that led to that decisions.
- The third section is called **algorithm design** and is entirely focused on the description of the most important algorithms identified and used in the system.
- The fourth section is called **user interface design** and will focus on how the informations will be displayed to the user and how the user can navigate between the different views (this is done by means of an UX diagram)
- The fifth section is the link between the RASD and the DD, is called **requirement traceability** and its purpose is to make explicit the mapping between the requirements specified in the RASD and the components identified in the DD.
- The **appendix** contains the list of software used to make this document, along with the time spent on this project for each of us and the revision history.

2. Architectural design

2.1. Overview

This chapter describes the architectural choices that are being considered through the development of this system. The system will be described starting from the high level components and dependencies with other existing systems. Then, a more detailed description of each component, their interfaces and how they interact is provided. Finally, the end of this chapter will focus on the most important patterns used according to the chosen architectural style.

2.2. High level components and their interaction

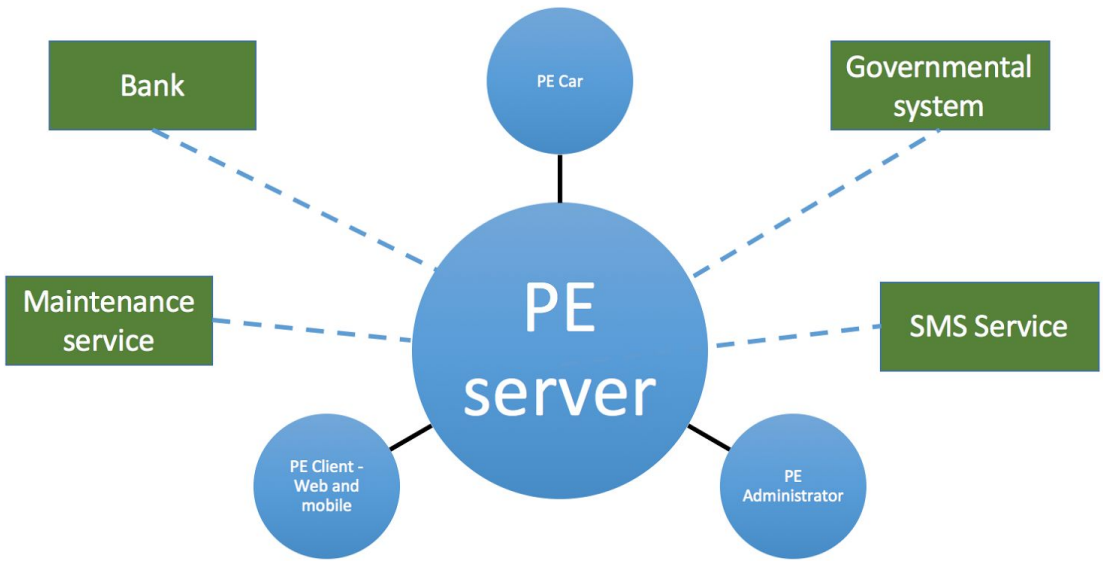
The main components that make up the PE system are:

- Database management system
- Web server
- Application server
- Client and administrator applications
- Car application

In addition to these, PE server interacts with the following external systems:

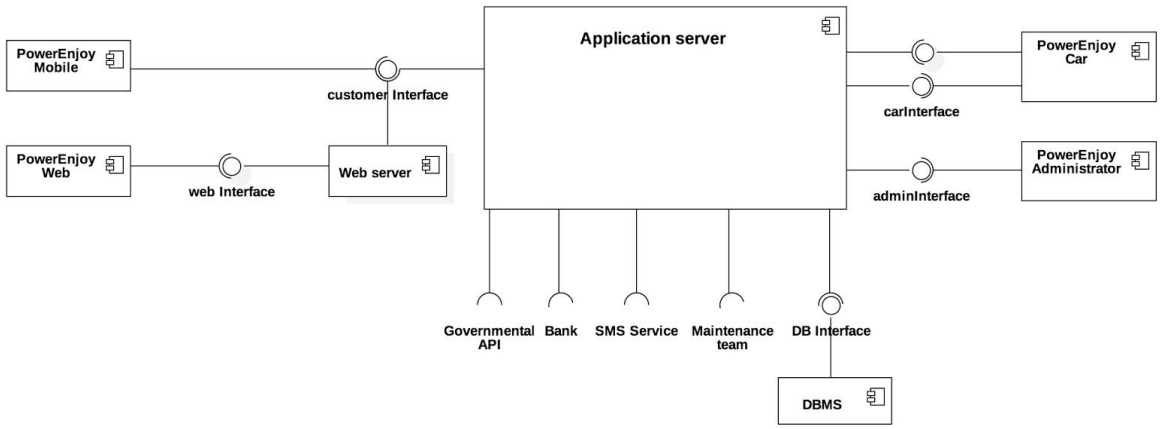
- Bank
- Governmental system
- Maintenance service
- SMS service

All these components will be further detailed in this document.



2.3. Component view

The following component view formalizes the architecture of PE system and the interactions among its components. The following sections will describe the functionality and dependencies of all the identified components.



2.3.1. PE Mobile

This is the component that will be responsible for the communication between handheld devices (e.g. the user's smartphone) and the rest of the system. To do this, PE Mobile will exchange REST messages with the *customerInterface* and will render the user interface based on the replies that PE server will provide.

Depending on the specific device (Android, iOS, Windows Mobile) the component will use specific frameworks and libraries, but the logic will be the same for each platform. As an example, Android application will use the builtin Android SDK platform, iOS will use the Foundation Framework for the core and UIKit for the user interface and Windows Mobile application will use the .NET framework.

Mobile application uses the *customerInterface* to communicate with PE server and will depend also on external libraries like GoogleMaps API to visualize the map.
The MVC pattern will be applied to structure the mobile application.

2.3.2. PE Web

This component will be responsible for the interaction between users who are only equipped with a browser, as opposed to a fully fledged Android or iOS or Windows Mobile device. For this reason, PE web needs to take into account that some devices will be equipped with old hardware and software, usually with poor performance. So this component will be as light and responsive as possible.

The *webInterface* allows all components from the network (such as PE web) to send HTTP requests, and responds to those requests by sending back HTML pages and static resources like CSS and JavaScript files.

This application will depend on the browser Geolocalization API.

2.3.3. PE Car

The PE car component will be very similar to PE mobile. Only a single (Android based) platform will be supported, because (as we already specified) the cars will be equipped with an Android-enabled board computer. This component will exchange REST messages with the *carInterface* (as opposed to the *customerInterface* used by PE mobile) which allows to perform operations such as sending the current position, battery status, and so on.

PE car will render the user interface based on the replies that will be provided by the *carInterface*. PE Car will also implement some business logic, such as communication recovery after a dropdown caused by external world. This business logic will grant more reliability and security concerns.

2.3.4. PE Administrator

This component is an internal application, not reachable from the Internet for security reasons. This application will be used by paid service administrators (employees) who will physically be located in the PE headquarters. PE Administrator is a web application, thus the call center operators will use a browser. This decision was taken to get rid of several issues concerning ad-hoc applications, such as: dependency from the OS, need to update the application for each computer and longer developing time.

In order to access data, the PE administrator component will programmatically access the *adminInterface*. This means that the app will employ software bindings which allows it to connect to the DBMS and to use Object-Relational Modeling when querying and updating records. The interface will not be a RESTful one, because we don’t plan on implementing different multiple applications and user interfaces (employees will be trained to use the existing user interface), but also to simplify development by making use of frameworks for building admin interfaces (which e.g. already implement all of CRUD operations) as much as possible.

2.3.5. Web server

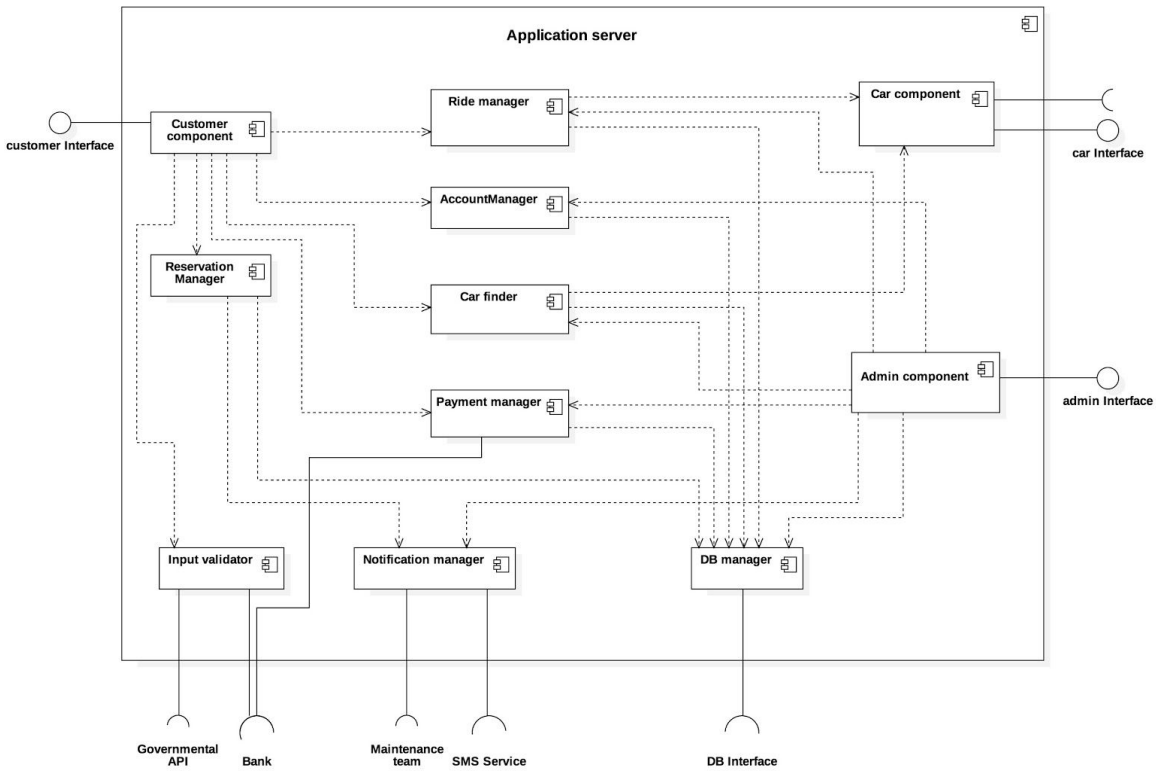
As mentioned already, the *webInterface* is meant to provide a way to obtain HTML responses when executing HTTP queries.

We could decide to do client-side rendering of the PE web application, by emulating what PE mobile does. For example: PE web could employ a JavaScript web application (written in AngularJS, ReactJS, or EmberJS) which exchanges REST messages with *customerInterface*, and then renders everything using JavaScript logic. However, since we are assuming that PE web can have poor performance, this solution is not feasible.

For this reason, the *webInterface* will basically be a way to avoid having PE web spend CPU time on rendering the page. Furthermore, this will guarantee that even the oldest browsers, which are not supported by recent JavaScript frameworks, will be supported by PE web.

2.3.6. Application server

Application server is the most important component in the Power EnJoy architecture. It contains all the business logic and performs all the computations, decisions and interactions needed for the system to work properly. It receives the input from different sources (user, administrators and cars) and it must be able to distribute the computation to different subsystems as much as possible.
For this reason we decoupled the application server in several subsystems, as depicted in the following figure.



Every sub component is obtained by identifying the functional units in the system, thus this decomposition is independent from the technology used to implement the application server. A possible implementation, though, can be done by using an Enterprise Java Bean (EJB) for every subcomponent. Such beans can be stateless and uses cookies to handle the communication with the user.
With such a configuration, depending on the server load, JEE can use instance pooling to distribute the computation across multiple containers, thus increasing the throughput. If the beans are also replicated in different machines, this method automatically improves the reliability of the overall system.

The following sections will describe the main functionalities of each sub component.

Customer component

Customer component oversees all the communication to and from the users. All the incoming requests are redirected to and managed by other components. As a transactional system, this component takes care of possible breakouts of other components and handles all the possible errors that may occur in runtime, thus increasing the reliability of the overall system.

Reservation manager

Reservation manager's main function is to handle all the requests related to a reservation. More precisely, it manages new reservations made by the user, the possibility to cancel a previous reservation and also takes care of the expiration of the user's reservation.

An important aspect of this component are concurrent requests. In fact, it would be possible that many requests on the same car can arrive at the same time, but only one of them should be accepted (one car can be reserved only by at most one user). This problem becomes even more relevant if this component's functionality is spread across multiple devices (or containers) to increase the throughput.

A solution to this problem is to use a distributed lock protocol, similar to the ones widely used in distributed database systems. In this way a distributed lock for a given car is requested and a reservation is accepted only if - after the lock is granted - the car is available.

Ride manager

Ride manager handles all the lifetime of a user's ride, namely from the *start driving* to the *stop driving* events. Between these two events, the user can unlock the car as many times as he want and the car will periodically send its position in order to keep track of the user's movements.

Account manager

Account manager is responsible of what concerns the user account, namely the registration of a new account, the updates of the user's informations, the confirmation of the phone number, the activation/deactivation of the user, the login and logout activities.

During the registration phase, this component is responsible to check the validity of all the submitted informations. This part will then be delegated to *Input Validator* component.

It is also its duty to periodically check the expiration date and the validity of the ID card and driving license provided by the user during the registration. Leaving out this check could lead to problems with the law, because a user could potentially drive without an up-to-date driving license.

Car finder

Car finder's main function is to efficiently answer to queries concerning the position of the cars (e.g. get all cars in a specified area). Since this query will be very frequent, Car finder

should use a fast algorithm and a proper data structure to ensure fast response time. The data structure and the related algorithm used by this component are well explained in the algorithm design section (section 3) of this document.

Input validator

The main functionality of this component is to offer an easy way to validate data inserted by the user. Input validator relies on external systems such as:

- bank: checks the validity of a specified payment method
- governmental system: checks the status of the driving license and the ID card of the user

Notification manager

Notification manager handles the communication to and from the user by means of email and SMS messages. SMS services are sent using an external system that offers this functionality.

This component also exposes a function to register a component as a delegate. This way, the Notification manager component knows what to do when it receives a message.
An example of the delegation method is the following: when a new registration is completed and the phone number verified, *Customer component* registers itself to be the delegate for any text message received by that user's phone number. Thus, when *Notification manager* receives a text message, checks the phone number and calls the related delegate, in this case a *Customer component* instance will be called.

Payment manager

This component handles all the operations related to money, that is payments and reimbursements. For this purpose, it interacts with the bank's systems.

Since the payment will be triggered monthly, this component must keep track of the pending bills that should be charged to the user. Then, when the user decides to pay (or the trigger is fired after a month), a single checkout is sent to the bank containing all the pending bills not already paid by the user.

Car component

Car component is the only component that directly interacts with PE Cars. Its main functionality is thus to handle the interaction between the system and the cars.
It is also responsible to keep track of the position of every car and raise a notification in case a problem occurs or a car is running out of battery and an intervention by the maintenance team is required.

Admin component

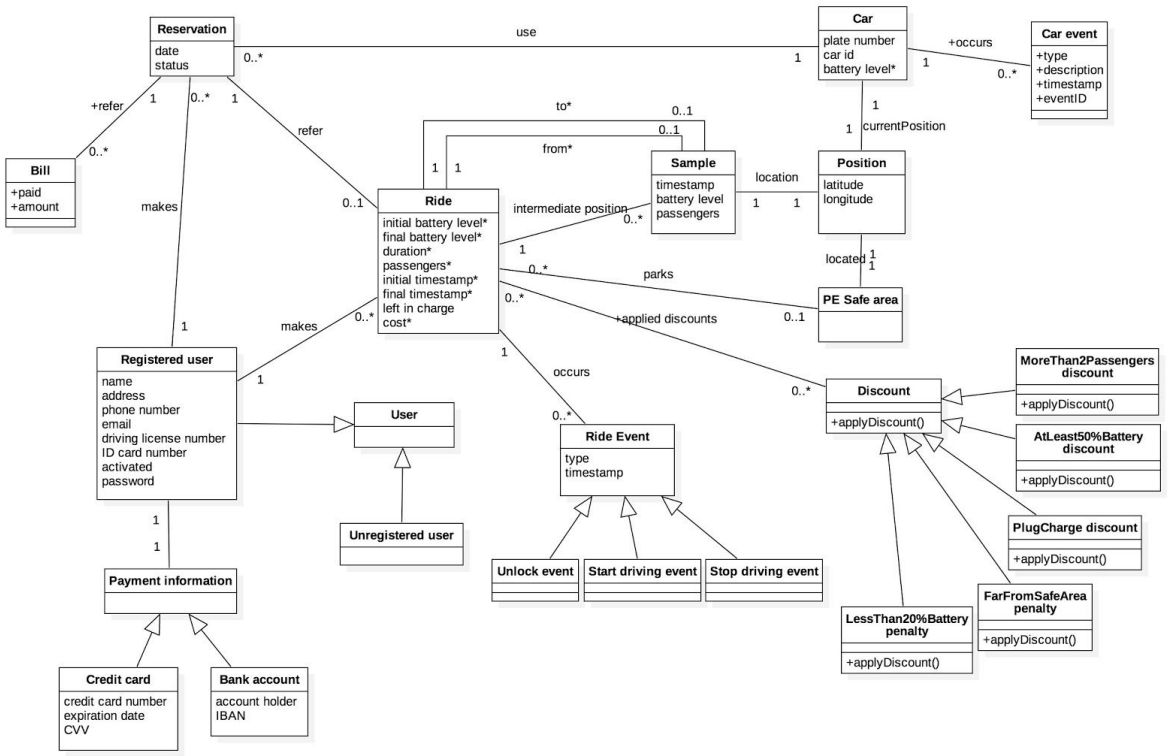
Admin component interacts with the admin application, and as well as customer component its job is to oversee admin requests and forward the requests to the other components. The admin component, differently from the customer component, is not accessible from the outside, so it requires fewer checks to guarantee a certain level of security.

DB manager

DB Manager offers a series of methods to access the database and performs updates, inserts, deletions and lookups. Nearly all the components depends on DB Manager, but this dependency will usually be transparent to the developer through the use of libraries like JPA, that offers an abstraction from the data.

2.3.7. DBMS

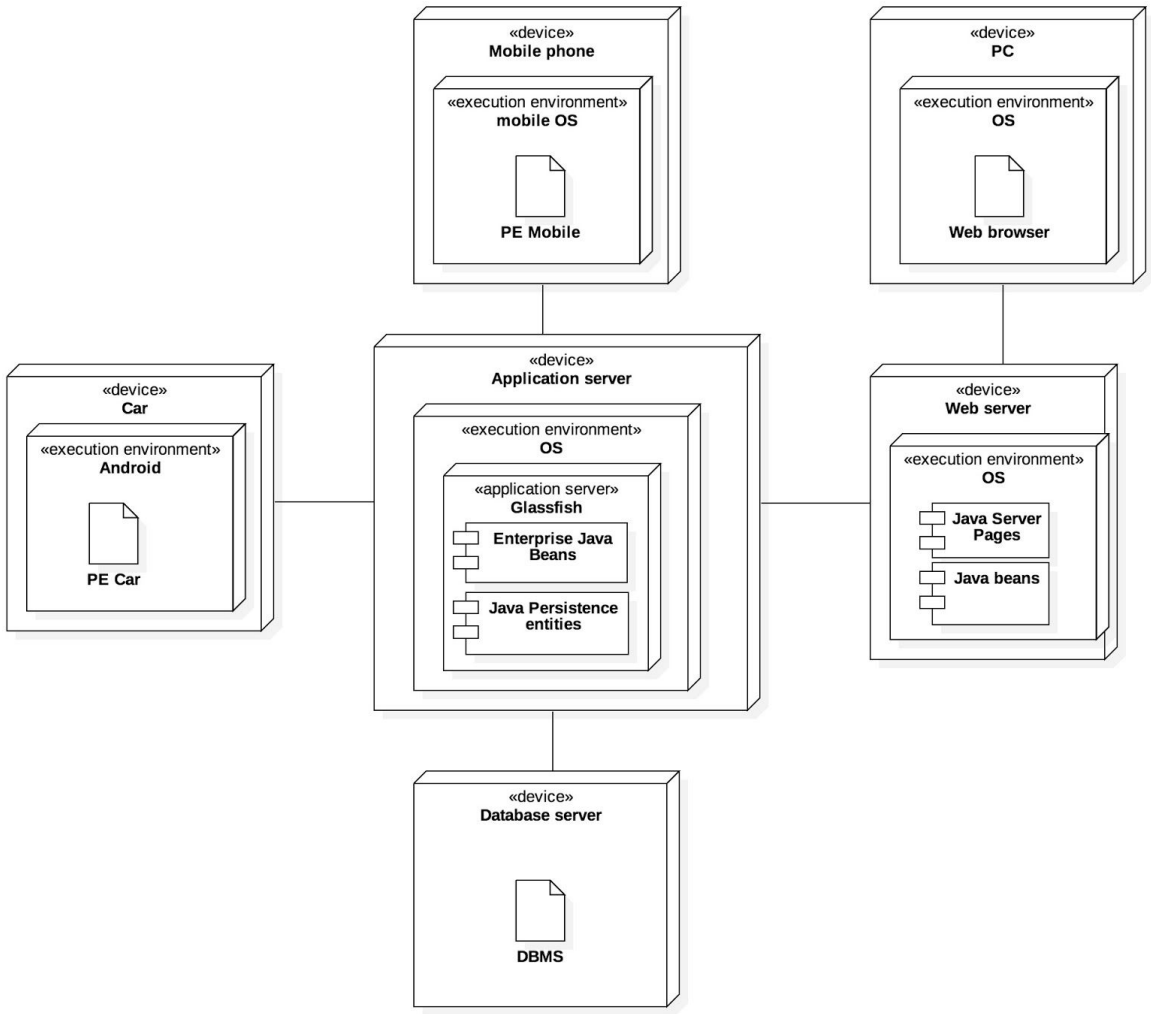
The database management system is the component that takes care of persistent storage of data. Since this application is not data-intensive we opted for a relational database like MySQL or PostgreSQL.
A possible structure for the database is the following:



Every class in the diagram will be mapped in a JPA Entity or similar, so that the management of the database will be mostly transparent to the developer.

2.4. Deployment view

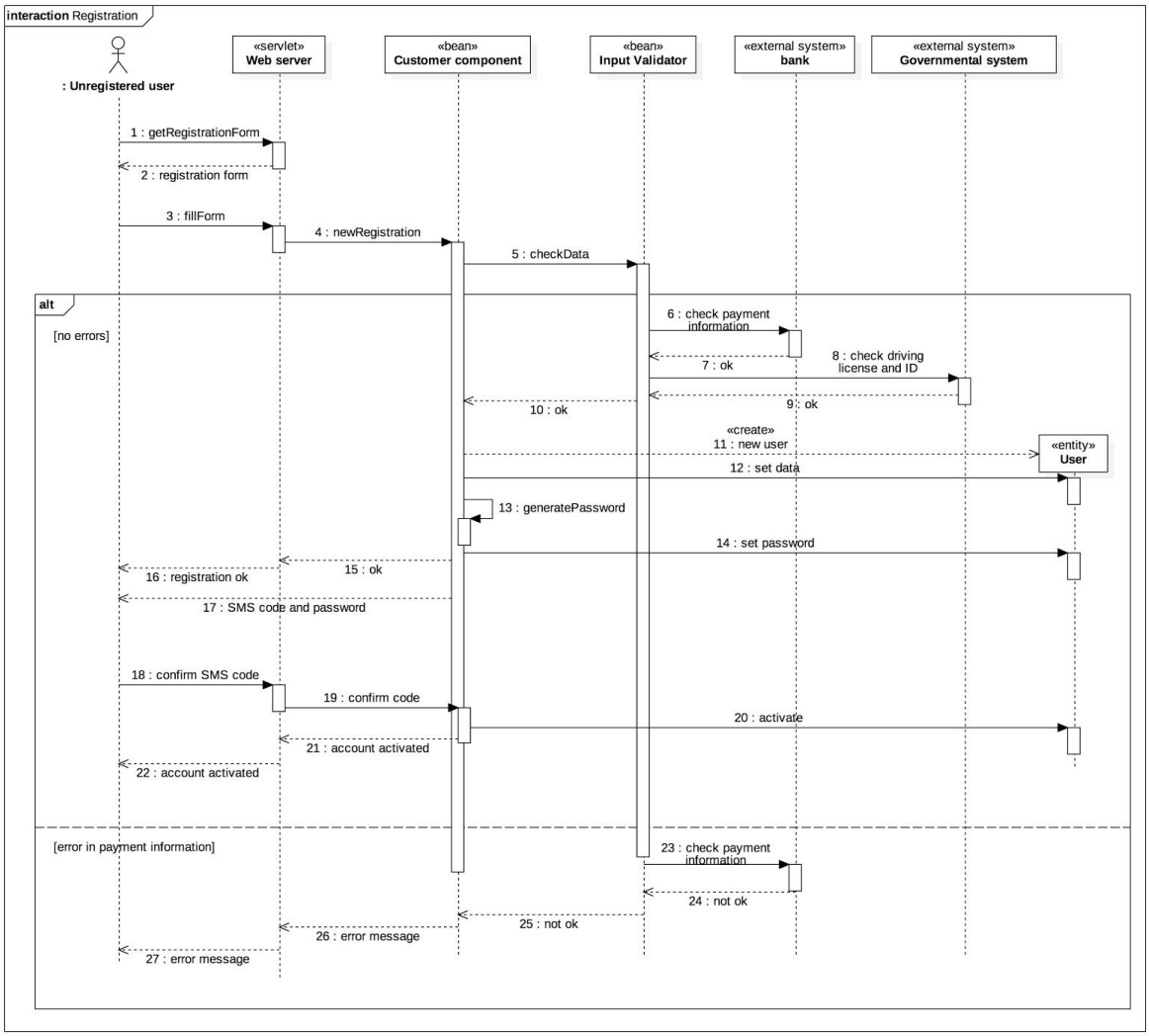
The deployment view is shown in the following figure. As previously stated, the system will use four tiers, allocating the web server and the application server in different devices.



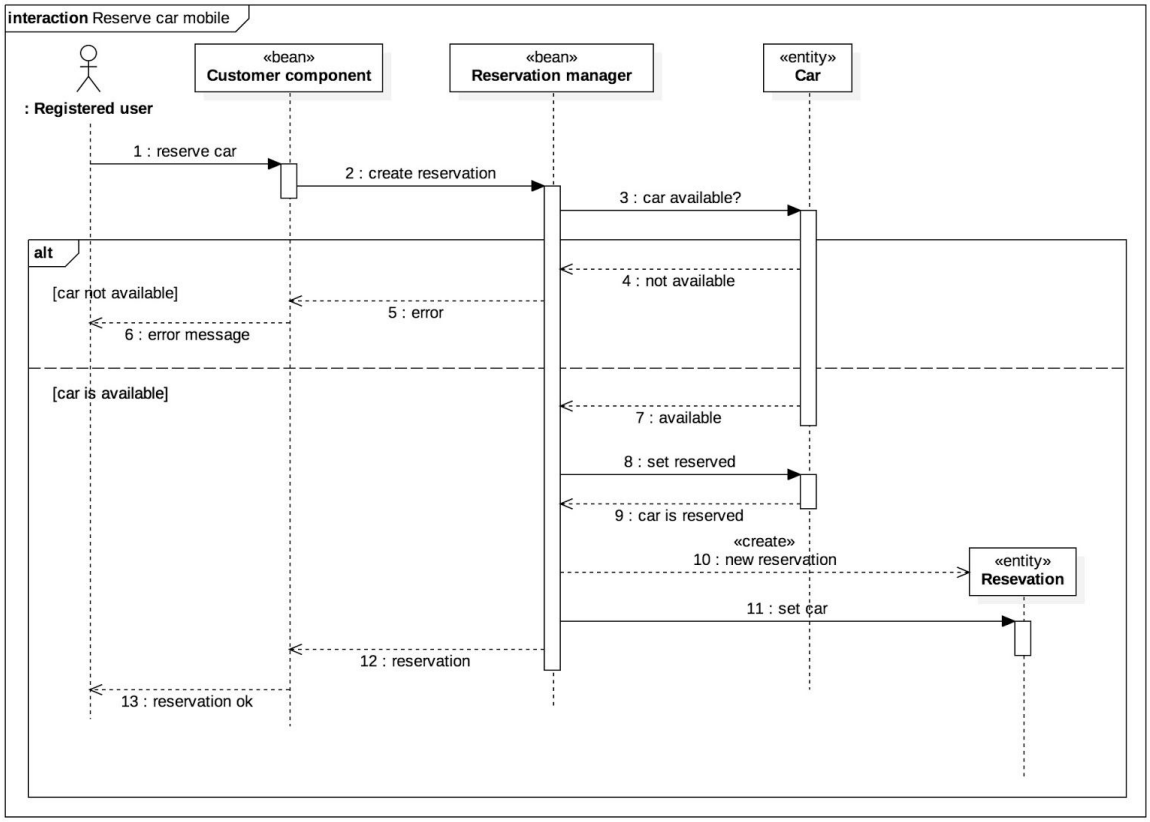
2.5. Runtime view

To better understand how these components interact, we propose several sequence diagrams that should give a more comprehensive idea of how the system dynamically evolves. Some of the following interactions will not go into the very specific details of every components, either because it is a widely known procedure and it is not critical or because we wanted to focus only to certain aspects of the interaction that we found more important.

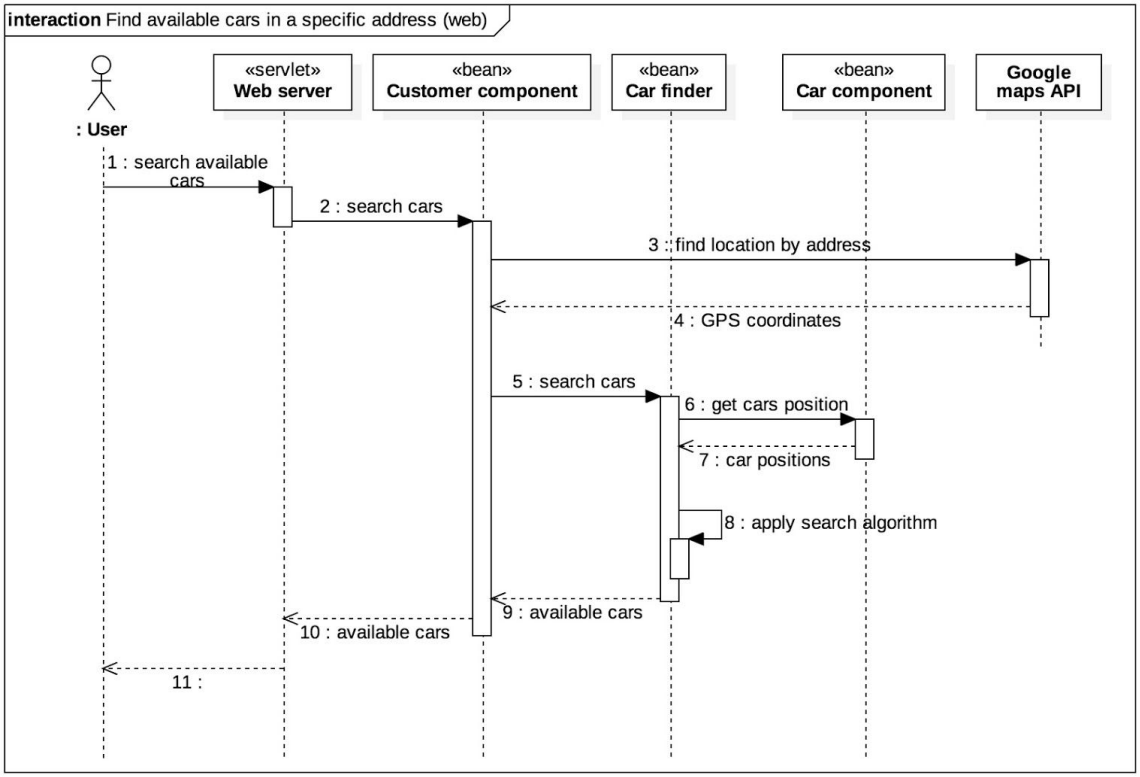
2.5.1. Registration



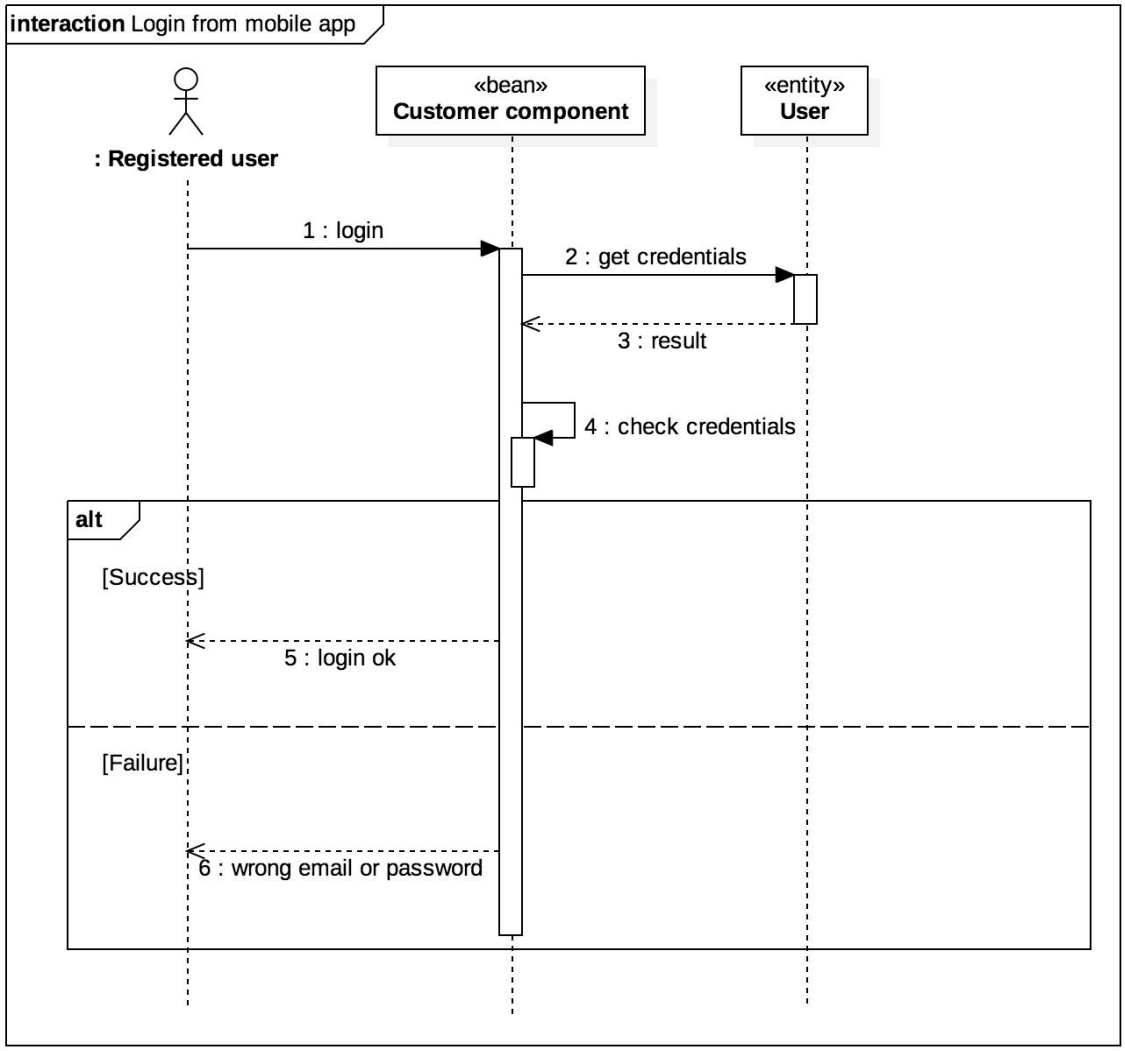
2.5.2. Reserve car mobile



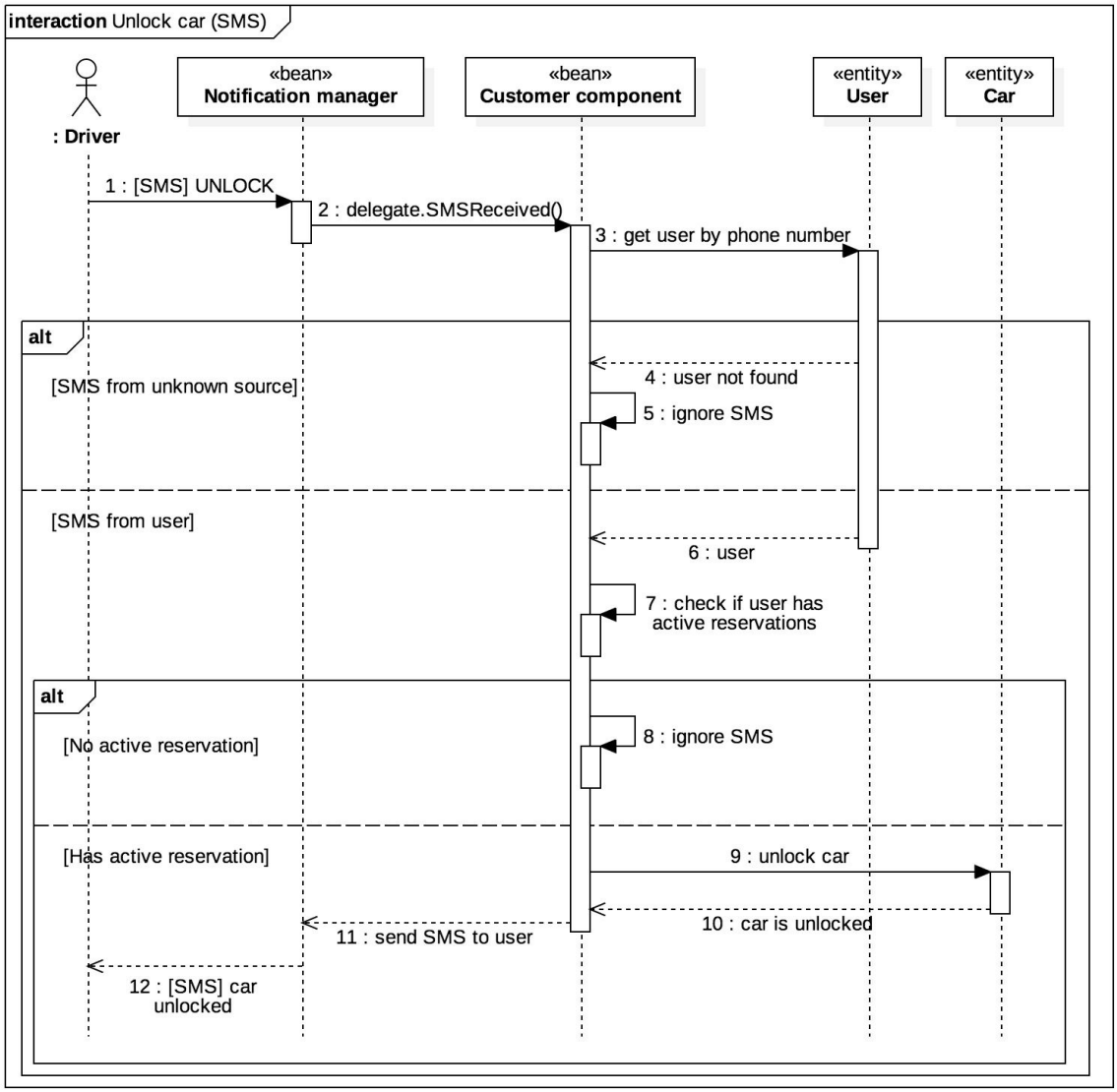
2.5.3. Find available cars in a specific address (web)



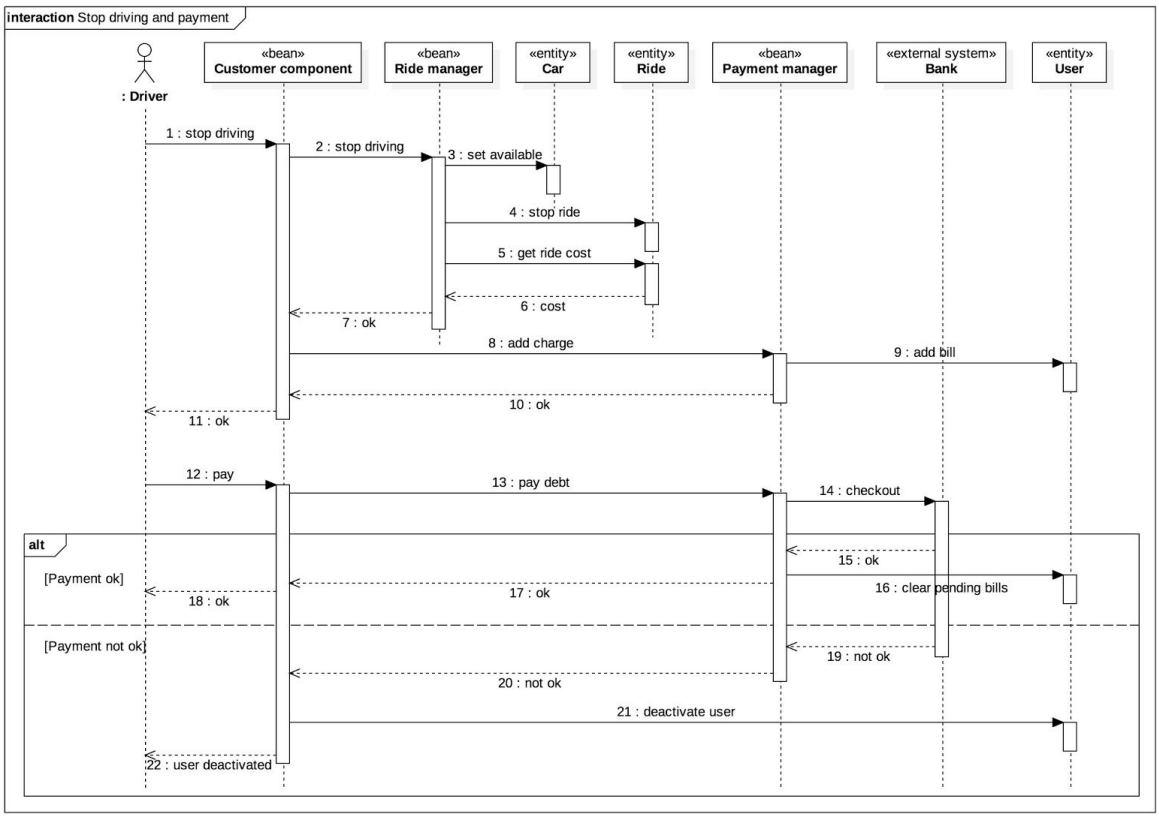
2.5.4. Login from mobile app



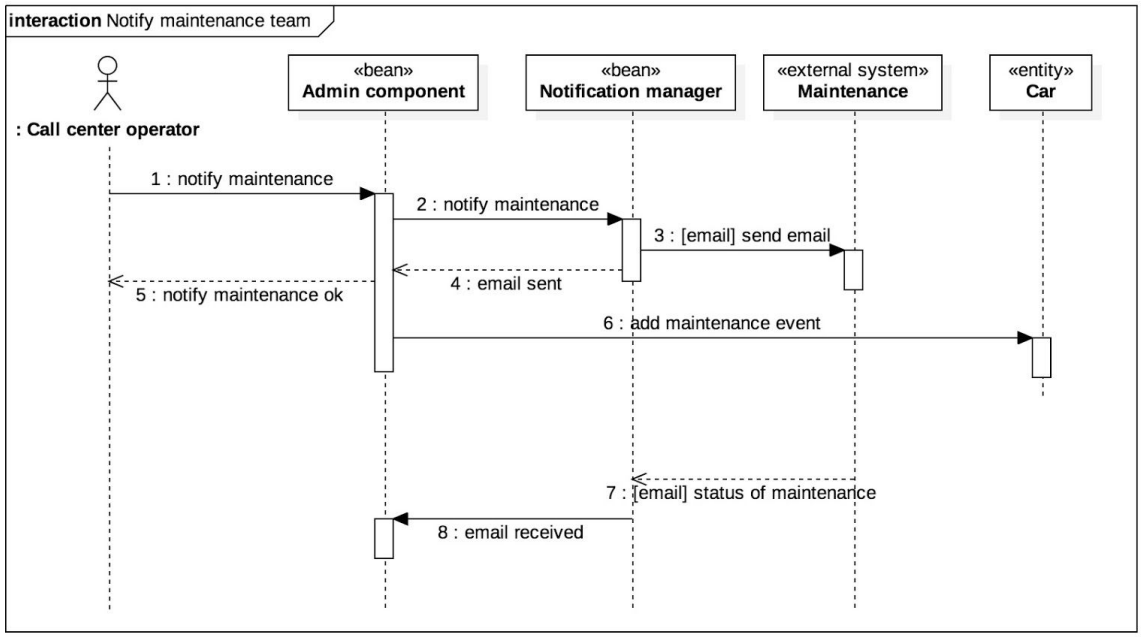
2.5.5. Unlock car (SMS)



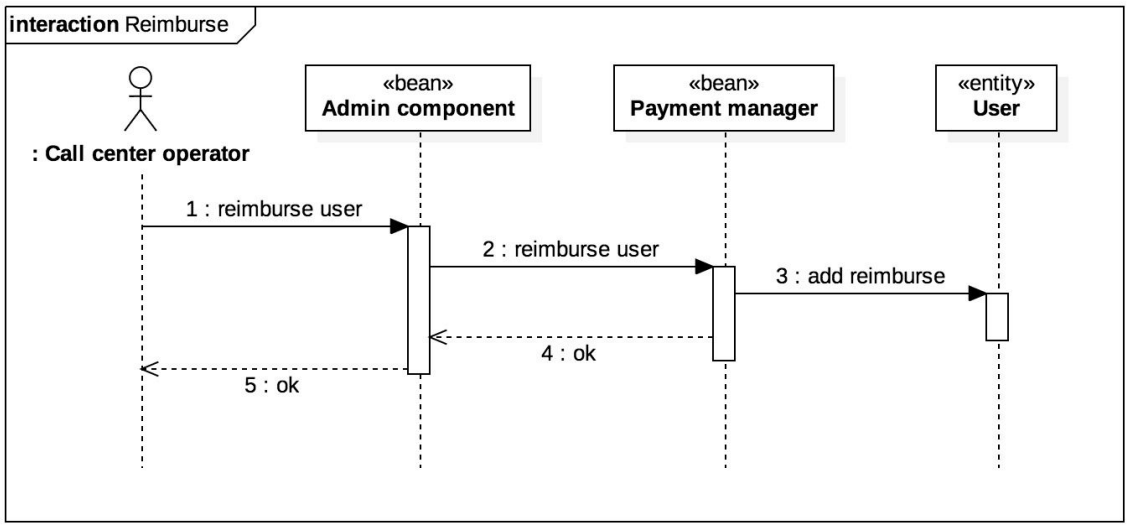
2.5.6. Stop driving and payment



2.5.7. Notify maintenance team



2.5.8. Reimburse



2.6. Component interfaces

This section gives a brief description of all the interfaces that Power EnJoy communicates with.

2.6.1. customerInterface

This interface is responsible of the communication with the users, either they are using the mobile app or the web app.

Functionalities	Method	Parameters	Responsible component

Creates new profile	`register()`	`name : String`
		`driving license : String`
		`ID number : String`
		`phone number : String`
		`email : String`
			`Account manager`

Confirms phone number	`confirmCode()`	`userID : Integer` `SMScode : String`	`Account manager`
Validates the credentials of the user	`login()`	`phone number : String` `password : String`	`Account manager`
Expires the user session	`logout()`	`userID : Integer`	`Account manager`

Edit profile informations	`editAccount()`	`userID : Integer`
		`new name : String`
		`new driving license : String`
		`new ID number : String`
		`new email : String`
			`Account manager`

Stop driving	`stopDriving()`	`rideID : Integer`	`Ride manager`
Start driving	`startDriving()`	`reservationID : Integer`	`Ride manager`
Returns informations about a ride	`getRideInfo()`	`rideID : Integer`	`Ride manager`

Search for cars in specified position	`searchCars()`	`latitude : Float`
		`longitude : Float`
		`range : Float`
			`Car finder`

Search for cars in specified address	`searchCars()`	`address : String` `range : Float`	`Car finder`
Reserves a car	`reserve()`	`userID : Integer` `carID : Integer`	`Reservation manager`
Cancels a reservation	`cancelReservation()`	`userID : Integer` `reservationID : Integer`	`Reservation manager`
Returns informations about a past reservation	`getReservationInfo()`	`reservationID : Integer`	`Reservation manager`
Pay all hanging bills	`pay()`	`userID : Integer`	`Payment manager`
Unlocks the car	`unlockCar()`	`rideID : Integer`	`Car component`

2.6.2. webInterface

This interface offers the exact same functionalities of `customerInterface`. The differences are due to the protocol used for the communication and that the `webInterface` encapsulates also informations for the presentation layer (HTML pages).

2.6.3. carInterface

This interface handles the communication between the Car and the application server. This communication is bidirectional but it is not symmetric, thus the interfaces will be presented separately.

Communication from Car to PE

Functionalities	Method	Parameters	Responsible component

Updates the position of the car	`updateCarPosition()`	`carID : Integer`
		`latitude : Float`
		`longitude : Float`
			`Car component`

Notifies the server that the battery is running low	`notifyBatteryLow()`	`carID : Integer` `batteryLevel : Float`	`Car component`

Communication from PE to Car

Functionalities	Method	Parameters	Responsible component
Get car coordinates	`getPosition()`	–	`Car`
Changes the car status	`setStatus()`	`status : Enum(Reserved, Available, NotAvailable)`	`Car`
Unlocks the car	`unlock()`	–	`Car`

2.6.4. adminInterface

Admin interface is used by Call center operators via PMA to manage and take care of the users and the cars on road. Offers some additional functionalities compared to `customerInterface`.

Functionalities	Method	Parameters	Responsible component
Login	`login()`	`username : String` `password : String`	`Account manager`
Call driver	`callDriver()`	`userID : Integer`	`Notification manager`

Notify maintenance team	`notifyMaintenance()`	`adminID : Integer`
		`description : String`
		`carPosition : Position`
			`Notification manager`

Call emergency or police	`callEmergency()`	–	`Notification manager`

Search for cars in specified position	`searchCars()`	`latitude : Float`
		`longitude : Float`
		`range : Float`
			`Car finder`

Search for cars in specified address	`searchCars()`	`address : String` `range : Float`	`Car finder`
View all car status	`carStatus()`	–	`Car finder`

Reimburse user	`reimburse()`	`adminID : Integer`
		`userID : Integer`
		`description : String`
		`amount : Float`
			`Payment manager`

Charge user	`chargeUser()`	`adminID : Integer`
		`userID : Integer`
		`description : String`
		`amount : Float`
			`Payment manager`

View all user details	`userDetails()`	`userID : Integer`	`Account manager`

Deactivate user	`deactivateUser()`	`adminID : Integer`
		`userID : Integer`
		`description : String`
			`Account manager`

View all current rides	`currentRides()`	–	`Ride manager`

2.6.5. External interfaces

Power EnJoy is also built upon external interfaces, that are extensively used to offer the needed service to the users and to administrators. For each dependency, the most important methods used by PE will be listed.

SMSService

SMSService offers the ability to send and receive text messages with these methods:

`sendSMS(number : String, text : String)`	sends SMS to users.
`SMSReceived()`	polls received SMS.

Governmental API

The `Governmental API` offers the following interface in order to check the validity of the documents and be on the side of the law:

`checkIDCard(IDCard : String)`	checks whether an ID card is valid or not.
`checkDrivingLicense(drivingLicense : String)`	checks if the driving license is valid and get the expiration date.

Bank

`Bank` is considered as a service and offers the following interface:

`checkPaymentInformation(method : PaymentInformation)`	checks the validity of the payment information.

`withdraw(method : PaymentInformation, amount : Float)`	withdraw the required amount from the specified payment method.

Maintenance team

The maintenance team is informed by means of email and phone calls, so there are no methods to call yet.

DB Interface

The `DBInterface` is used transparently by the JPA, that maps an entity (e.g. Ride, User, Car) with a database table, thus handling all the required insertions, updates and deletions.

GoogleMaps API

GoogleMaps API offers the following methods:

`getMap(area : GeoRectangle)`	returns the map of the specified area. A GeoRectangle is a rectangle whose vertices are coordinates identified by latitude and longitude
`getCoordinateFromAddress(address : String)`	returns the latitude and longitude of the specified address

2.7. Selected architectural styles and patterns

2.7.1. Architectural pattern: MVC

MVC (Model View Controller) is an architectural pattern that is widely used in software development that requires user interfaces. The main actors that are defined in this pattern are:

- *Model:* the part of a system that handles the application logic. It usually interacts with a DBMS to store persistent data. PE uses JPA and Entities to represent the model of the entire application.
- *View*: the component that handles how the data is displayed to the final user. Every model can have different views, each of them focuses on a specific perspective or is intended to be used for different kinds of user types. In this application there are several views: mobile applications and car application use OS-dependant user interfaces while the web application and the admin application use HTML and CSS to display data.
- *Controller*: components that handles user interactions and the communication between view and model. Typically the controller reads the user input from the view and updates the model accordingly.

The main advantage of using this approach is the *separation of concerns,* that usually reduce the *coupling* among these elements while keeping high *cohesion* within them.

Therefore, it would be possible to modify a component without affecting the others, thus increasing *maintainability*. In fact, user interfaces are more prone to changes while business logic usually has longer life. In this way it is also possible to parallelize the development process.

2.7.2. Architectural style: Client-server

Client-server is the most common architectural style for distributed applications. This architectural style defines two roles:

- Client: requires a service and asks to the server.
- Server: receives the client requests and provides the service.

Typically, with this architectural style, the client starts the conversation and the server answers after the required computations. This is a centralized system, because every message will be processed by the server and it is the only component that takes decisions. If the application server goes down, there is no way for a user to use the service anymore, thus the server should be reliable and available. Reliability and availability will be granted by a replication of the server.

Power EnJoy system will use client-server architectural style for almost all the functionalities offered because:

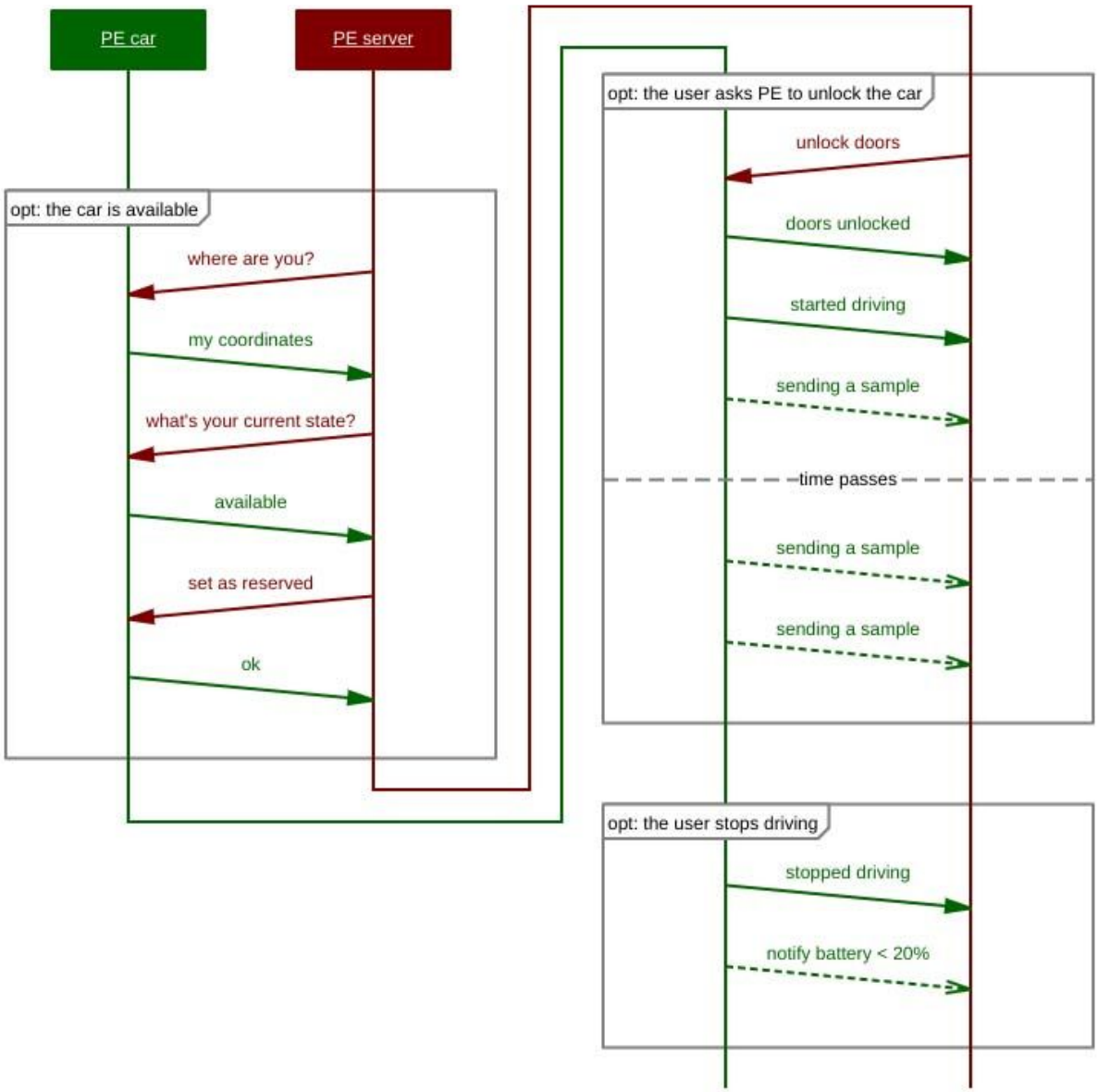
- It is a distributed application since there will be a number of cars, all cooperating to provide the service, which will interact with the system.
- Among all distributed architectures taken into account, this is the simplest one that does its job. Simplicity is a good criterion because it usually lead to quicker development and fewer errors.
- There is a lot of thoroughly tested software and many hardware components that can be used to develop the system according to this architectural style.

2.7.3. Car - server protocol

It is noteworthy to specify how the interaction between PE cars and the PE system takes place. The communication is bidirectional, thus either the car or PE can start a conversation and provide services. This architectural choice was made because there could be situations when the car server have to contact the car directly and vice versa. Therefore each of the car and the application server takes both the role of client and server.

This two-way communication also improves the reliability because the information about the position, for example, is sent as soon as possible, reducing the window of uncertainty, instead of waiting the next poll from the server.

The following picture shows an example of the communication between the car and the application server:



2.7.4. Three layered - four tiered application

The client-server model can be subdivided in many logical layers. Each logical layer performs a specific operation. The most common logical layers are: presentation layer, business layer and data layer. A logical layer can then be mapped to one or more physical systems (called *tier*). The number of logical layers and physical tiers may vary depending on the service offered.

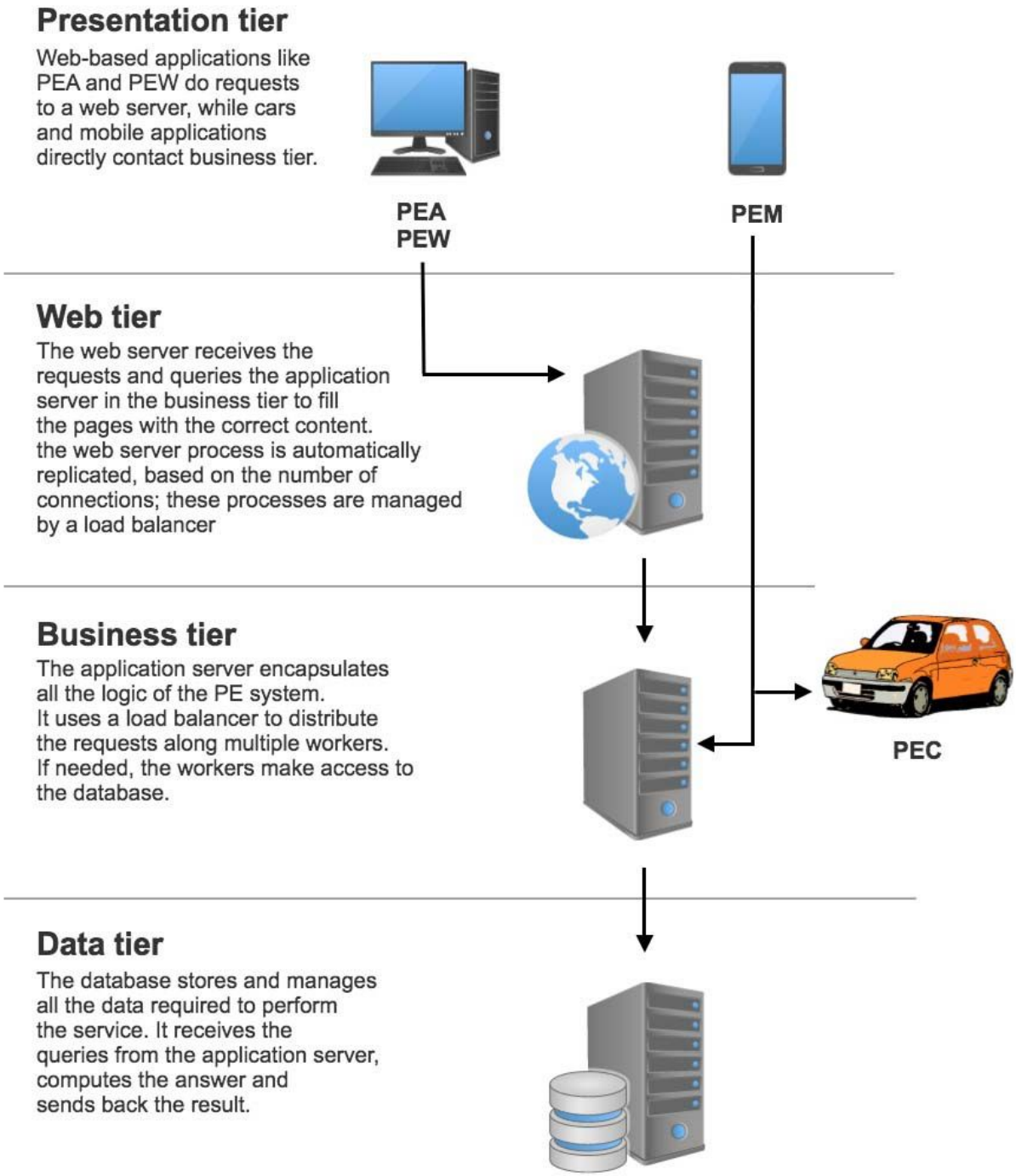
Power EnJoy system will use a four-tiered architecture, divided as follow:

- **Presentation tier**: this level takes care of displaying the informations. Presentation layer is distributed upon multiple devices, since there are web and mobile applications, each with their own interface.

- **Web tier**: this level takes care of encapsulating the business logic in a way that can be displayed on a browser. More precisely, the web server receives HTTP requests from a browser, forwards the request to the business tier and returns a HTML web page with the required informations.
- **Business tier**: this is where the business logic is located. The application server takes all user inputs, validates and processes the answer and provide the needed service.
- **Data tier**: this tier takes care of persistently storing the data. This tier is devoted to a DBMS.

The presentation layer is split across presentation and web tiers, while business layer and data layer are bound to business tier and data tier respectively.

We decided to split the presentation layer in four tiers instead of the most common three tiers because we wanted to further mark the boundary between the logic and the presentation layer.



It is worth to mention that PEC is depicted in the business tier and not in the presentation tier. This is because PEC implements both the presentation layer (how it displays the informations on the car touchscreen display) and a part of the business layer, because the car is not completely a thin client but is programmed to have some interaction with the server.

3. Algorithm design

3.1. Efficient car lookup

The PE server will frequently need to access the location of one (or more) of the PE cars, available or otherwise, that are stored in the system's primary or secondary storage.

The location of a car is represented by two real numbers, one representing latitude and the other one representing longitude. To ease interoperability with external libraries, the unit we will use to store this data will be degrees.

The most straightforward way to implement the lookup operation of cars given a position (acquired via GPS, for example) is to just perform a `SELECT` query on the entire set of car records, then sorting them according to a function that returns the distance between the car and the given position, and then filtering only the nearest cars. This solution has mainly two disadvantages:

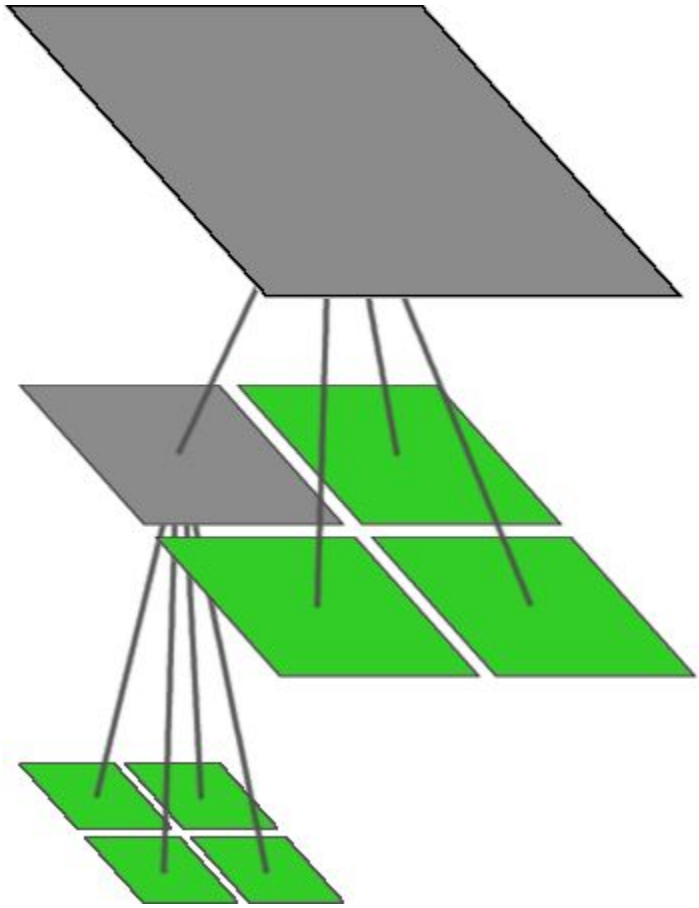
- It requires us to perform a complete lookup on the car records and an expensive sorting operation.
- It requires us to constantly keep the car records up to date. This means that we would need to actually perform a write operation in secondary storage each time we receive a new sample that represents a car movement.

To improve upon this solution, we will adopt the *proxy* design pattern and introduce a layer between the DBMS and the application server which will serve as both a *cache* and an *index* for our set of car records, and will thus greatly optimize common operations such as:

- The lookup of one or more cars: since this layer serves as *index*, the lookup will be very efficient.
- The update of a car's position: since this layer serves as a cache, the update will actually happen in main memory and not in secondary memory (at least not immediately).

This layer will be entirely in main memory, and will retain a copy of the location and the numeric ID of each car (available and otherwise) present in the system.

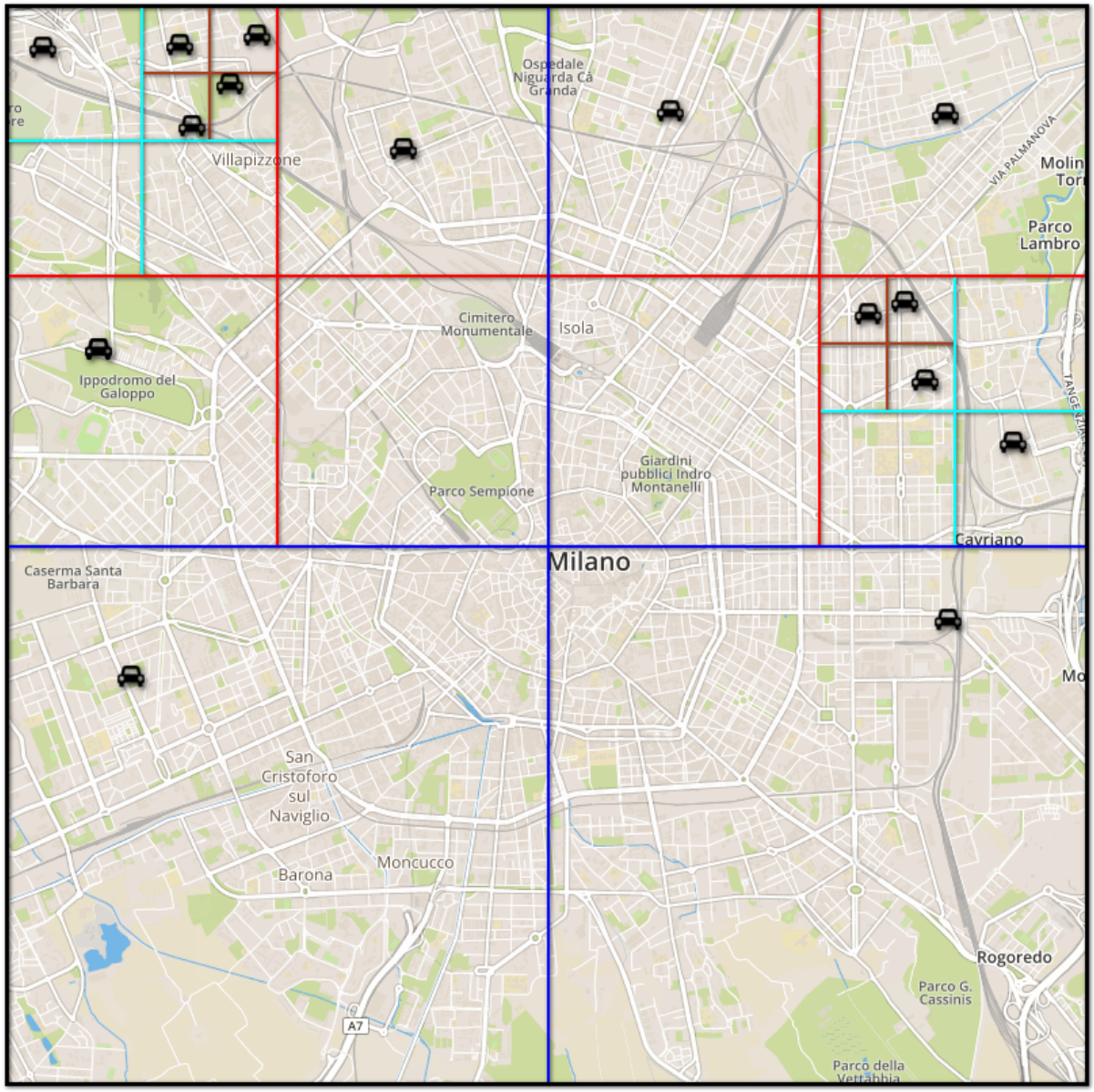
This data will be maintained in a *Quadtree*-like data structure, where each node in the tree will represent a specific (square-shaped) subarea of the city. A node will either contain a single car, or it will further divide its subarea in four equal spaces:



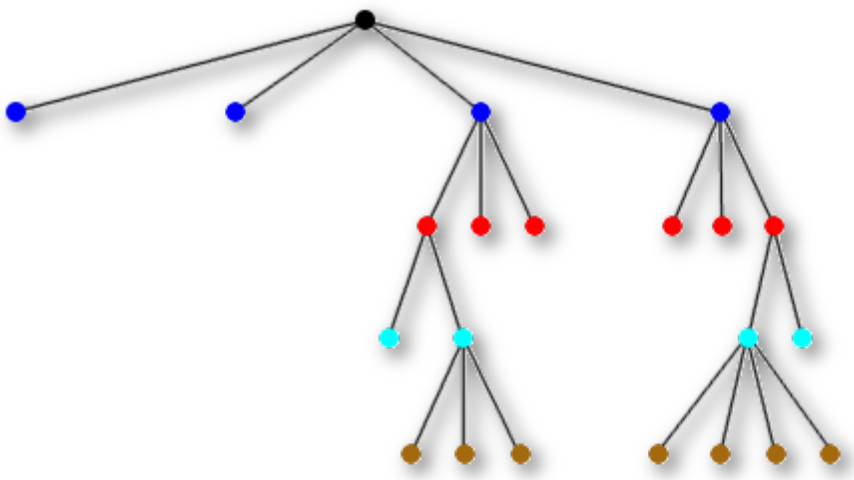
In the very first level, we will store an area wide enough to span *at least* the entire city. It does not matter if the city is not square-shaped: it will only mean that some parts of the data structure will never be populated: this will not significantly affect performance.

If there's only one car, the first level will only contain a reference to that car (ID, latitude, longitude, available). However, in the much more likely scenario where there are more than a single car in the entire city, the first level will contain 4 memory pointers that will point to different levels, each of them representing 25% of the original area.

The following is a visual representation of what our data structure may look like when 15 cars are present in the system:



Which, in RAM, corresponds to the following tree:



3.1.1. Construction

Since the data structure is not persistent, it will need to be reconstructed at each startup. In order to build it, we will use the following algorithm:

```
procedure QuadtreeBuild
   Quadtree = {empty}
     For C = 1 to N           ... loop over all cars
       QuadInsert(C, root)    ... insert car C in quadtree
     end for
```

3.1.2. Insertion of a new car

To insert a new car in the data structure, we will use the following algorithm:

```
procedure QuadInsert(C, X)
   ... Try to insert car C at node X in quadtree
   ... By construction, each leaf will contain either
   ... 1 or 0 cars
   if the subtree rooted at X contains more than 1 car
      determine in which child Y of node X the car C lies in
      ... Y is either the top left, top right, bottom left,
      ... or the bottom right "quadrant"
      QuadInsert(C, Y)
   else if the subtree rooted at X contains exactly 1 car
      ... X is a leaf
      create four children for node X in the Quadtree
      ... X is not a leaf anymore
      move the car already in X into the child in which it lies
      let Y be child in which car C lies
      QuadInsert(C, Y)
   else if the subtree rooted at X is empty
      ... X is a leaf
      store car C in node X
   endif
```

The complexity of the QuadInsert algorithm is proportional to the distance between the root and the leaf in which the car resides (that is: the execution time will be higher if the insertion happens in a very crowded area).

We can safely assume that the cars will be distributed in a mostly uniform way, with some clusters in the busiest areas (or when there are important events somewhere). However, even in the event of having all the cars clustered together, the execution time will still be at most logarithmic.

For example, in order to get 1m x 1m tiles out of the entire surface of the Earth and Moon combined, our quad tree would have $log_4(5.5 \times 10^{14}) \simeq 24.4832 \simeq 25$ levels.

Even assuming an accuracy higher than 1 squared meter, in order to tell the position of two distinct cars apart, we are using a logarithmic amount of bits (e.g. a 64 bit float number). So, having more levels than the amount of bits would mean that some cars are closer than the machine precision (which is $2^{-53} \approx 1.11e\text{-}16$). With 64 bits we have about 16 decimal places of tolerance. To compare, let's note that with just 8 decimal places we have a 1.1 millimeters tolerance. With 13 decimal places we have a tolerance of 1 angstrom, around half the thickness of a small atom.

We will thus never have more levels than the number of bits of our float data type, because we are dealing with cars (who are not small) and we assume that the GPS systems installed in those cars will always work correctly and consistently.

3.1.3. Retrieval of a car

We will store direct pointers as a hash map (which maps integer IDs to memory addresses), in order to optimize the lookup of a specific car in the quad tree. This is useful when we need to quickly retrieve the node in which a car is located, in order to make updates.

3.1.4. Deletion of a car

To delete a car we will use the following algorithm:

```
procedure QuadRemove(C)
   ... Retrieve the node in which C is located
   X = QuadRetrieve(C)
   ... X is a leaf
   delete car C from node X
   while X is not the root
      X = parent of X
      if X contains 0 cars
         delete the four empty children of X
         ... X is now a leaf
      else
         break
   endwhile
```

3.1.5. Movement of a car

This operation will happen very often (whenever a car sends a new sample). We will basically move the car in the tree without actually telling the DBMS about this update.

For our specific application we are dealing with items that move in a very predictable way: a car will generally move a few meters at a time, as opposed to other items, like particles. We

will exploit this useful property to greatly improve the average running time of the `QuadMove` operation:

```
procedure QuadMove(C, C')
   X = QuadRetrieve(C)

   if C' is still inside the area delimited by node X
      just update the location from C to C' inside node X
   else
      QuadRemove(C)
      QuadInsert(C')
   endif
```

In the (very likely) case in which the car just moves a few meters and is still inside the "designated square" of the quad tree, we will only do 1 retrieve operation (constant time) plus 1 update operation (constant time). The tree will only need to be reshaped if the car goes outside its leaf's designated square and, even when that happens, the running time will be given by 1 deletion (logarithmic time) plus 1 insertion (logarithmic time).

3.1.6. Retrieval of all cars inside a "query area"

Another very important operation is: querying all the cars in a given area. This is useful whenever a user is looking for a car, and wants to see which cars are available near their location. The following procedure returns a list of cars, and its execution time is linear in the number of cars returned.

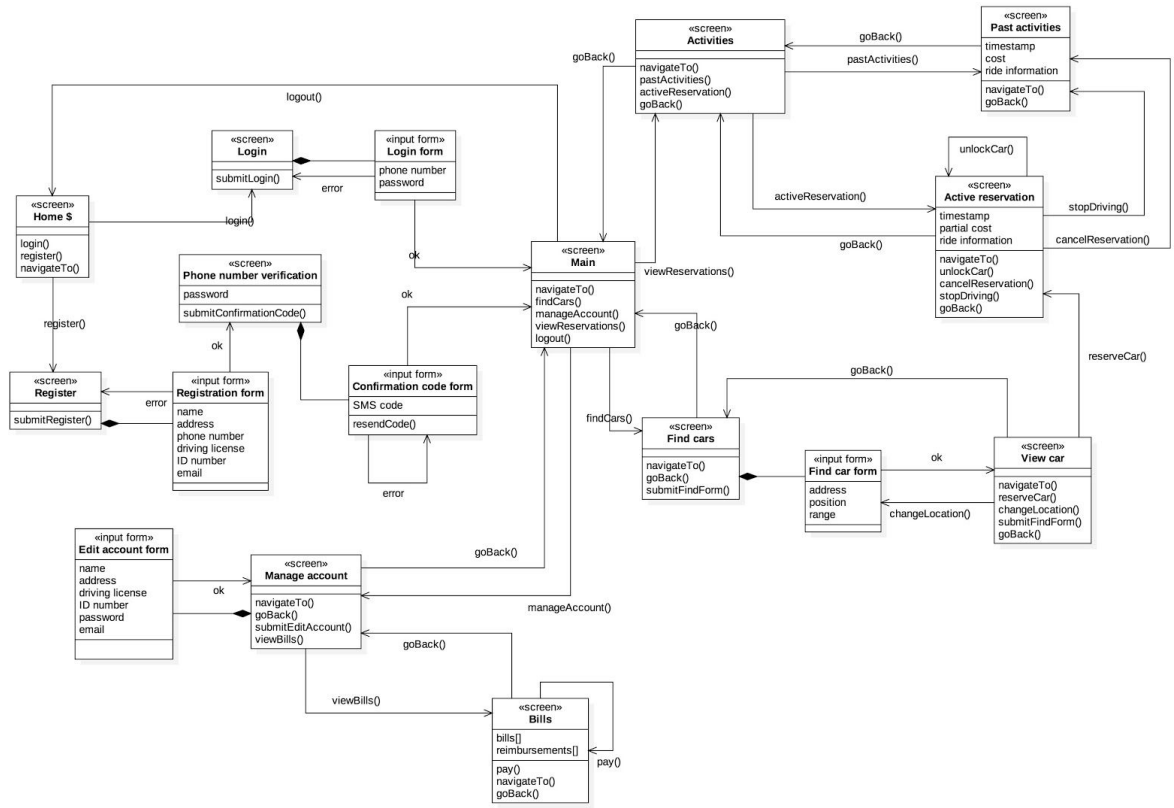
```
procedure QuadList(S, X)
   ... S is the "query square", that is: the interesting area
   ... X is the root node, initially is set to root

   answer = []            ... empty list
   Y = S ∩ area(X)  ... intersection between S and the area of X

   if Y ≠ ∅          ... non-empty intersection
      if X is a leaf
         answer += [all cars that are inside Y]     ... 0 or 1 car
      else
         answer += QuadList(S, top left of X)
         answer += QuadList(S, top right of X)
         answer += QuadList(S, bottom left of X)
         answer += QuadList(S, bottom right of X)
      endif
   endif

   return answer
```

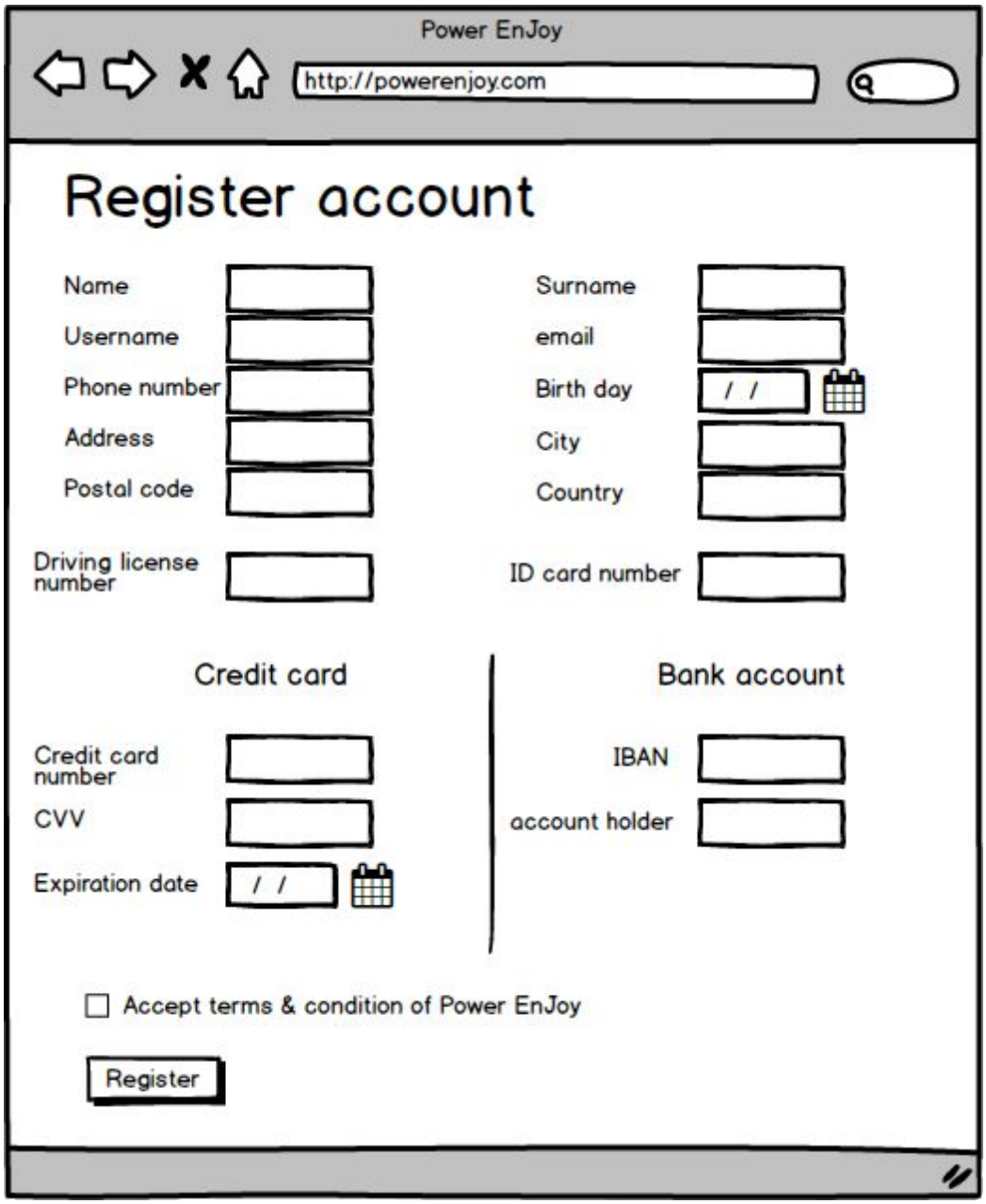
4. User interface design

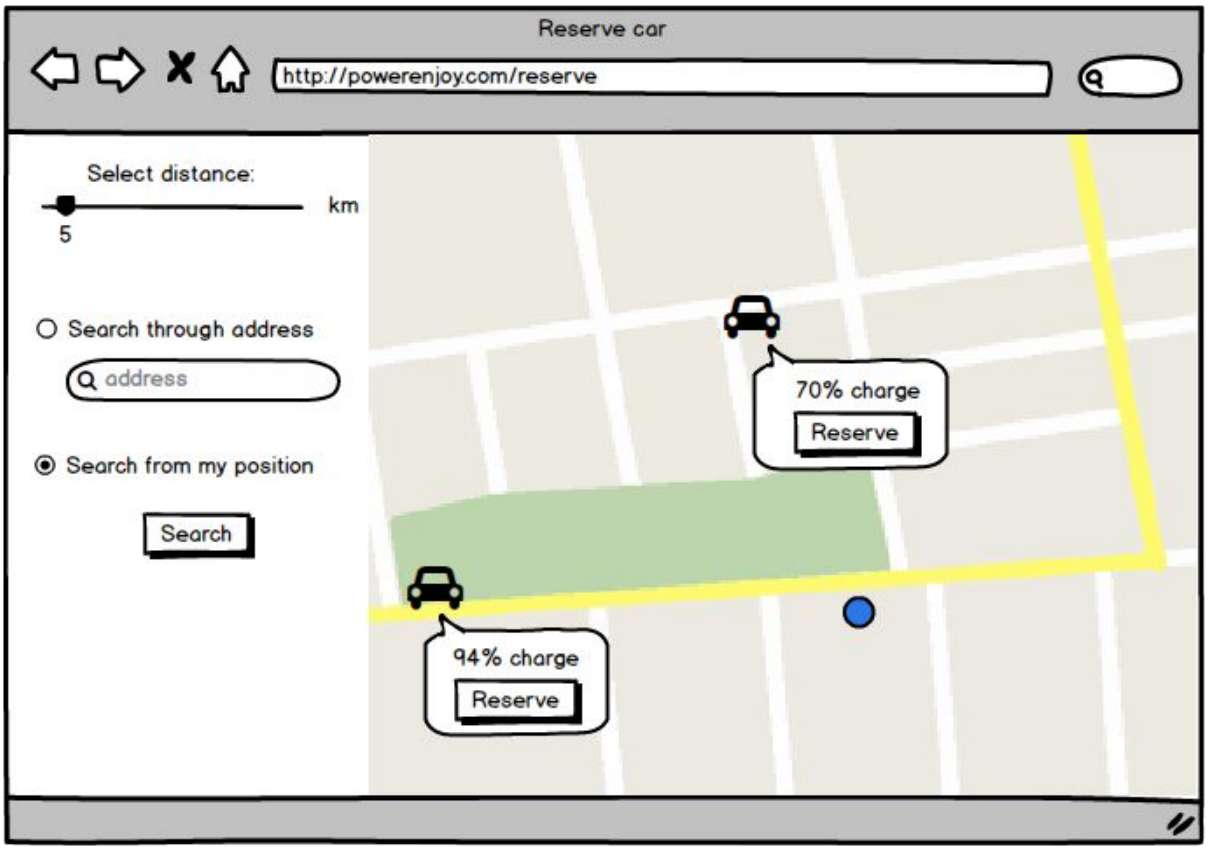
User interface design is an important aspect of the development of a software system that should interact with many users. A good user interface should be concise, responsive and good-looking. In the following sections we will provide a UX diagram that illustrates the navigation through the applications and some mockups of the UI we have designed.

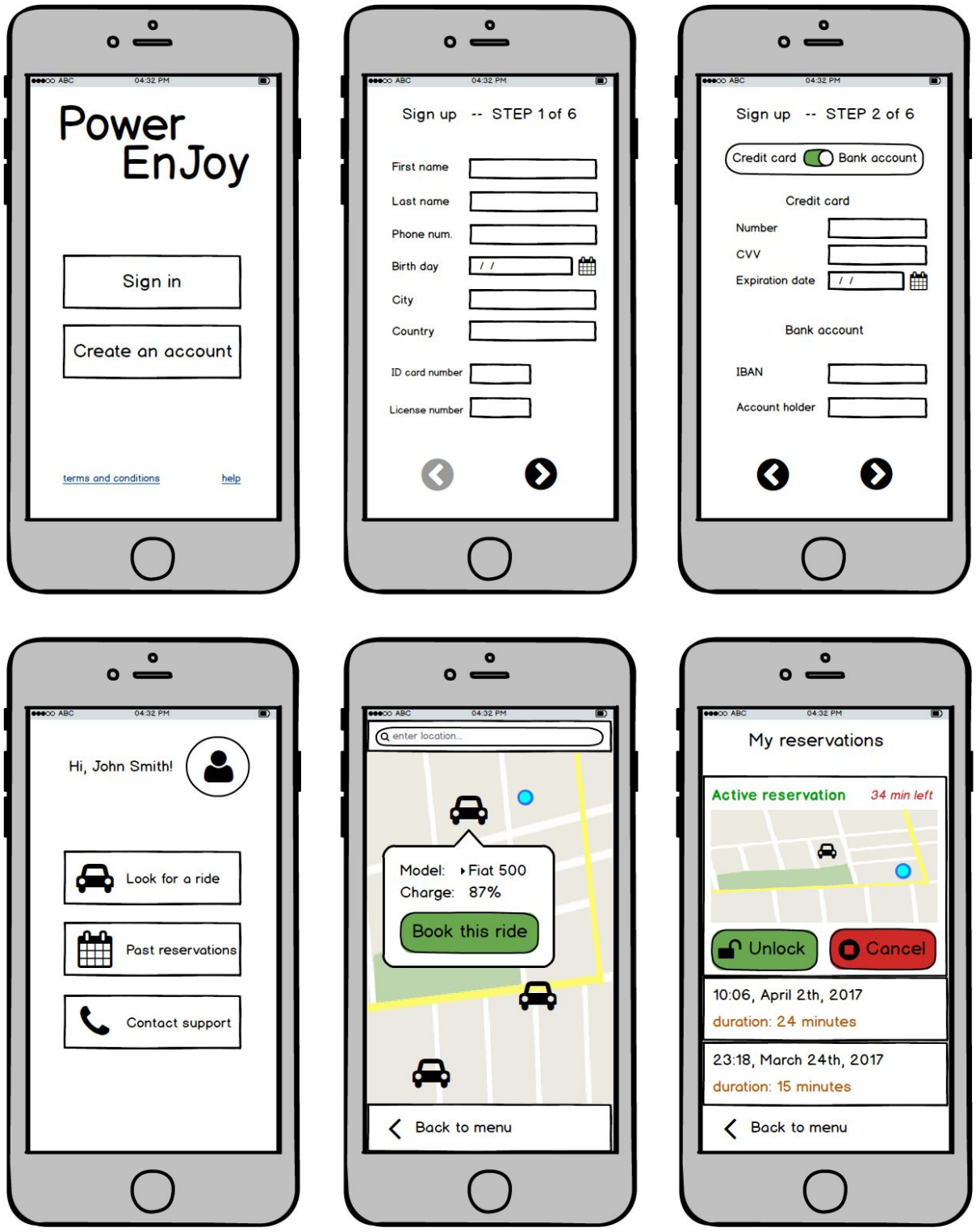
4.1. PE Mobile and PE Web

These applications shares the same flow of events, but graphically they will be very different. In fact PEW will use HTML and CSS to arrange the UI, while mobile applications will use native frameworks (depending on the platform).



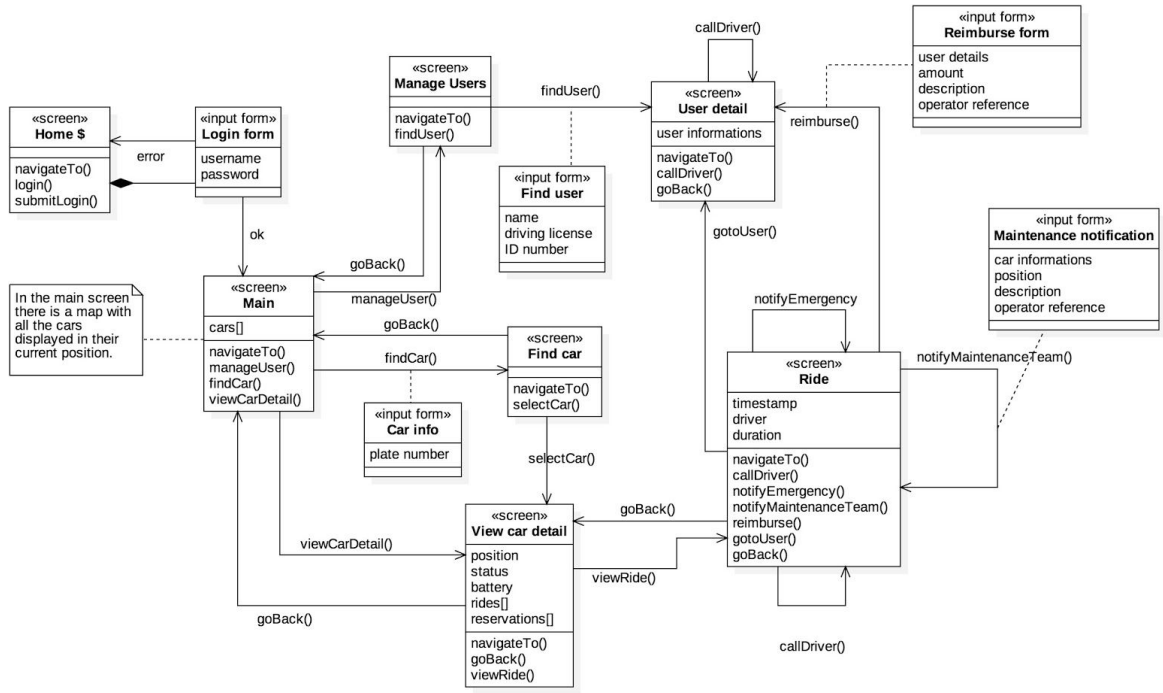

Power EnJoy
http://powerenjoy.com
Register account
Name
Surname
Username
email
Phone number
Birth day
/ /
Address
City
Postal code
Country
Driving license number
ID card number
Credit card
Bank account
Credit card number
IBAN
CVV
account holder
Expiration date
/ /
Accept terms & condition of Power EnJoy
Register

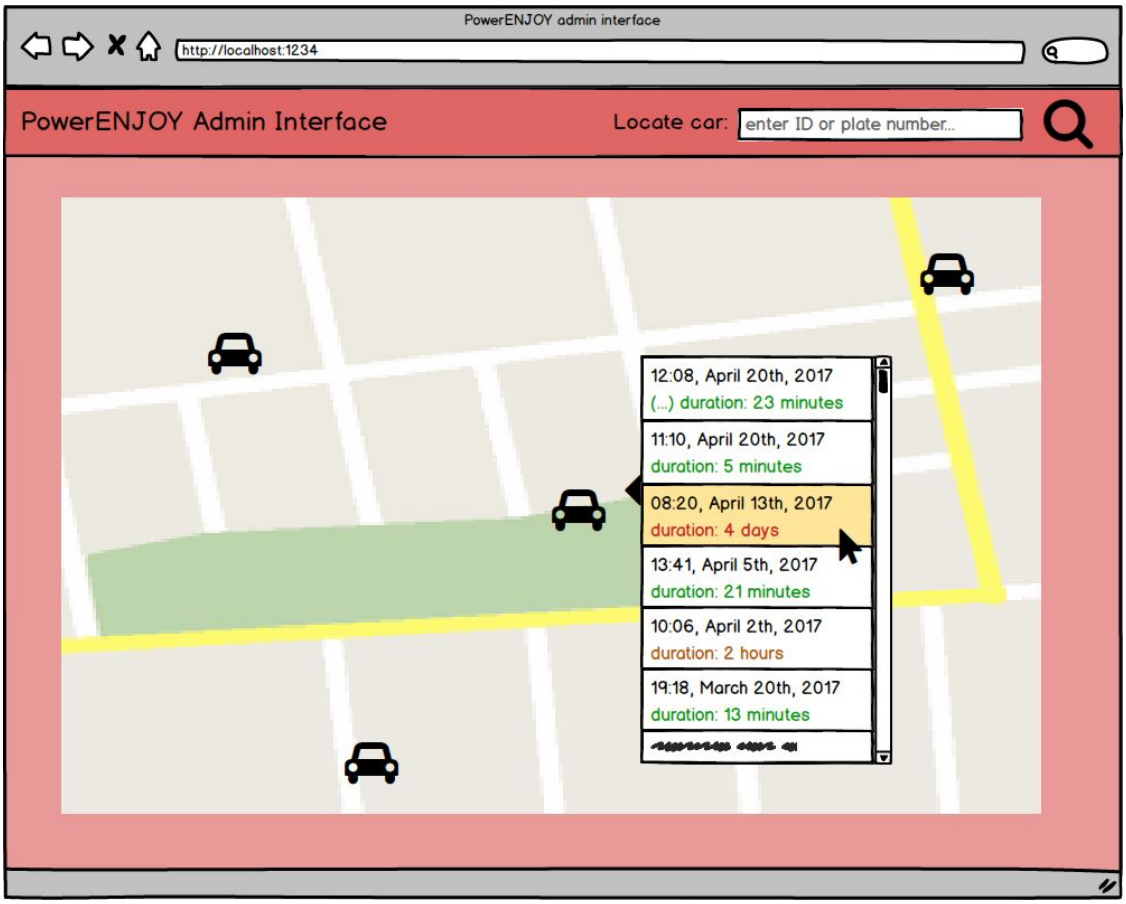

Reserve car
http://powerenjoy.com/reserve
Select distance:
km
5
Search through address
address
Search from my position
Search
70% charge
Reserve
94% charge
Reserve

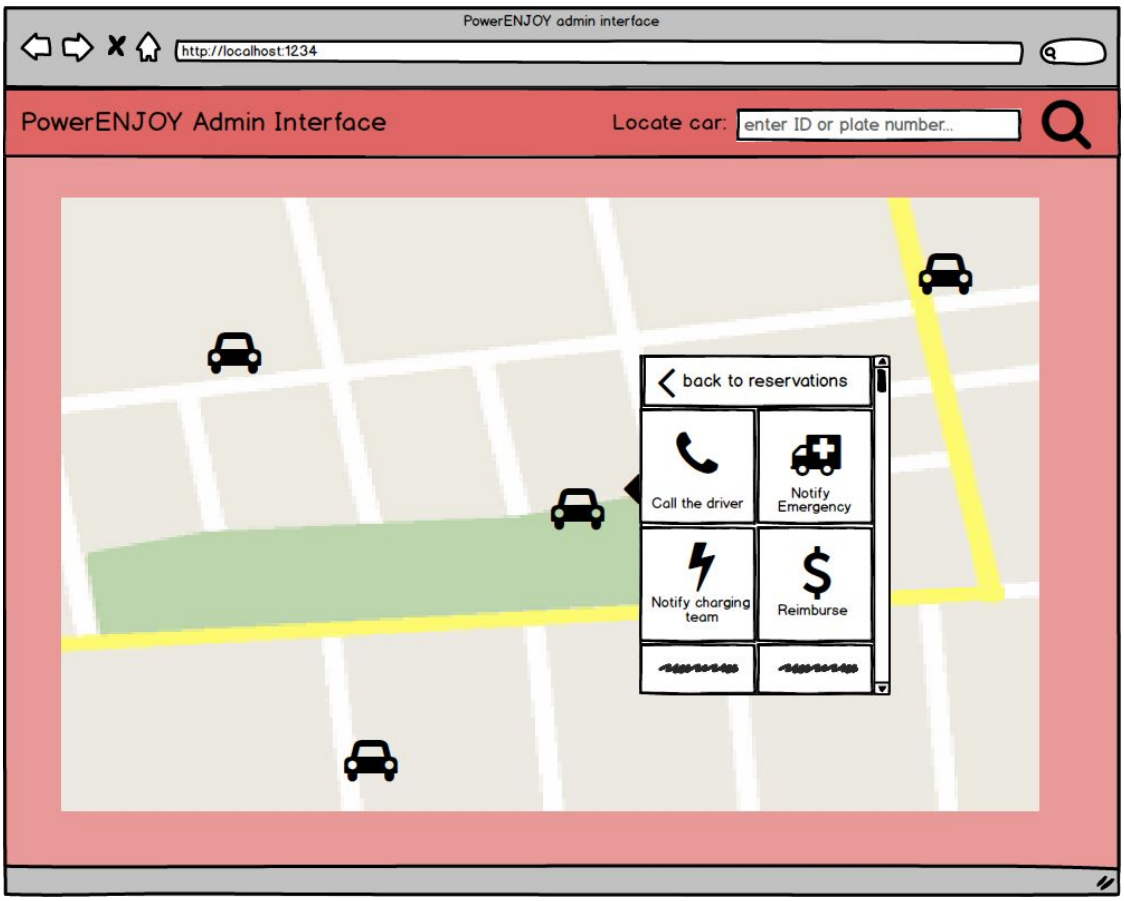

Power EnJoy
Sign in
Create an account
terms and conditions
help
Sign up -- STEP 1 of 6
First name
Last name
Phone num.
Birth day
City
Country
ID card number
License number
Sign up -- STEP 2 of 6
Credit card
Bank account
Credit card
Number
CVV
Expiration date
Bank account
IBAN
Account holder
Hi, John Smith!
Look for a ride
Past reservations
Contact support
enter location...
Model: Fiat 500
Charge: 87%
Book this ride
Back to menu
My reservations
Active reservation
34 min left
Unlock
Cancel
10:06, April 2th, 2017
duration: 24 minutes
23:18, March 24th, 2017
duration: 15 minutes
Back to menu

4.2. PE Administrator

Administrator application will be used by call center operators inside the actual PE headquarter. It is a web-based application and offers some additional functionality with respect to the user application.

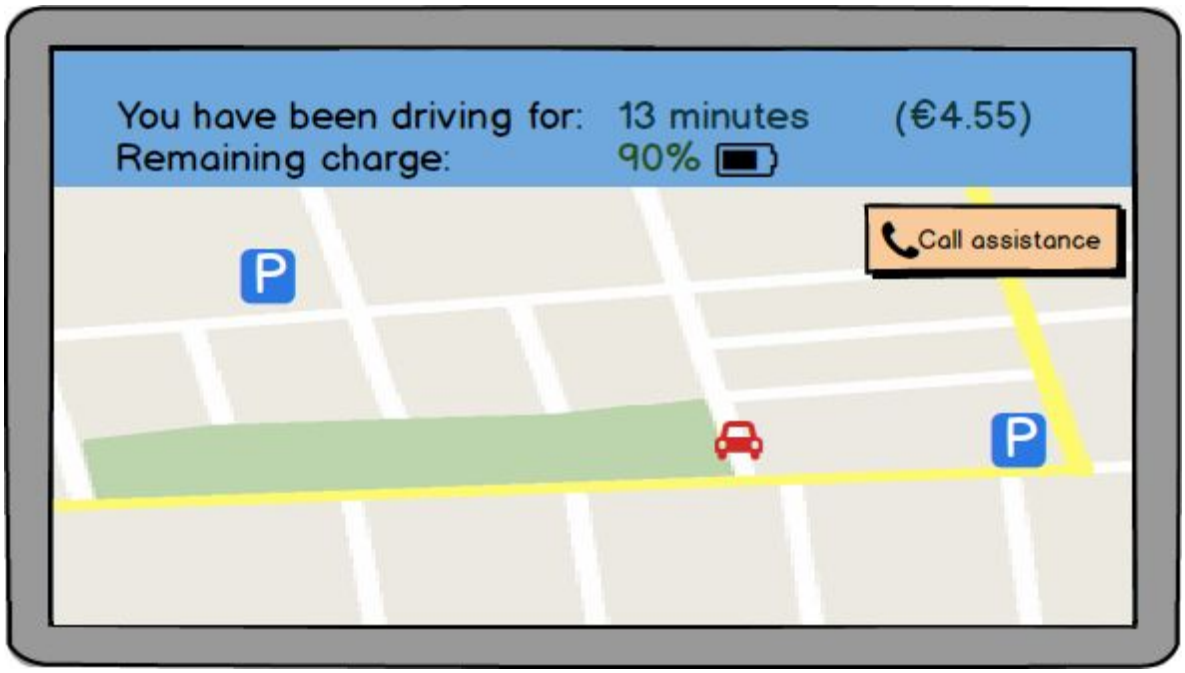



PowerENJOY admin interface
http://localhost:1234
PowerENJOY Admin Interface
Locate car:
enter ID or plate number...
12:08, April 20th, 2017
(...) duration: 23 minutes
11:10, April 20th, 2017
duration: 5 minutes
08:20, April 13th, 2017
duration: 4 days
13:41, April 5th, 2017
duration: 21 minutes
10:06, April 2th, 2017
duration: 2 hours
19:18, March 20th, 2017
duration: 13 minutes


PowerENJOY admin interface
http://localhost:1234
PowerENJOY Admin Interface
Locate car:
enter ID or plate number...
back to reservations
Call the driver
Notify Emergency
Notify charging team
Reimburse

4.3. PE Car

PE Car offers a minimal interface based on the Android SDK. The interaction between the user and the car is reduced to the minimum because the user can use his/her mobile phone to interact with the car.



5. Requirements traceability

This section investigates how the functional and non-functional requirements stated in the RASD have been concretely taken into account during the design process.

5.1. Functional requirements

In the following table we mapped the functional requirements specified in the RASD document to the components defined in this document.

	Application server											Front-end applications			
Requirement	**Customer component**	**Ride manager**	**Account manager**	**Reservation manager**	**Car finder**	**Payment manager**	**Input validator**	**Notification manager**	**Car component**	**Admin component**	**DB Manager**	**PEC**	**PEA**	**PEW**	**PEM**
R1.1														X	X
R1.2					X										
R1.3									X			X			
R1.4	X				X									X	X
R1.5	X								X					X	X
R1.6					X				X						
R1.7				X							X			X	X
R1.8				X					X		X	X			
R1.9				X	X				X		X	X			
R1.10				X		X		X			X			X	X
R1.11					X				X						
R1.12	X	X						X	X					X	X
R1.13	X							X	X			X			X
R1.14									X			X			
R1.15	X	X						X	X		X	X			X
R1.16	X		X			X					X				
R1.17									X			X			
R1.18												X			
R1.19		X							X			X			
R1.20		X							X			X		X	X
R1.21	X			X				X	X			X		X	X
R1.22	X		X	X										X	X
R1.23			X				X			X	X				
R1.24	X		X			X	X							X	X
R1.25	X		X				X				X			X	X
R1.26	X		X								X			X	X
R2.1		X			X				X		X	X			
R2.2							X								
R2.3		X				X			X		X	X			
R2.4		X				X			X		X	X			
R2.5			X				X			X					
R3.1			X		X	X		X	X	X	X		X		

5.2. Non functional requirements

- **Performance**: since the number of requests to PE server will reach high peaks in certain hours or days (e.g. during a public transportation strike), PE should guarantee high performance for the most common requests, namely the search for available cars, car tracking and updates to car position. For this reason we chose a proper data structure and designed a suitable algorithm for these common queries as explained in details in the section 3 of this document. These algorithms will be implemented in the *Car finder* component.
- **Reliability**: this can be guaranteed up to a certain level by replicating the beans in the application server in different machines.
- **Availability**: PE application server should guarantee 99.99% availability. This is a critical aspect especially during strikes or other 'special' days where the number of requests may increase dramatically. To achieve this result, PE system will take use of replicating EJB over multiple machines.
- **Security**: The following procedures have been taken into account to comply PCI Data security standard:
    - PE administrator interface will be accessible only from the PE headquarter.
    - Encryption of passwords and personal informations of the users
    - Monitoring of the accesses to the application server.
- **Maintainability**: this aspect is enforced by the MVC pattern we have used and by decoupling the application server in functional components. We also managed to apply the Strategy design pattern when modelling the discounts so that administrators can add/remove new discounts or penalties.
- **Portability**: the choice of deploying a web application for users and call center administrators and the ability for users to unlock cars with text messages have been made in this direction.
- **User interface and human factor**: the user interface is designed to be simple and has at most 4 nested levels. This means that any user can get to the right functionality within 4 steps.

6. Appendix

6.1. Used tools

- Google Docs (http://www.docs.google.com) for this document
- Star UML (http://staruml.io/) for the UML modelling, in particular use case, statechart, activity, class and sequence diagrams
- Alloy Model 4.2 (http://alloy.mit.edu/alloy) to generate the world and proving its consistency
- Balsamiq Mockup (http://balsamiq.com/products/mockups/) for the sketch on the user interfaces of this software system

6.2. Hours of work

The following are the total number of hours spent by each member of the group:

- Andrea Battistello: 28 h
- William di Luigi: 18 h

6.3. Revision history

Version	*Date*	*Revision description*	*Revision notes*
1.0	10-12-2016	Final draft	-
1.1	11-12-2016	First release	Minor corrections