POLITECNICO DI MILANO

POWERENJOY

SOFTWARE ENGINEERING 2

Design Document

Authors:
Giancarlo COLACI
Giulio DE PASQUALE
Francesco RINALDI

Supervisor:
Elisabetta DE NITTO

January 15, 2017

Contents

1 Introduction

1.1 Revision History

Version	Date	Author(s)	Summary
1.1	14/01/2017	Giancarlo Colaci, Giulio De Pasquale, Francesco Rinaldi	Name refactoring
1.0	12/12/2016	Giancarlo Colaci, Giulio De Pasquale, Francesco Rinaldi	Initial Release

1.2 Brief description

This document describes design process for PowerEnjoy project based on the requirements described in the RASD. It is divided into four sections, following *IEEE* standard:

- **Architecture Design**, which describes the structure of the system, highlighting software elements and relations among them and their properties;
- **Data Design**, which describes the chosen data structures, dictated by attributes and relationships among data objects;
- **Interface Design**, which describes either internal or external program interfaces;
- **Procedural Design**, which describes procedural details using graphical notations that facilitates translation to code.

1.3 Purpose

The purpose of this document, the Design Document, is to give more technical details than the RASD about PowerEnjoy system. In more detail, this document is addressed to software developers to let them better understand what is to be built and how it is expected to be done. The main goal of this document is to completely describe the system-to-be by detecting and describing the adopted architectural style, the high-level components of the software-to-be and how these components are distributed on the architecture's tiers and how they communicate and interact with each other, according to the dictates and the guide-lines provided by the RASD.

1.4 Scope

The aim of this project is to develop a web/mobile application for PowerEnjoy, in order to let any registered and active user to reserve a car nearby and drive it in a simple and effective way. Thanks to a smarter uniform distribution of cars in the city that maximizes the probability to find a car nearby and to our competitive prices and some special discount, every user will experience a totally new feeling of freedom and will enjoy saving while driving. No car parking ticket. No refuelling. No road tax. No hidden costs. PowerEnjoy will make it possible.

1.5 Definitions, Acronyms, Abbreviations

Below there are definitions of some terms that will be used in the document, in order to avoid any ambiguity in their use and their understanding.

API: Application Programming Interface; it is a common way to communicate with another system.

DBMS: DataBase Management System, a software that control the creation, maintenance and use of a database. (e.g. MySQL)

DD: Design Document.

EJB: Enterprise JavaBeans, a managed server-side component architecture for modular construction of enterprise applications.

***EJB Container*:** a software that implements Sun EJB specifications in order to correctly manage session and entity beans.

***Entity bean*:** a distributed object that have persistent state.

JavaBeans Components: Objects that act as temporary data stores for the pages of an application.

JAX-RS: Java API for RESTful Web Services, a Java programming language API specification that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.

JDBC: Java Database Connectivity, a Java API that allows the application to communicate with a database.open source database manager system, a software capable of managing data.

JSF: JavaServer Faces, a Java specification foRevision History

r building component-based user interfaces for web applications.

JSP Pages: text documents that contain two types of text: static data, which can be expressed in any text-based format (such as HTML, SVG, WML, and XML), and JSP elements, which construct dynamic content.

HTML: HyperText Markup Language, a markup language for building web pages.

HTTP: Hypertext Transfer Protocol, an application protocol used by web browsers.

MVC: Model View Controller.

MySQL: open source database manager system, a software capable of managing data.

RASD: Requirements Analysis and Specifications Document.

***URL*:** Uniform Resource Locator.

***Push notification*:** it is a notification sent to a smartphone using the mobile application, so it must be installed.

Push service: it is a service that allows to send push notifications with own API.

REST: REpresantional State Transfer.

RESTful: REST with no session.

UX: User eXperience.

UI: User Interface.

BCE: Business Controller Entity.

1.6 Reference documents

- Requirements Analysis and Specification Document produced before: [RASD.pdf]
- Specification Document: [ASSIGNMENTS AA 2016-2017.pdf]

1.7 Document Structure

1. **Introduction**: this section introduces the design document. It contains a justification of his utility and indications on which parts are covered in this document that are not covered by RASD.
2. **Architecture Design**: this section is divided into different parts.
    - *Overview*: this sections explains the division in tiers of our application.
    - *High level components and their interaction*: this sections gives a global view of the components of the application and how they communicate.
    - *Component view*: this sections gives a more detailed view of the components of the applications.
    - *Deploying view*: this section shows the components that must be deployed to have the application running correctly.
    - *Runtime view*: this section shows the course of the different tasks of our application through several sequence diagrams .
    - *Component interfaces*: this section shows the interfaces between the components.
    - *Selected architectural styles and patterns*: this section explain the architectural choices taken during the creation of the application.
    - *Other design decisions*
3. **Algorithms Design**: this section describes the most critical parts via some algorithms. Pseudo code is used in order to hide unnecessary implementation details in order to focus on the most important parts.
4. **User Interface Design**: this section presents mockups and user experience explained via UX and BCE diagrams.
5. **Requirements Traceability**: this section aims to explain how the decisions taken in the RASD are linked to design elements.
6. **Appendix**: in this section will be listed the different tools we used and the hours of work spent by each member of the team.

2 Architectural Design

2.1 Overview

The PowerEnjoy system will be developed on the Java 2 Enterprise Edition platform, one of the most important technology used to build large scale, distributed, component-based, web-enabled and multi-tier applications. J2EE provides also robustness, efficiency and *portability* of code because it is based on Java technology and standard-based Java programming APIs: in this way, we will be able to "write once" our application and "run it anywhere". The system will be designed according to the traditional multi-tier architecture, in order to guarantee:

- *flexibility*: by separating the business logic of an application from its presentation logic, a multi-tier architecture makes the application much more flexible to changes;
- *maintainability*: because of their independence and the different technology used in each layer, changes should have no effect on any other layer and can be managed by independent teams with different skills.
- *reusability*: separating the application into multiple layers makes it easier to implement re-usable components.
- *scalability*: a multi-tier architecture allows distribution of application components across multiple servers thus making the system much more scalable

In detail, PowerEnjoy architecture will be made up of four logical levels: the **Client** tier, the **Web** tier, the **Business** tier and the **EIS** tier.

- The **Client** tier consists of application clients that make *requests* to the server, which processes the requests and returns a *response* back to the client. The Client tier is accessible via *mobile application* (that interacts directly with the application server via RESTful API) or by a *web browser* (that run JavaScript and apply Cascading Style Sheets (CSS) before interpreting pages marked up with HTML), and it interacts with the Web tier through an HTTP connection, eventually sending data provided by the user.
- The **Web** tier consists of components implemented with JEE technology that handle the interaction between clients and the business tier. Its main tasks are collect HTTP requests from users of the client interface and return appropriate anwers with HTML pages, generated with the data received from the components in the business tier.
- The **Business** tier consists of Enterprise JavaBean (EJB), software components that encapsulate all the logic of an application. This tier uses the Java Persistence API (JPA) to access the database, abstracting the relational model implemented by the database in a model of data objects, in order to allow the interaction with the application.
- The **EIS** tier consists of a Database Management System (DBMS) in charge of create, manipulate, query and extract data over a database. The business tier uses the Java Database Connectivity (JDBC) to have access to the resources located on the database.

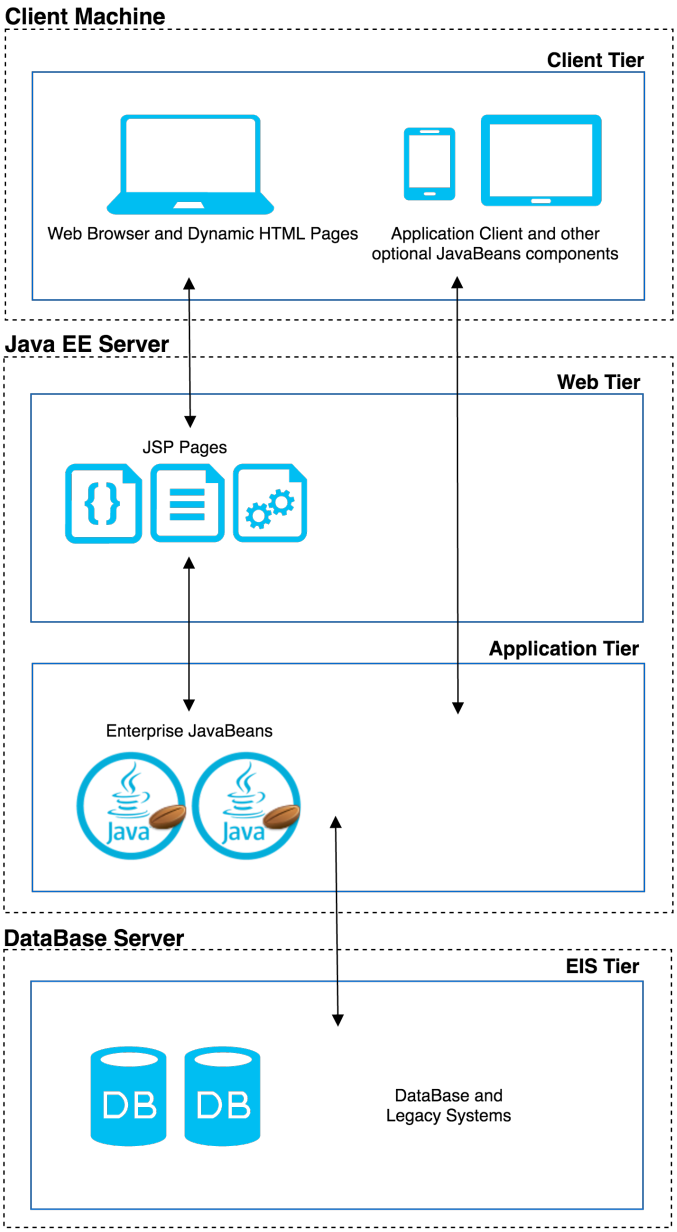

Figure 1: MultiTier Architecture

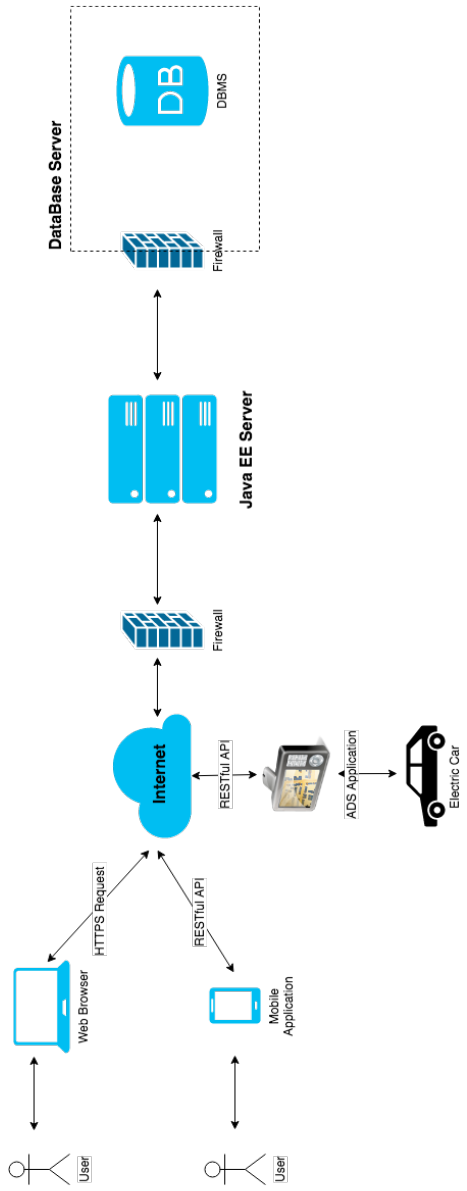

Figure 2: Hardware Representation

2.2 High level components and their interaction

The high level components architecture is composed of different elements. The client can initiate the communication with the server from his mobile application or from the webpage of the application. This communication is made in a synchronous way since the client, who initiates the communication, has to wait the answer of the application that acknowledge him that his request has been taken into account. The application will receive two kinds of requests from the client: direct ones, if he's using the mobile application, or redirected ones by the webserver, if he's using the web browser. The application will later send an asynchronous message to the client to specifiy the status of his reservation. The application communicates also with another type of component, the cars. The application can send synchronous messages to the cars to check their status and eventually reserve one of them for each user or manage any kind of maintenance.

The interactions between the main components are shown in figure:

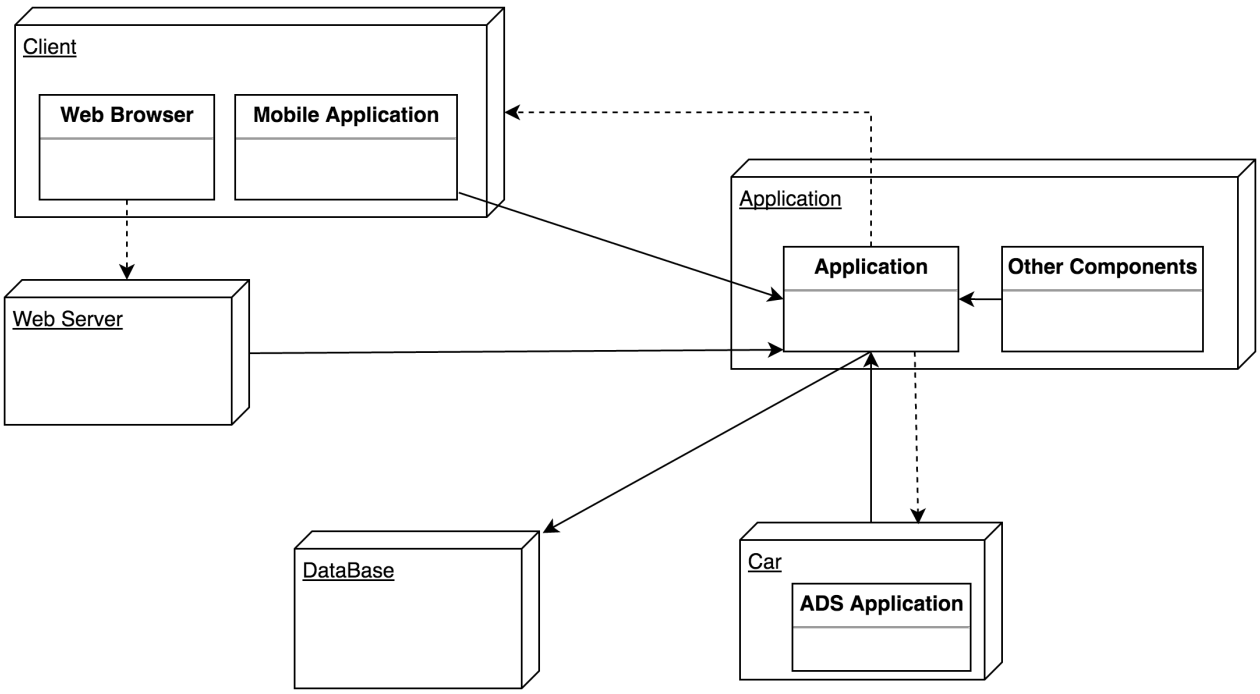

Figure 3: High Level Components

2.3 Component view

We adopted a *top-down* approach to separate, at least from a logical point of view, different set of functionalities. Once the system was decomposed into several sub-systems, we adopted a *bottom-up* approach in order to create more modular and reusable components. Here are listed the most relevant components the system has been split into.

- **PowerEnjoyApp GUI**: this is the graphic interface which let the user send request to the server and receive response via mobile application;
- **PowerEnjoyWeb GUI**: this is the graphic interface which let the user send request to the server and receive response via web browser;
- **Authentication Manager**: this is the component responsible for the customer registration and login system. It handles activities such as creating a new customer account, log into the system or deleting an existing account;
- **AccountInformation Manager**: this is the component responsible for the customer profile management. It handles activities such as editing personal and billing data, enabling/disabling money saving option and consulting reservation history;
- **CheckAvailability Manager**, this is the component responsible for searching available cars in a specific location;
- **Reservation Manager**: this is the component that manages each reservation. It handles activities such as creating a new reservation, checking the active reservation information (such as the remaining time until it expires, the reserved car's position and status or the elapsed rental time) and terminating the active ones;
- **Car Manager**: this is the component that check and manage the status of each car by communicating with them through the ADS of each present on each car. If a car needs any kind of maintenance, it will send a request to an external maintenance service;
- **ADS_Application Manager**: this is the component which guarantee the correct communication between the Server and the ADS Application installed on each ADS on each electric car;
- **Payment Manager**: this is the component that calculates the fees for the rides at the end of each reservation and send a payment request to an external service;
- **Data Manager**: this is the component where the data is managed and viewed;
- **Notification Manager**: this is the component that manages all the notifications.

Our application will use Google Maps API to manage information about the actors' position.

The interactions between the main components are shown in figure:

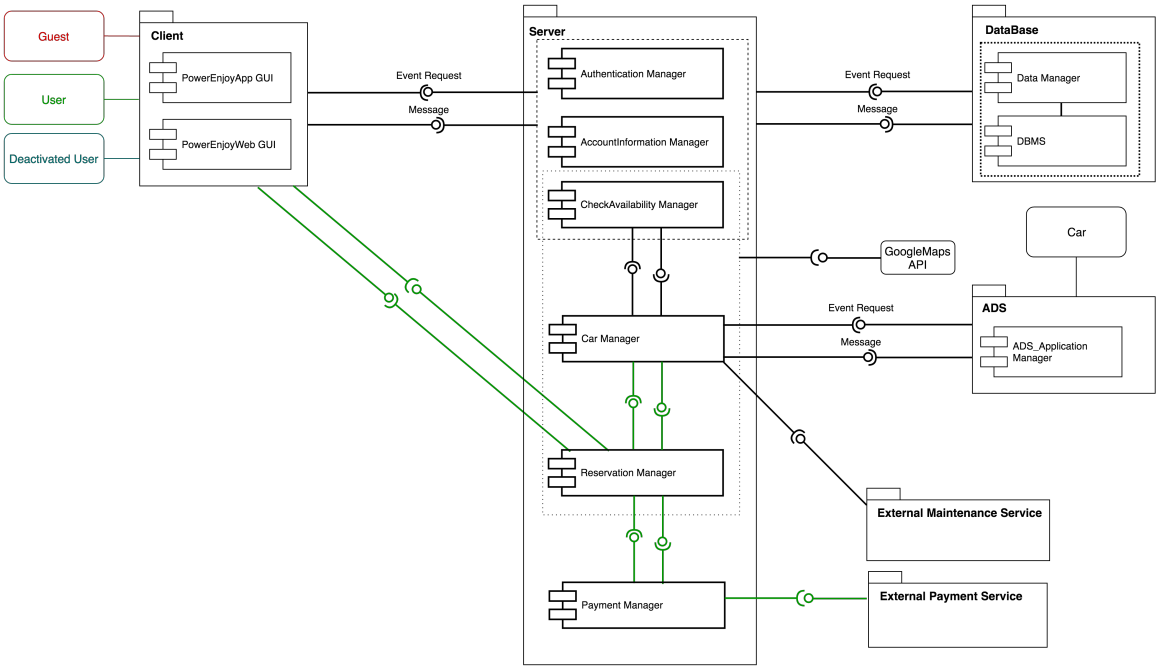

Figure 4: Component View

In this representation, in order to avoid redundancy due a duplication of the same elements, we decide to use a different color for the actors to enlight the fact that only an active user can have the possibility to reserve a car.

2.4 Deployment view

In this section in shown how the software components are distributed with respect to the hardware resources available in the system. The mobile applications communicate directly with application server via RESTful API while the browser applications communicate with the Web server that receives users' requests, interacts with the Business server and provides HTML pages in order to start the user experience. The system can retrieve information from each car communicating with their ADS via RESTful API. The Business server contains the application logic and the modules that handle the main functionalities of the system. Every module is accessible only by the allowed users. All the data needed for the operations handled by the modules are retrieved from the database in the Data server. The various interactions with the database are monitored by the DBMS in order to guarantee the ACID properties of the database.

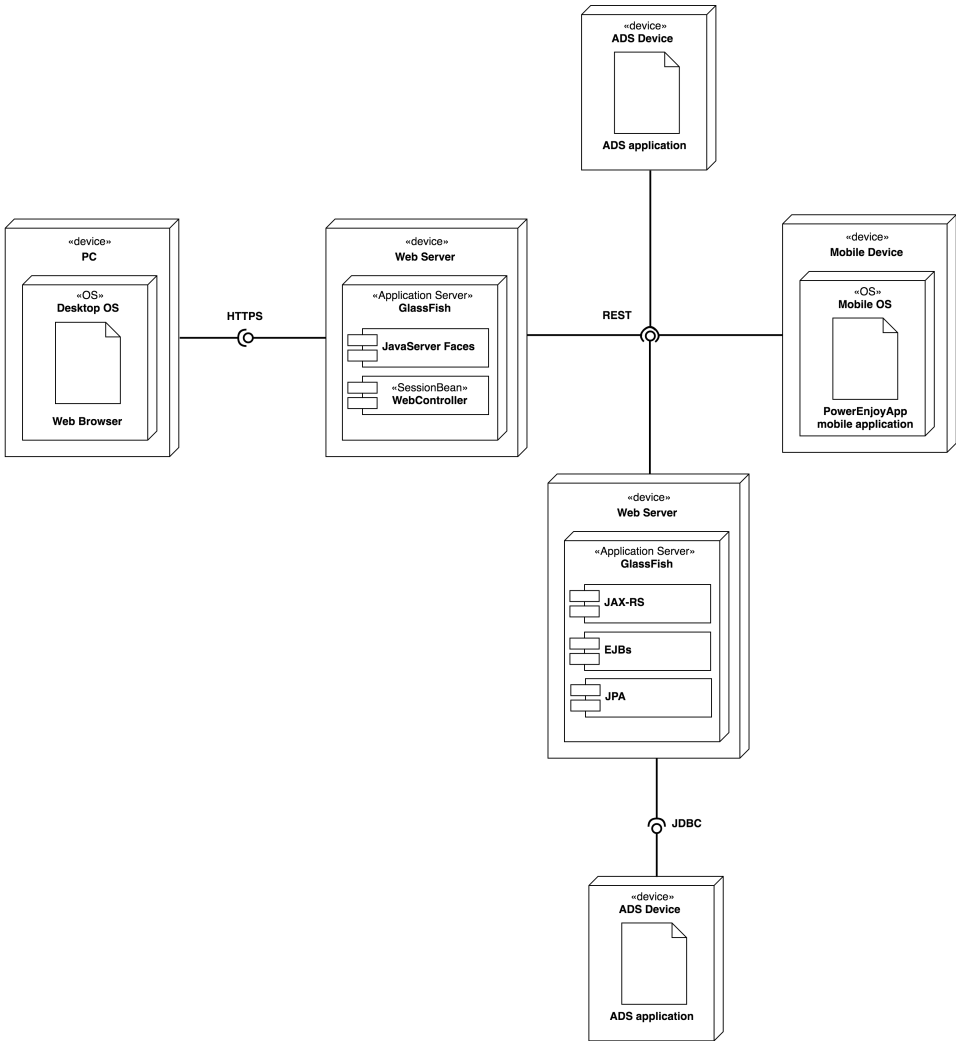

Figure 5: Deployment View

2.5 Runtime view

In this section are explained through Sequence diagrams some of the most relevant interactions between components and actors of the system. Like in the RASD document, some assumptions are made in order to simplify the representation and clarify some aspects; by the way, some of the most important features, if considered implicit in a Sequence diagram, are explicitly represented in another one.

Login Sequence Diagram

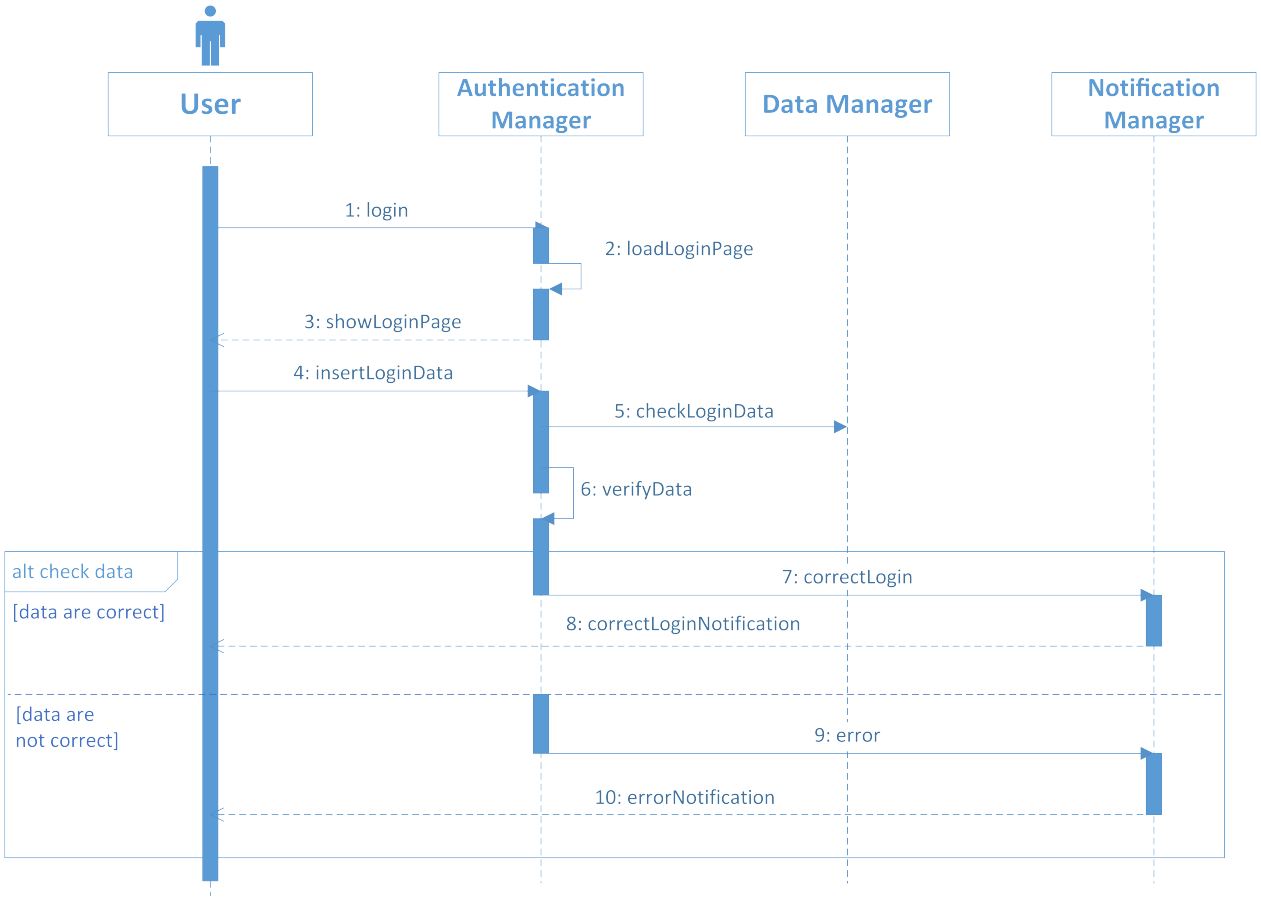

Figure 6: Login Sequence Diagram

Registration Sequence Diagram

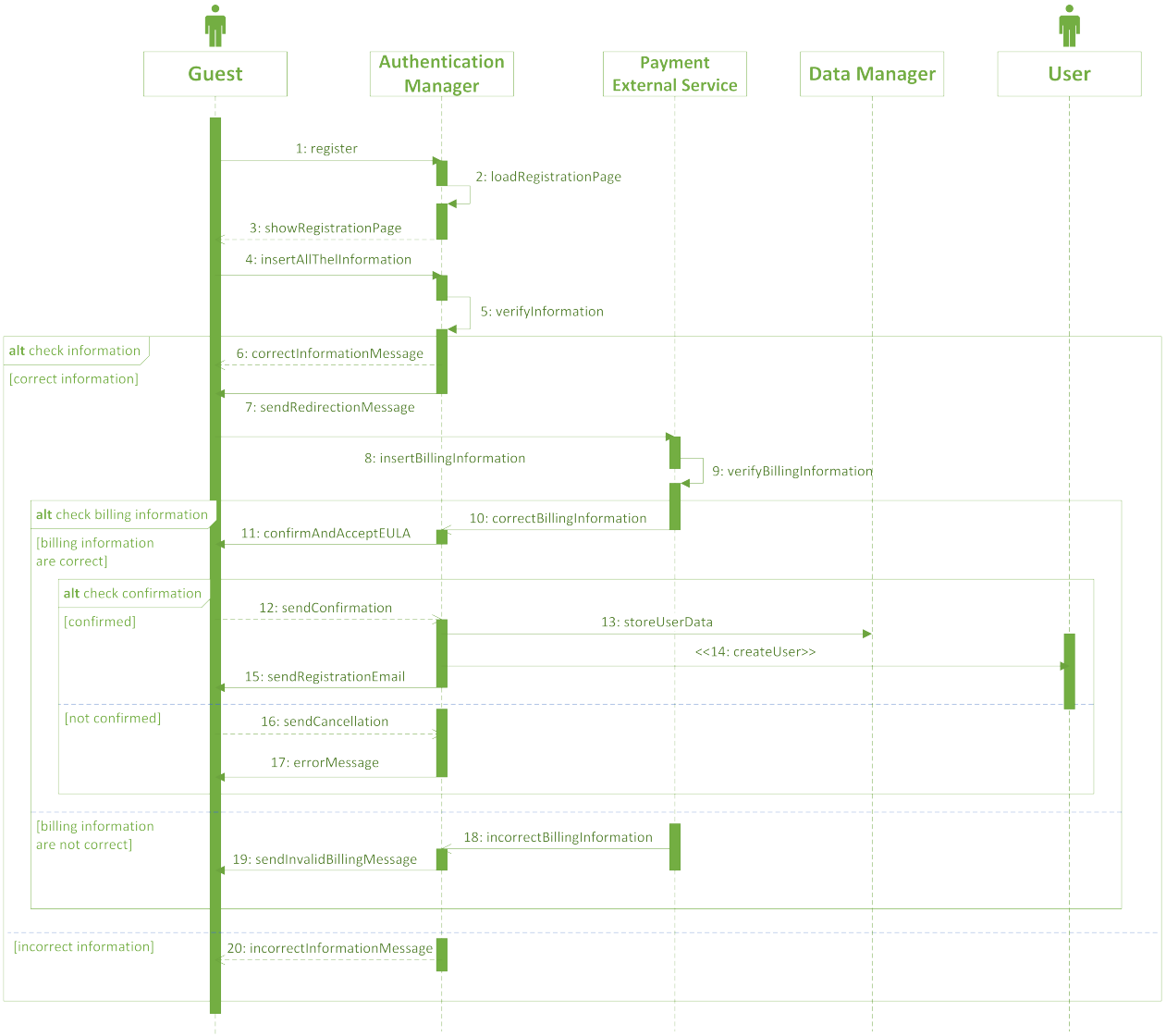

Figure 7: Registration Sequence Diagram

Check Cars' Availability Sequence Diagram

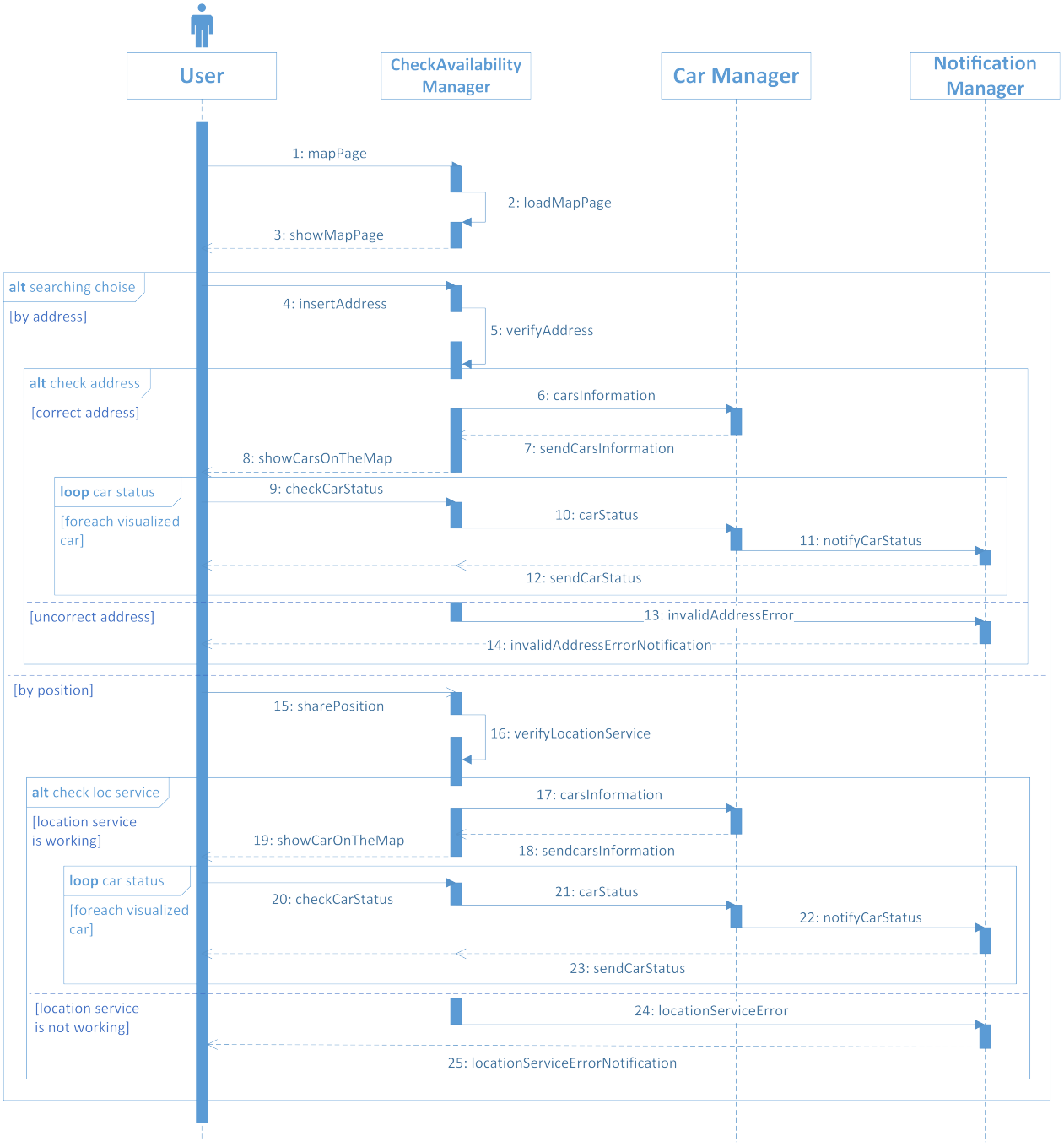

Figure 8: Check Cars' Availability Sequence Diagram

Modify Personal Information Sequence Diagram

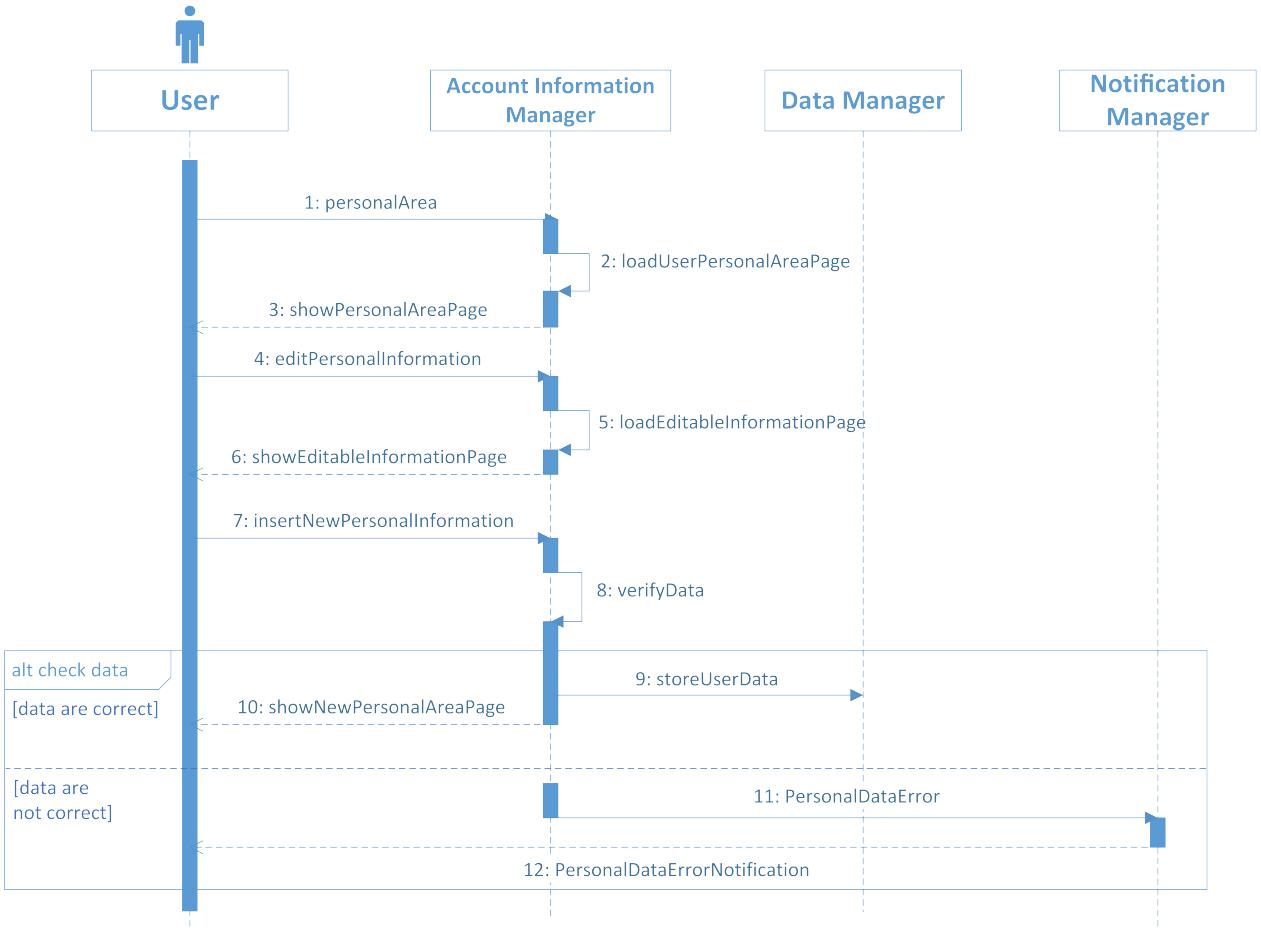

Figure 9: Modify Personal Information Sequence Diagram

Car Reservation Sequence Diagram

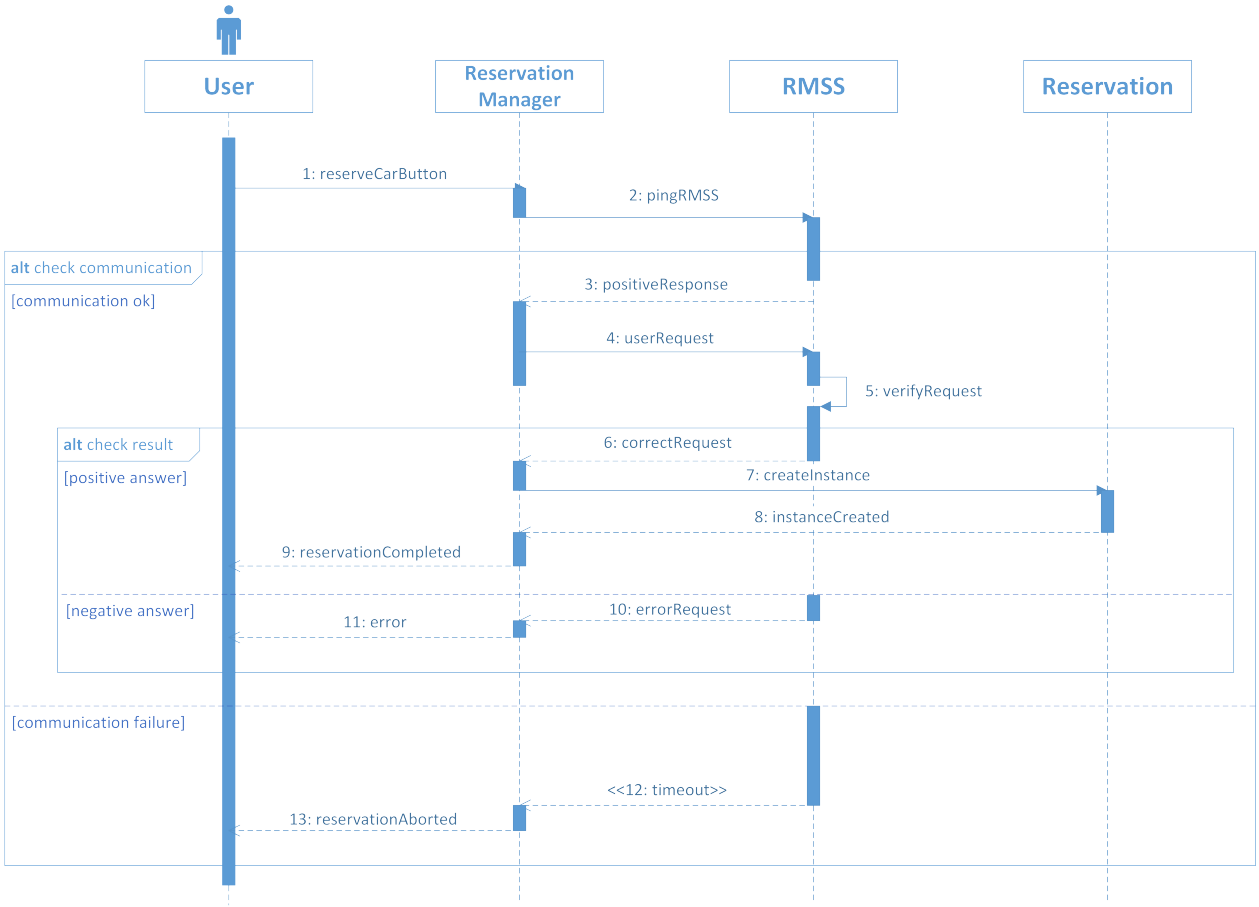

Figure 10: Car Reservation Sequence Diagram

Car Unlock Sequence Diagram

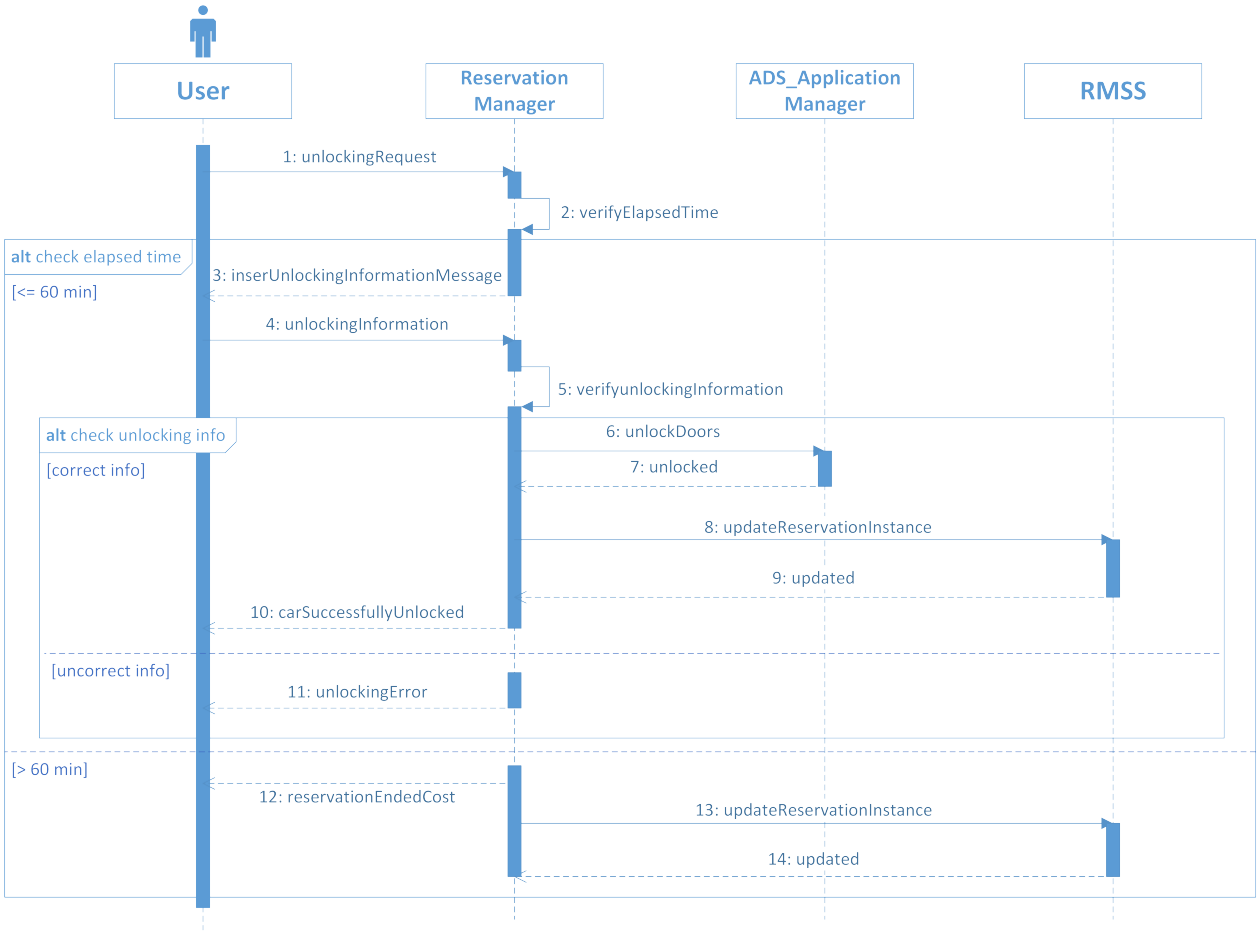

Figure 11: Car Unlock Sequence Diagram

Car Rental Sequence Diagram

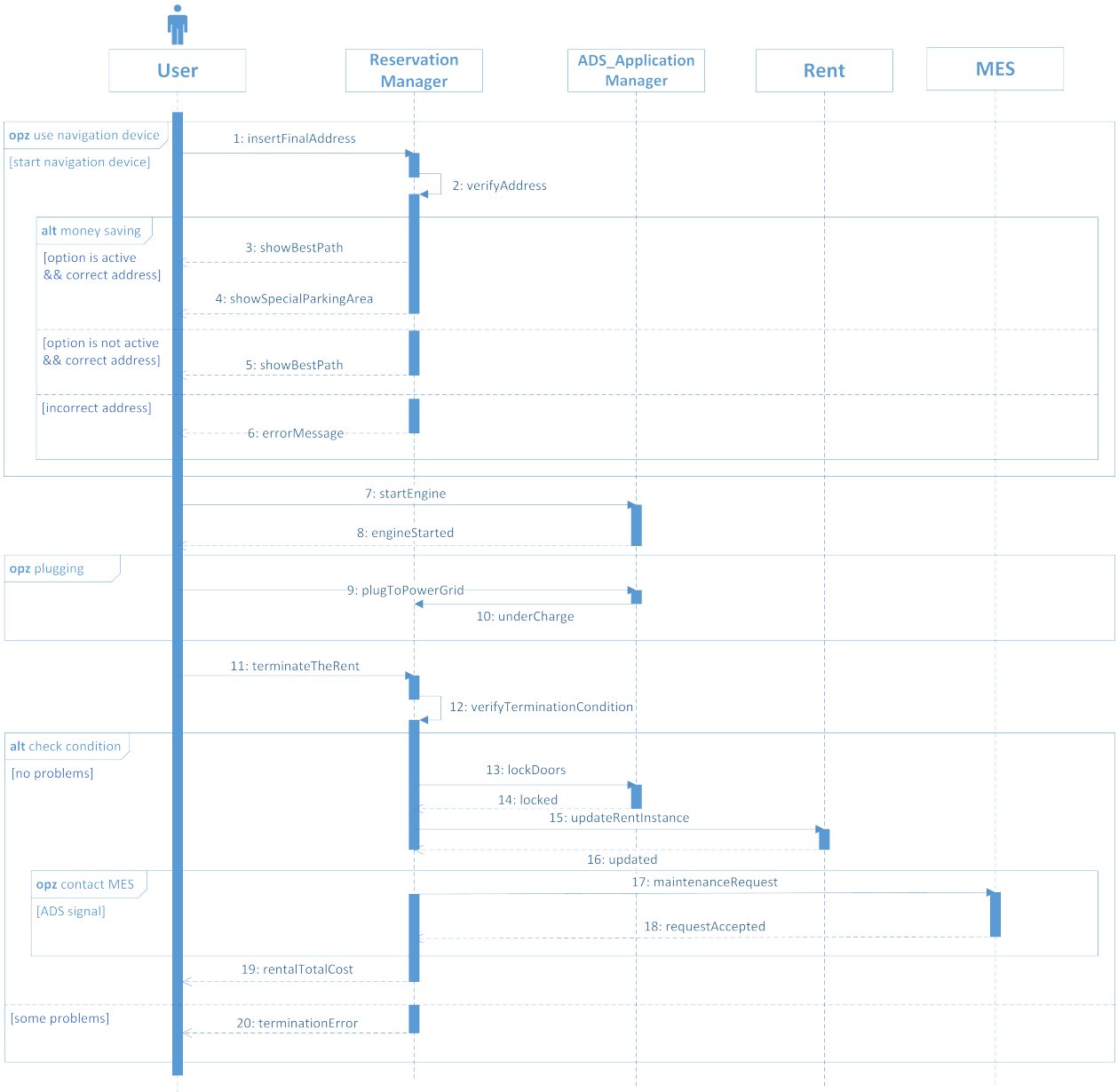

Figure 12: Car Rental Sequence Diagram

2.6 Component interfaces

In this section we're going to describe the interfaces through which the various components of the system communicate. For each interface we'll provide a short but exhaustive description of the main exposed methods.

Authentication Manager

Functions implemented by **Authentication Manager**:

`register(UserData)`: This function creates a new `User` instance in the system with all the information provided by the user during the registration phase. If the entered data are correct, an email is sent to the user address to confirm the correct registration to the service.

`login(Credentials)`: This function allows any registered user to log into the system using his *username* and *password*.

`checkCredentials(Credentials)`: This function verifies the *username - password* combination is correct. If the credentials are correct, the function returns a token to be used in the future requests to identify the user. Otherwise, an error is returned.

AccountInformation Manager

Functions implemented by **AccountInformation Manager**:

`remove(User)`: This function deletes the `User` from the system.

`activate(User)`: This function restores the privileges of the `User`.

`deactivate(User)`: This function removes some of the privileges from the `User`.

`editProfile(User, UserData)`: This function updates the existent `User` instance in the system with all the new information provided by the `UserData` object.

`getHistory(User)`: This function allows a registered `User` to consult his complete requests history.

`enableMSO(User)`:This function enables the Money Saving Option for the `User`.

`disableMSO(User)`:This function disables the Money Saving Option for the `User`.

CheckAvailability Manager

Functions implemented by **CheckAvailability Manager**:

`getUserPosition(User)`: This function will retrieve the information about the user's position through the Google Maps API.

`getAvailableCars(Location)`: This function will retrieve the needed information through the Car Manager and it will return a list of *available* cars near the user's `Location`.

Reservation Manager

Functions implemented by **Reservation Manager**:

`checkVerificationCode(User, int)`: This function verifies the `User` - *verification code* combination is correct.

`getUserPosition(User)`: This function will retrieve the information about the user's position through the Google Maps API.

`getReservableCars(Location)`: This function will retrieve the needed information through the Car Manager and it will return a list of *reservable* cars near the user's `Location`.

`startReservation(Car, User)`: This function creates a new instance of `Reservation`, tying together the `Car` and `User` instances passed as argument, and notifies it to the `User` through the Notification Manager.

`checkReservationStatus(Reservation)`: This function retrieves all the information about the `Reservation` instance passed as argument, and shows them to the `User` through the Notification Manager.

`endReservation(Reservation)`: This function terminates the current `Reservation` and send a payment request to the Payment Manager.

`startRent(Reservation)`: This function creates a new instance of `Rent`, updating the `Reservation` instance passed as argument, and notifies it to the `User` through the Notification Manager.

`checkRentStatus(Rent)`: This function retrieves all the information about the `Rent` instance passed as argument, and shows them to the `User` through the Notification Manager.

`isTerminable(Rent)`: This function will check through the Car Manager if all the conditions to end the `Rent` are respected (for example if nobody is still in the car, and so on..) and it will return `true` if the rent is terminable, `false` otherwise.

`endRent(Rent)`: This function terminates the current `Rent`, notifies it to the user through the Notification Manager and sends a payment request to the Payment Manager.

ADS_Application Manager

Functions implemented by **ADS_Application Manager**:

`getStatus()`: This function asks the ADS to check the current `Status` (*available*, *reserved*, *in_use*, *unavailable*) of the car.

`getDamages()`: This function queries the ADS to check the eventual car's damages through the sensors installed in there.

`getPosition)`: This function queries the ADS to check the current position of the car.

`getPassengers()`: This function queries the ADS the number of passengers actually inside the car.

`getPowerGrid()`: This function queries the ADS to check if the car is plugged into the power grid.

`lockDoors()`: This function queries the ADS the lock the car's doors and to change the `Status` of the car into *available*.

`unlockDoors()`: This function queries the ADS the unlock the car's doors and to change the `Status` of the car into *in_use*.

`updateCarStatus(Status)`: This function queries the ADS to change the car's `Status` into the one passed as argument.

Car Manager

Functions implemented by **Car Manager**:

`contactMaintainanceService(Car)`: Every time the Car Manager retrieve information about a car's status, if necessary, it will send a maintainance request to an external service through this function.

`ping(Car)`: This function queries a `Car` to check whether it is online.

`carsInRadius(Location)`: This function retrieves all the cars near the `Location` passed as argument.

Payment Manager

Functions implemented by **Payment Manager**: `applyReservationFees(Reservation)`: This function will apply the fees for each `Reservation` instance. Actually, the user can terminate a reservation for free, but in the future if PowerEnjoy would decide to apply a fee for the termination of a reservation, it would be very easy to modify this value. If the user picks up the car, his reservation terminates and the system will create another instance `Rent`: no fees will be applied to transform a `Reservation` into a `Rent`. `calculateRentFees(Rent)`: This function will calculate the fees for each `Rent` instance. In this case we have to consider eventual discounts and other additional fees before sending the payment request to the external payment service. It would not be difficult in the future to change some parameters according to the PowerEnjoy policy. `calculateRentDiscount(Rent)`: This function will calculate the eventual discount for each `Rent` instance. `calculateAdditionalFees(Rent)`: This function will calculate the eventual additional fees for each `Rent` instance.

Data Manager

Functions implemented by **Data Manager**:

`store(Data)`: This function allows any authorized entity to store data into the DataBase.

`retrieve(Data)`:: This function allows any authorized entity to read data from the DataBase.

`delete(Data)`:: This function allows any authorized entity to delete data from the DataBase.

`edit(Data)`:: This function allows any authorized entity to modify data into the DataBase.

Notification Manager

Functions implemented by **Notification Manager**:

`notify(User, Notification)`: This function is used to notify a message to the user.

Other Specifications

- The front-ends of the system (the mobile app and the web application) communicate with the Application Server using the back-end interface implemented as a RESTful interface over the HTTPS protocol.
- The users' browsers will communicate with the web server via HTTPS requests. Any unencrypted request will be denied.
- The RESTful interface is implemented in the Application server using JAX-RS and uses XML as the data representation language.
- The Application Server communicates with the DBMS via the Java Persistence API over standard network protocols.
- The Application Server is configurable by means of a XML configuration file, that will contain any other setting useful in the implementation phase.

2.7 Selected architectural styles and patterns

2.7.1 The Architecture

The architecture of the system has been previously described at higher level.

As we said, we adopted a **multi-tier architecture** in which every tier is hosted on a different machine: the *Client* tier is on the pc or smartphone the user access the system from, the *Web* tier is hosted on a separate server used only for this purpose, while another server hosts the *Business* tier. Finally a last server contains the *Data* tier.

This division between all levels has been made in order to allow update and maintenance on a particular tier without affecting the others, and to prevent the whole system to interrupt in case of fault of a component. We also adopted the *thin client* paradigm to the interaction between user's machine and the system: in this way all the application logic is on the application server, which has sufficient computing power and is able to manage concurrency issue efficiently.

2.7.2 The Desing Pattern

The Design pattern used for this system is the **Model-View-Controller** (MVC).

This choice has been made mainly for the flexibility that this pattern provides: the *model*, the *view* and the *controller* are separated into different components that can be modified with a minimum impact with respect to the others, helping to achieve a better scalability and offering the possibility to add new functionalities to the system or to apply changes to the code. In more detail:

- The **Model** contains the representation of data and the application logic, so all the functions that manipulates data.
- The **View** makes the model suitable for the user interaction, providing a GUI: every user will access the system and all its functionalities through this component. Furthermore, it provides the possibility to have different representations (or views) for the same data, depending on the user's preference or on the user's permissions.
- The **Controller**, finally, is responsible for the communication between the View and the Model: in particular it responds to user's actions and invokes changes on the model.

2.8 Other design decisions

2.8.1 Storage of passwords

Users' password will not be stored in plain-text, but they will be hashed and salted with cryptographic hash functions. This provides a last line of defense in case of data theft.

2.8.2 Google Maps

The system uses an external service, Google Maps, to offload all the geolocalization, distance calculation and map visualization processes. The reasons of this choice are the following:

- manually developing maps for each city is not a viable solution due to the tremendous effort of coding and data collection required;
- Google Maps is a well-established, tested and reliable software component already used by millions of people around the world;
- Google Maps offers APIs, enabling programmatic access to its features;
- Google Maps can be used both on the server side (calculation, shortest paths, traffic, incident reporting) and on the client side (map visualization);
- the users feel comfortable with a software component they know and use everyday.

3 Algorithm Design

In this section we will focus mostly on two important themes: the calculus of the reservations' and rents' fees and the uniform redistribution of the cars. It is very important to implement the former in a dynamic and easy-to-change way because of the continuously changing business strategies and it is fundamental to give a lot of attention to the latter because it would considerably increase the quality of the service and incentivize the virtuous behaviours of the user, according to the core values of PowerEnjoy.

3.1 The uniform distribution of cars

Let's start from the attempt to redistribute in a uniform way the cars into each city.

The first kind of solution we will propose is valid from a theoretical point of view, but it is not so suitable to our service. We will explain it anyway to give a deep and better understanding of the problem and of our reasoning.

To guarantee an effectively uniform distribution of the cars, we would act in this way:

1. For each point (intended as a couple of two values: **latitude** and **longitude**) inside the city (intended as a *closed ensemble of points*), we would calculate the minimum distance with the first available car found.

2. Among these minimum distances, we would take in account the greatest one and we would put a new available car in that exact point.

By iterating these two steps, we would guarantee an uniform distribution of cars, starting from a 2-dimensional space (the city) already filled in part with other cars, unmovable.

However this solution is not a feasible one because it would be necessary to use an ingent calculus power to effectively substain a similar amount of operations. Perhaps it would be possible to achieve the same result with a smaller quantity of points (taken randomly or in a less accurate way) or in the opposite way: we could calculate the maximum distant point from each available car inside the city, and decide to address the user there.

However this theoretical solution won't take in account two important facts:

1. the first one is that we cannot address the user into a specific point (intended as above, a couple of *latitude* and *longitude*) but we have to address him/her into a recharging station *near* the point, so we will be able anyway only to achieve an *approximately* uniform distribution of cars.

2. the second one is that the user will never revolutionize his destination, so we have to address him into a point not so *far* from his final destination.

So we abandoned the utopia of a *perfect* uniform distribution of cars, and we focused mainly on the distribution of the car taking in account the PowerEnjoy power grid stations.

We first ranked them according to the *utility function* of the user, in agreement with the point of view of some experts: let's take in account Milan, the first city in which we offered our service. As showed in figure, we ranked the main zones of Milan giving them a number from 1 to 6, according to the attractivness of that place for our clients.

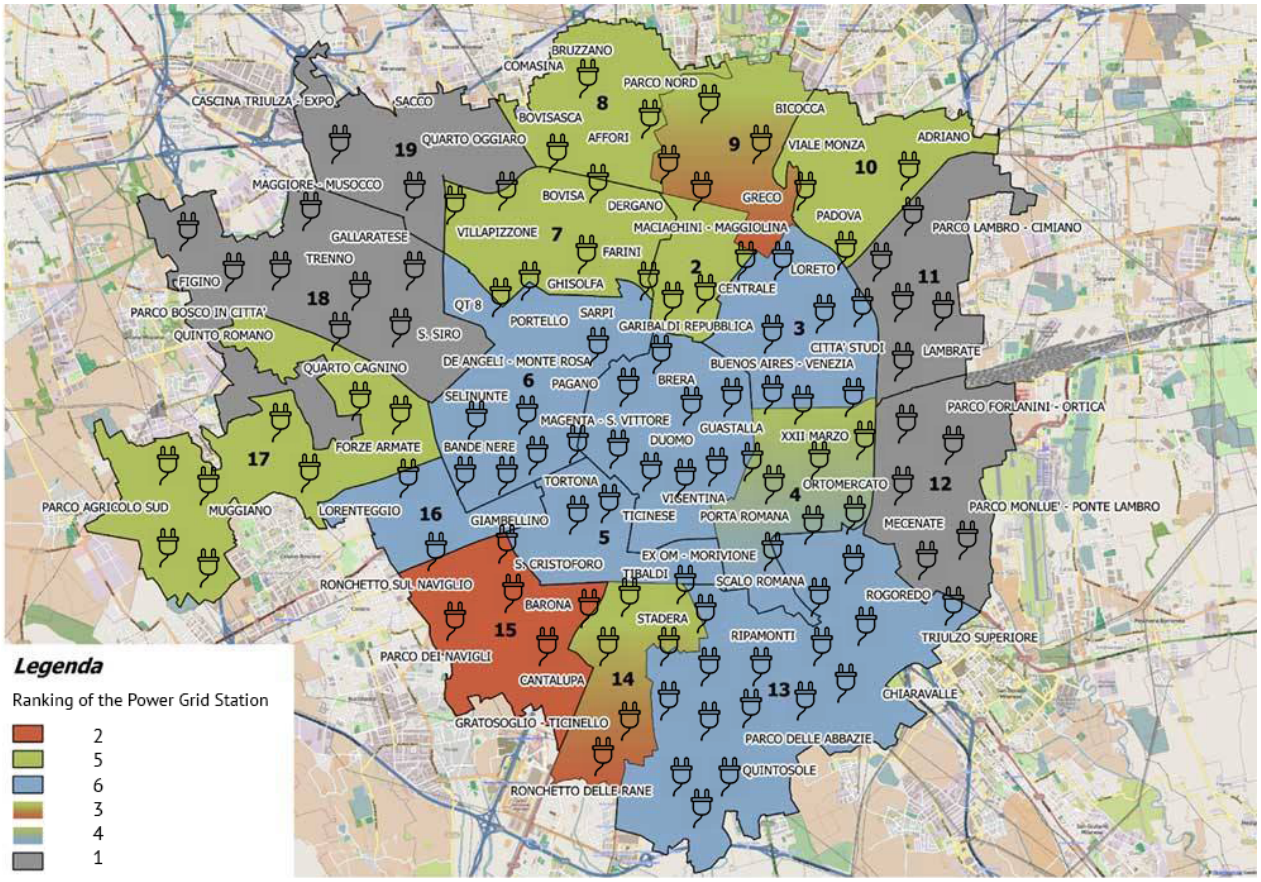

Figure 13: Ranking of the Power Grid Stations in Milan

Then we defined a customized radius of 500 - 1000 meters for each power grid station and we covered the whole map of Milan by drawing one circumference for each one of them, according to their given radius, as showed in figure.

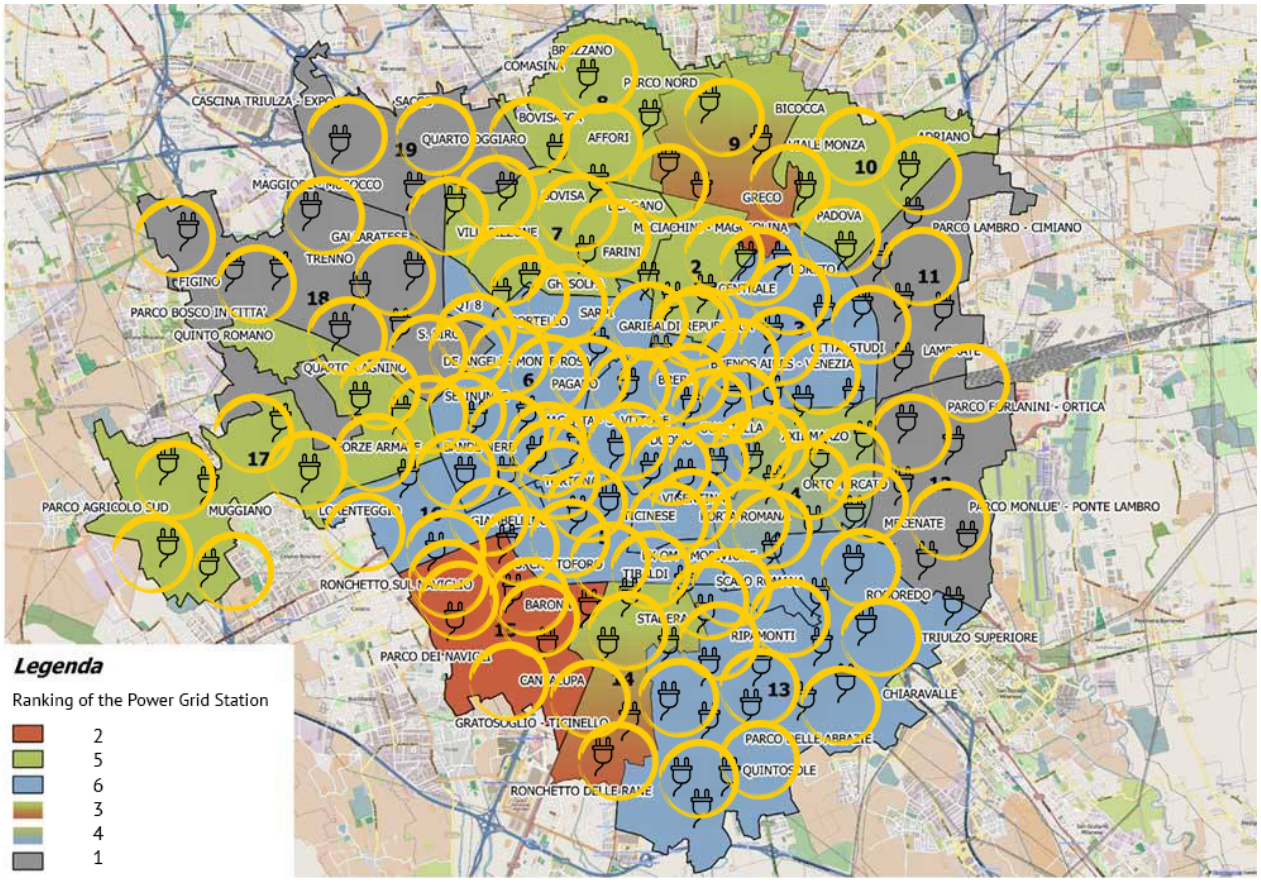

Figure 14: Radius of the Power Grid Stations in Milan

In this way, when a user with the enabled **Money Saving Option** would input his/her final destination at the beginning of a rent, the system would select a special parking area among all the ones present in the city by following these steps:

1. The system will retrieve the latitude and longitude of the final destination of the user;
2. The system will list all the circumferences (with the relative power grid station) that contains that specific point;
3. If there is only one station (distant in the worst case more or less 500-1000 meters from the final destination of the user), the system will select it and will provide information about it to the user;
4. If there isn't any station with free power plugs, a message of error will be showed to the user;
5. If there is more than one station that contains that specific point, the system will select the one with the maximum score according to this formula:

$$score = freeSlot * [(ranking - 2) + (10 - distance/100) * 2]$$

where:

- **freeSlot** is the number of available power plugs;
- **ranking** is the number from 1 to 6 assigned to the zone in which is contained the selected power grid station;
- **distance** is the quantity of meters that separates the final destination of the user from the selected power grid station.

1. Example n°1:

   The final destination of the user is in piazza delle Meraviglie, 34. At this address correspond the point with latitude x and longitude y, and there are 4 power grid stations that contain it. The power grid stations are characterized by the following information:

   (a) 4 freeSlot, ranking 3, distance from the final destination of the user 500m;
   (b) 3 freeSlot, ranking 4, distance from the final destination of the user 350m;
   (c) 1 freeSlot, ranking 5, distance from the final destination of the user 650m;
   (d) 2 freeSlot, ranking 3, distance from the final destination of the user 250m;

   By applying the formula written above, we would obtain the following score:

   (a) 44
   (b) 45
   (c) 10
   (d) 32

   The system will select the power grid station n°2 and it will provide information to the user about to get there.

2. Example n°2:

   The final destination of the user is in piazza Trifoglio, 12. At this address correspond the point with latitude z and longitude w, and there are 2 power grid stations that contain it. The power grid stations are characterized by the following information:

   (a) 0 freeSlot, ranking 5, distance from the final destination of the user 100m;
   (b) 2 freeSlot, ranking 4, distance from the final destination of the user 450m;

   By applying the formula written above, we would obtain the following score:

   (a) 0
   (b) 26

   The system will select the power grid station n°2 and it will provide information to the user about to get there.

In this way we would achieve our goal to distribute cars in the smartest way we were able to think about.

3.2 How to apply the Reservation fees

In this section we will analyze the structure of the code that will handle the payment for the reservation fees.

applyReservationFees(Reservation)

```
/*  This function will apply the fees for each Reservation instance. Actually, the user can
    terminate a reservation for free, but in the future if PowerEnjoy would decide to apply a fee
    for the termination of a reservation, it would be very easy to modify this value. If the user
    picks up the car, his reservation terminates and the system will create another object Rent:
    no fees will be applied to transform a reservation into a rent. On the other hand, if the
    reservation expires before the user picks up the car, the system will apply to him a fee of
    one euro. As said before, it would not be difficult in the future also to change this amount
    of money.
*/

function applyReservationFees (Reservation res){
   if (res.isExpired()){
      //  This function will return a 'true' value if the reservation is expired because of the
          time.
      paymentHandler(res.User, res.expirationFee);
      //  This function will send a request of payment to the external payment service with as
          input the ID of the user and the amount of money.
   }
   else {
      paymentHandler(res.User, 0);
      //  This case covers both the cases in which the user decides to terminate the reservation
          and in which he unlocks the car's doors.
   }
```

3.3 How to calculate the Rent fees

In this section we will analyze the structure of the code that will handle the payment for the reservation fees.

calculateRentDiscount(Rent)

```
/*  This function will calculate the eventual discount for each rent instance. */

void function calculateRentDiscount(Rent ren){

     if (ren.atLeastTwoPassengers()){
     //  This function will return a 'true' value if almost two passengers were in the car.
     ren.totalCost *= 0.9;
     //  In this way we will apply a 10% of discount to the client.
  }

  if (ren.report.batteryLevelAtTheEndOfRent >= 0.5){
     //  This function will return the percentage of battery at the end of the rental.
     ren.totalCost *= 0.8;
     //  In this way we will apply a 20% of discount to the client.
  }

  if (ren.report.MSOactived() && ren.report.finalDestinationNotEmpty()){
     //  The first function will return a 'true' value if the user enabled the Money Saving Option
         at the beginning of his/her rent. The second funtion will return a 'true' value if the
         user input his/her final destination.
     if (ren.carParkedAt(ren.report.specialArea) && ren.carLeftUnderCharge()){
     //  The first function will return a 'true' value if the user parked the car in the special
         area the system provided at the beginning of the rent. The second function will return a
         'true' value if the car is left under charge.
        ren.totalCost *= 0.6;
     //  In this way we will apply a 40% of discount to the client.
     else if (ren.carLeftUnderCharge()){
     //  This function will return a 'true' value if the car is left under charge in another
         recharging area (not the special one provided by the system at the beginning of the rent).
        ren.totalCost *= 0.7;
     //  In this way we will apply a 30% of discount to the client.
     }
  }
  else{
     if (ren.carLeftUnderCharge()){
     //  This function will return 'true' if the car is left under charge in a recharging area.
        ren.totalCost *= 0.7;
     //  In this way we will apply a 30% of discount to the client.
     }
  }
}
```

calculateAdditionalFees(Rent)

```
/*  This function will calculate the additional fees for each rent instance. */

void function calculateAdditionalFees(Rent ren){
   if (ren.report.batteryLevelAtTheEndOfRent <= 0.2 ||
      ren.report.distanceFromTheNearestPowerGridStationInKm >= 3){
      //  The first value is the same explained above
      //  The second one is the distance from the nearest power grid station expressed in km
      ren.totalCost *= 1.3;
      //  In this way we will apply a 30% of additional fee to the client.
   }
}
```

calculateRentFees(Rent)

```
/*  This function will calculate the fees for each Rent instance. In this case we have to consider
    eventual discounts and other additional fees before sending the payment request to the
    external payment service. It would not be difficult in the future to change some parameters
    according to the PowerEnjoy policy.
*/

function calculateRentFees (Rent ren){
   ren.totalCost = ren.report.totalTime * ren.report.pricePerMinute;
   // We will first calculate the Total Cost as the multiplication of the Total Time of the Rental
       and the Actual Price Per Minute. In the future it would be easy to modify them, according
       to the PowerEnjoy policy.

   calculateRentDiscount(ren);
   // In this way we will apply all the discount.

   calculateAdditionalFees(ren);
   // In this way we will apply all the additional fees.

   paymentHandler(ren.User, ren.totalCost);
   //  This function will send a request of payment to the external payment service with as input
       the ID of the user and the amount of money .

}
```

4 User Interface Design

In this section we will provide an overview on how the user interfaces of our system will look like.

4.1 UX Diagram

The UX Diagram shows the interactions between the different screens of the User Interface of the clients. The services are both accessible via web and mobile application.

Figure 15: UX Diagram

4.2 User UI

The following mockups represent the user UI, regarding both the Web Browser and the Mobile Application interfaces.

4.2.1 Home Page

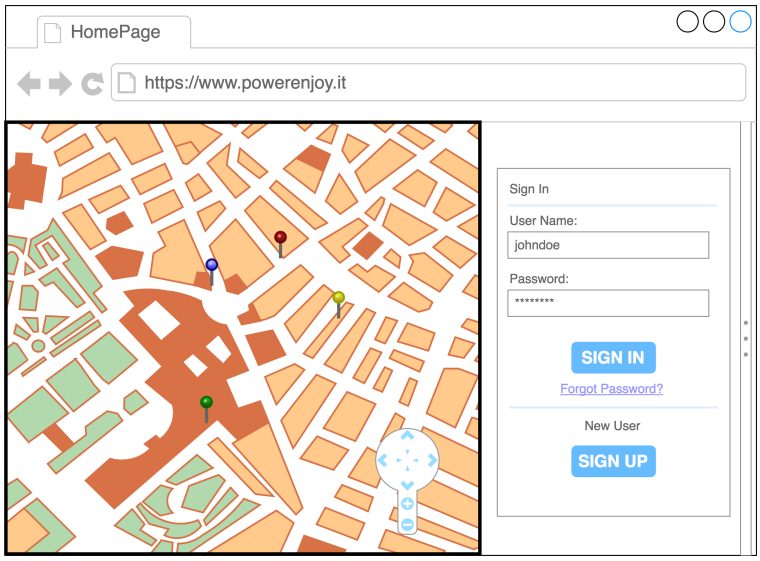

Figure 16: Home Page - Web Browser

4.2.2 Login

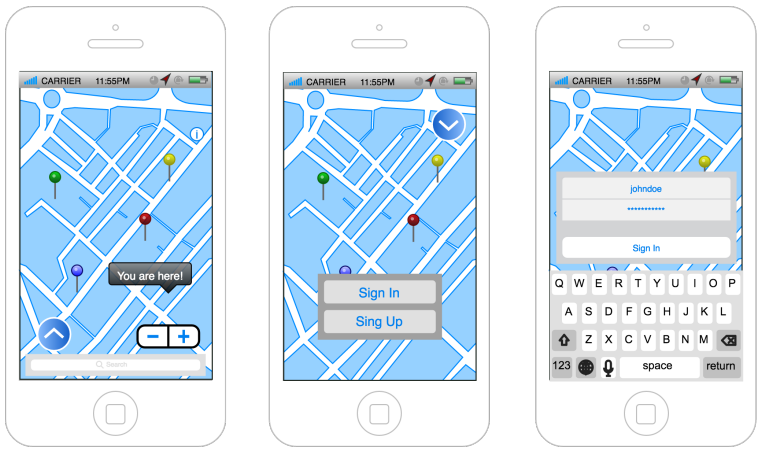

Figure 17: Login - Mobile Application

4.2.3 Registration

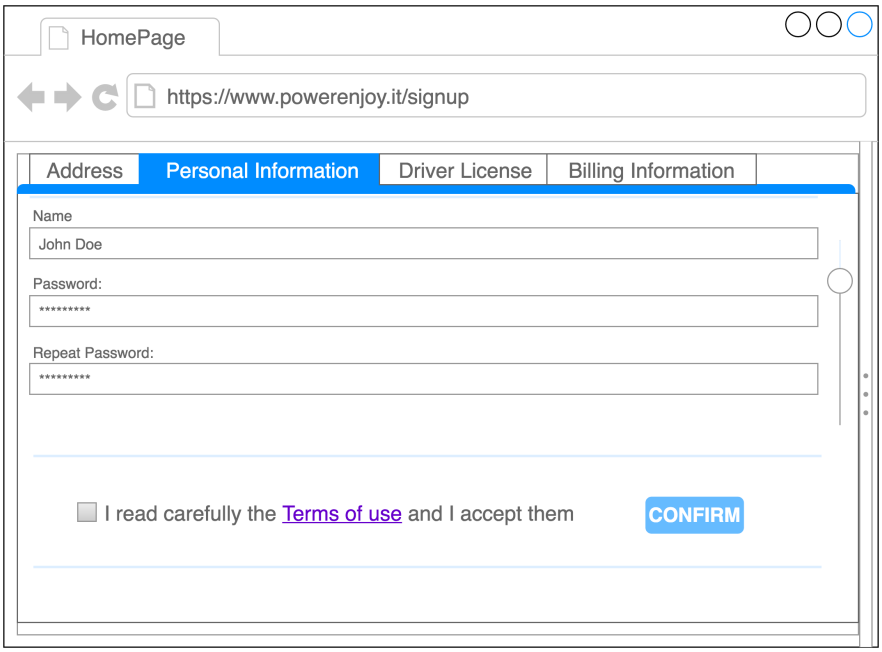

Figure 18: Registration Page - Web Browser

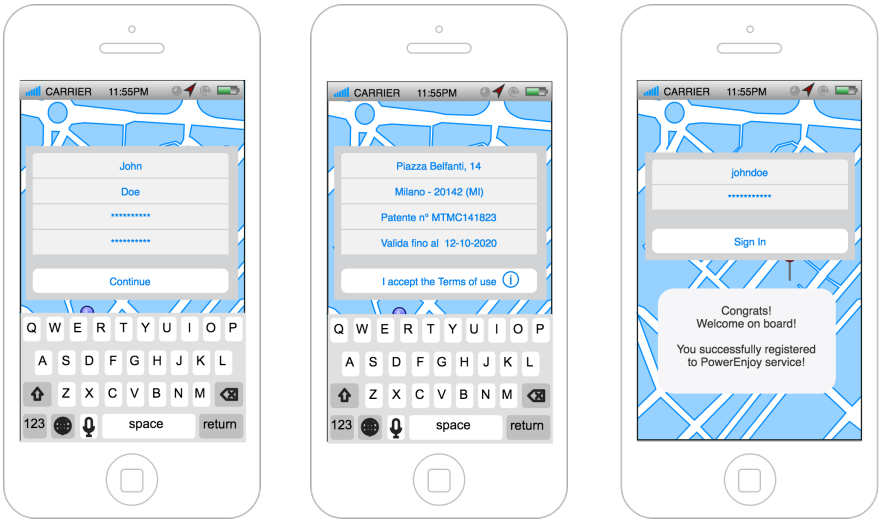

Figure 19: Registration Page - Mobile Application

4.2.4 Edit Personal Information

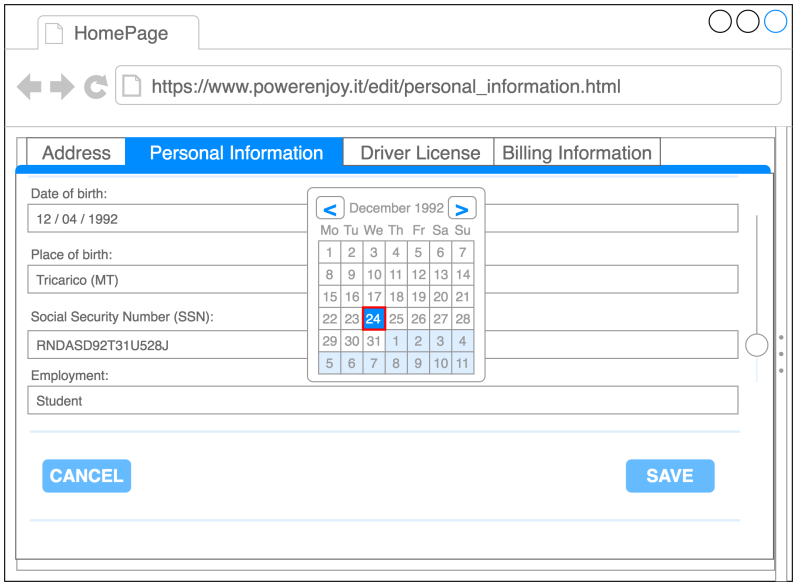

Figure 20: Edit Personal Information - Web Browser

4.2.5 Edit Billing Information

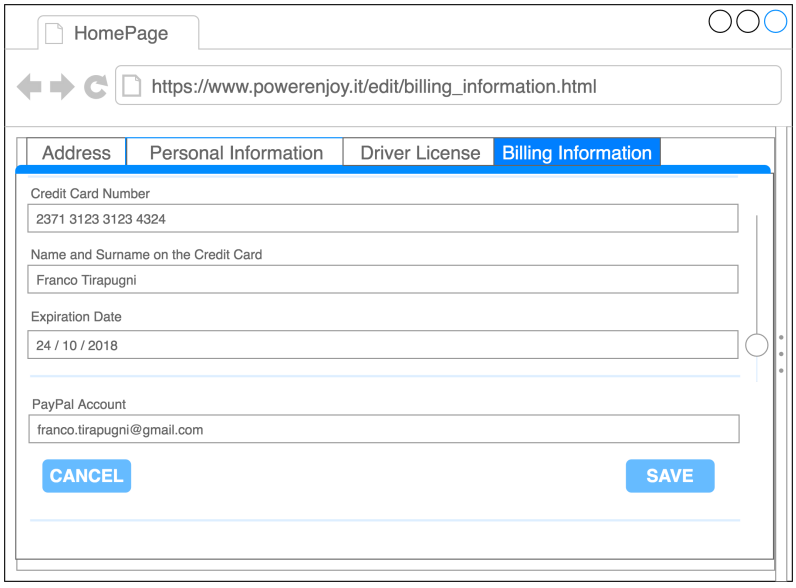

Figure 21: Edit Billing Information - Web Browser

4.2.6 Enable / Disable the Money Saving Option

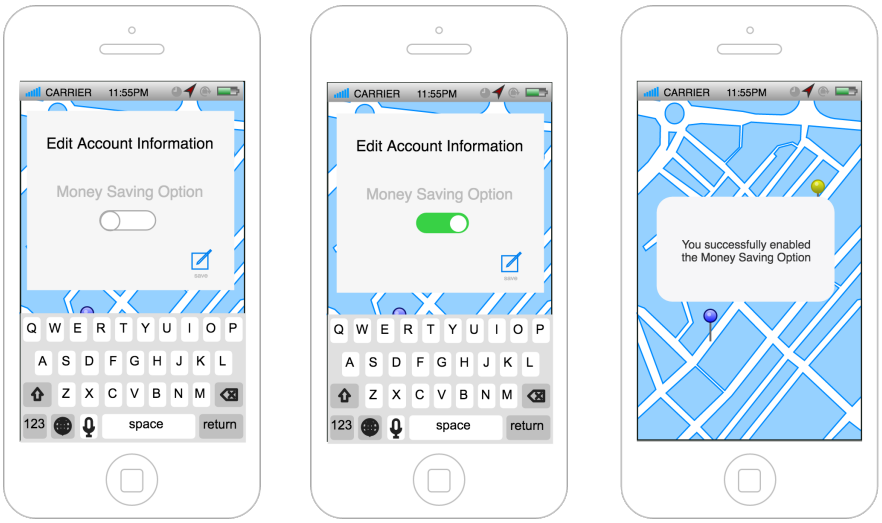

Figure 22: Enable Money Saving Option - Mobile Application

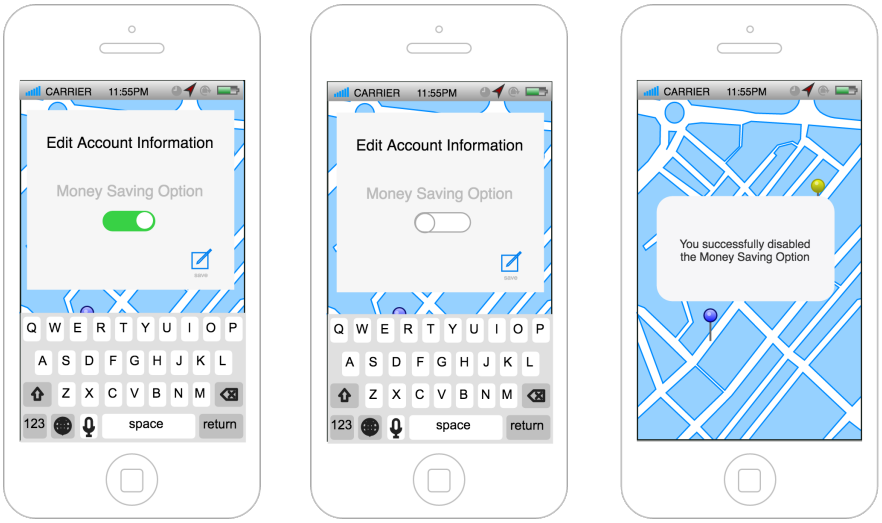

Figure 23: Disable Money Saving Option - Mobile Application

4.2.7 Start Reservation

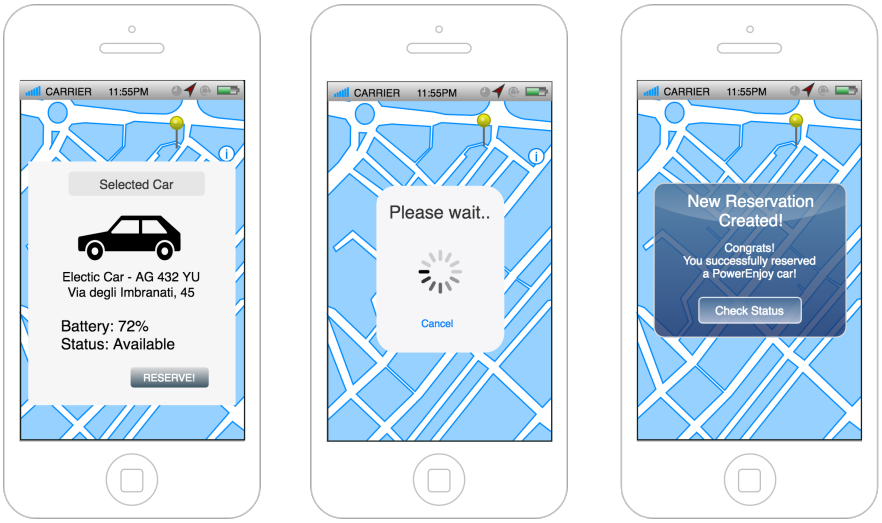

Figure 24: Start Reservation - Mobile Application

4.2.8 Terminate Reservation

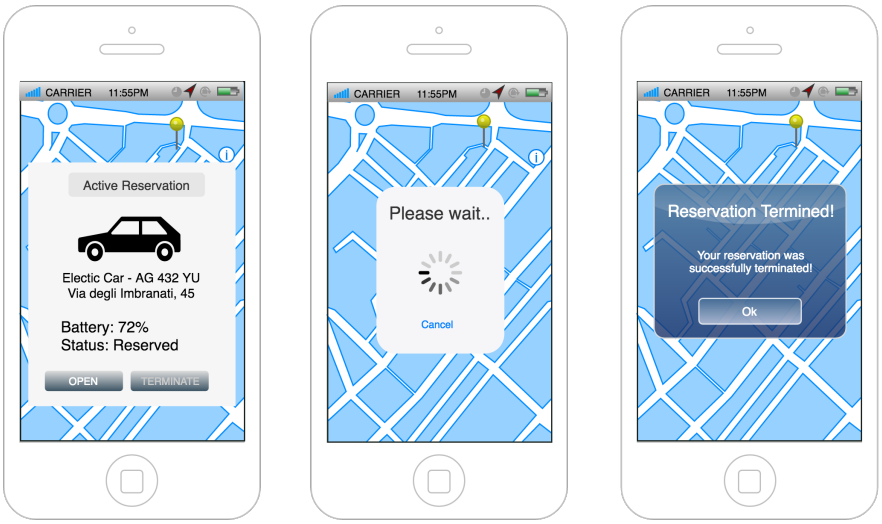

Figure 25: Terminate Reservation - Mobile Application

4.2.9 Unlock the car

Figure 26: Unlock Car's Doors - Mobile Application

4.2.10 Lock the car

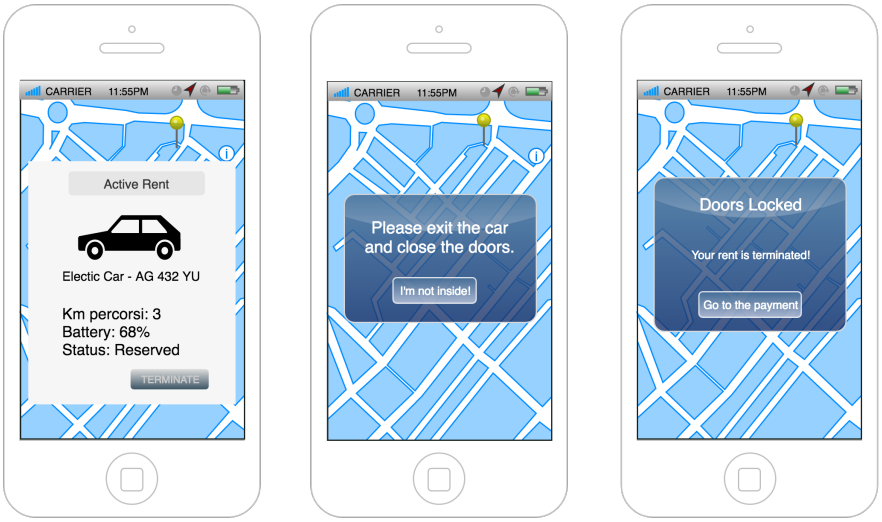

Figure 27: Lock Car's Doors - Mobile Application

5 Requirements Traceability

In the RASD document we have defined all the system's requirements: in this section we explaine how they map into the design elements defined in this document.

Component (DD)	**Requirements (RASD)**
Authentication Manager	4.1.1 Registration of a guest to the system 4.1.2 Login of a user into the system
AccountInformation Manager	4.1.9 Modify the profile information
CheckAvailability Manager	4.1.3 Find the location of available cars in a specified area
Reservation Manager	4.1.4 Book a car with the possibility to cancel the reservation 4.1.6 Be notified of active reservation status 4.1.7 End the rental
Car Manager	4.1.3 Available cars in a specified area
ADS_Application Manager	4.1.4 Book a car with the possibility to cancel the reservation 4.1.5 Open his/her reserved car 4.1.6 Active reservation status
Payment Manager	4.1.8 Know the total cost at the end of the rental
Data Manager	4.1.1 Registration of a guest to the system 4.1.2 Login of a user into the system
Notification Manager	4.1.6 Be notified of active reservation status

6 Appendix

6.1 Tools used

We used the following tools to produce this document:

- **LaTex** as typesetting system to write this document
- **LyX** as editor
- **Visio Professional** and **draw.io** to draw all the diagrams

6.2 Hours of work

Project Hours

Data	Colaci	De Pasquale	Rinaldi
21/11/2016	3	3	3
24/11/2016	2,5	2,5	2,5
25/11/2016	4	4	4
27/11/2016	3,5	3,5	3,5
30/11/2016	2,5	2,5	2,5
04/12/2016	5	5	5
05/12/2016	6	6	6
07/12/2016	5,5	5,5	5,5
09/12/2016	4	4	4
10/12/2016	5,5	5,5	5,5
11/12/2016	6	6	6
TOT:	47,5	47,5	47,5