Power EnJoy
Design Document

Niccolo' Raspa, Matteo Marinelli

December 9, 2016

Software Engineering 2 Course Project

Contents

1 Introduction

1.1 Purpose

This is the Design Document for the Power Enjoy Service. It's aim is to provide a functional description of the main architectural components, their interfaces and their interactions, together with the algorithms to implement and the User Interface Design. Using UML standards, this document will show the structure of the system and the relationships between the modules. This document is written for project managers, developers, testers and Quality Assurance. It can be used for a structural overview to help maintenance and further development.

1.2 Scope

PowerEnjoy is a digital management system for a car-sharing service that exclusively employs electric cars. It allows registered clients (Power Users) to use a vehicle paying only on the basis of the actual use during each individual rental. For a more detail description of the domain and the requirements please refer to the Requirement and Specification Document.

The software system is divided into four layers, which will be presented in the document. The architecture has to be easily extensible and maintainable in order to provide new functionalities. Every component must be conveniently thin and must encapsulate a single functionality (high cohesion). The dependency between components has to be unidirectional and coupling must be avoided in order to increase the reusability of the modules.

Futhermore, to increase cohesion and decoupling as much individual components must not include too many unrelated functionalities and reduce interdependencies.

1.3 Definitions, Acronyms, Abbreviation

RASD: Requirements Analysis and Specification Document.

DD: Design Document.

DBMS: Relational Data Base Management System.

DB: Database layer,

UI: User Interface.

Backend: Term used to identify the Application server.

Frontend: The components which use the application server services (web frontend and the mobile applications).

SOA: Service Oriented Architecture.

JDBC: Java DataBase Connectivity.

JPA: Java Persistence API.

EJB: Enterprise JavaBean.

ACID: Atomicity, Consistency, Integrity and Durability.

1.4 Reference Documents

This document refers to the following documents:

- Project rules of the Software Engineering 2 project
- Requirement Analysis and Specification Document (from the previous delivery)

1.5 Document Structure

This document is structured in five parts:

Chapter 1: Introduction. This section provides general information about the DD document and the system to be developed.

Chapter 2: Architectural Design. This section shows the main components of the systems with their subcomponents and their relationships, along with their static and dynamic design. This section will also focus on design choices, styles, patterns and paradigms.

Chapter 3: Algorithm Design. This section will present and discuss the main algorithms for the core functions of the system, independently from their concrete implementation.

Chapter 4: User Interface Design. This section shows how the user interface will look like and behave, by means of concept graphics and UX modeling.

Chapter 5: Requirements Traceability This section shows how the requirements in the RASD are satisfied by the design choices, and which compontents will implement them.

2 Architectural Design

2.1 Overview

This chapter provides a comprehensive view over the system components, both at a physical and at a logical level. This description will follow a top-down approach, starting with the description of the high-level components and their relations and interactions. We will then reason and describe the single components and the functionalities they must implement. We will especially focus on the components that implement the core logic of our application and using sequence diagrams we will describe the runtime behaviour of the system.

We will also include deployment diagrams to show the physical implementation of the system.

2.2 High Level Components

Before describing the actual system architecture we introduce the high level components of our application. Following the requirements and the specification listed on the RASD, we identify what components are needed in order to implement them and only after we have detected them, we will focus on the architecture and explain our architectural decision and the technologies chosen. It's important to follow this process to generalize our design as much as possibile and abstract from implementation details, in this way we are able to describe only the essence of our system.

The starting point to detect the main components is the Class Diagram described in the RASD.

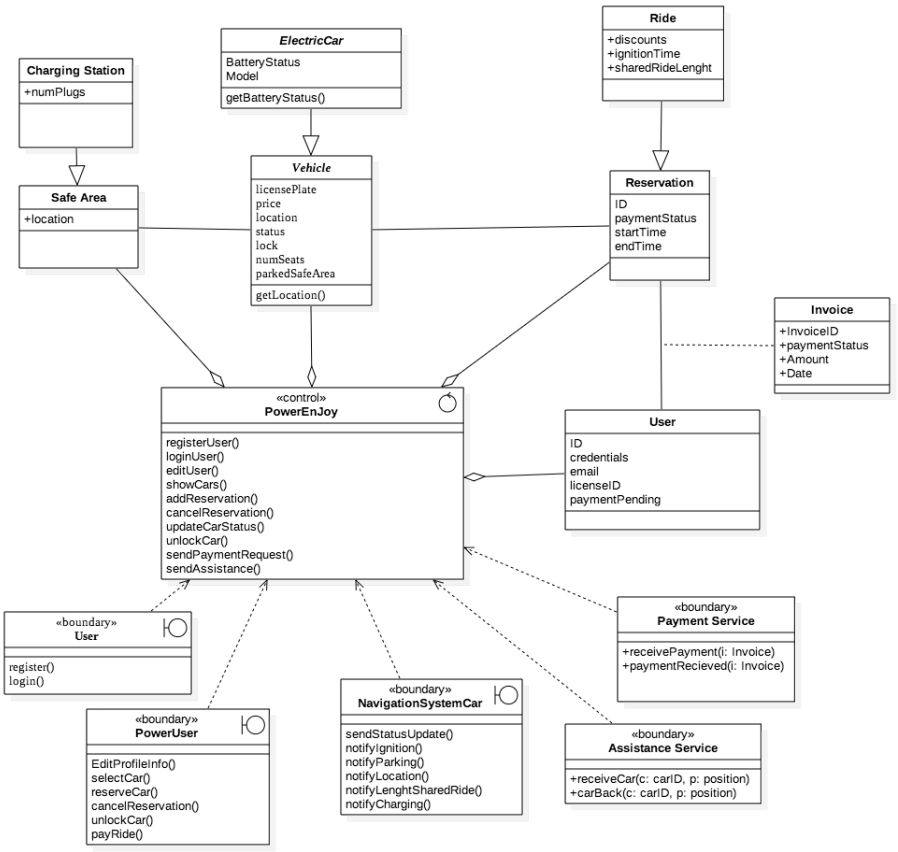


Figure 1: Class Diagram

From the diagram we can identify the following high level components:

Client and External Services The boundaries of the Class Diagram shows the normal users of the application (User and Power User) and the External Services who interacts with the control object to provide additional functionalities.

Application Server: The control object is implemented in this component. It contains all the logic for the system application. It will implement all the required functionalities and communicate both with the clients and the external services.

Database: This component is responsible for data storage and retrieval of all the entities rapresented in the top part of the diagram. It will not implement any logic but it stores all the information needed for the correct functioning of our service. It must guarantee ACID properties and be accessible from the Application Server.

In the figure below we rapresent all the components listed above in a layered fashion, highlighting the relations among the different parts:

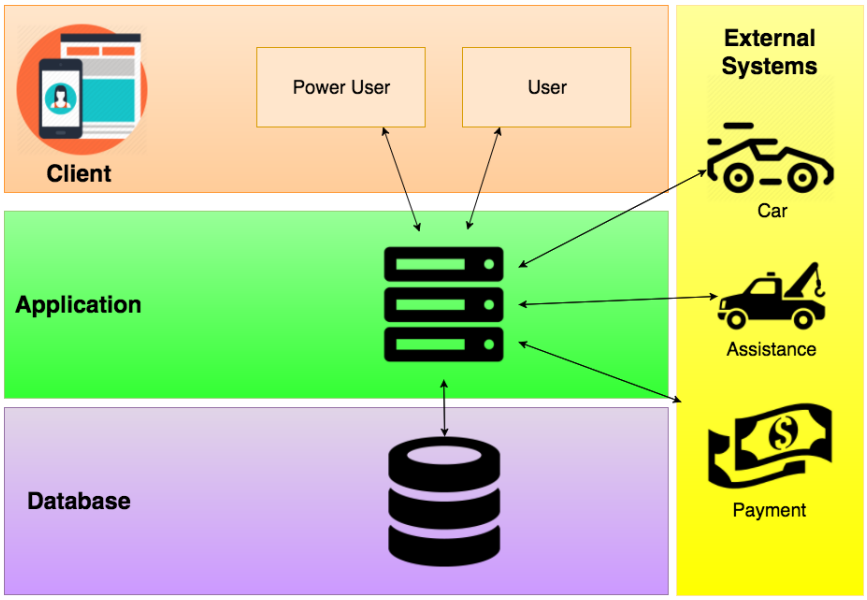


Figure 1: High Level Components

2.3 Component View

In this subsection we will look inside every single component and describe all the internal subcomponents. It's important to identify the relevant modules without increasing granularity to much. This will allow to have an efficient load balance in the present and it will be easy to integrate new functionalities in the future.

In a divide-and-conquer fashion for every component we will specify the implementation chosen and at the end we will how to connect the single components.

2.3.1 Application Server

This components implements the logic of the Power Enjoy Application, it's the core of our business and in this part of the document we'll explore the subcomponents inside. To provide a natural continuation from the RASD, we will start from the Class Diagram, and we will focus on the control object. We will look at the singles control functions, logically group them in cohesive groups and map them in modules of our system.

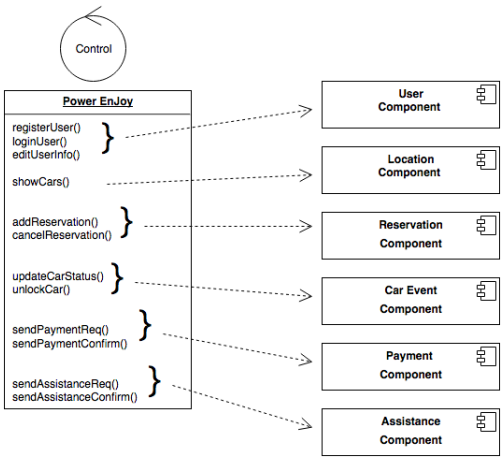


Figure 2: Mapping of control function to subcomponents of our system

A brief desription of the functionalities expected from each module:

User Component This module provides the logic for a User and a Power User regarding all the user management features, namely: user login, user registration, user deletion, user profile editing. It will perform all the validation of the credentials received before inserting them into the system.

Location Component This module handles the position of each car and user of the system. It's used to show cars on the map and to perform proximity checks to unlock vehicles. It also stores location of Safe Areas and Charging Stations.

Reservation Component This module is responsible to manage all the current reservation and to accept new reservations from Power Users. and for avoiding undesiderable behaviours (reservation of an unavailable car, double-booking and multiple reservations). It's strongly connected to the Car Event Component. This module keeps track of all the ride informations.

Car Event Component This modules interfaces with the API of the car on board system. It's a "low level" component that collects all the car data for other components to use and can be used to remotly control the vehicle. It will signal to the interested component all the events of the car (car locking/unlocking, motor ignition, malfunctioning).

Payment Component This modules interfaces with the API of the Payment Service to request payments and receive confirmations. It will not perform the fee calculation but it will receive the final price and check for price variations via other components It will also flag/unflag users as banned in case of Pending Payment.

Assistance Component This modules interfaces with the API of the Assistance Service to request assistance (recharge on site, fix malfunctions, bring car back from unsafe to safe areas) and receive confirmations once the assistance is provided. Once a car is fixed it will update the car informations (e.g. new position, new battery level).

Time Component This is an utility component, it will be responsible for timing features such as expiration of the reservation and unsafe park timing. This helps to avoid the necessity of introducing stateful components.

Implementation Choice

This component will be implemented using:

- Jave Enterprise Edition 7 (JEE7) - the platform incorporates a design based largely on modular components running on an application server which is a natural consequence from the description above. It also provides support for large-scale, multi-tiered, scalable, reliable, and secure network applications. This modularity helps to handle such complex system and it make easy to insert the functionalities in the future.
- Enterprise JavaBeans (EJB) to encapsulate all the business logic of the modules described above.
- GlassFish as the Application Server - the server provides services such as security, transaction support, load balancing and supports the JEE7 platform.

The next figure will show the Application Component implemented as session beans logically grouped in EJBContainers.

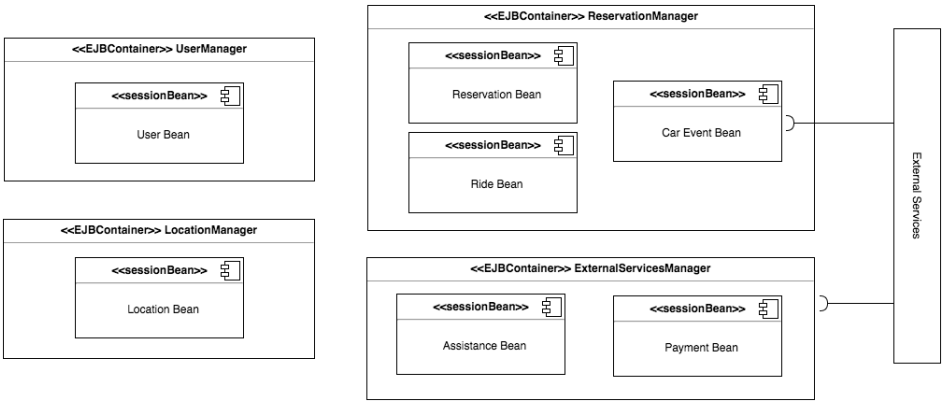


Figure 3: Application as Java Entity Beans

For the sake of clarity some information have been hidden from the diagram aboce but can be found in the next sections.

In Section 2.5 we will provide more details on the interfaces of each module.

In Section 2.7 using UML Sequence Diagrams we highlight the relationship between each component showing the runtime behaviour of our system.

2.3.2 Database

To design the DBMS is auspicable to start from the Class Diagram proposed in the RASD. The upper section of the diagram, Power Enjoy excluded, describe the information the system need to process to guarantee a correct and efficient service. Is clear that this set of informations must be persistent.

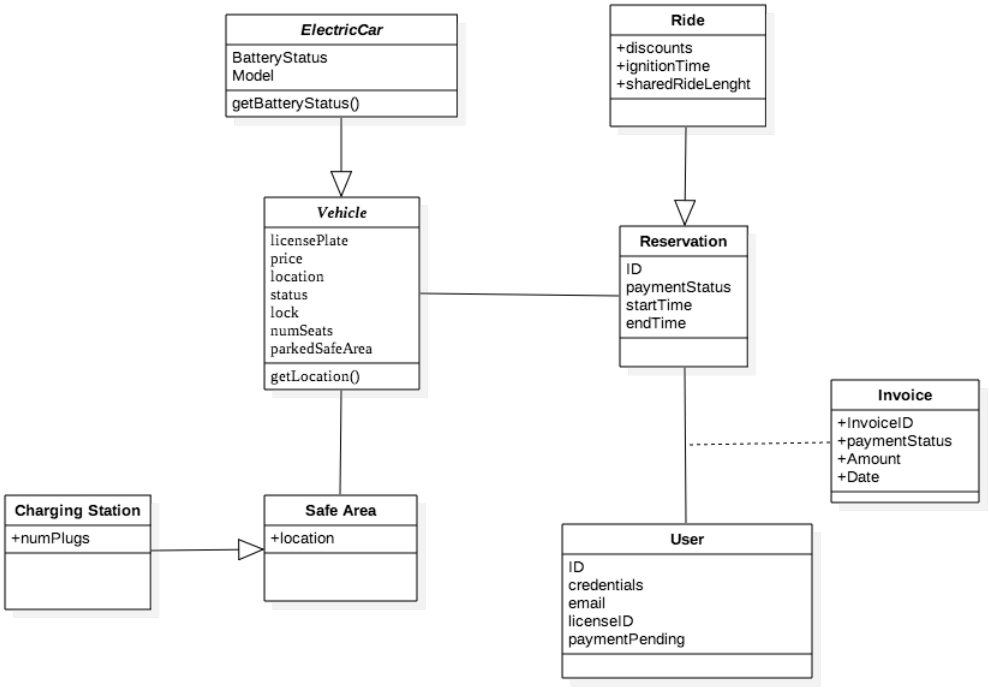


Figure X: Entities in the class Diagram

The DBMS must guarantee the correct functioning of concurrent transactions and the ACID properties; a relational DBMS is sufficient to handle the data storage required by the application. The database structure will be here described by a ER Diagram.

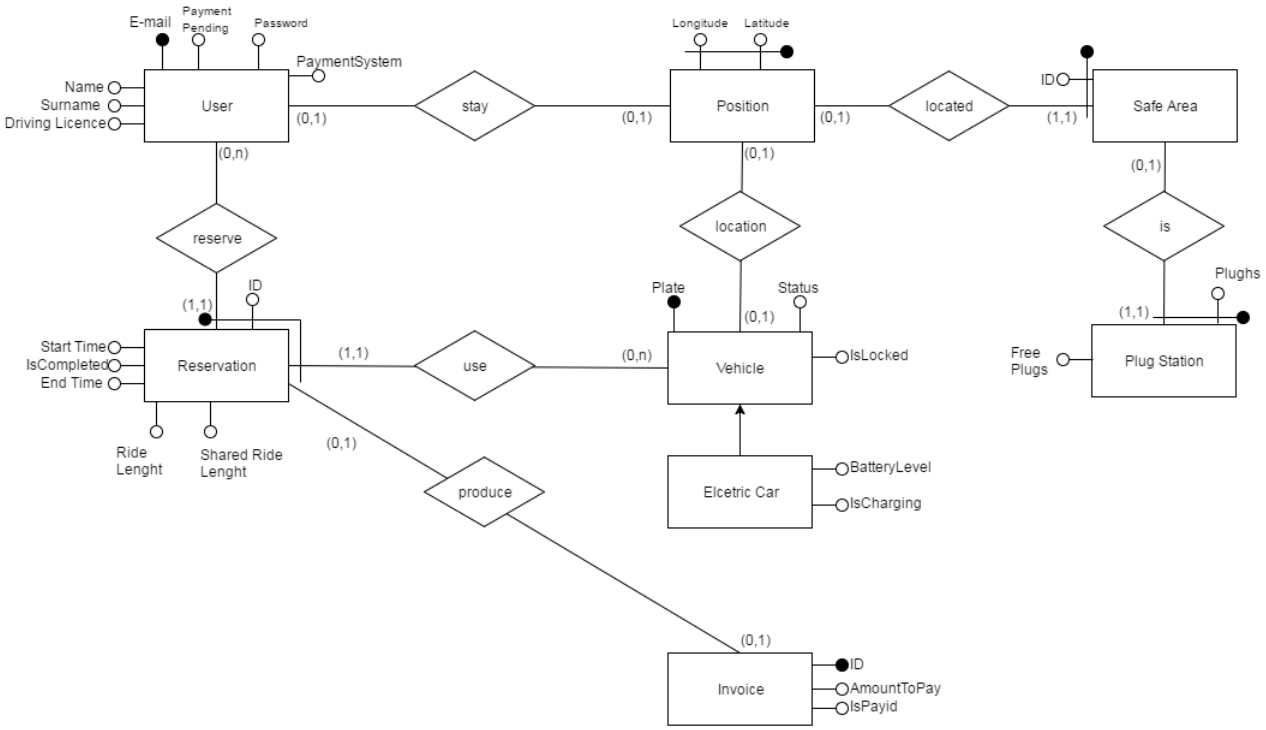


Figure X: Entities in the class Diagram

Class Diagram suggests eight main entities: User, Invoice, Vehicle, ElectricCar, Safe Area, Charging Station, Reservation, Ride. The division between Vehicle and ElectricCar was added to give the possibility to add, in the future, different types of Vehicles. As for the Application Server, the division of reservation and ride is redundant and only one entity will be created.

In addition to the class Diagram we've added the entity Position which stores the position of Users, Vehicles and Safe Areas.

Implementation Choice

This component will be implemented using:

- MySQL as the relational DBMS. It was chosen for its Scalability, Flexibility, High Performance and High Availability.
- MySQL also provides connectors and drivers (JDBC) that allow all forms of applications to make use of MySQL as a preferred data management server.
- The Java Persistence API (JPA) will be used inside the Application Server as an interface with the database to perform object-relation mapping and Database access.

2.3.3 External Services

In this section we focus on the components which are not part of our system but with which the system depends to implement some functionalities (please refer to the RASD for a more detailed description).

Car On Board System

Every Power Enjoy vehicle comes with a pre-installed embedded system which registers and notifies all the car activity. This components provides an API to monitor and remotly control every vehicle. This component implements some logic and it's connected over GSM to the Car Bean of the Application Server. The main functionalities that provides are the following:

1. Provides an interface to access every relevant information of the car (e.g. battery level, position)
2. Automatically signals relevant events to the Car Bean (e.g. engine ignition, parking, malfunctioning).
3. Automatically shows/updates the price on the screen.
4. Automatically locks the car when the car is parked with no people inside.
5. Calculates the lenght of every ride (from ignition to parking).
6. For every ride it calculates the lenght of a shared ride. Everytime at least 3 weight sensors in the car seats are on, the car increases this time counter

The communication will be implemented via RESTful APIs.

Assistance Service

Power Enjoy is in charge of the management of the car-sharing system. All the secondary functionalities (recharging vehicles onsite, bringing cars back from unsafe areas and fixing malfunctions) are handled by an assistance service. This components provides an API to handle all the assistance request.

The interaction between Application Server and Assistance Service will be bidirectional: the application server sends the information about a malfunctioning car and the type of assistance required and the Assistance service must notify the application server when the assistance is provided.

The communication will be implemented via RESTful APIs.

Payment Service

Every Power Enjoy user, in order to use our service, is required to have an account registered to a third party payment service, with a valid payment method. This components provides an API to handle all the payments functionalities. The interaction with such system is standard: the Payment Bean requests a payment specifying the fee and the user account, the Payment System will manage

the payment process (including debt collection in case of negligent users) and will notify the Payment Bean when the payment was executed. A user with a pending payment will be prevented to reserve another vehicle.

The communication will be implemented via RESTful APIs.

2.3.4 Client

In this subsection we'll focus the client component. Power Enjoy is a car sharing service therefore it must be implemented with mobility in mind. Since the majority of the mobile devices have a GPS module and we need to have access to the user position for our application, it makes sense to require that the main user has our mobile application installed.

Mobile application can handle all the functionalities required by the application server alone but to have a more efficient service, adding a Web Page seems to be an optimal strategy to provide more visibility and accessability. The Web Page become an optimized platform to manage all the side functionalities, in particular the profile informations management and all the payment informations.

This suggests a modification of the architecture, introducing a new tier with a Web Server Layer who is a bridge between Web Page and Application Server. Separating Application Server and Web Server improves scalability as we expect Power Enjoy usage to grow in differents regions and in this way we are able to separate task and optimize each layer individually to support increasing loads.

In conclusion, the subcomponents are the following:

Web Browser: Using a web browser the user is able to communicate with the Web Server to obtain the required web pages.

Web Server: This component does not contain any application logic, it's used to provide a web interface interface to the user. It helps to separate presentation from logic.

Mobile Application: This component is used by a Power User to use all the functionalities of the Power Enjoy Service. It's a really thin client that interacts with the Application Servier.

Implementation Choice Web Server

- WebServer runs on Glassfish with JavaServer Pages (JSP), this was done to provide consistency with the Application Component.
- The communication to the Application Server will be done using JAX-RS (in the Application side) to implement proper RESTful APIs.

Implementation Choice Mobile App

- The mobile app will be written with Cordova which is a mobile application development framework which is free, open source and it allows to target

multiple platforms with one code base (with access to Access native device APIs). This will allow to have reusable code across platforms abstracting from different platforms. It's the best cost-effective solution to target nearly every phone or tablet on the market today and publish to their app stores directly.

2.4 Component Interfaces

Application Server

QUESTA DA FARE ASSOLUTAMENTE, LE ALTRE FORSE NON SERVONO

2.5 Final System Architecture

Starting from the Class Diagram at the beginning of the design process we had a three tiers client-server based architecture, but in the process of defining the single complement we introduce a Web Tier. This process triggered more modification that led to the final architecture that will be presented in this section.

From a Three Tier to a Four Tier Architecture

In Section 2.3.4 we explain the necessity of introducing a Web Tier to improve scalability and to separete presentation from logic. The architecture will now include the following tiers:

Client Tier This layer contains the mobile app and the web browser

Web Tier This layer contains contains the Web Server to provide a web interface interface to the user

Application Tier This layer contains all the logic for the application

Database Tier This layer contains all the persistent data

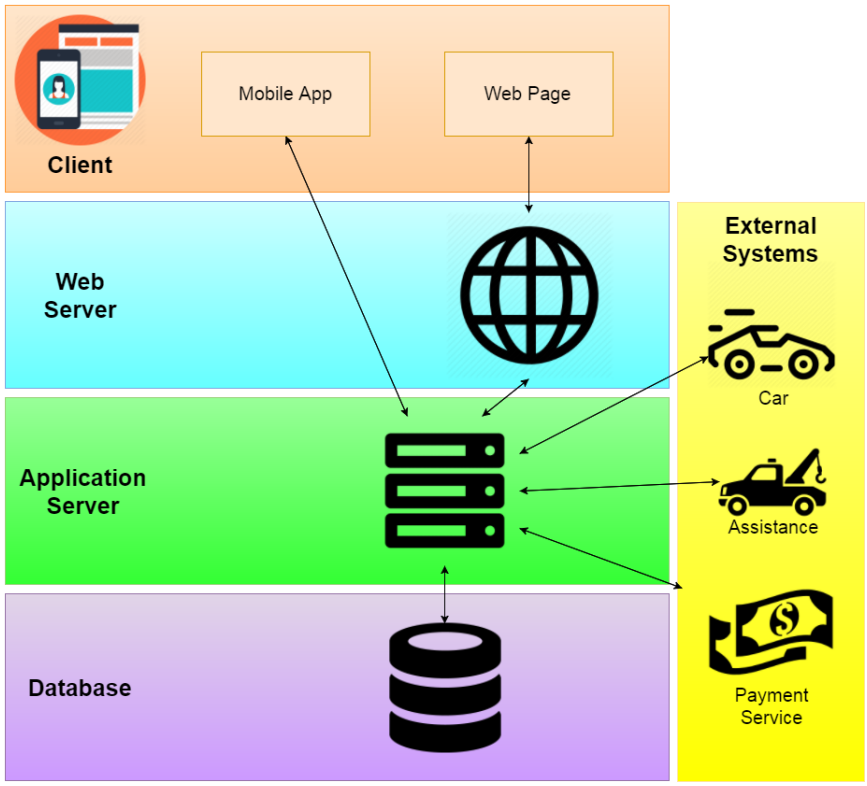


Figure 1: High Level Components

Moving to a Service Oriented Approach

The problem of our architecture is that the Application Server is the bottleneck of our system. Every other components is in relation with it, therefore it's performance is stricly related to the performance of our overall system. But since every components expects different functionalities from the Application Server we can parallelize using threads and split the work load among different services. To exploit the benefits of a service-oriented architecture we have to split the database among the different components.

The next figure shows which database table will each component manage.

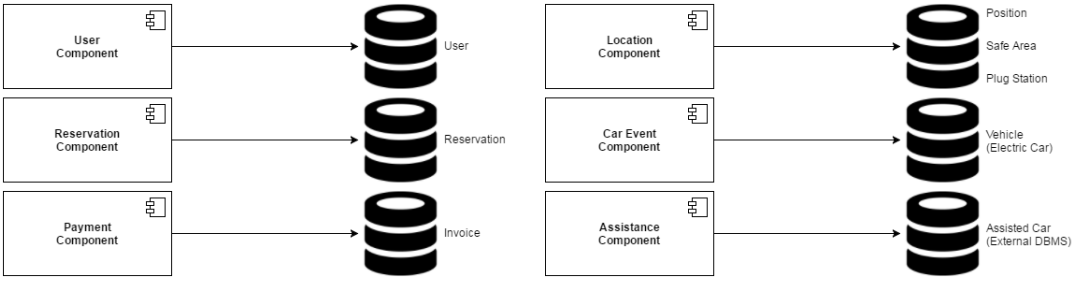


Figure 1: Application Layer with Service Oriented Approach

We believe this is a succesfull architecture for the following reasons:

- It's a more clean architecture. Every component implements a service and provides an interface to all the other services.
- Changing/optimizing each module will not affect the whole system as long as we mantain the same interface for each component.
- It's very flexibile, it's will be easy in the future to add new functionalities.
- We can divide the databases among different regions (e.g. for the city of Milan we don't need to keep track of the cars in Turin)

The next figure rapresent the whole system architecture:

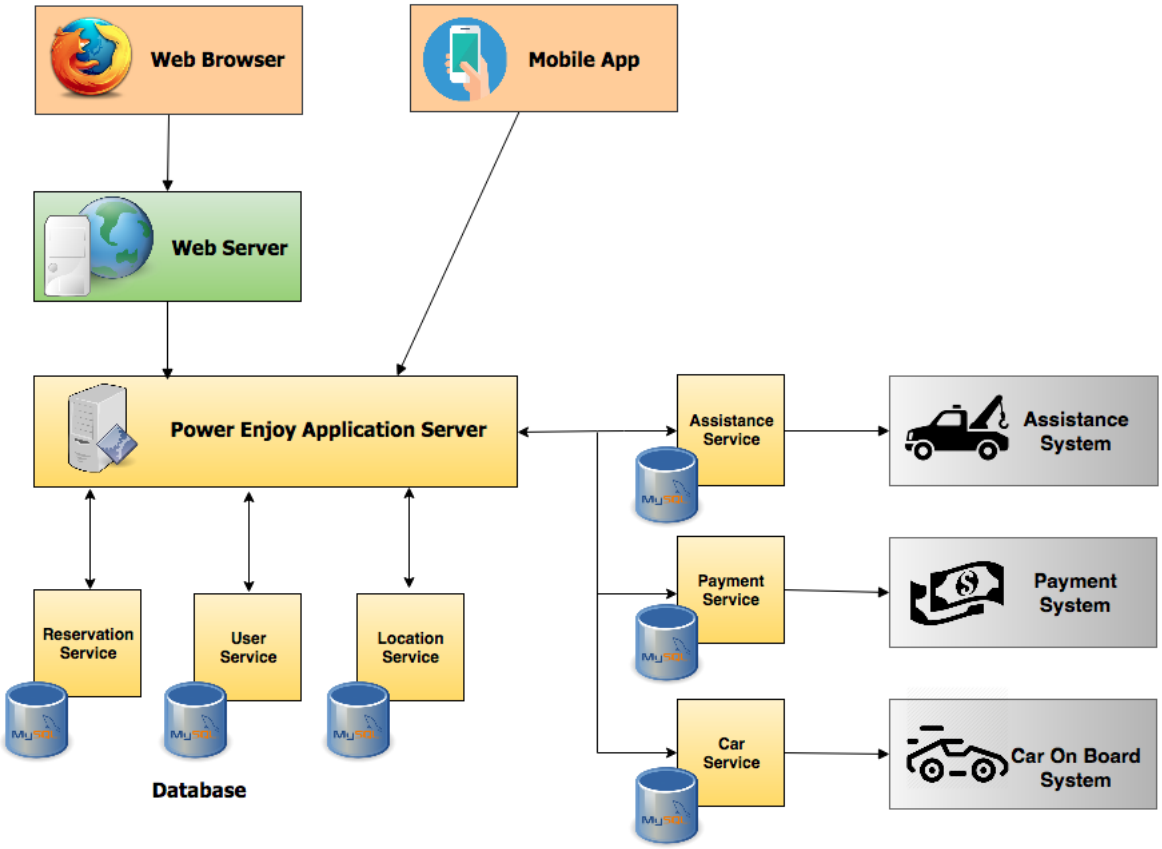
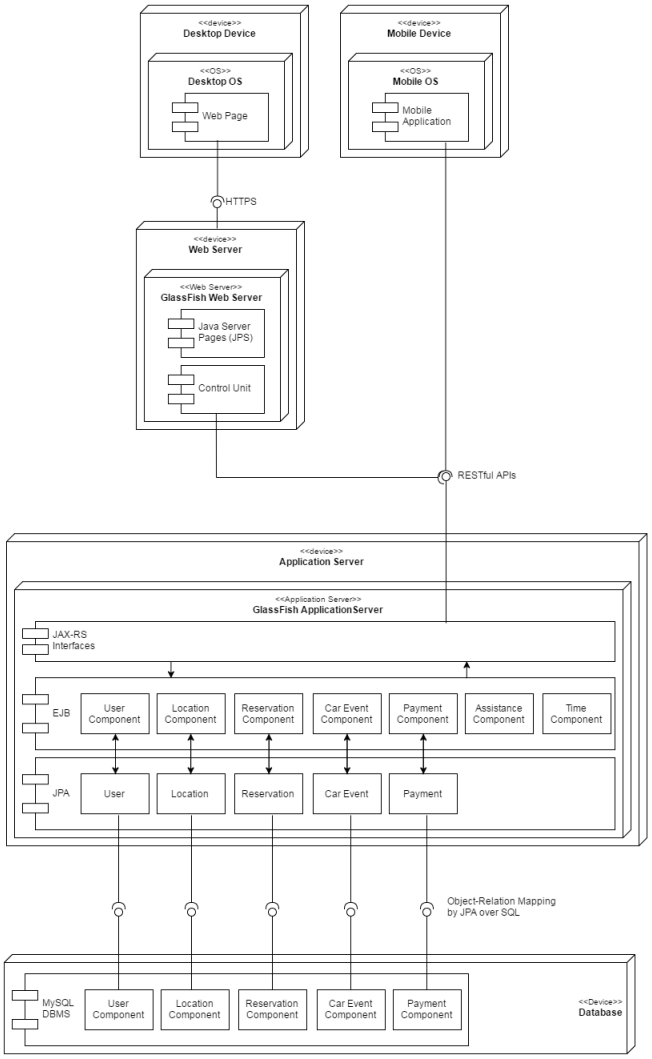


Figure 1: Application Layer with Service Oriented Approach

2.6 Deployment View

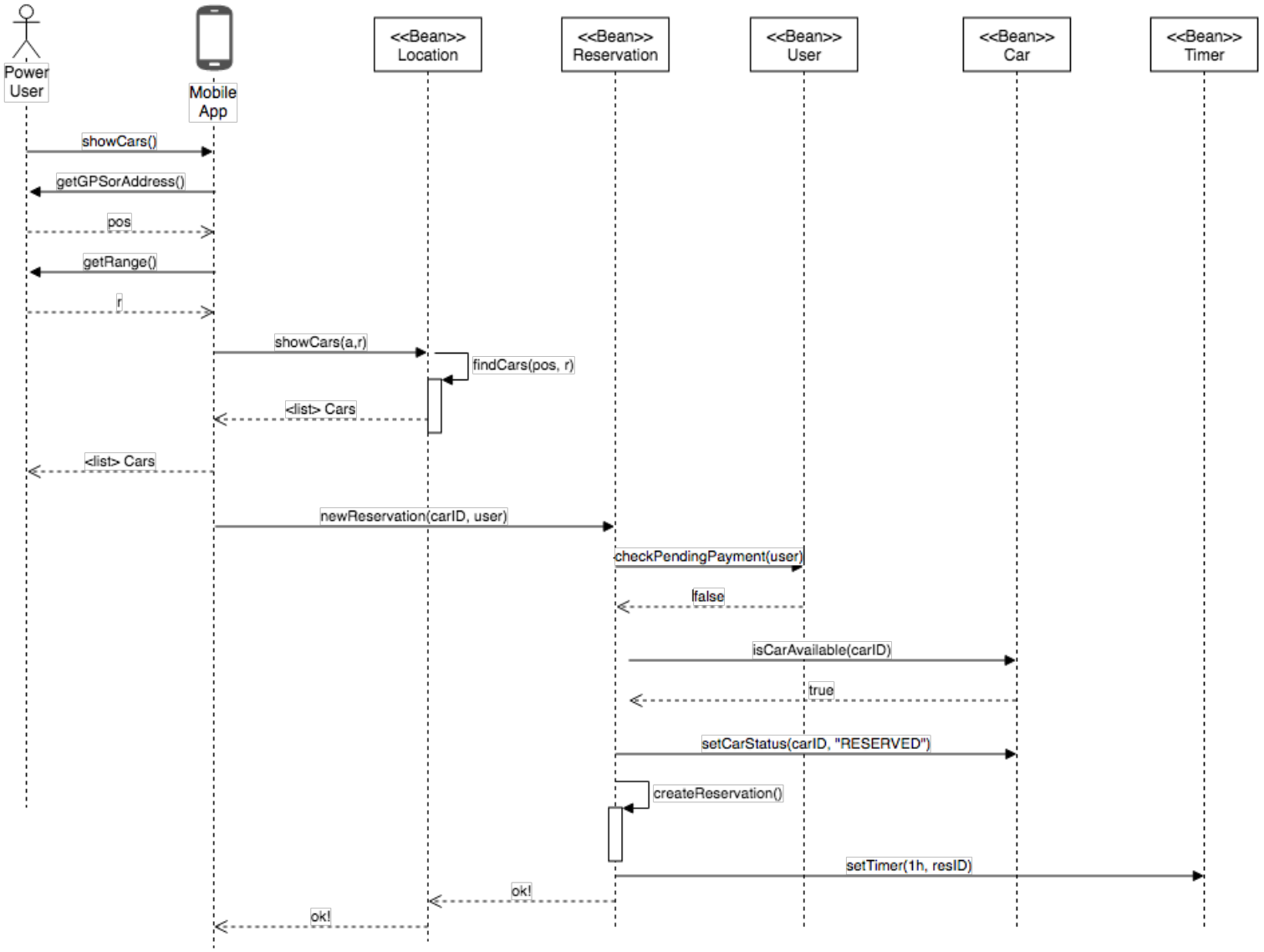
In this subsection we'll move on the physical side of our application, showing the deployment diagram for the whole system.



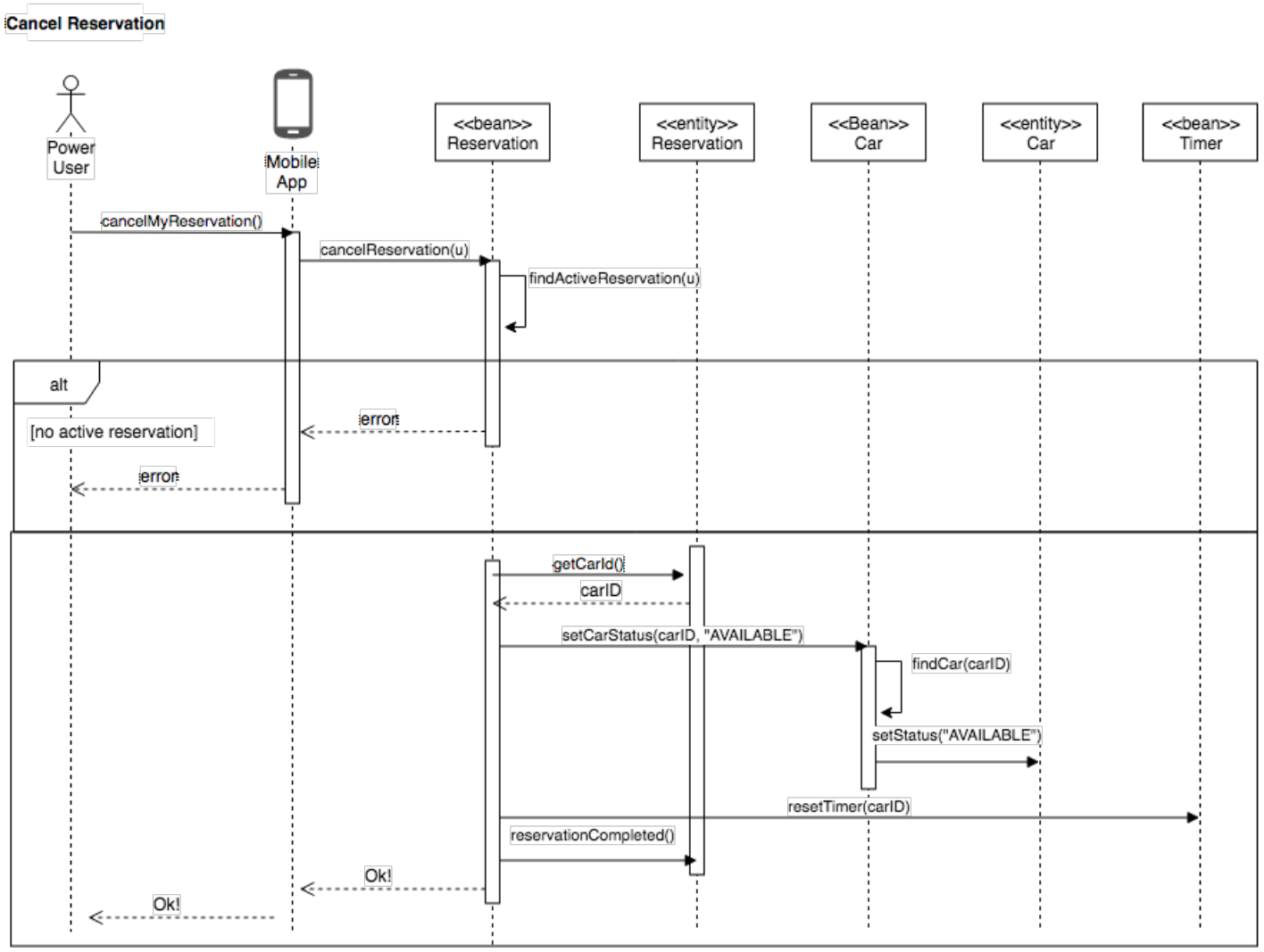
2.7 Runtime View

In this section we will describe the dynamic behaviour of the system. In particular, it will be shown how the software and logical components defined in section 2.3 interact one with another, using sequence diagrams for the more meaningful functionalities of the system. We decided not to represent the database in the sequence diagram, because the interaction with the database is totally abstracted by the entities via the Java Persistence API.

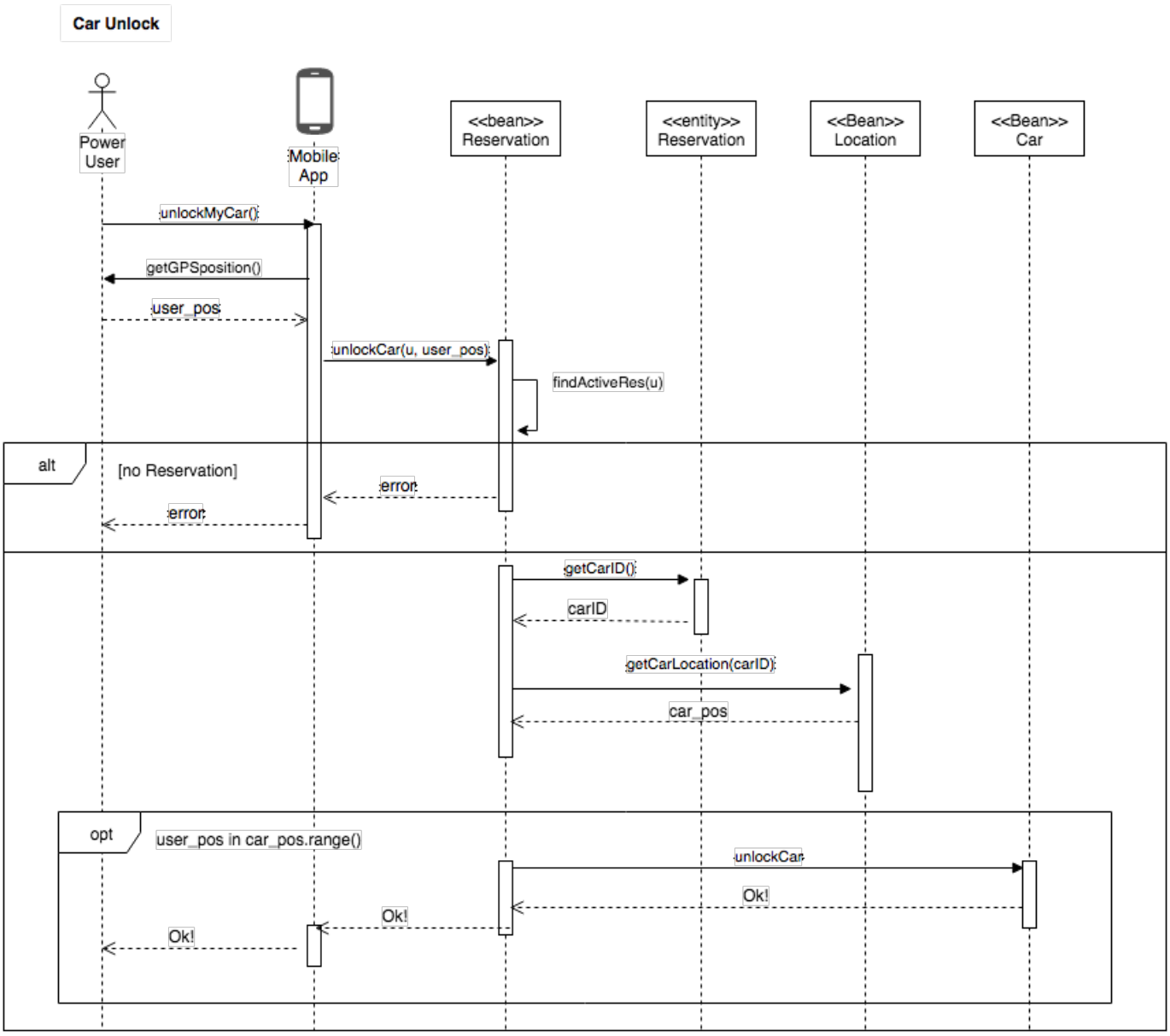
Create Reservation



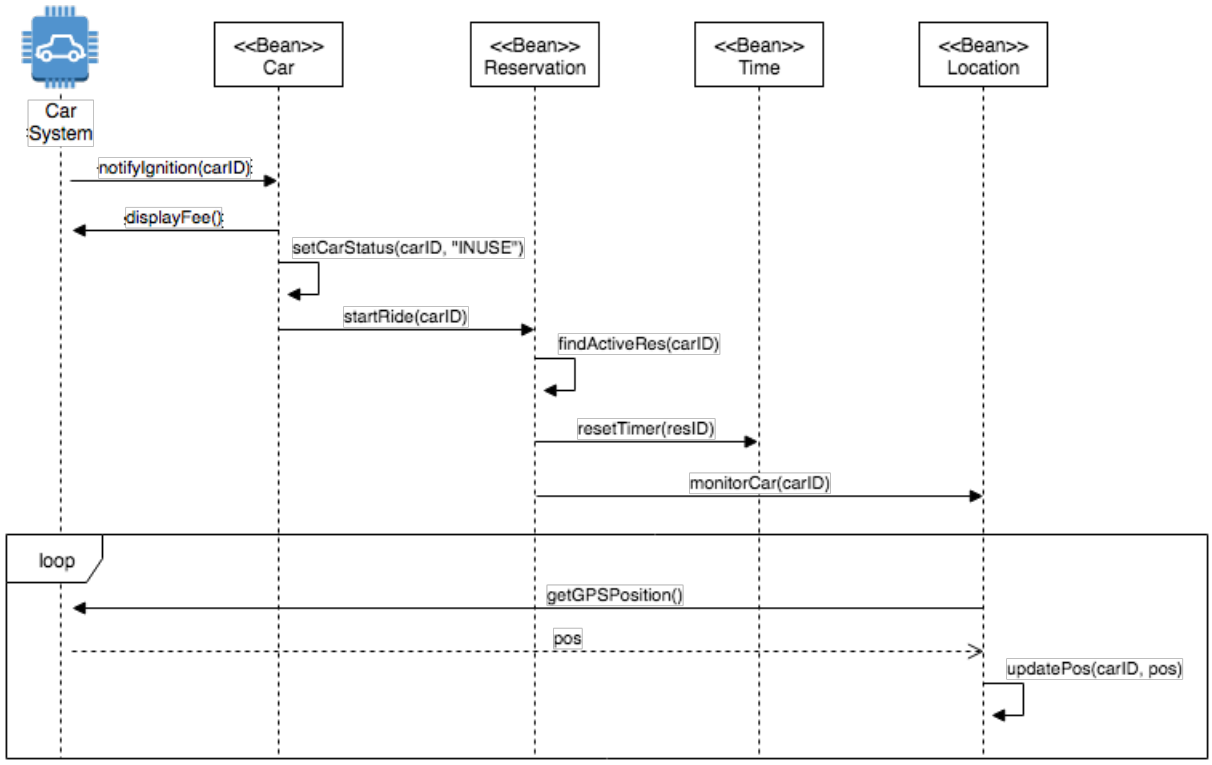
Cancel Reservation



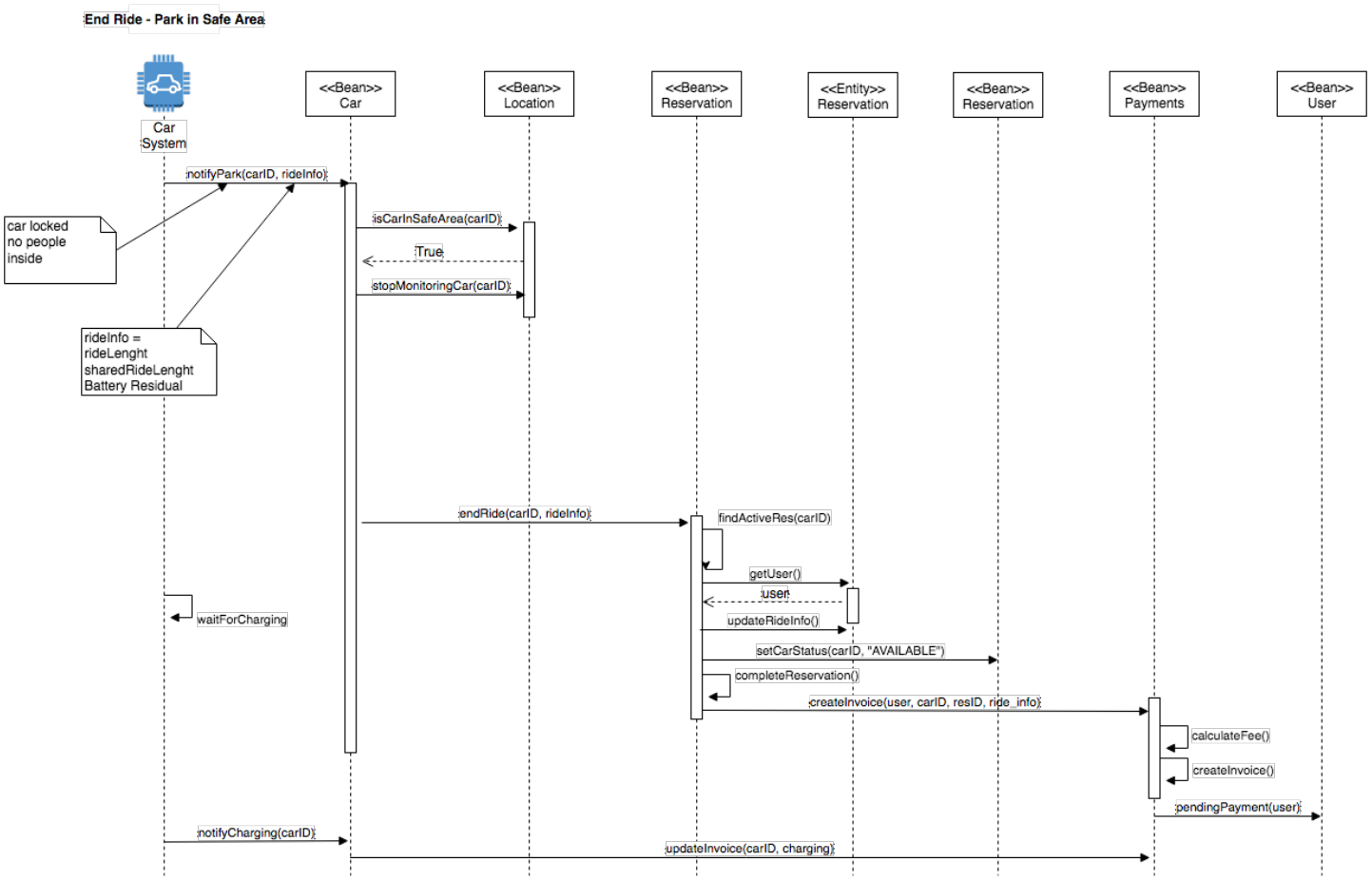
Unlock Car


Car Unlock
Power User
Mobile App
<<bean>> Reservation
<<entity>> Reservation
<<Bean>> Location
<<Bean>> Car
unlockMyCar()
getGPSposition()
user_pos
unlockCar(u, user_pos)
findActiveRes(u)
alt
[no Reservation]
error
error
getCarID()
carID
getCarLocation(carID)
car_pos
opt
user_pos in car_pos.range()
unlockCar
Ok!
Ok!
Ok!

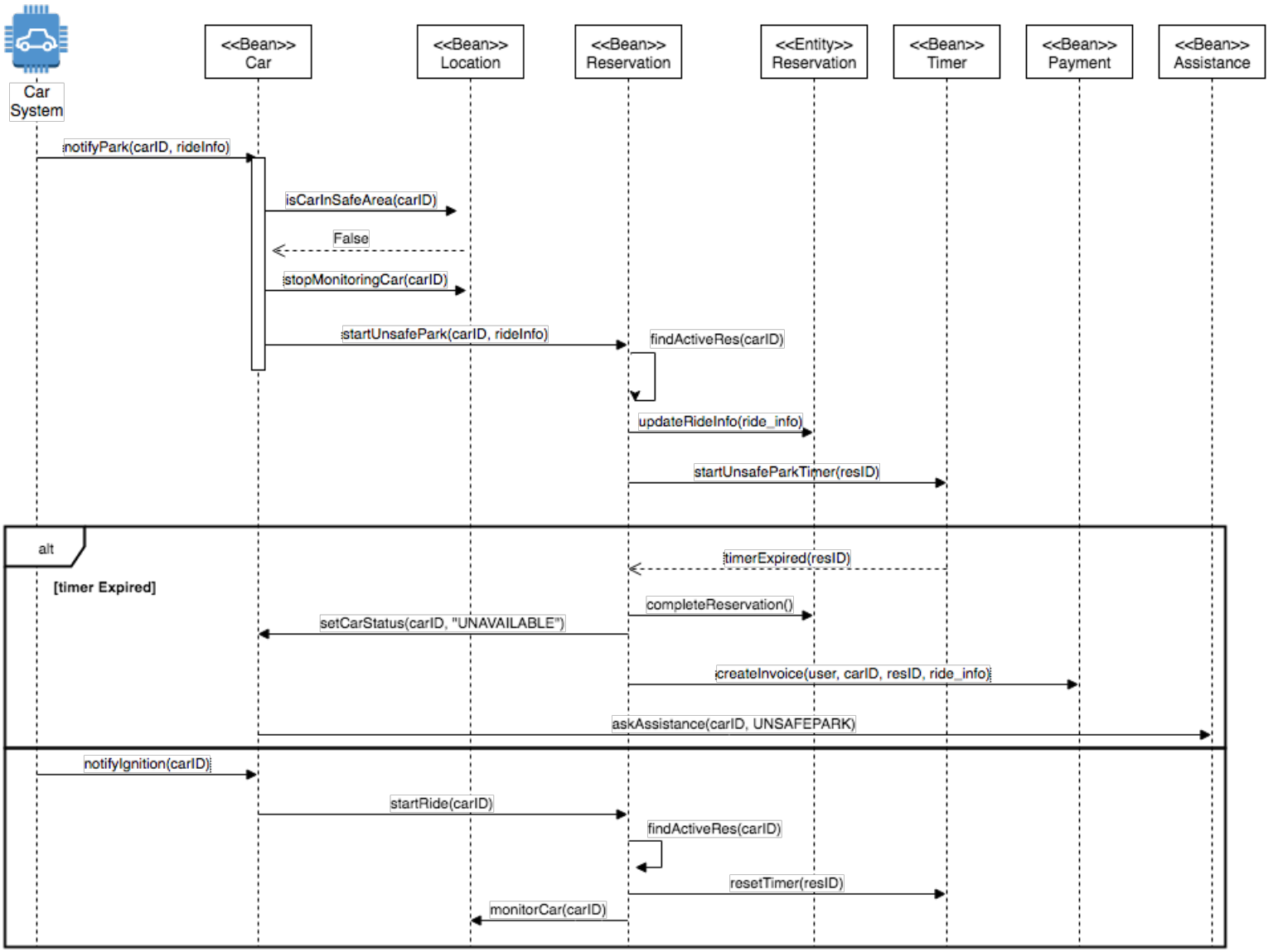
Start Ride


Car System
<<Bean>> Car
<<Bean>> Reservation
<<Bean>> Time
<<Bean>> Location
notifyIgnition(carID)
displayFee()
setCarStatus(carID, "INUSE")
startRide(carID)
findActiveRes(carID)
resetTimer(resID)
monitorCar(carID)
loop
getGPSPosition()
pos
updatePos(carID, pos)

End Ride (Safe Park)



End Ride (Unsafe Park)



3 Algorithm Design

In this section we give some guidelines for the programmers for the most crucial part of the application.

Object Relation Mapping and Searches In the runtime diagrams usually a component is asked to find a specific entity from an identifier, for example the Reservation Component needs to find active Reservation Entity from a User ID or a Car ID. In order to make this process more efficient we could introduce and HashMap to keep track of the active reaservation. This will avoid going back to the database everytime and will result into a faster information retrieval and a overall better system performance.

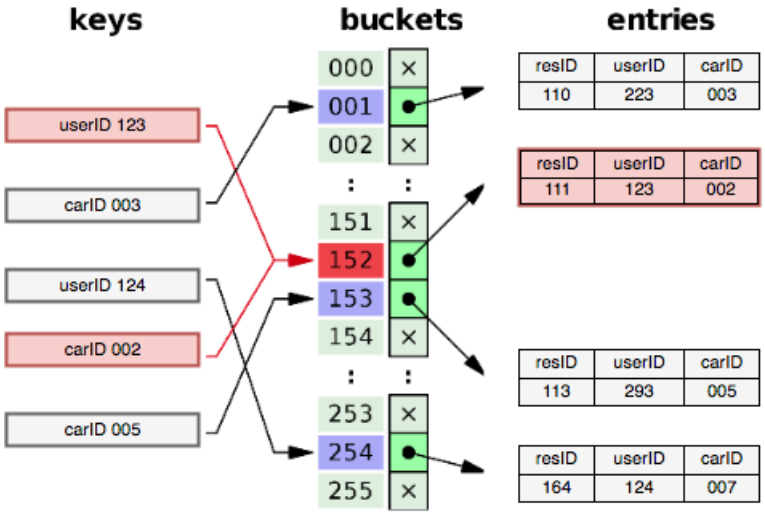


Figure 1: Application Layer with Service Oriented Approach

Searching Car on a Map The location component as a method called `findCars(position, range)` which given as inputs a gps position and range returns a list containing all the cars in such area. A naive approach would be to search the entire Location Database and check if a car belongs to such area, but this is very expensive in term of time.

```
badFindCars(position, range)
        FOR-EACH c:cars IN locationDB
                <create list L>
                p = c.getPosition()
                IF p in area(position, range)
                        <append c to L>
        return L
```

A better search requires more logic. We divide the map of the zone of interests in cells forming a grid as shown in the next figure.

Figure 1: Grid View

Since the lenght of the cell is fixed, we can limit the search only in zones covered in the area. This requires that the Location DB keeps track of the zone in which the car is currently in.

Figure 1: Grid Search

```
FindCars(position, range)
        <create list L>
        zones = <findZones from position,range>
        FOR-EACH c:cars IN zones
                p = c.getPosition()
                IF p in area(position, range)
                        <append c to L>
        return L
```

4 UI Design

4.1 Mobile Application

The mobile application, as usual, will have a recognizable icon that can be added on the desktop. When the application is opened, the display will show the login screen.

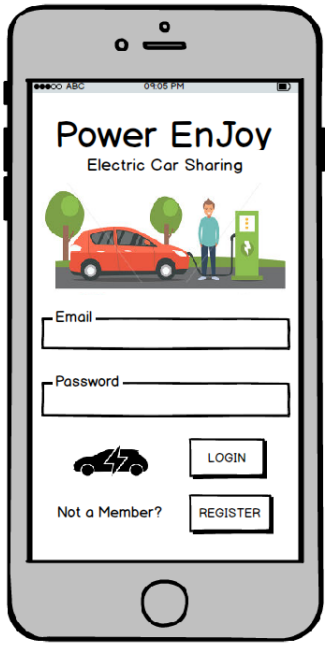


Figure 1: Mockup

E-mail and Password are required for the login. Insert wrong credential cause the refresh of the page with a notification explaining the problem, without giving information about the wrong field. In this screen is possible to register new Users by the button "REGISTER". Clicking leads to the registration screen, it is a form who must be filled entirely and correctly to have a successfull regstration. The fields must be at least "name", "Surname", "Driving Licence", "E-Mail" and "Payment System", add other fields could be halpfull but not strictly necessary. If some datas are not acceptable, a notification will be displayed and the form will be reuploaded. After the login, the display show the Main Screen.

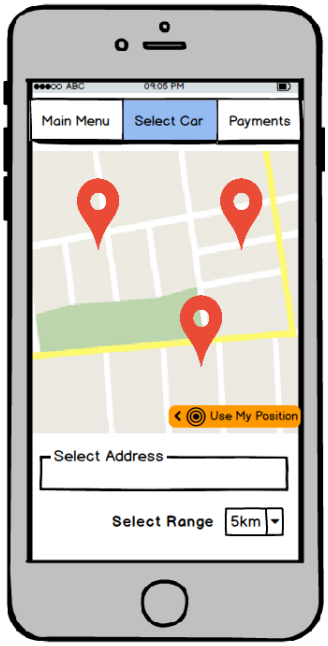


Figure 1: Mockup

On the screen is possible to see the map of the city, if is available the center will be the user position, if it is not available the center will be the center of the city. In the upper side there is a menˇ with the sections "Main Menˇ", "Select Car" and "Payments". Main Menˇ open a list of fields. It contains the field "Modify Profile" that lead to a screen equal to the Registration Screen previously described. In the Main Menˇ list is auspicable to put all functionalities not strictly related to the reservation and the payment, this will make more clean and simple the application. Payments allows to pay the unpayed fee, if there is. If there are no unpayed fee will be shown a message like "You payed the last fee, there is nothing to pay more". If there is a payment pending the massage will be "There is a pending payment. You have to pay 5Ä (example)." and a button will start the payment procedure. Select car is the basic screen of the application. In this section is possible to set the center point of research and the research range, user should be guided to choose his position as center even if is possible to put any position as center of the research. When center and range are available for the server, it starts to search cars. Cars will be shown on the map and user can click on them to see the car informations.

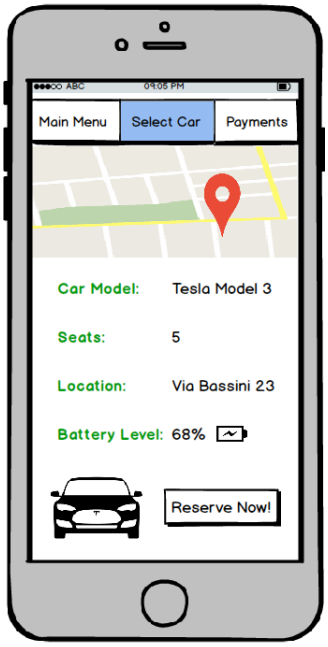


Figure 1: Mockup

Informations must be about the battery and the location, could be usefull add other information as model and number of sets. A button "Reserve Now" will start the reservation. When the reservation starts the display will show a one hour countdown, the timer indicates the time to the expiration of the reservation. An unlock button and a cancel button will be in the screen as well. If the unlock button is pushed not near enough, will be dislayed a message that notify the problem. When the car is ignited the countdown stops and the application go in standby, in the logic the application switch from reservation to ride. Only the application is in standby, not the entire phone. Application will wake up when the car is turned off. If the car is in an unsafe area, "You left car in an unsafe area" will be displayed togheter wit an unlock button and a one hour countdown. At the end of the countdown the ride will end If the car is in a safe area the ride will end. When a reservation or a ride ends, a payment notification is shown. It should be like "Your reservation has ended. Your fee is 8Ä. Thanks for using Power Enjoy Service". A confirmation button allows to go back to the Main Scree. The payment message is not shown if the reservation is canceled. Is important to underline that on the payment notificatio will NEVER appear the word "ride": ride is usefull for the logic but its existence is useless for the user, so is usefull to avoid the use of two terms. Every time is not specified, the "go back" button of every mobile system will accomplish the "go back" function.

The application must be as simple as possible, the main idea is to make every

marginal utilities, like the main menˇ options, obscured by the main functionality. This reasoning increase usability thanks to the focus on the reservation.

The language to write the application obviously will change from a operating system to another. The application must run on Android and iOS. For Android the language will be Java and for iOS the language will be Swift.

4.2 Web Page

Web Page is a support of the application. The presence of a dowload link should be as constant as possible.

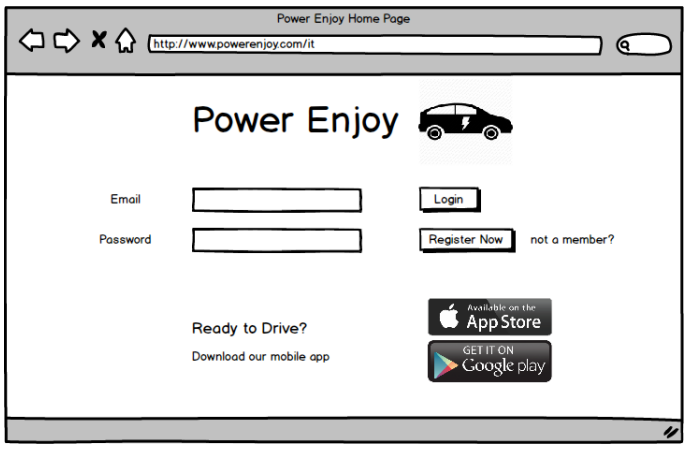


Figure 1: Mockup

The main purpose of the web page is to give the possibility to the user to have a more efficient way to manage side functions. The app will focus on the reservations, and the web page allow to have a better interface for the options of the main menˇ list. The main page is very similar to the access screen of the application and the consequences of the buttons are the very same.

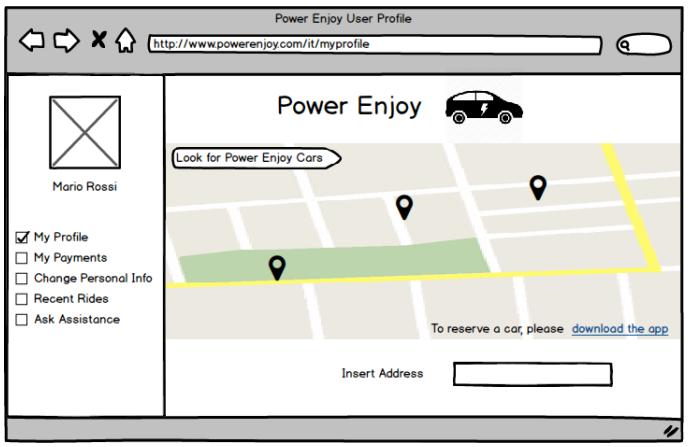


Figure 1: Mockup

On the main page the is a list on the left. From the lists fields is easy to manage the profile, from the personal information to the history of payments. On the main page is also possible to see Power Enjoy Cars distribuited on the map. This possybility doesn't allow to reserve a car, this is possible only via application

The web page should be written in XML or JSON, to allows quick and efficient data transfer through textual data files over HTTPS.

5 Requirements Traceability

Explain how requirements defined in the RASD map to the design elements that you have definited in this document.

6 Effort Spent

Niccolo' 23 Ore