DD - Software Engineering 2

POLITECNICO
MILANO 1863

PowerEnjoy

Marini Alberto
862838
alberto2.marini@mail.polimi.it

Marrone Matteo
810840
matteo.marrone@mail.polimi.it

Sabatelli Antonella
875666
antonella.sabatelli@mail.polimi.it

December 11th, 2016

Politecnico di Milano

Contents

Chapter 1

1 Introduction

1.1 Purpose

The Design Document here presented is meant to provide relevant information regarding the architectural layers, components and interface of the Power Enjoy system to be developed as well as a description of the interaction occurring between the different architectural parts at different levels, directed to project managers, developers, testers and Quality Assurance staff; within the DD well make use of graphical representations such as component views.

1.2 Scope

The PowerEnjoy system to be built will be able to offer a service of car sharing within the city of Milan involving electric cars only. Our description will not only include a description of the architectural tiers with a gradually increasing amount of detail, but will specify the relations between the tiers and the cars, characterized by a limited AI.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Session Bean: is a component of the application logic used to model business functions.
- Stateless Session Bean: no state is maintained with the client.
- Stateful Session Bean: the state of an object consists in the values of its instance variables. They represent the state of a unique client/bean session.When the client terminates, the bean is no longer associated with the client.
- Singleton Session Bean: is instantiated once per application and exists for the whole application lifecycle. A single bean instance is shared across and concurrently accessed by clients.
- Java Server Faces: a component-based MVC framework built on top of the Servlet API.

1.3.2 Acronyms and Abbreviations

- RASD: Requirements Analysis and Specication Document
- Java EE: Java Enterprise Edition.
- MSO: Money Saving Option.
- REST: Representational State Transfer.
- XML: eXtensible Markup Language
- EJB: Enterprise Java Beans.
- UX Diagram: User Experience Diagram.
- DB: the database layer, handled by a RDBMS.
- UI: User Interface.
- MVC: Model-View-Controller.
- JDBC: Java DataBase Connectivity.
- JPA: Java Persistence API.
- MITM: man in the middle

1.4 Document Structure

The document is divided in seven parts, as of requirements:

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 sub-components 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 in detail the algorithms designed for the system functionalities, 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 of the DD.

Chapter 2

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. The system will be described starting with high-level components in Section 2.2. This high-level design will be detailed through Section 2.3. Section 2.4 will put some attention on the deployment of the system on physical tiers, and Section 2.5 will describe the dynamic behaviour of the software. Section 2.6 will focus on the interface between dierent components of the system. The design choices and patterns used in the aforementioned sections will be presented and discussed in Section 2.7.

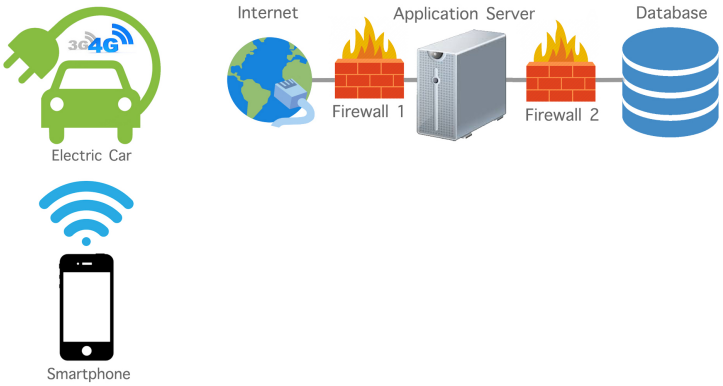

Figure 1: 3-tier architecture

2.2 High level components and their interaction

We make use of a 3-tier architecture composed of a client tier, a business tier and a database tier, the two last tiers being mocked on a cloud platform. The Client Tier contains the mobile application clients, who can be characterized as thick since only the presentation layer of the application is located within it. The mobile Android app interacts with the business tier, where the application layer is located, and with the car, which communicates with the business tier as well (for instance while exchanging information collected through the sensors).The application server within the business tier in turn hosts the entirety of the application logic (that is, all the algorithms, the policies and the computations take place on the application server) under

the form of Enterprise Java Beans and Java Entities, manages the requests sent by the clients and the info received by the car, producing appropriate responses. The application server acts as a DBMS as well and as such interacts with the data tier, within which the data layer is comprised, through a Java Persistence API.

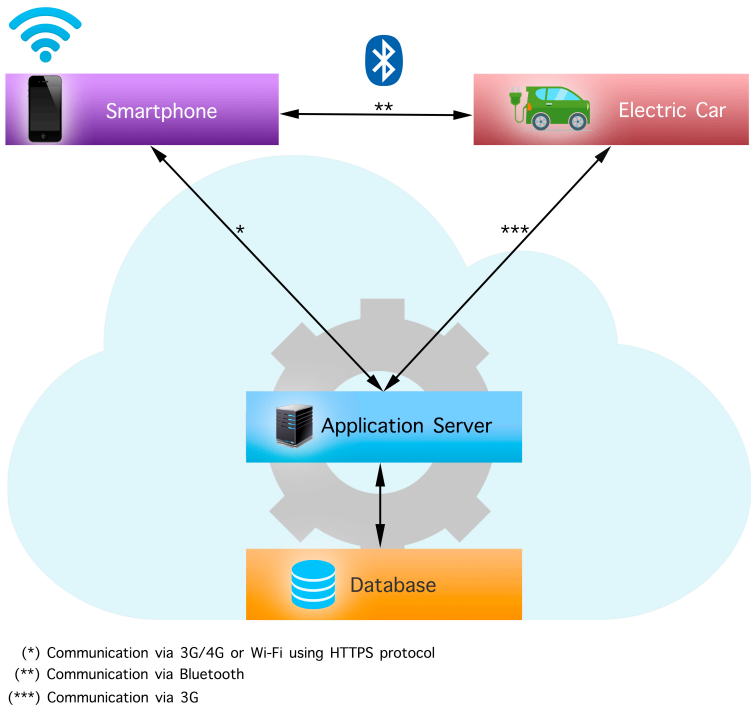

Figure 2: Layers of the system

2.3 Component view

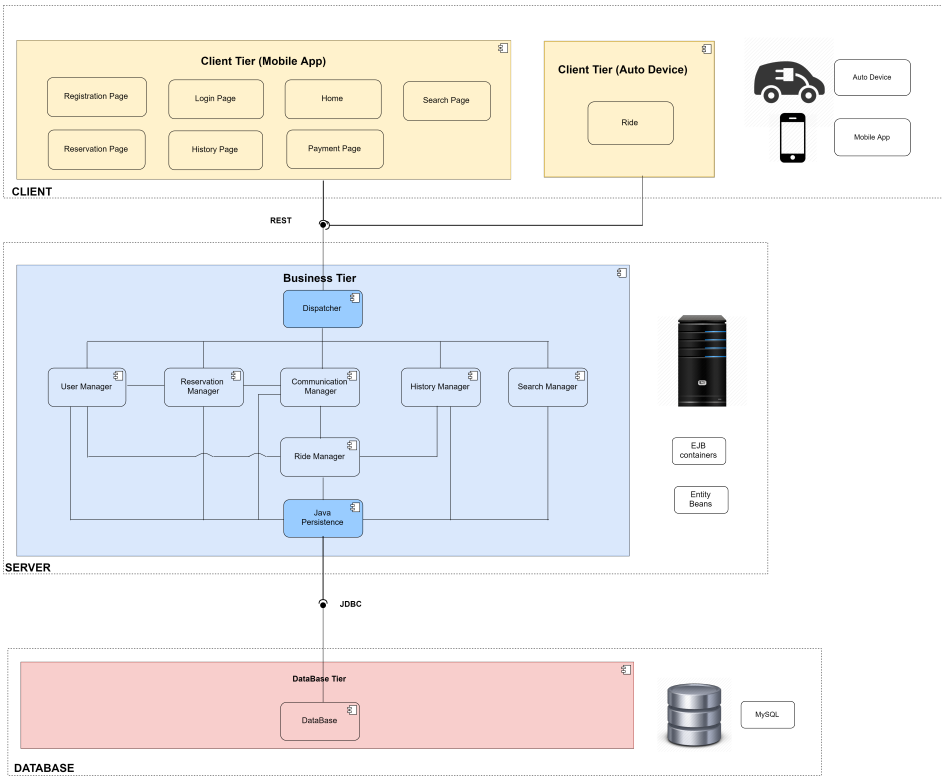

Figure 3: High Level Components view and their interaction

2.3.1 Application server

The application server is implemented in the business logic tier using *Java EE* and runs on *GlassFish Server*. The access to the DBMS is not implemented with direct SQL queries: instead, it is completely wrapped by the *Java Persistence API* (JPA) which provides an object-relation mapping through entity beans.

In accordance to RESTful paradigm, the business logic is implemented by custom-built stateless *Enterprise JavaBeans* (EJB); the statelessness of the beans leads to a significant simplification of the operations carried out by the business beans in what is de facto a rather basic event driven service oriented architecture, preventing as well loss of data in case of instance failure. Beans used are listed below:

User Manager

Functions comprised in this bean are related to user management features such as user login, user registration and modification of user history and characterization.

Search Manager

This bean is in charge of the searches management: it creates new search objects and its subobjects as well (such as the maps).

Reservation Manager

This bean is mainly meant to manage creation and destruction of reservations (both upon user input and due to timer expiration), but contains functions addressing the need to manage damage reports, too.

Ride Manager

When a car is picked up in time by the right user a new ride is created and its details (such as its MSO enabling and its price calculation) are managed through methods contained in this bean.

Communication Manager

This bean is mainly in charge of the interaction with the cars and the safe areas software and triggers events according to the information retrieved by the sensors.

History Manager

This beans functions are meant to create, analyze and export logs regarding different aspects of the application.

Dispatcher

This component forwards the requests coming from the clients to the correct manager and the responses coming from the beans to the correct client.

Configuration Bean

This bean is a singleton and its only duty is to read the server conguration le and provide the value of conguration options to other components.

2.3.2 Database

The database tier runs MySQL 6.0 and uses InnoDB as the database engine. As an external component which only needs to be congured and tuned in the implementation phase, no details regarding its internal design will be provided.

All the persistent application data is stored in the database.As the dynamic behaviour of the data is handled entirely by the Java Persistence API in the Business Application tier, foreign key constraints and triggers are not utilized.

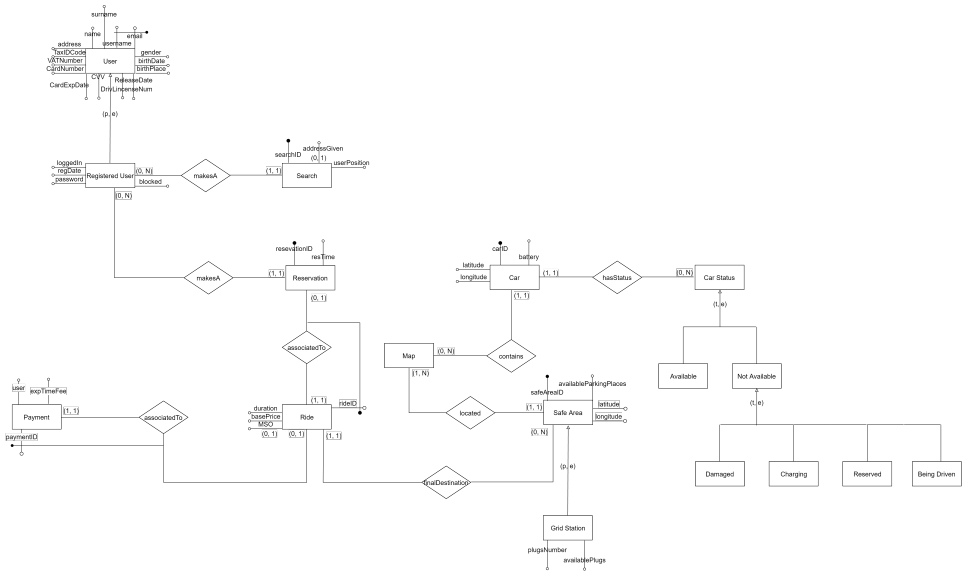

Figure 4: Entity Relationship diagram of the DB

2.3.3 Mobile Device

The mobile client implementation depends on the specic platform. The Android application is implemented in Java and mainly uses android.view package for graphical management. the application mainly consisting of a controller which translates the inputs from the UI into remote functions calls through RESTful APIs; the controller also manages the interaction with the GPS component LocationListener interface, whereas android.bluetooth package is employed to exploit functions related to Bluetooth. The iOS application works similarly, but it is implemented in Swift and manages the UI interface using UIKit framework; moreover the interaction with the GPS component is carried out using CoreLocation framework.

2.3.4 External Component Cronjob

A cronjob will be scheduled to run every 2 minutes to check whether cars in a charging state have reached a battery level greater than 75% of the maximum and, if so, change its status to available.

2.3.5 Component Car

The software embedded in the cars (that is, what will be referred within the deployment view as the Power Enjoy Auto App) acquired by the service was developed by the producer of the car itself using the Snapdragon™ Automotive Development Platform (ADP) configured with the Android OS, adapted by us so to enable it to interact with our cloud and is mainly used for the communication of the data picked up by the sensors toward the application server, the managing of the pairing process leading to the opening of the car after its reservation and navigation, inclusive the calculations of the shortest path within the MSO. Information is provided by the application server to the cars only in two instances, that is, when the car receives the pairing code after the reservation and when the location of the chosen station is pointed to the car within the MSO context.

2.4 Deployment view

This diagram shows the deployment view of the software product. This diagram is only depicts the distinction between client machines, server machines and database machines at large, because of the early stage of system developing.

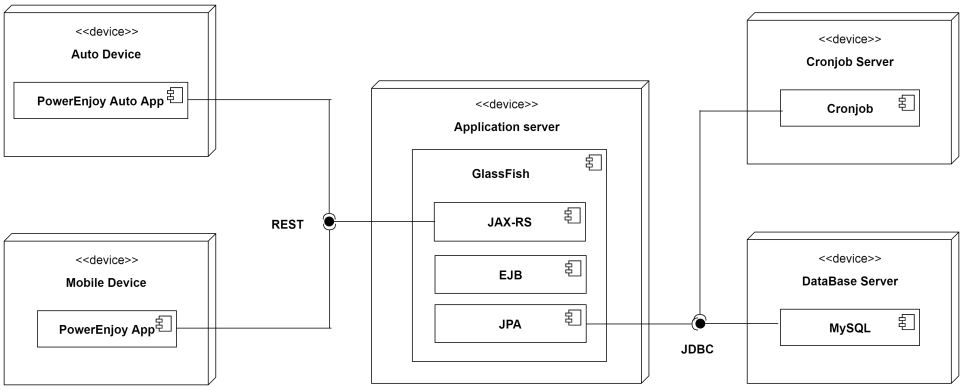

Figure 5: Deployment diagram

2.5 Runtime views

In this section it will be shown how the software and logical components interact one with another. We decided to don't represent the database in the sequence diagram, because the interaction with the database is totally abstracted by the entities via the *JPA* (Java Persistence API).

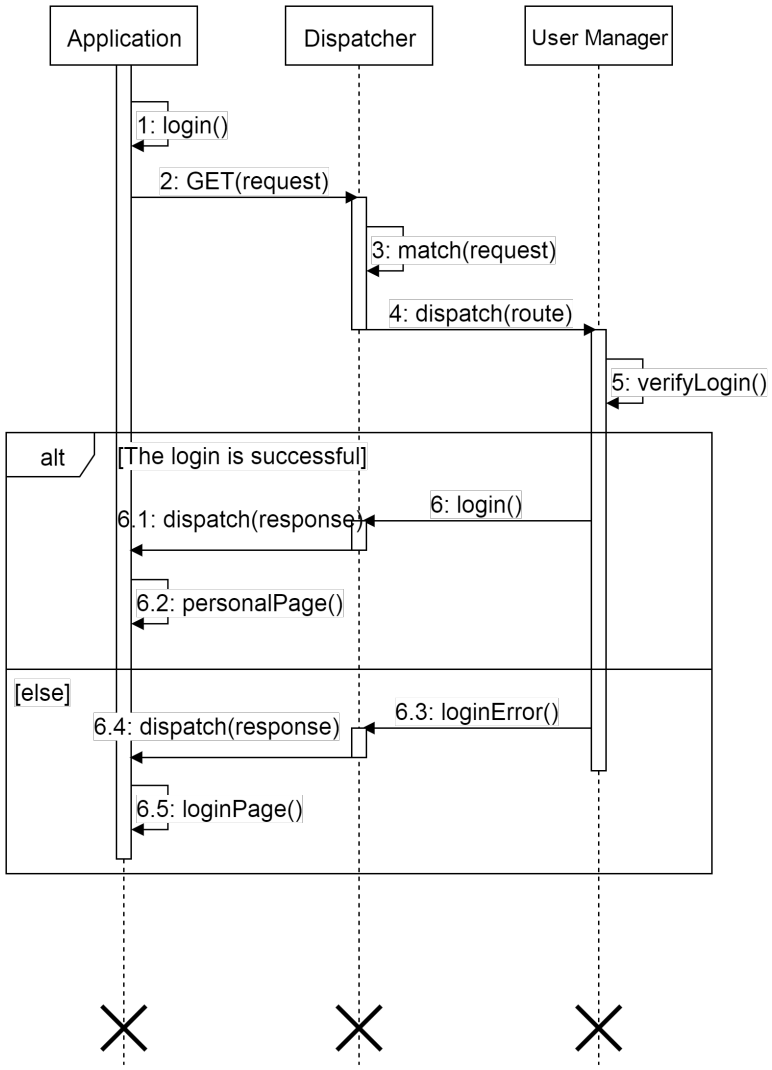

Figure 6: Login procedure for a registered user via a mobile device

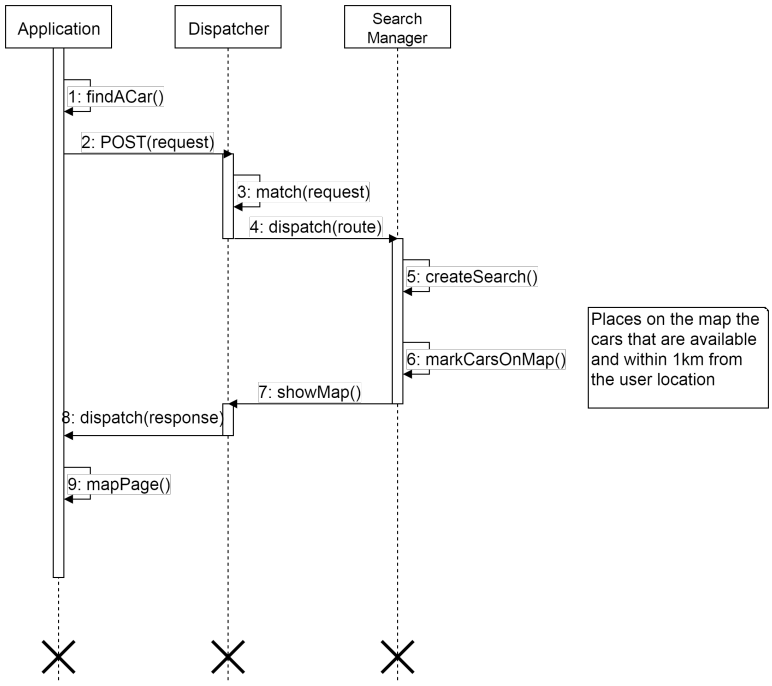

Figure 7: Search procedure for available cars, via mobile device, depending from user position

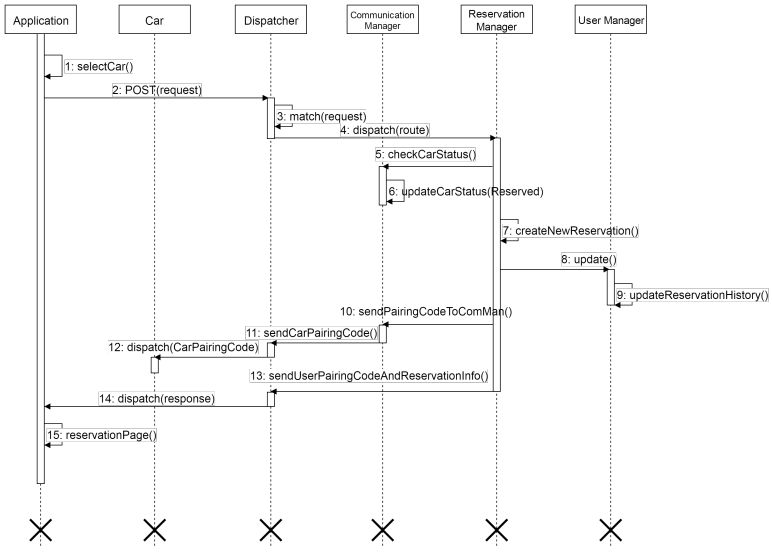

Figure 8: Reservation procedure to book a car from a mobile device

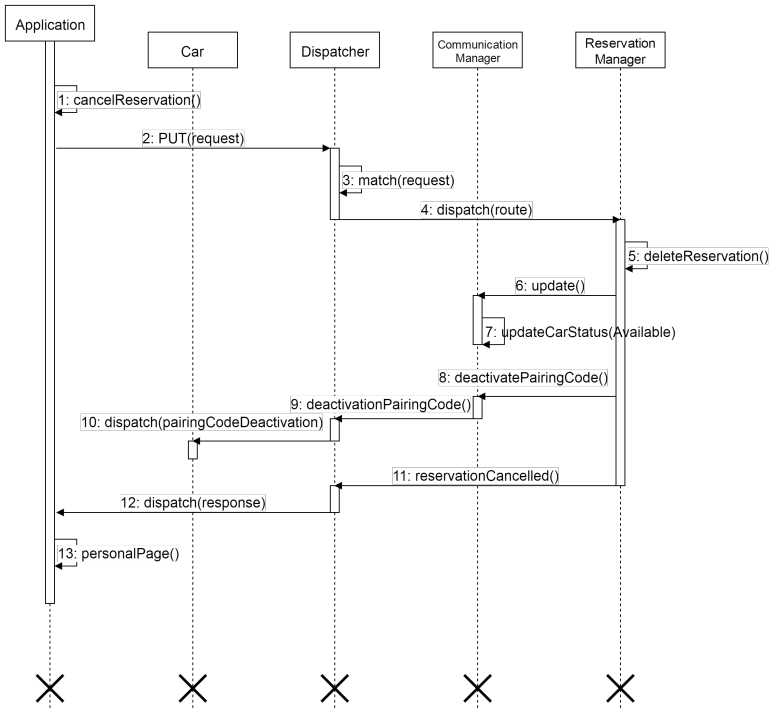

Figure 9: Procedure to cancel a booking within 15 minutes, from a mobile device

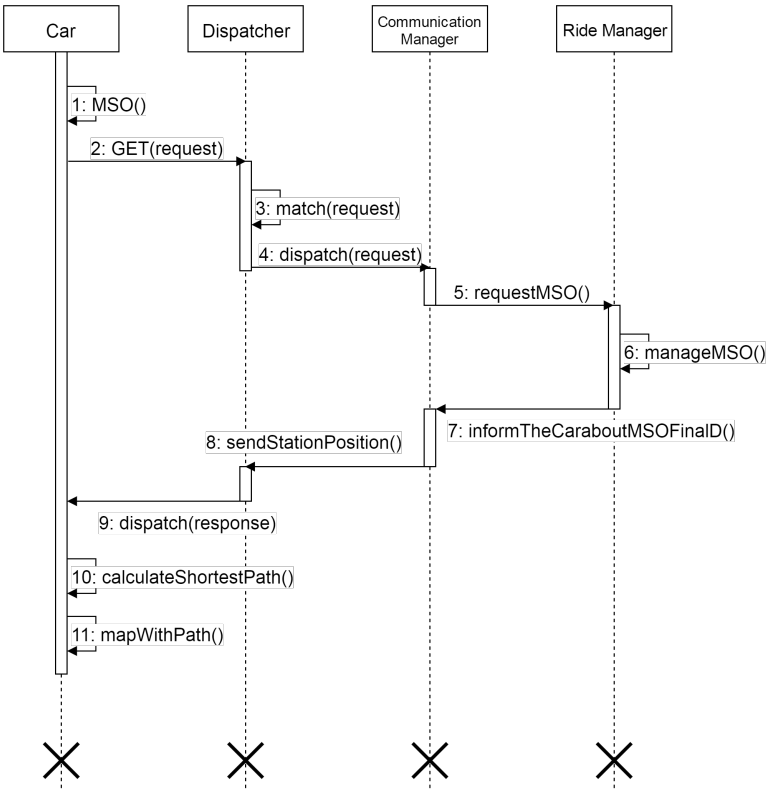

Figure 10: Procedure to activate the Money Saving Option, before the beginning of the ride, from auto device

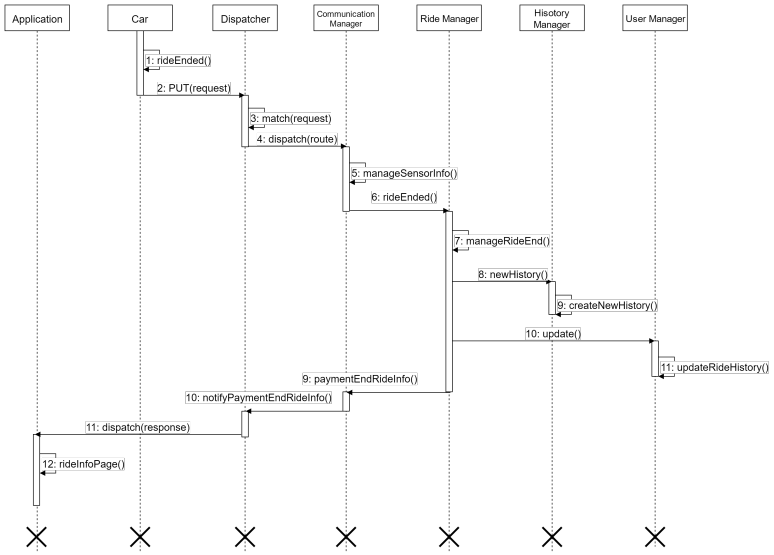

Figure 11: Conclusion of a ride, notified by the auto device

2.6 Component interfaces

The application server communicates with the DB via the Java Persistence API over standard network protocols. Thus, the DB and the application server layers can be deployed on dierent tiers, as well on the same one.

2.6.1 User Manager

+createnewUser(name, surname, username, credit card number, CVV, credit card expiration date, gender, birthday, address, birthplace, TAX code, VAT number, driving license number, driving license photo, driving license expiration date, driving license issue date): User
This function creates a new User.

+createnewRegisteredUser(user, password): RegisteredUser
This function creates a new RegisteredUser from an existing user, being invoked after the existence of the driving license provided by said user has been proved.

+createNewPassword(): String
This function creates a password to be sent to a newly registered user, thus returning a string obtained from a randomization algorithm.

+verifyLogin(username, password): boolean
This function determines whether a couple username-password submitted through the login interface does actually match one such couple stored in database and allows login accordingly.

+verifyRegistrationForm(name, surname, username, credit card number, CVV, credit card expiration date, gender, birthday, address, birthplace, TAX code, VAT number, driving license number, driving license photo, driving license expiration date, driving license issue date): boolean
This function checks whether the fields to be compulsorily filled in the form were filled in with acceptable values (for example, a not-already-existing-in-the-database username and a not-already-existing-in-the-database TaxID-Code) and by mean of an image analysis process checks if the data inferrable from the driving license photo submitted matches the data in the form. If the check result is positive the function return true and the data is forwarded to an agent who will check if the driving license matches an existing one.

+block(username): void
This function changes the value of the attribute blocked in a RegisteredUser to true.

+updateRideHistory(username, new ride): void
This function adds the ride passed as a parameter to the list of rides of the

user.

+updateReservationHistory(username, new reservation): void
This function adds the reservation passed as a parameter to the list of reservations of the user.

2.6.2 ReservationManager

+createNewReservation(user, car): Reservation
This functions creates a new reservation and associates it to the user (checking it is not blocked) and the car received as parameters, creating the timers and returning said reservation.

+deleteReservation(reservation): void
This functions deactivates the timers associated to an existing reservation and updates the status of the car reserved.

+Timer1ExpirationManagement(): boolean
This function is a task to be activated when the 15 minutes timer for the deletion of a reservation expires, and allows for the disabling of the deletion option.

+Timer2ExpirationManagement(): boolean
This function is a task to be activated when the 60 minutes timer for the pickup of a reserved car expires, it requests the deletion of the reservation.

+manageDamageReport(damageImages, Reservation): void
This function raises a flag signalling the existence of a car damaged after a damage report is sent in, and requests the deletion do the reservation.

2.6.3 RideManager

+ManageRideBeginning(reservationID, baseprice, MSOselected, finaldestination): void
See the pseudoJava code in algorithm design section.

+ManageRideEnd(ride, twoOrMorePassengers, charging): void
See the psudoJava code below in algorithm design section.

+ManageMSO(latcenter, longcenter, ride): GridStation
See the pseudoJava code below in algorithm design section.

2.6.4 CommunicationManager

+manageSensorInfo(weightSensed, BeltsFastened, car, BatteryLevel, Location, pluggedIn, locked)
This functions processes the data picked up by the car sensors and invokes updateCarStatus when necessary, providing at the same time information needed to carry out other procedures: for instance, if the weight sensed by the car and the number of the fastened belts as perceived by the car match

+updateCarStatus(car, new carStatus): void
A method to update the car Status. See UML Class Diagram in the RASD quoted for information regarding the possible statuses.

+updateSafeAreaStatus(safeArea, newvalueofAvailableParkingSpots): void
A method to update the safe area parameters.

+informTheCaraboutMSOFinalD(car, GridStation): void
This function allows for the communication to the car regarding the final destination MSO-related determined within a manageMSO context.

2.6.5 HistoryManager

+createNewHistory(ride, payment): History
Creates a history entity.

+exportHistory(user, componenttype): History
Allows for the history exportation towards other components.

+PatternDeduction(histories): Payment ([Average Payment]), SafeArea(Area Where Cars Are More Often Parked)
This function is used for data analysis. It takes an arrayList of histories as a parameter and from that it potentially infers pattern characterizing the rides, and derives the return values here listed, that is the average price of a ride or the safe area most frequently chosen.

2.6.6 SearchManager

+createSearch(user, latcenter, longcenter): Search
It creates a searchEntity.

+markCarsOnMap(cars): Map
See pseudo Java code below.

2.6.7 Dispatcher

+match(request, deviceID, deviceType)
According to the request type and the deviceType(car or mobile phone) a proper response procedure is invoked by this function, the device source being identified thorugh its ID.

+dispatch(deviceID, response, deviceType)
This function dispatches a response previously requested towards the correct destination., the device source being identified thorugh its ID.

2.6.8 Configurator Bean

The application server is congurable by means of a XML conguration le through the configurator bean. The conguration le defines:

- the boundaries of the area where the PowerEnjoy service is active, expressed as polygons (list of coordinates);
  - the locations of the safe areas and the grid stations among them;
  - the initial location of the cars;
- the credentials of the user that can access the database;
- the host, port and name of the database;
- the network settings of the application server (listening port, host, ...);
- any other settings that will be useful in the implementation phase;

2.7 Selected architectural styles and patterns

RESTful architecture: as previoulsy mentiones, we make use of RESTful paradygm to transfer over HTTP web resources between client and server.

Client-Server: Cars and mobile phone softwares are both styled as clients, though of different type, issuing requests to the server and managing the proper responses.

Thin client: since only a rich user interface and the aforementioned controller are to be developed on the mobile device client side(all the application logic is on the application server), clients can be seen as thin clients: this should prove advantageous as it will ease updates and favors usage by devices with low processing power, thus expanding the pool of potential users.

Model-View-Controller (MVC): as hinted before in the mobile application section the widespread model-view-controller design pattern will be

followed in coding the clients, in order to properly decouple the different parts of the application.

Pattern State: another common design pattern employed will be the pattern state to efficiently manage the state of the object of type car. Please refer to the RASD (see last section od the document)to observe as this was shown in the UML class diagram related to this project.

Pattern Singleton: the object Map shown in the Class Diagram needs to be instantiated only once during the configuration process and as a consequence will be treated as a singleton using the Singleton creational design pattern. Again, refer to our RASD to see how this was represented through the UML class diagram.

2.8 Other design decisions

2.8.1 Maps

The system relies on an external service, **Google Maps**, for geolocalization, distance calculation and map generation and visualization processes, generally trusted by virtue of being common and widely tested. Maps API will be used both on the server side (for map generation, distance calculation, exc.) and on the client side (in the context of map visualization), and since it is reasonable to assume that most users will have some degree of experience with the function this will determine a higher level of the application usability.

2.8.2 Security

2.8.2.1 External Interfaces Side

PowerEnjoy app implements a login authentication to protect user informations. In particular, each user password is saved using hashing mechanism. This methodology provide good level of security, even if the system doesn't require anything about the password strongness. So, it could be developed a system that requires a strong password with 8 or more characters mixing numbers, uppercase and lowercase letters and special symbols. The password is static and the user isn't involved in changing the password. The system will ask to user to change the password every 3 months.

Among the future possible implementations we took into account figures a login procedure including captcha, to prevent potential attack from botnets, and other multi-factor authentication system using for example biometric authentication through fingerprint or retina scan.

2.8.2.2 Application Side

On the application side, the register and login procedures implement a filtering system. However, malicious users could fill the form with SQL code to have access to hidden information, using *SQL injection* methodology.
PowerEnjoy implements *https* connection to guarantee communication confidentiality and integrity and also mutual authentication. SSL is resistant to **man in the middle attack**(MITM) but need a server certificate signed by a Certification Authority(CA).

2.8.2.3 Firewall

Two firewalls of the packet filtering type will be located respectively between the client tier and the business tier and between the business tier and the data tier; details regarding access policies and ports are listed in the table below. Host-Based IDS will be developed as well on each car.

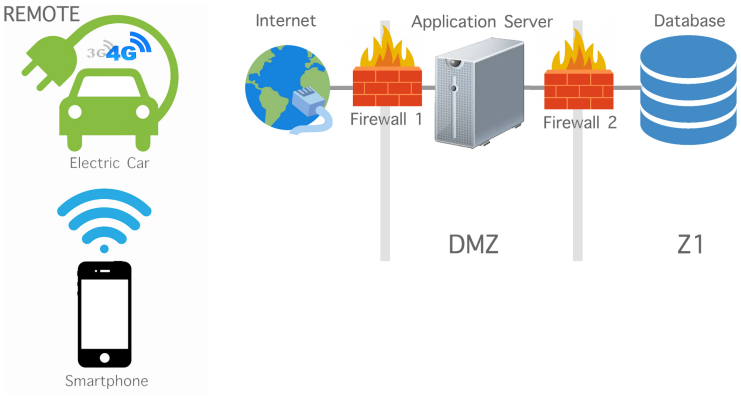

Figure 12: Firewall

2.8.2.4 Details regarding the choice of hosting server side on a cloud platform

All the system will be hosted on a cloud platform like **Amazon EC2**, in Frankfurt. This solution gives us more scalability and could reduce costs, compared to dedicated servers. In fact, this cloud solution let us focus more on the quality of the service; maintaining an high level of performance, especially in case of high traffic. (We have chosen to host the system in Frankfurt, because the legislation on data is quite similar to Italy. At the same time, a backup of the data will be hosted also on some servers in Ireland, to protect the service from potentially disastrous loss of data.

Table 1: Firewall configuration table

Firewall	SRC IP	SRC Port	Direction of the1st packet	Dst IP	Dst Port	Policy	Description
ALL	ANY	ANY	ANY ->ANY	ANY	ANY	DENY	Default deny on all firewalls
FW1	ANY	ANY	PUB ->DMZ	AS_IP	80	ALLOW	The application server is publicly reachable
FW2	AS_IP	ANY	DMZ ->Z1	DB_IP	CUS-TOM	ALLOW	The application server reaches the DB
FW1	AS_IP	ANY	DMZ ->PUB	RE-MOTE_IP	443	ALLOW	The application server reaches the remote device
FW2	DB_IP	ANY	Z1 ->DMZ	AS_IP	443	ALLOW	The DB reaches the application server