Software Engineering 2 Project: PowerEnJoy
DD: Design Document

December 11, 2016

- Alessio Mongelluzzo
- Michele Ferri
- Mattia Maffioli

Contents

1 Introduction

1.1 Purpose

This is the Design Document for the PowerEnJoy application. Its aim is to provide a functional description of the main architectural components, their interfaces and their interactions, together with the algorithms to implement the application logic 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

The system supports a car-sharing service employing electric cars.

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.

Design patterns and architectural styles will be used for solving architectural problems in order to simplify the system comprehension and avoid misunderstandings during the implementation phase.

1.3 Definitions, Acronyms, Abbreviations

- **RASD**: Requirements Analysis and Specification Document.
- **DD**: Design Document (this document).
- **RDBMS**: Relational Data Base Management System.
- **DB**: the database layer, handled by a RDBMS.
- **UI**: User Interface.
- **Application server**: the layer which provides the application logic and interacts with the DB and with the front-ends.
- **Back-end**: term used to identify the Application server.
- **Front-end**: the components which use the application server services, namely the web front-end and the mobile applications.
- **Web server**: the component that implements the web-based front-end. It interacts with the application server and with the users' browsers.
- **MVC**: Model-View-Controller.
- **JDBC**: Java DataBase Connectivity.

- **JPA**: Java Persistence API.
- **EJB**: Enterprise JavaBean.
- **ACID**: Atomicity, Consistency, Integrity and Durability.
- **MVC**: Model-View-Controller design pattern.

1.4 Reference Documents

This document refers to the project rules of the Software Engineering 2 project, to the DD assignment, and to the previous deliverable (RASD).

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 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.

2 Architectural Design

2.1 Overview

In this chapter we will analyse the components that build up PowerEnJoy system. To begin with, a high-level analysis is carried out, showing how the main blocks interact. Then the single components are detailed in chapter 2.3 and the relation between logic layers and physical tiers is described in chapter 2.4. In chapter 2.6 we anlyse in depth the interfaces that allow the different components to communicate. The dynamics of each feature of the system is detailed in chapter 2.5.

2.2 High level components and their interaction

A client/server architectural style is chosen to undergo the need for a distributed application. In particular, the server side is divided into three main components, to separate the data, the application logic and the presentation. These three components are:

- a DBMS, that manages the storage and retrieval of all the data needed by the application to work;

- an Application Server, that contains all the application logic to provide the services requested by clients;

- a Presentation Server, whose job is to provide the web interface for the users.

In particular, the DBMSr communicates with the Application Server by providing the data needed for the computation, but the Data Server does not implement any kind of application logic. The Presentation Server communicates with the Application Server but it does not contain any application logic as well. The application layer is service-oriented, in order to increase the independence among the services and the global fault tolerance of the architecture.

From the client side, there can be three kinds of clients:

- Mobile, that request services through the mobile application (PowerEnJoy Mobile);

- Browser, that request services through the web front-end (PowerEnJoy Web);

- Employee App, that request employee's functionalities;

Mobile clients are fatter, as they directly communicate with the Application Server of the server side and incorporate the presentation; instead, in the case of Browser clients the GUI is distributed between the two sides and this kind of client, which is therefore thinner, has to go through the Presentation Server in order to request the services of the Application Server.

The high level components of the system are:

- **Mobile application:** the mobile client on the user's device.
- **User's browser:** it represents the browser of the user's computer.
- **Web server:** it provides the web interface the user accesses through his/her browser.
- **Application server:** it contains the application logic: all the functionalities and algorithms that make the system work.
- **(R)DBMS:** (Relational) Database Management System. Software application that interacts with databases. It allows to query, update and administrate the database.
- **Database:** it is the layer where all data are stored. It is a relational database, therefore ACID properties must be guaranteed.

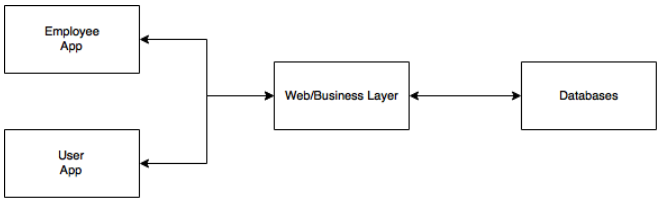


Figure 2.1: High level representation of the system architecture.

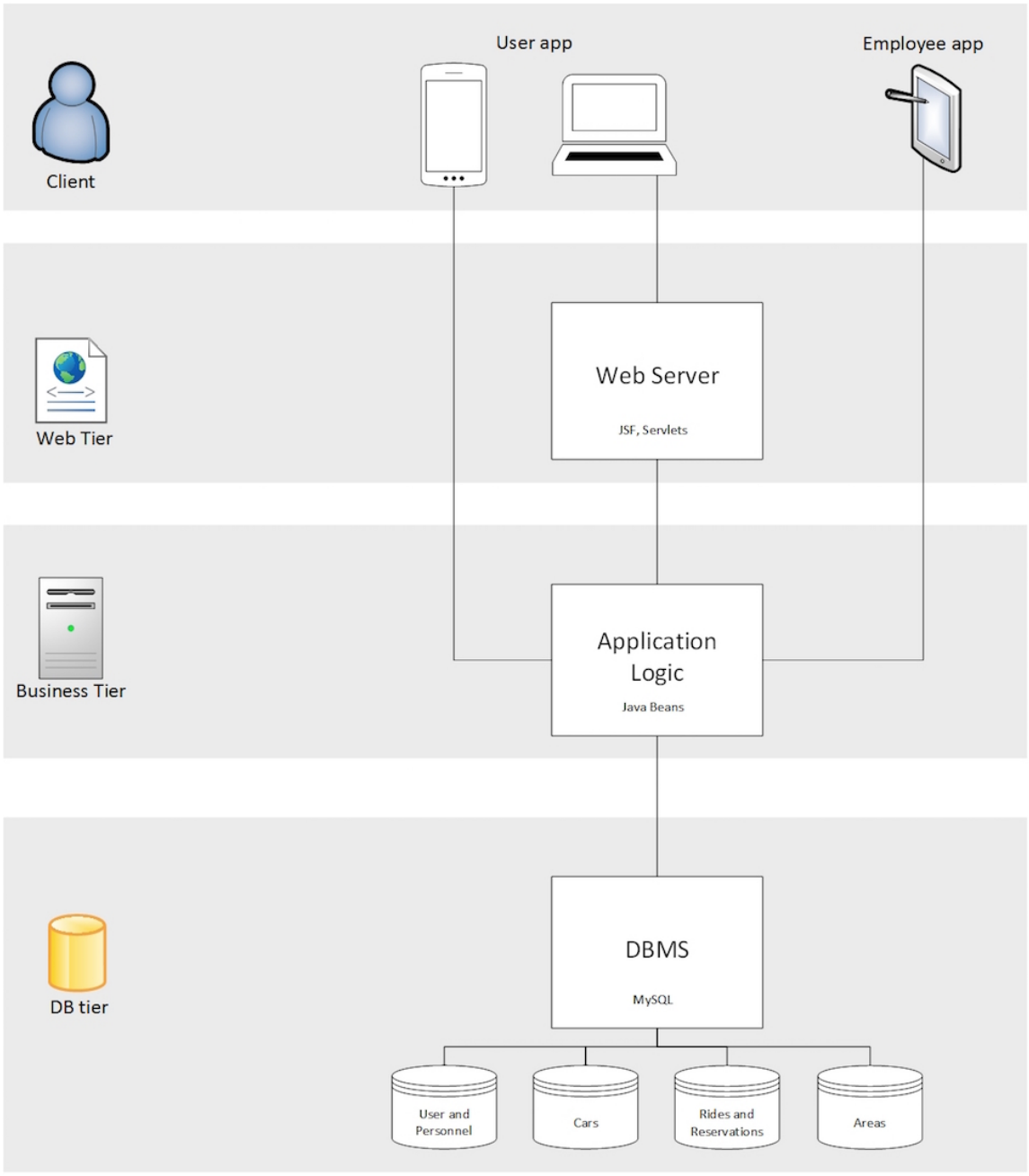


Figure 2.2: Representation of high level components detailed with JEE components.

2.3 Component view

The system architecture is based on 4 main components:

- **Mobile app**

  The mobile client implementation depends on the specific platform. The iOS application is implemented in Swift and mainly uses UIKit framework to manage the UI interface. Instead, the Android application is implemented in Java and mainly uses android.view package for graphical management. The application core is composed by a controller which translates the inputs from the UI into remote functions calls via RESTful APIs.

- **Web Server**

  JSF (Java Server Faces) is used to implement the web server. JSF is a MVC-based framework to perform web pages presentation. The web tier only works as a presentantion layer: RESTful APIs are used to interface with the application server, where all the business logic is set.

- **Application Server**

  The following diagram is meant to show the flow of information from boundary classes to database entities. However, the system has to be developed so that the interaction between boundary and control classes is filtered by an orchestrator whose task is to dispatch boundary requests to the corresponding application manager (i.e. a session bean in jee environment).

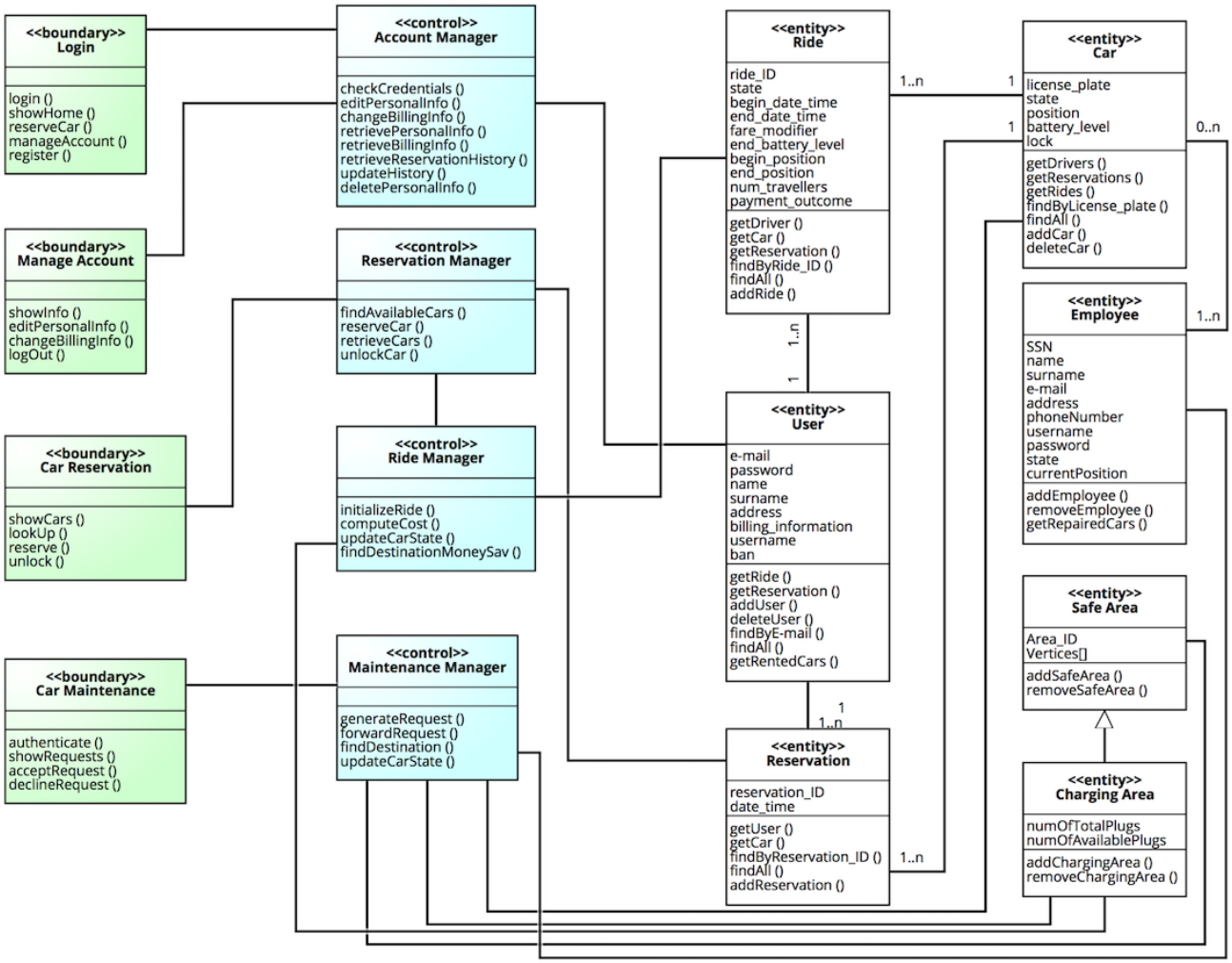


Figure 2.3: Boundary-Control-Entities diagram representing the interaction between UI, business components and database entities.

Client requests are forwarded to the orchestrator. This software module dispatches requests to the correct application module so that the client layer results fully decoupled from the application layer: further modules can be added in the future without affecting the system structure.

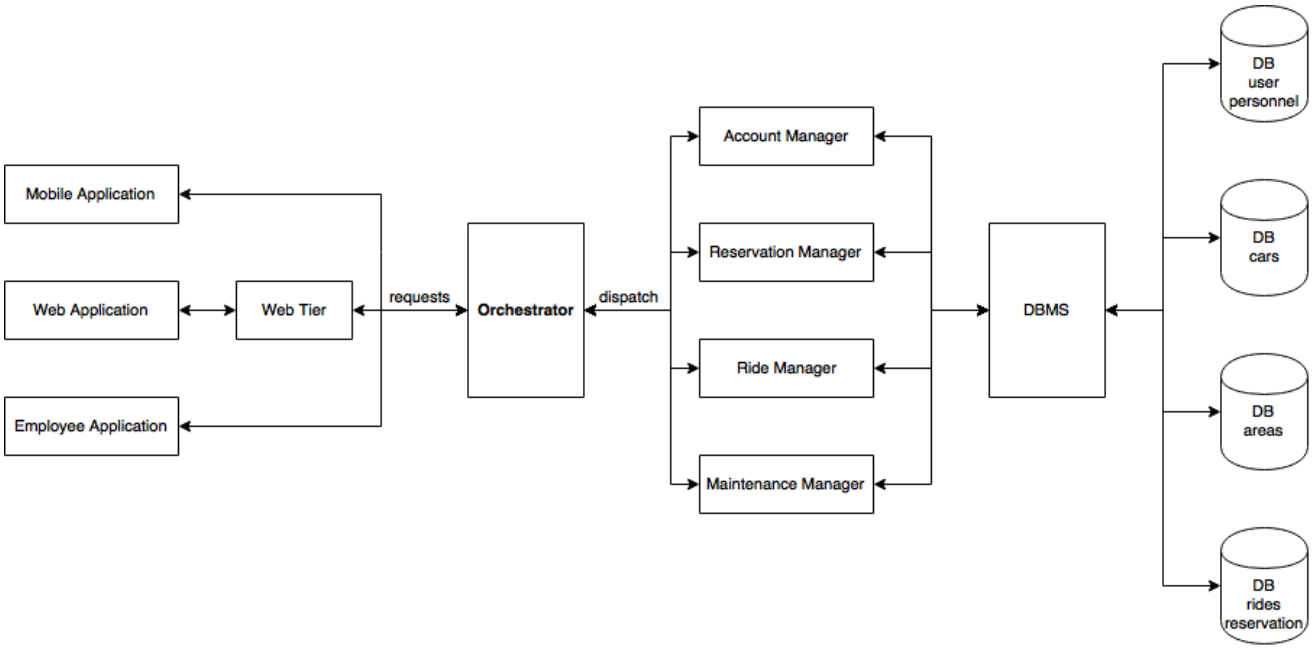


Figure 2.4: A high-level representation of client-server interaction w/ orchestrator.

Account Manager

The task of this module deals with account editing and information retrieval. It interacts with the "user and personnel" DB accessing and modifying the "user" entity.

It allows to log in/register to the system, consult/modify personal information, delete profile.

Reservation Manager

The task of this module deals with car reservation.

It interacts with the "rides and reservation" DB creating new "reservation" entities every time a new reservation is made by a user.

It allows to look for a car, make a reservation, unlock cars managing the reservation logic.

Ride Manager

The task of this module is to manage rides. It has to compute the ride cost taking into account possible fare modifiers, i.e. discounts/extra charges.

It interacts with the "rides and reservation" DB creating new "ride" entities every time a new ride is started.

It allows to start a ride, even with money-saving option enabled, complete a ride, pay for the ride.

Maintenance Manager

This module allows PowerEnJoy to manage its employees so that every time a car needs maintenance a request for intervention is created.

It implements an Observable-Observer pattern to track battery levels and cars position.

It interacts with the "user and personnel" DB, "cars" DB, "areas" DB in order to find availbale employees, check battery levels and cars position, update cars states and employees' history.

It allows the system to dispatch requests to the nearest available employee, who can accept/decline.

- **Database**

MySQL Community Edition is used as RDBMS and InnoDB is used as storage engine. The database tier communicates with the application logic tier: object-relations mapping is managed using Java Persistence API. Access to data must be secure: only authorized users with valid credentials can retrieve their personal data. The conceptual schema of the database is represented by the following E-R diagram.

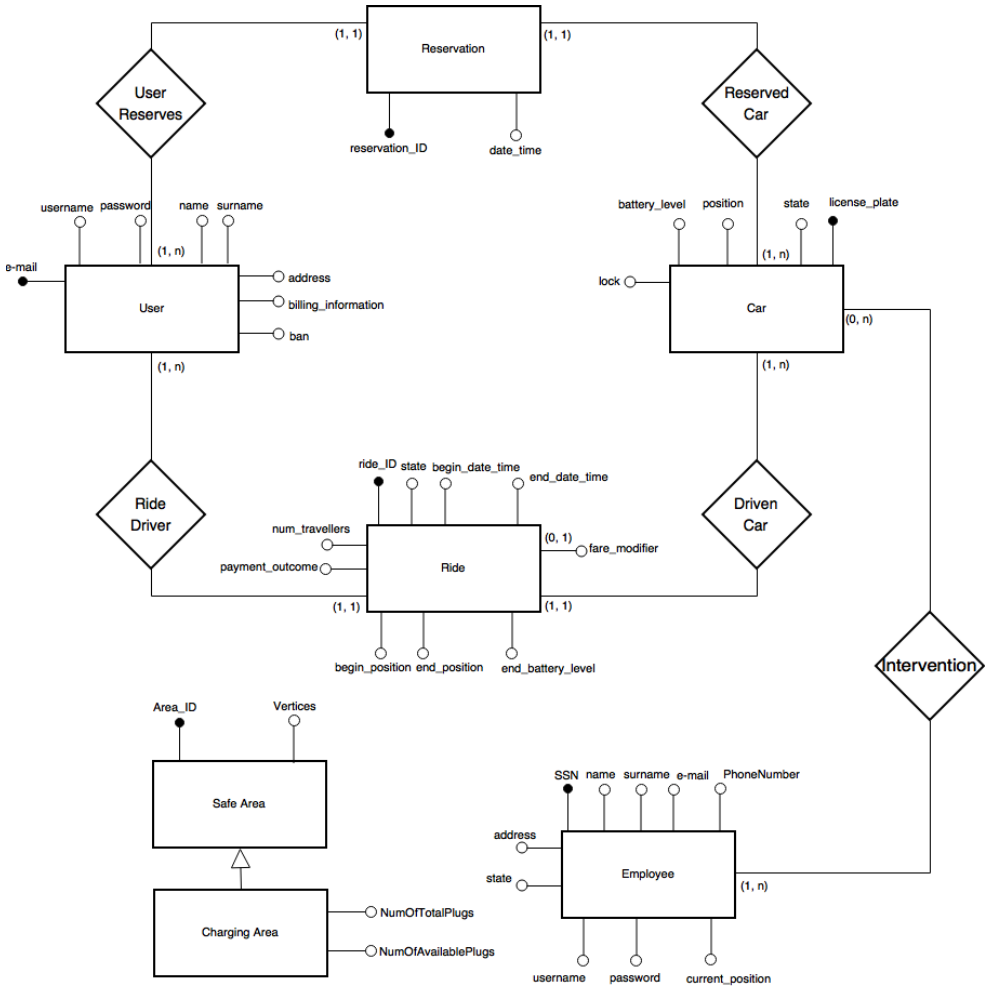


Figure 2.5: ER diagram of the database relations

Four subject-oriented databases are used to store data:

- User and Personnel DB
- Cars DB
- Areas DB
- Rides and Reservations DB

2.4 Deployment view

The allocation of software components to hardware components has to be as follows:

Web Server and Application Server are set on two different tiers.

Database Server is set on its own tier.

The communication with the application server uses JAX-RS to allow RESTful API client side. The application server interacts with the database server with SQL DBMS.

The following diagram represents the deployment of the system:

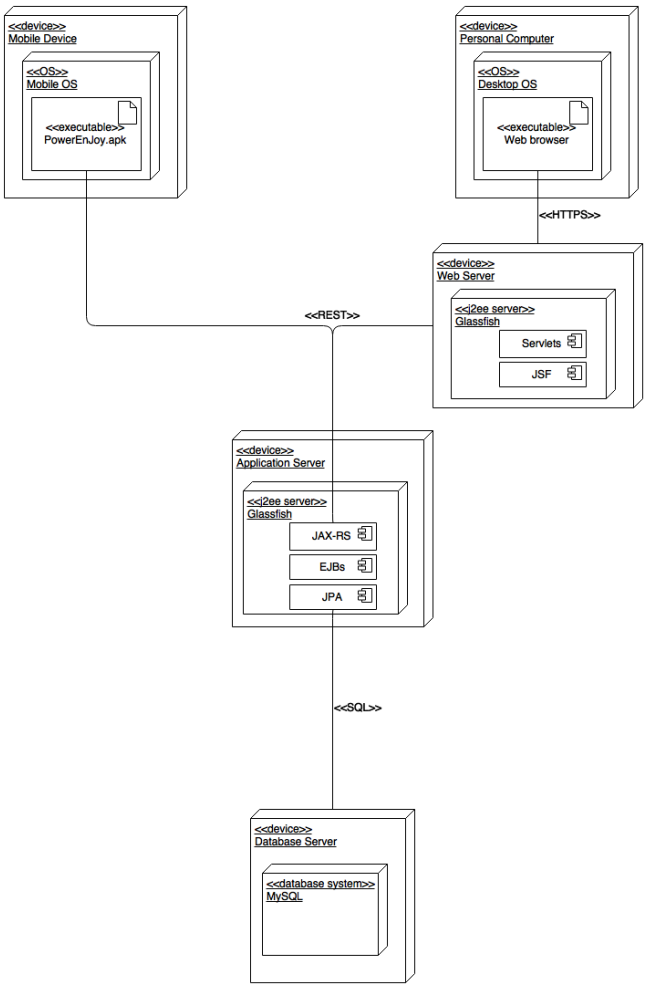


Figure 2.6: The deployment diagram of the system.

2.5 Runtime view

We represent the dynamic behaviour of the system with sequence diagrams. Each diagram shows a specific feature of the application.

2.5.1 Web Registration

This diagram shows the behaviour of the application when a user performs a registration action. We chose to represent the system dynamics in case of web application. The same diagram would state the mobile app behaviour if we do not consider the Registration servlet.

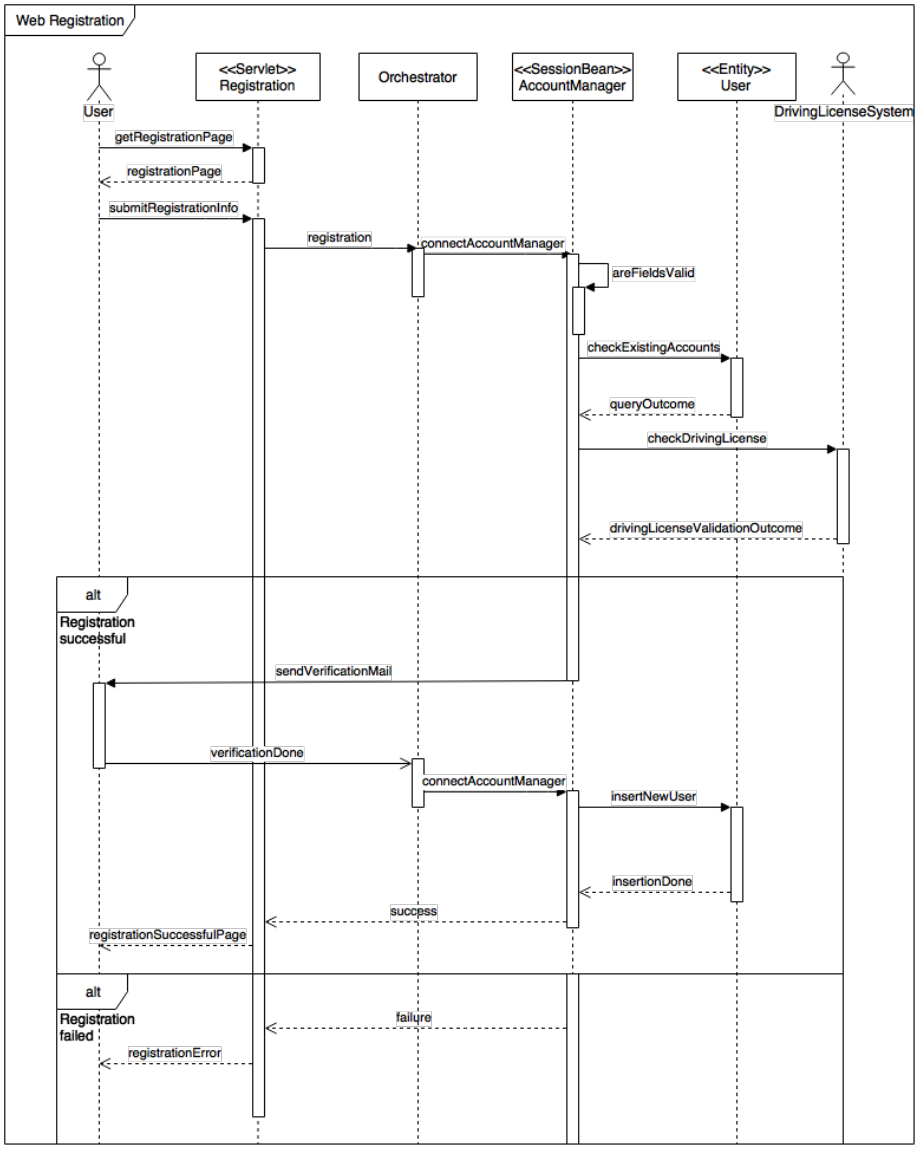


Figure 2.7: Sequence Diagram for registration action.

2.5.2 Web Login

This diagram shows the behaviour of the application when a user performs a login action. We chose to represent the system dynamics in case of web application. The same diagram would state the mobile app behaviour if we do not consider the Login servlet.

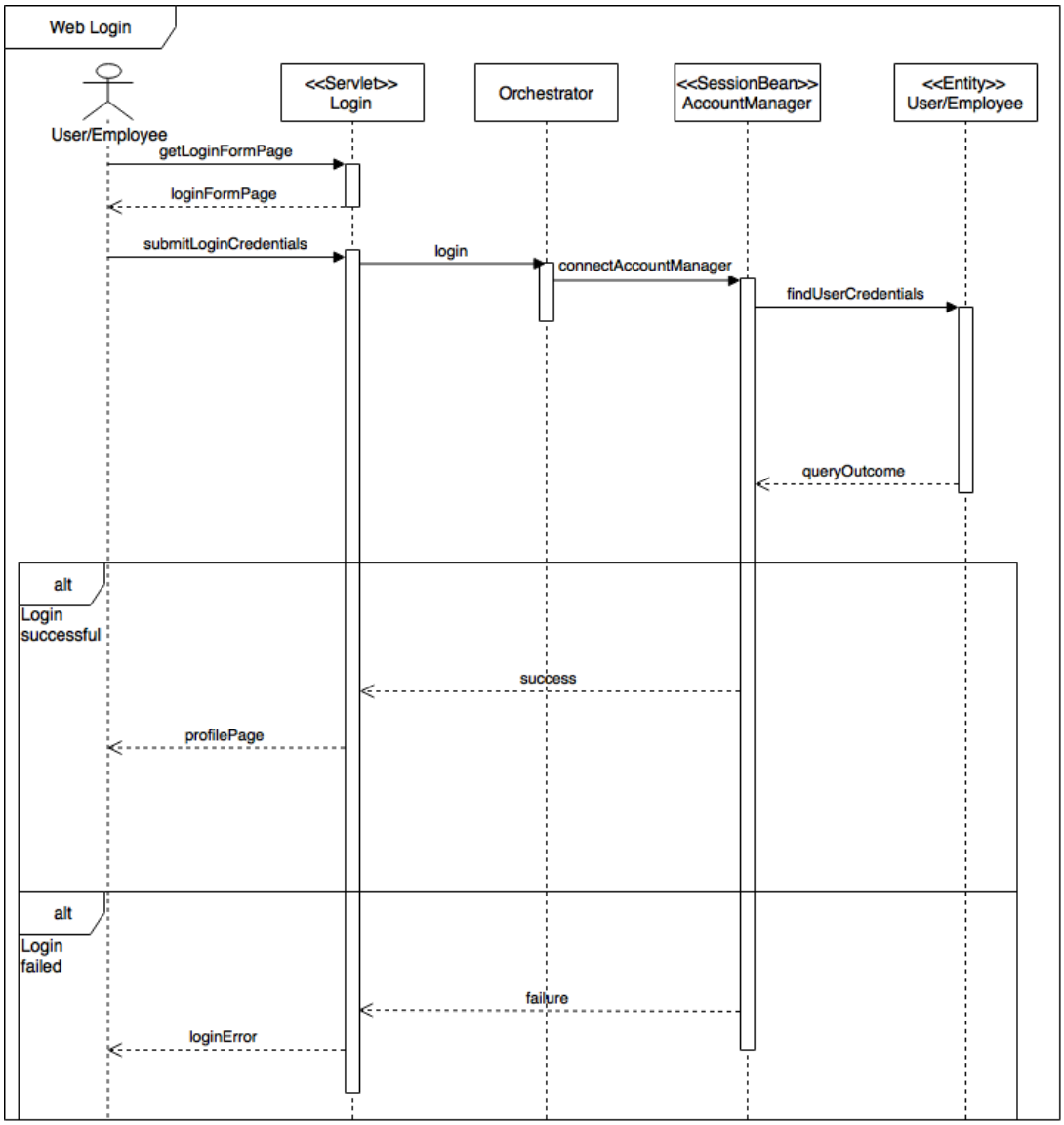


Figure 2.8: Sequence Diagram for login action.

2.5.3 Web lookup-reserve

The following diagram represents the behaviour of the application when a user looks for a car to reserve. We chose to represent the system dynamics in case of web application. The same diagram would state the mobile app behaviour if we do not consider the Lookup servlet.

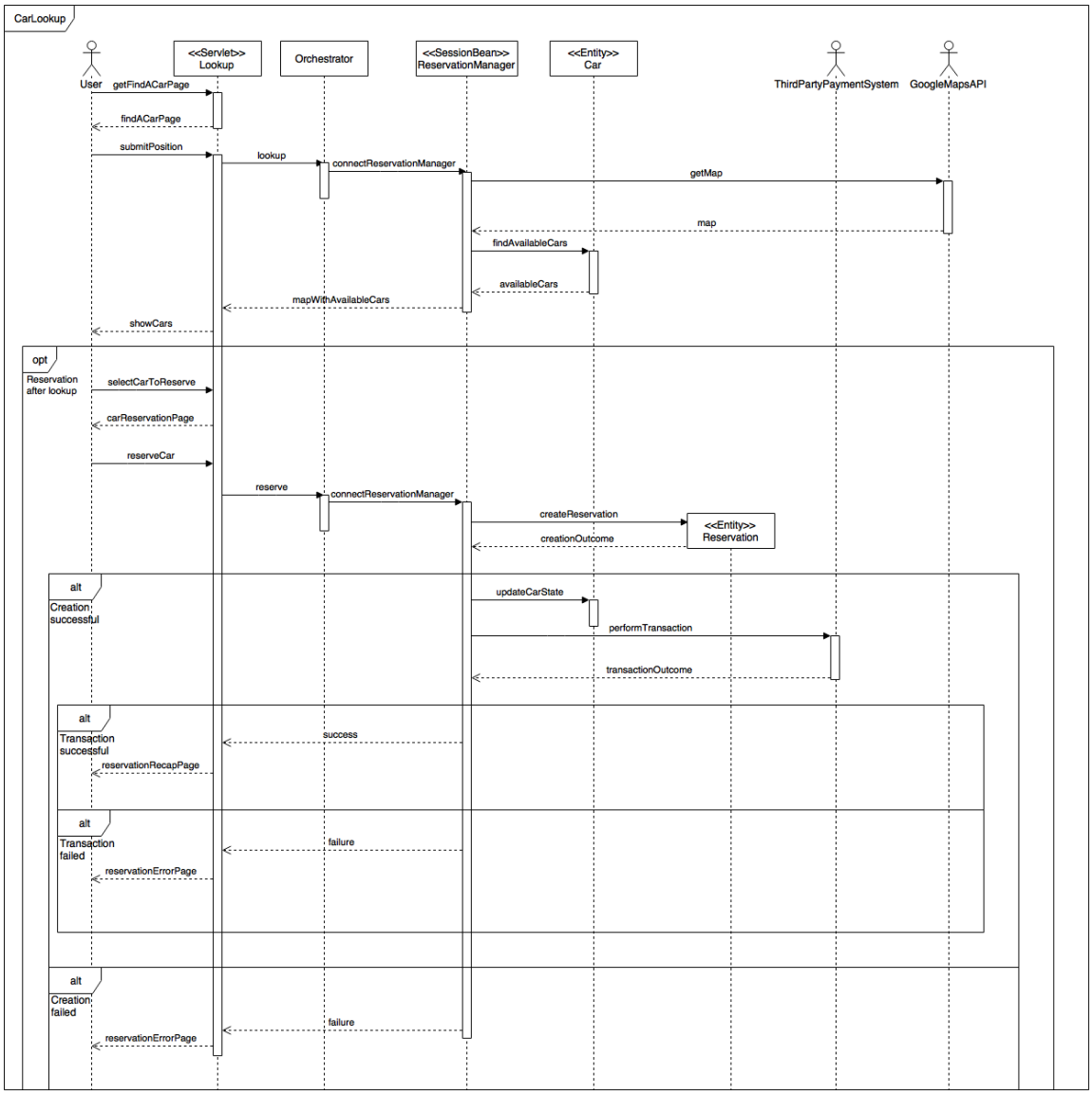


Figure 2.9: Sequence Diagram for car look-up and reservation.

2.5.4 Unlock/Lock and Ride

This diagram shows how the system behaves when a user wants to unlock a reserved car and perform a ride:

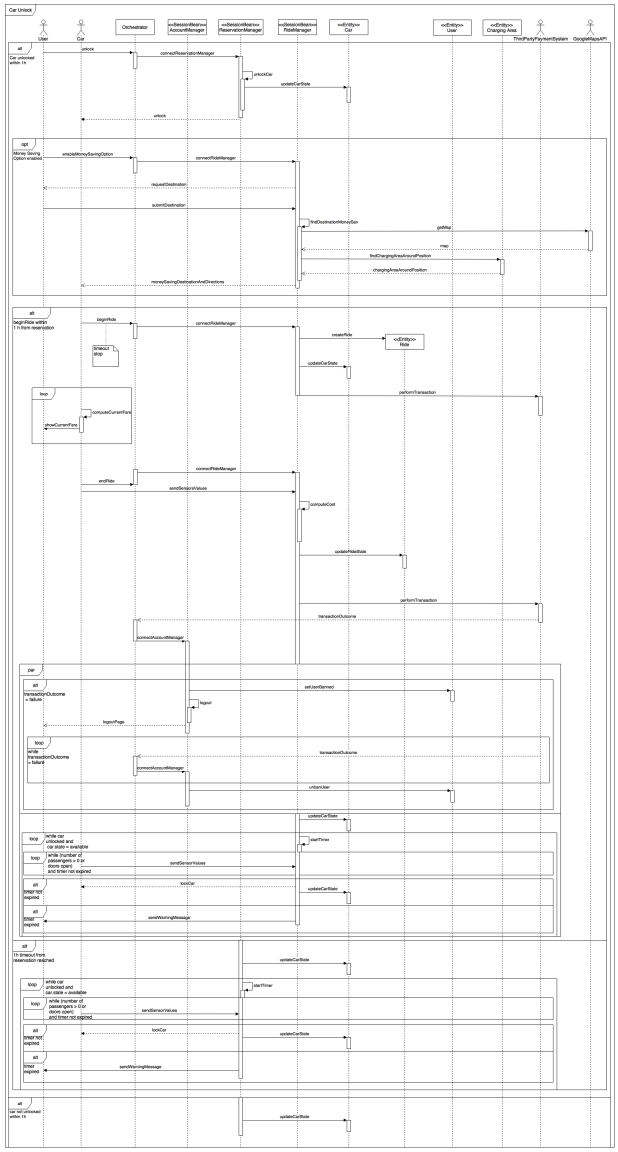


Figure 2.10: Sequence Diagram for car unlock and ride management.

2.5.5 Web User profile management

The following diagram shows the dynamics of the system in case a user wants to manage its personal information:

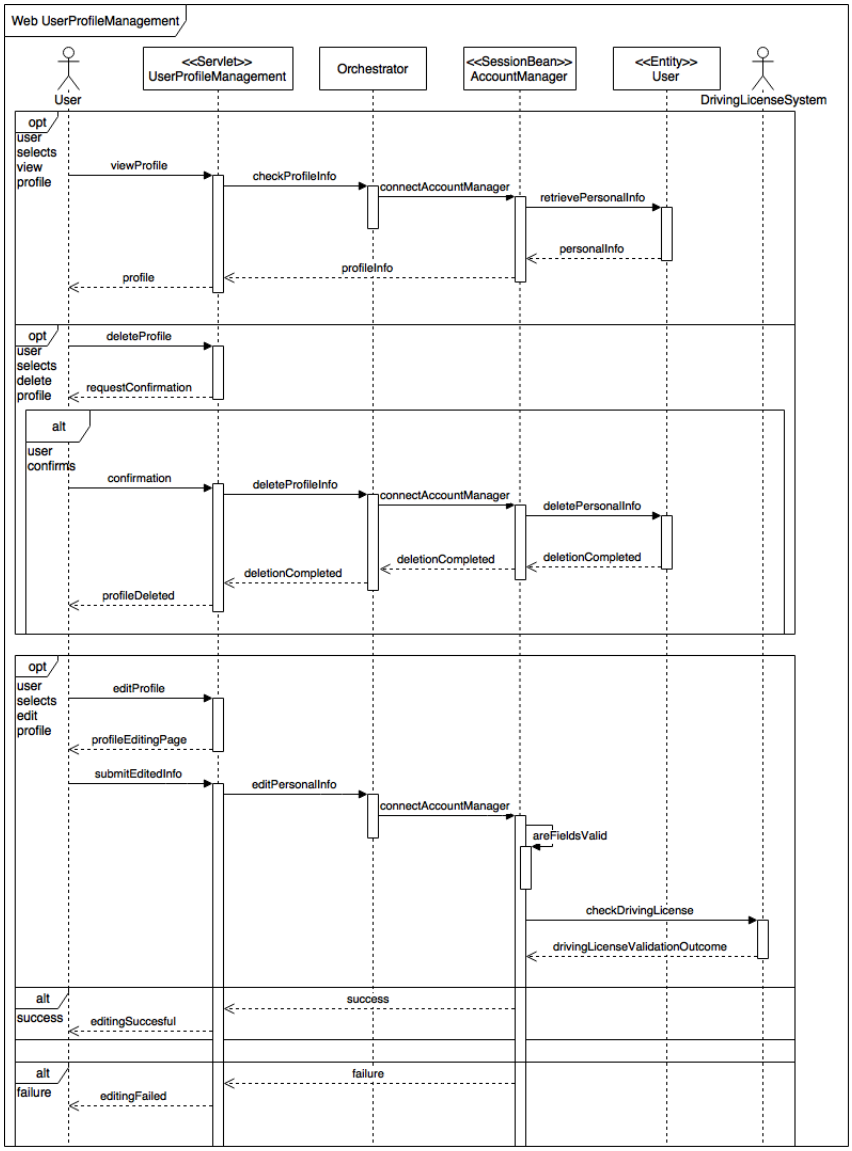


Figure 2.11: Sequence Diagram for profile info management.

2.5.6 Car Maintenance

This diagram represents the flow of actions performed by the system when cars maintenance is needed:

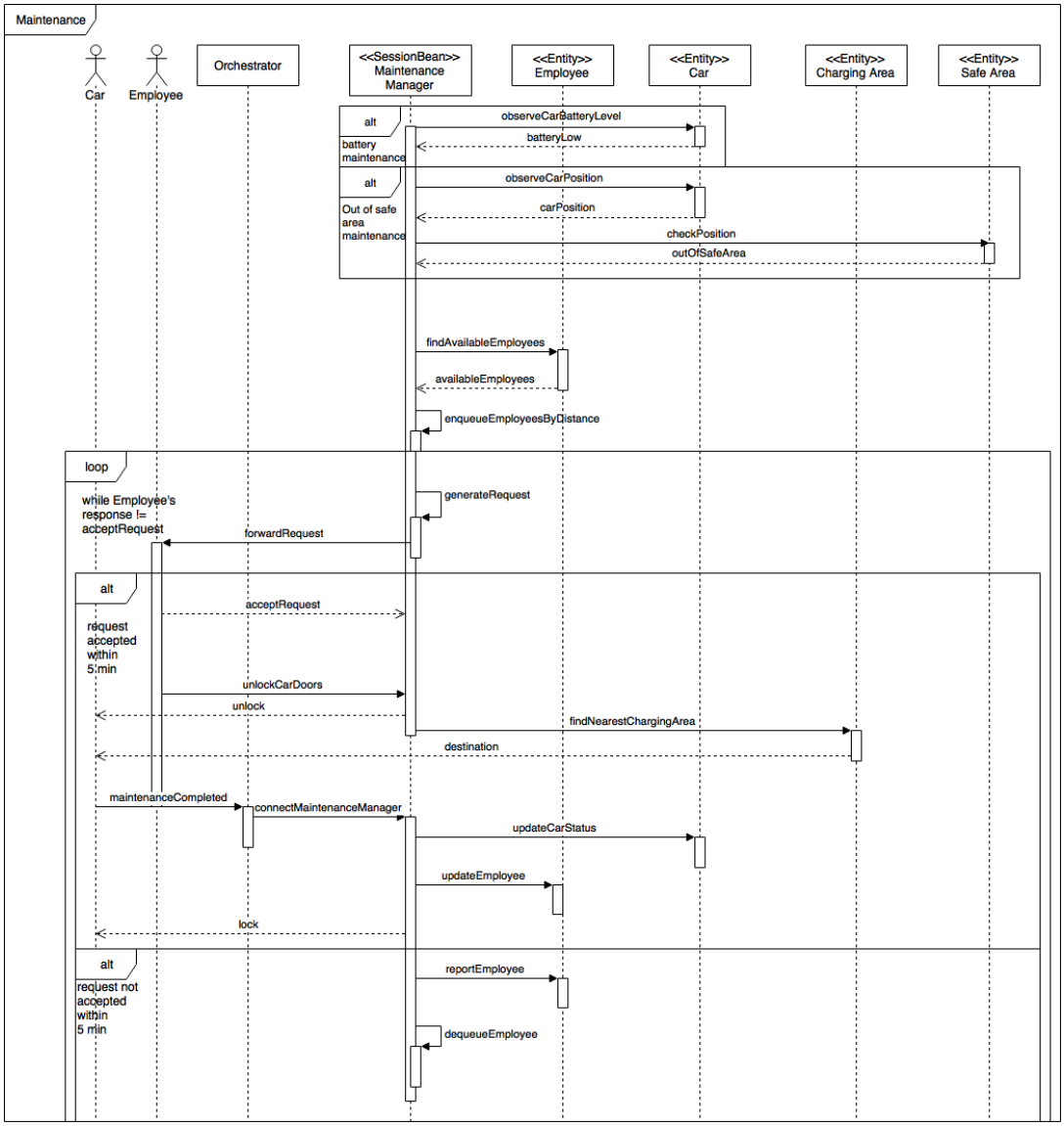


Figure 2.12: Sequence Diagram for car maintenance.

2.5.7 Database Maintenance

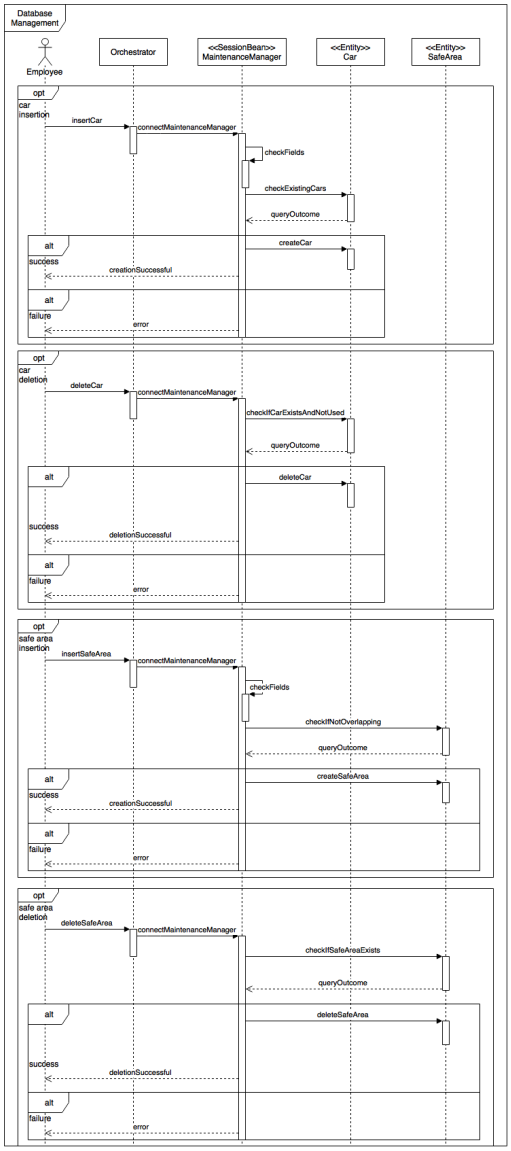


Figure 2.13: Sequence Diagram for cars insertion/deletion-safe areas insertion/deletion.

2.6 Component interfaces

2.6.1 Front-end to Application Server

The Mobile app and the Web App must communicate with the application server through a RESTful interface, which is implemented in the application server using JAX-RS. XML is used as the data representation language.

The REST resources and the offered functionalities are implemented as follows:

- Account Manager
  - **registration:** this function takes as input the personal information of the user and his/her payment information and driving license. It checks whether the input information is valid. If information is correct and no conflicts are spotted with already registered users a new User tuple is created in the database, otherwise an error message is returned.
  - **login:** this function takes as input a couple username/e-mail and password and checks whether there is an existing tuple in the database with the given username and password and the ban field set to false. If so, the access to the app services is granted. Otherwise, an error message is returned.
  - **viewProfile:** this function retrieves the user's profile information from the User relation in the database and returns them.
  - **editProfile:** this function allows the user to modify retrieved data: it takes as input the modified personal/payment information or driving license number. It checks whether provided information is valid and, if so, updates the corresponding user tuple in the database. Otherwise an error message is returned.
  - **deleteProfile:** this function deletes the tuple corresponding to the caller user in the database. It returns a confirmation message, then a final log-out is performed.
  - **ban:** this function takes as input a user and sets his/her "ban" attribute to true. It then performs a log-out to the user.
  - **unban:** this function takes as input a user and sets his/her "ban" attribute to false.
  - **generateEmployee:** this function generates special username and password for an employee and returns them.
- Reservation Manager
  - **findAvailableCars:** this function takes as input a position and queries the "Car" database in order to return all the cars whose state is "available" and whose position is near to position parameter.

- **reserve:** this function takes as input the user and a car and updates the selected car tuple in the database changing the attribute "state" to "reserved". It creates a new "reservation" tuple in the database and calls for the digital payment service to perform the pre-reservation deposit. It eventually returns a confirmation message once the operation is accomplished. If the operation fails an error message is returned.
- **unlock:** this function takes as input a car and updates the selected car tuple in the database setting its "lock" attribute to false.

• Ride Manager

- **createRide:** this function takes as input a car and a user and creates a new ride tuple in the database whose attribute "state" is "active". It calls for the digital payment service to refund the pre-reservation deposit.
- **computeMoneySavDest:** this function takes as input the location specified by the user and returns the optimal destination, i.e. the location of a charging area with optimal:
  * distance from destination parameter
  * number of available plugs
  * distribution of cars in the city
- **computeCost:** this function takes as input the time of the ride and car sensors values and returns the cost the user must pay considering fare modifiers.
- **terminateRide:** this function takes as input a ride and updates its state to "completed". It calls for the digital payment service to perform the payment transaction. If the operation is successful, then a confirmation message is returnes. Otherwise, it calls for the account manager to ban the user and an error message is returned.
- **lock:** this function takes as input a car and updates the selected car tuple in the database setting its "lock" attribute to true.

• Maintenance Manager

- **findAvailableEmployees:** this function takes as input a car and queries the database in order to returns a collection of employees whose attribute "state" is "available", ordered by distance from the car.
- **generateRequest:** this function takes as input a car and an employee and generates an intervention message to send to the employee.
- **forwardRequest:** this function takes as input an intervention request and sends a notification message to the employee.
- **findDestination:** this function takes as input the GPS location of the car and returns the optimal destination, i.e. the location of the nearest charging area to the car.

2.6.2 Browser to Web Server

All the communications between users' browser and web server are implemented with HTTPS requests, thus all the traffic must be protected with SSL encryption.

2.6.3 Application Server to Database

The communication between the application server and the DBMS must be via Java Persistence API (JPA). Thus, object-relation mapping is performed by JPA and JPQL (Java Persistence Query Language) allows to write queries that work regardless of the underlying data store.

2.7 Selected architectural styles and patterns

The following architectural styles have been used:

Client-server: the front end (mobile app and employee app) are clients communicating with the application server. The browser supporting the web app is a client communicating with the web server. The application server behaves as a client quering the database (server).

Service-Oriented Architecture (SOA): the way components interact with the application server is thought to be service-oriented. The single components are analysed from a high-level point of view depending on the service the offer. SOA allows to easily extend the system by building and adding independent modules to the core. Moreover, being the modules decoupled, their testing can be carried out independently module by module.

Model-View-Controller (MVC): MVC pattern is followed throughout the whole system design. The clients are front-end components (views) interacting with logic components (controllers) which drive the information flow and the information retrieval from the database (model).

2.8 Other design decisions

2.8.1 Password storage

Users' passwords are not stored in plaintext, but they are hashed and salted with strong cryptographic hash functions.

2.8.2 Maps

The system uses an external service, *Google Maps*, to calculate distances and visualize maps. The reasons of this choice are the following:

- manually developing maps is not a viable solution due to the enormous effort required to code and collect data;
- *Google Maps* is a well-established, tested and reliable software component already used by millions of people around the world, and therefore the users feel comfortable with a software they know and use everyday;

- *Google Maps* offers APIs, enabling programmatic access to its features, and can be used both on the server side (calculations, shortest paths...) and on the client side (map visualization).

3 Algorithm Design

3.1 Final price calculation with discounts and additional charges

As stated in the RASD, discounts are applied to the fare of users who have a virtuous behaviour, and additional charges are applied to users who have an inconvenient behaviour instead.

Specifically, the discounts and additional charges pre-defined by the systems are:

- 30% discount if the car is plugged into a plug within a 2 minutes time frame from the moment the engine stops.
- 20% discount if the battery level at the end of the ride is measured to be at least 50% of the total charge.
- 10% discount if the number of people in the car during the ride is measured to be 3 or more.
- 30% additional charge if the location of the car is more than 3 km away from the nearest charging area or if the battery level is lower than 20% of the total charge.

This algorithm is run by the back-end every time the state of a ride is set to completed. The state of a ride is automatically set to completed as soon as the engine is stopped by the driver.

The input data are:

- r: the ride which has to be checked for discount or additional charge eligibility
- D: list of discounts
- A: list of additional charges

The algorithm returns the final price the user has to be charged with.

The price per second is supposed to be a system constant.

A discount has an integer value between 1 and 100 representing the value of the discount in percentage. Similarly, an additional charge has an integer value between the minimum representable integer value and -1.

Discount/additional charge applicability can be checked by calling a method on the modifier itself, that takes r (the ride) as input and returns *true* or *false* depending on whether that modifier is applicable or not.

In order to make the algorithm as reusable as possible, and ease the future insertion of new discounts/additional charges or the removal of existing ones, the algorithm iterates over a generic list of discounts, and a generic list of additional charges.

The algorithm works as follows:

1. D is sorted by descending value (e.g. 30, 20, 10...).
2. The standard price is calculated as:

$$price = (r.endDate - r.beginDate) * PRICEPERSECOND \quad (3.1)$$

3. For every discount d_i in D:
   (a) if the discount is applicable on r:
      i. update *price* according to this formula:

$$price = price * (1 - \frac{d_i.value}{100}) \quad (3.2)$$

      ii. exit the loop.
4. For every additional charge a_i in A:
   (a) if the additional charge is applicable on r:
      i. update *price* according to formula (3.2).
5. Return *price*, which is the final price the user has to be charged with.

Notice that every additional charge is applied, while only the largest discount is applied. This is achieved by sorting the discount list by descending value at step 1 of the algorithm and exiting the loop as soon as the first discount of that list is applied.

Also notice that because of the way discount and additional charge values are represented, the formula for calculating the updated price is the same in both cases and can be isolated in a separate function.

3.2 Money saving ride destination calculation

As stated in the RASD, money saving option allows the driver to specify the destination of the ride. The system will then suggest an optimal charging area to park at, in order to get a discount on the fare.

The applied discount would be the one already described in Section 3.1, Final price calculation with discounts and additional charges, for plugging the car within a 2 minutes time frame from the moment the engine stops.

The money saving option serves the purpose of simplifying the operation of driving to a charging area, because such an area is suggested to the user.

The suggested charging area has to be optimal depending on:

1. closeness to the user-specified destination;
2. availability of plugs;
3. uniform distribution of cars in the city.

This algorithm is run by the back-end every time the user enables the money saving option and inputs the destination that he/she wants to reach.

The input data are:

- *dest*: the coordinates of the location the user wants to reach (latitude, longitude).
- A: list of charging areas.
- w_d: the weight associated to the distance of the charging area from the user-specified destination.
- w_p: the weight associated to the availability of plugs at the charging area.
- w_c: the weight associated to the number of cars near the charging area.

The algorithm returns the charging area which is calculated to be the best w.r.t. the three constraints stated above.

The default value of w_d and w_c is 1, and the default value of w_p is 0.5. Different weight values can be used for increasing/reducing the influence of a certain factor (e.g. $w_d = 2$ means that the distance of the charging area from the user-specified destination would impact the choice twice as much as with the default value).

The number of cars which are "near" a certain charging area can be calculated by calling a specific method on the charging area itself. The method returns the number of cars which are located within a certain radius from the charging area (including the ones which are plugged in the charging area itself).

The algorithm works as follows:

1. Initialize two variables:
   (a) a, the charging area to be returned (initially null).
   (b) v, the value associated with a (initially$+\infty$)
2. For every charging area a_i in A:
   (a) if the charging area is full, meaning that $a_i.availablePlugs = 0$: skip to the next charging area; else calculate:

$$p = a_i.totalPlugs - a_i.availablePlugs \tag{3.3}$$

   which is the number of busy plugs at the charging area a_i.

(b) calculate the distance d between *dest* and a_i using a suitable formula for calculating distances between two Latitude/Longitude points, for example the Haversine formula. [1] [2].

(c) calculate the number c of cars near a_i by calling the method mentioned above.

(d) calculate the value v_i associated with a_i according to the following formula (weighted sum):

$$v_i = \frac{w_d d + w_p p + w_c c}{w_d + w_p + w_c} \tag{3.4}$$

(e) if $v_i < v$:

i. update a and v with the current values of a_i and v_i respectively.

3. Return a.

3.2.1 Example

Let's imagine a possible scenario where a user chooses the money saving option and the algorithm has to determine the best charging area according to the factors mentioned above:

[1] Haversine formula: `https://en.wikipedia.org/wiki/Haversine_formula`

[2] Calculate distance, bearing and more between Latitude/Longitude points: `http://www.movable-type.co.uk/scripts/latlong.html`

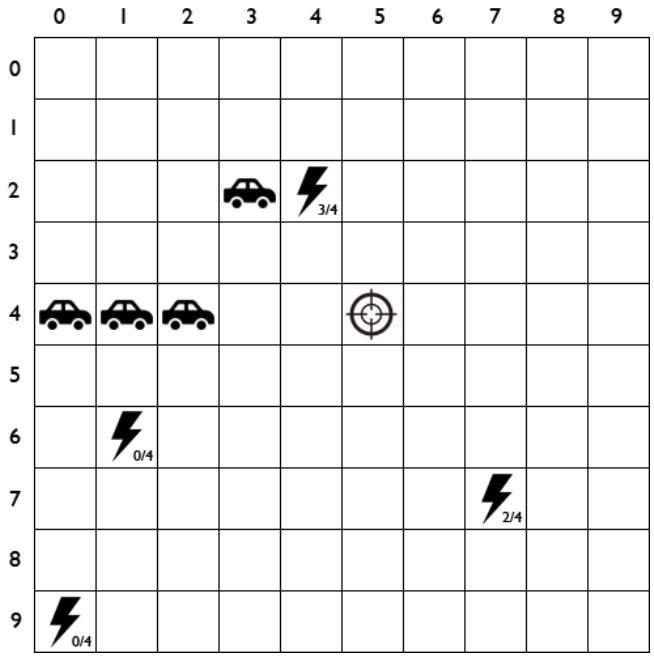


Figure 3.1: Simple example map showing position of user-specified destination, charging areas with number of busy and available plugs, and cars.

For the sake of simplicity, in this example the coordinates are not represented as Latitude/Longitude but as simple $(row, column)$ pairs.

Assuming the default values for w_d, w_p and w_c, and that the method calculating the number of cars which are "near" a certain charging area detects cars which are distant at most $\sqrt{5}$ units of distance, the intermediate values calculated by the algorithm at step 2 are:

Charging Area	d	c	p	**v**
$(9, 0)$	$\sqrt{50}$	0	0	2.828
$(6, 1)$	$\sqrt{20}$	3	0	2.989
$(2, 4)$	$\sqrt{5}$	4	3	3.094
$(7, 7)$	$\sqrt{13}$	2	2	2.642

The lowest value of v is the one associated with the charging area at $(7, 7)$, so that charging area is returned. It's not the nearest one, but it's much nearer than the one at $(9, 0)$ and it has only 2 cars nearby. The nearest charging area has 4 cars nearby and therefore it has a higher value.

4 User Interface Design

4.1 UX diagram

The UX diagram shows the different screens of the User Interface of the clients and the interaction between them.

The normal flow of a logged user who wants to reserve and use a car is:

1. look for a car on the map;
2. select the desired car and view car data;
3. confirm car reservation;
4. view reservation info such as the data of the reserved car, reservation date, and time left to unlock the car;
5. unlock the car from the reservation info screen.

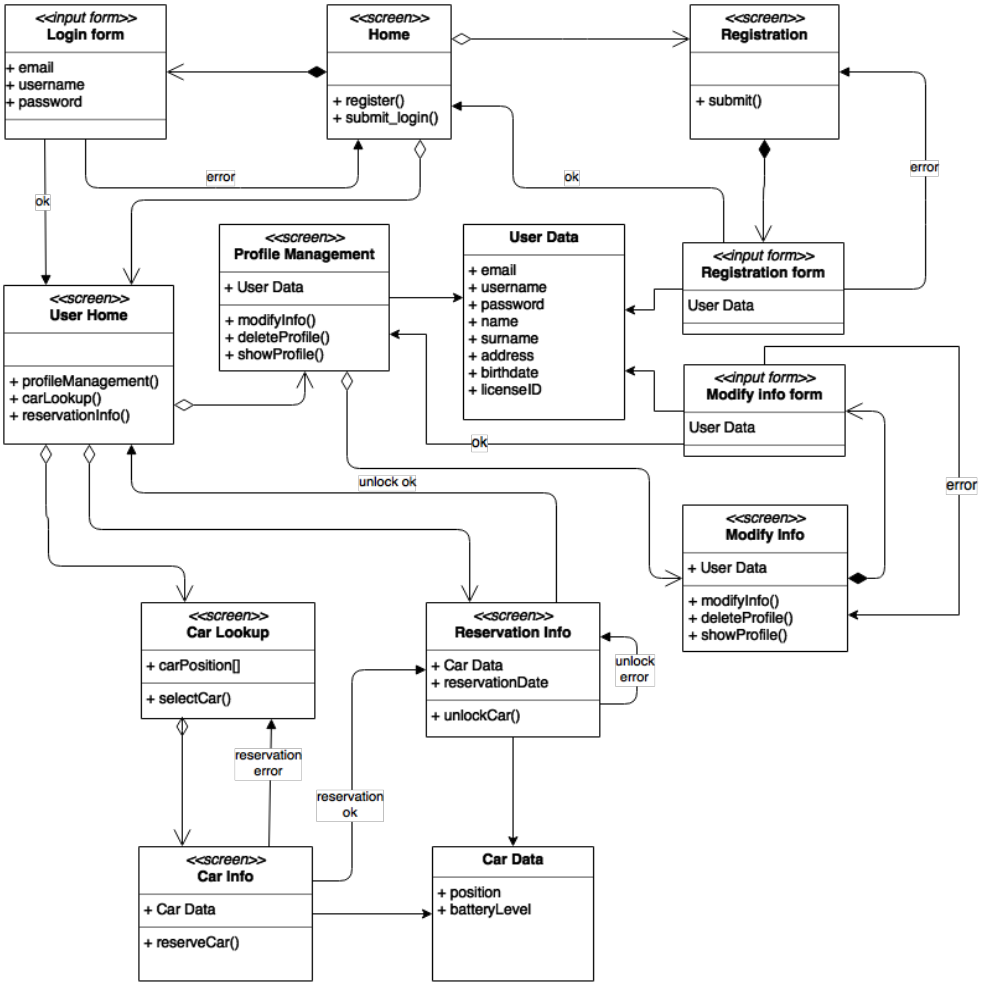


Figure 4.1: The UX diagram.

4.2 User Interface concept

4.2.1 Web interface

These mock-ups show how the user interface for the web application should look like on web browser.

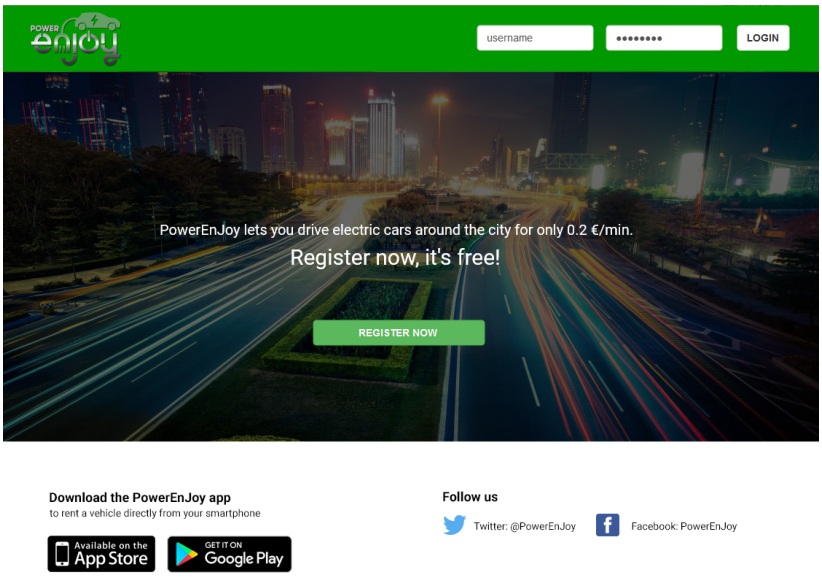


Figure 4.2: The home page of PowerEnJoy before log-in or registration.

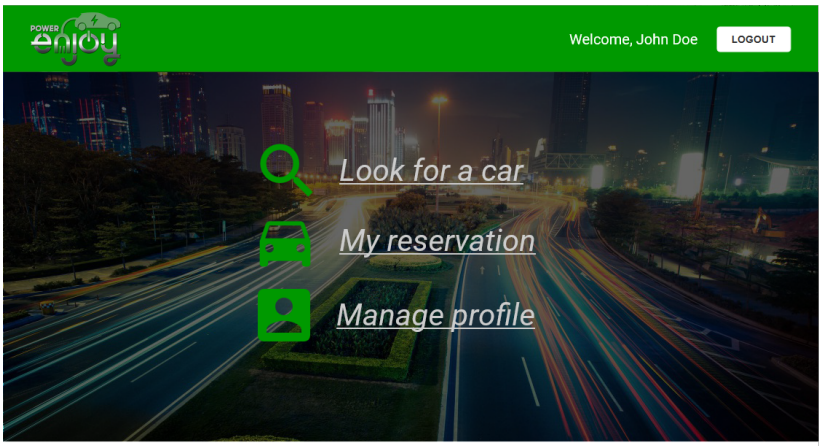


Figure 4.3: Home page for the logged in users linking to all the main functionalities in the web application.

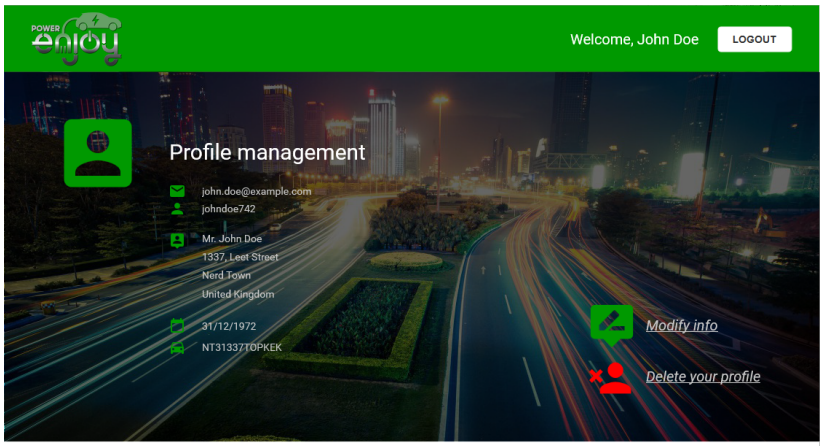


Figure 4.4: Profile management screen in the web application.

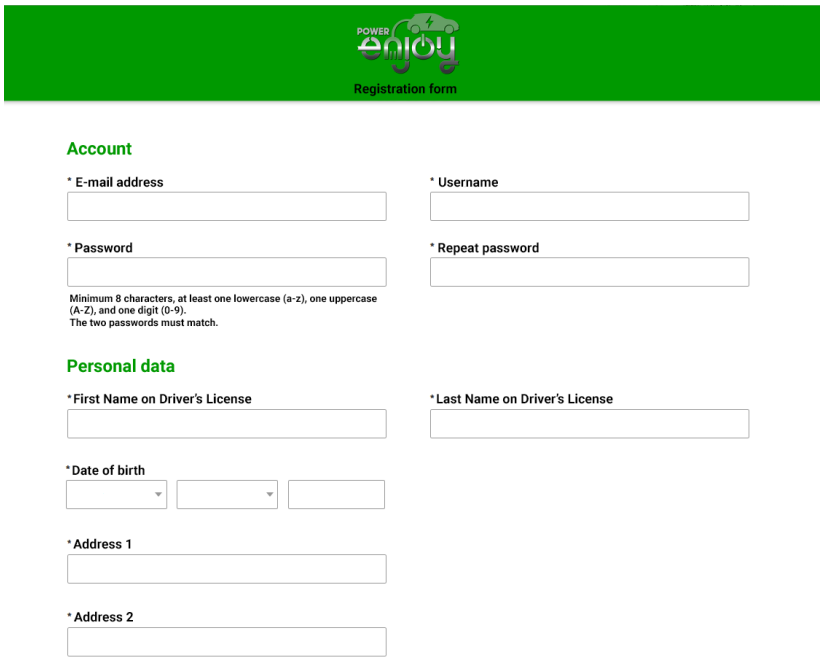
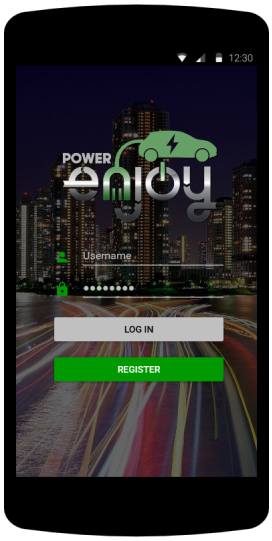


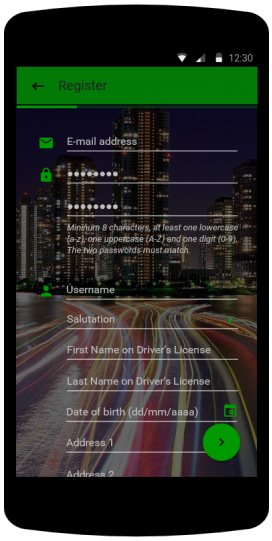
Figure 4.5: Registration screen for new users in the web application.

4.2.2 Mobile interface

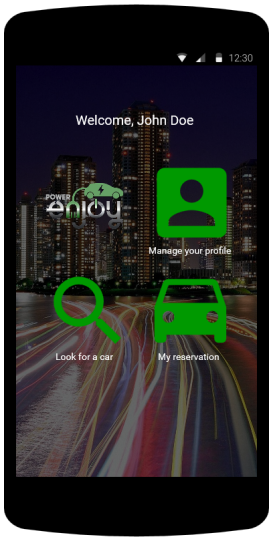
These mock-ups show how the interface of the PowerEnJoy mobile application will look like.



Initial login activity on the mobile application. This screen also presents the option for registering if the visitor does not have an account.

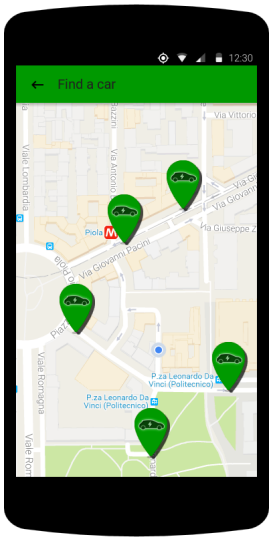


Registration activity for new users in the mobile application.

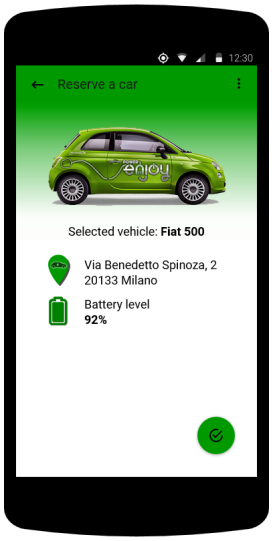


Initial activity for logged in users linking to all the main functionalities in the mobile application.

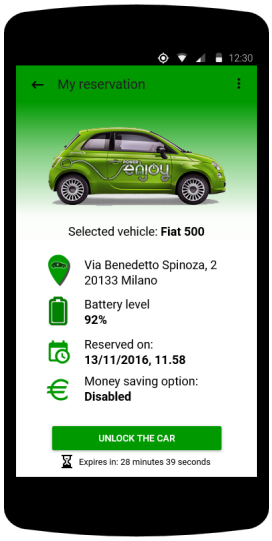
Profile management activity in the web application.



Car lookup activity in the mobile application. The car info is displayed when a marker on the map is clicked.



Car info activity in the mobile application. From this screen, the selected car can be reserved by clicking the green circular bottom right button.



My reservation activity in the mobile application. From this screen the car can be remotely unlocked by clicking the green button.

5 Requirements traceability

5.1 Functional requirements and components

The following table shows the mapping between specific requirements identified in section 3.2 of the RASD and business logic components described in the DD.

Account Manager	R1. On registration, a person must be asked: e-mail address, username, pass- word, name, surname, birth date, address, phone number, driving license number, billing information. R2. There mustn't be another user already registered with the same e-mail address, username, or driving license number. R3. The password must be at least 8 characters long and it must contain at least one lowercase character, one uppercase character, and one digit. R4. The password must be asked twice. The two passwords must match. R5. On login, the system must grant the user access to the account if and only if the following conditions are met: • (a) The inserted username corresponds to a username of an existing user, or the inserted e-mail corresponds to the registration e-mail of an existing user. • (b) The inserted password matches with that of the user identified above. R6. If the entered password is wrong, a new attempt can be made only 10 seconds later. R32. If the payment operation of a user fails due to insuffcient funds, then that user is banned until he/she pays it off. R33. If a user is banned, the system must prevent him/her from having access to the service. R34. The user must be able to view the information in his/her profile. R35. The user must be able to edit the information in his/her profile, according to information fields' constraints. R36. The user must be able to delete his/her profile. R37. The system must generate special credential for employees. R40. The system must store users' information into a database.

Reservation Manager	R7. The user can choose to find cars around his/her position, or around a specied address. R8. The user must be shown a map of the selected position, on which the locations of available cars are marked. R9. The system charges the user in advance with 1 EUR deposit. The deposit is given back to the user if he turns on the engine. R10. If an error occurs while charging the user with the down payment then an error message is displayed and the reservation operation is aborted. R11. The user must be able to click on a marked location on the map in order to view additional information on the car at that position. R12. The user must be able to view at least the exact address and the battery level of the selected car. R13. The user must be able to reserve the selected car by clicking on a button in the additional information screen. After the button has been clicked, the car state must be set to RESERVED and a new reservation must be created associated to the user and the selected car. R14. The user must not be able to reserve the selected car if there already exists a reservation made by the user himself/herself. R15. Any user who reserved a car in the last hour must be able to unlock the car he/she has reserved in one of the following ways: • (a) If the reservation has been made via mobile app, the user must be able to unlock the car by clicking on a button in the reservation screen of the mobile app. • (b) If the reservation has been made via web app, the user must be able to unlock the car by sending an SMS. R28. Any user who doesn't ignite the engine of the car he/she has reserved within an hour is charged with a fee of 1 EUR and his/her reservation is deleted.

Ride Manager	R17. The system must keep track of the time spent from when the engine turns on until the engine turns off. R18. At the end of the ride the system waits for possible discounts to be applied to the final fare, and then interacts with the third party payment service and notifies the user with the report of the payment operation. R19. The system locks the car if it detects that nobody is inside the vehicle and doors are closed. R20. If the system fails in locking the car (e.g. doors are not closed) then a warning message is sent to the respective user. R21. If a user unlocks a locked car from the inside, then the system locks the car again. R22. If the car is detected to be at a charging area, then the system must wait for a fixed timeout of 2 minutes or until the car is plugged before charging the user. R23. If the number of people inside the car is measured to be 3 or more, the system must apply a 10% discount on the final fare. R24. If the battery level at the end of the ride is measured to be at least 50% of the total charge, the system must apply a 20% discount on the final fare. R25. If the position of the car at the end of the ride is a charging area, and the car is plugged in within a 2 minutes time frame, the system must apply a 30% discount on the final fare. R26. If a ride is eligible for multiple discounts, only the largest is effectively applied to the final fare. R27. If the position of the car at the end of the ride is more than 3 km away from the nearest charging area or if the battery level is lower than 20% of the total charge, the system must apply a 30% additional charge on the final fare. R29. The user must be able to enable a money saving option as soon as he/she enters the car, by entering a specific destination. R30. The system estimates at which charging area the user has to park the car, depending on the specified destination and plug availability, in order to achieve a uniform distribution of cars. R31. The optimal route to reach the charging area is displayed to the user.

Maintenance Manager	R16. The system must inform an employee when a car is in low battery state. R38. Employees' account can lock and unlock every car. R39. Employees must be able to access an area where cars and safe areas can be inserted to/deleted from the database.

5.2 Non-functional requirements

Availability	The architecture we design does not take into account partitions or duplications of the application server, so we can ensure no more than the 98% availability we stated in the RASD.
Reliability	All the information stored in the database must be periodically dumped into a back-up server.
Maintainability	The SOA architecture grants high modules decoupling and facilitates future extensions.
Security	Encryption algorithms, like SSL, are used.
Portability	The architecture models are based on the J2EE framework. Java code can be ported across multiple platforms, thus Java driven Web Services may be developed on one platform, but deployed and executed on another.

A Appendix A

A.1 Software and tools used

- LaTeX for typesetting the document.
- LyX as a document processor.
- GitHub for version control and teamwork.
- Pencil for mockups.
- Draw.io for UML diagrams.
- Signavio for UML diagrams.

A.2 Hours of work

- Alessio Mongelluzzo: 35 hours
- Michele Ferri: 35 hours
- Mattia Maffioli: 35 hours