



Politecnico di Milano

AA 2016-2017

Software Engineering 2 project: "*Power Enjoy*"

Prof. Mottola Luca

DD

Design Document

11/12/2016 - version 2.0

Edited by:

Giulia Pennati **878686**

Paola Sanfilippo **809689**

Marco Siracusa **878083**

Contents

1. INTRODUCTION	4
A. Purpose.....	4
B. Scope.....	4
C. Definitions, Acronyms, Abbreviations	4
C.1. Definitions.....	4
C.2. Acronyms	6
D. Reference Documents	6
E. Document Structure	7
2. ARCHITECTURAL DESIGN	8
A. Overview	8
B. Component view.....	10
Component Diagram	10
Session Manager Explosion	11
C. Database logic view.....	13
D. Deployment view	14
E. Runtime view	16
E.1. Registration.....	16
E.2. Login.....	17
E.3. Reservation.....	18
E.4. Drive a car	19
E.5. Payment interaction	21
F. Component interfaces	22
G. Selected architectural styles and patterns.....	24
H. Other design decision.....	24
3. ALGORITHM DESIGN	25
4. USER INTERFACE DESIGN.....	29
B. UX diagrams.....	29
B.1. Desktop version.....	29
B.2. Mobile version.....	30
C. BCE diagrams.....	31
C.1 Desktop version.....	31
C.2. Mobile version.....	32
D. Ux diagram and component correspondence	33
D.1. Primary Home	33
D.2. User Home/Payment History/Reservation History	34
D.3. Registration	35
D.4. Research a car	36
D.5. Reserve a car	36
D.5. Unlock a car	37
5. REQUIREMENTS TRACEABILITY.....	38
6. EFFORT SPENT	39
7. REFERENCES	40

1. INTRODUCTION

A. Purpose

The Design Document's purpose is to give more detailed instructions of how PowerEnjoy system should be built. It is addressed to PowerEnjoy's developers as it contains functional descriptions of the system with more technical aspects than the RASD has. It focuses mostly on the architectural and interfaces' design.

B. Scope

This project aims to realize an electric car sharing service.

This service makes available to the users an entire fleet of electric cars, power grid stations furnished with plugs to recharge the cars and a maintenance team ready to intervene in case of need.

It also provides both a website and a mobile application, with which the registered users can access the system functionalities.

In order to access these system functionalities, a person should first register, inserting all the required data (that has to be correct), and then wait until the reception of an email containing a password.

The email (s)he provided during the registration with the password received via email are his/her credentials to log into the system and access all the functionalities.

A logged in user can make a research of a car within a certain area, the "Safe Area", choosing the one that most fits his/her needs, and reserve the desired one.

(S)He also can use the car according to the PowerEnjoy company's rules (that will be explained in the next pages) and can both enjoy the different available discounts and be charged with extra fees, limited to a discount of 30% (i.e. $+charges-discounts \leq 30\%$ total discount).

Discounts and charges policy is thought to reward those behaviors that simplify the availableness of usable cars (i.e. in the right place and with a sufficient battery level) and discourage the others.

C. Definitions, Acronyms, Abbreviations

C.1. Definitions

[D01] **Visitor**: a person who has not already completed the registration phase. An inaccessible account is assigned to him/her.

[D02] **Account**: a record into the system that identifies each visitor/user and contains all his/her data.

Each account can be either "Not owing" or "Owing" depending on whether the user has carried out all payments or not.

[D03] **Registration phase**: procedure that starts with a visitor that sends the required data and ends receiving a password back from the system (being allowed to log into that and use all the provided functionalities).

[D04] **User**: any person that, consequently a registration phase, has received back the password and so is able, after the login phase, to access all the system functionalities.

[D05] **Login**: procedure with which a user (already registered), inserting his/her username and password, accesses into the system in order to use all the provide functionalities.

[D06] **Car**: any electric vehicle of the provided fleet. This vehicle is equipped with all the required sensors to provide the information necessary to the system.

The car can be:

- Not reserved: has not been reserved yet, so it's ready to be reserved
- Reserved: has already been reserved, so there's another user that has reserved the car or is using it

And

- Plugged (to a power grid station)
- Not plugged (to any power grid station)

Parts of the car that are named throughout the text are:

- Doors
- Trunk
- Battery
- Internal screen
- Terminal

A car provides an interface to communicate with the system.

[D07] **Reservation**: action that makes the cars change their state from "reserved" to "Not reserved" and vice versa. It **starts** when the user receives the confirmation of the requests of the car's reservation done through the mobile application and **ends** when the user unlocks the car

[D08] **Reservation-time**: time elapsed from the beginning to the end of the reservation.

[D09] **Locking**: action requested by the system that consists in the locking of the car's doors and trunk too.

[D10] **Unlocking**: action requested by the system that consist in the unlocking of the car's doors and trunk too.

[D11] **Deploy**: action that intends the effective use of the service that the user is charged for. It **starts** from the unlocking of the car and **ends** on the locking.

[D12] **Deploy-time**: time elapsed from the beginning to the end of the deploy.

[D13] **Ride**: action that **starts** when the engine is switched on and **ends** when the engine is switched off.

[D14] **Ride-time**: time elapsed from the beginning to the end of the ride.

[D15] **Passenger**: person who, for a single ride, travels with the user inside the reserved car.

[D16] **Power grid station**: place equipped with company's plugs whereby the user can put the car on charge. These are placed only in the safe area and well distributed. Each one has a total number of plugs.

[D17] **Safe area**: places within the urban city area where the user is allowed to park the car in. It has been already defined before the deploy phase.

[D18] **Payment system**: external payment system that is supposed to carry out the transactions of the due amounts of money of the several users. It communicates with the developed system through an interface.

[D19] **Pledge**: Amount of money (100€) used as security for the fulfillment of the reservation debt. It is liable to forfeiture the damage of the car or the inability to afford the payment of the reservation.

[D20] **Data checking system**: system that checks the validity and completeness of the user's data. The feedback is positive if and only if the inserted data are completed with respect to the

required ones and valid with respect to the laws applied in the country where the system is running.

[D21] **Money saving option**: option that, when selected, drives the user to the power grid station that is nearest to the destination (s)he selected but with not less than half of the plugs available. If the user leaves and plugs the car in the suggested place, (s)he receives the discount.

[D22] **Maintenance team**: team that manages the cars that has been left with a percentage battery level less than 20%

[D23] **Research**: procedure that allows the user to see which cars (s)he can reserve. The showed cars have battery charge level more or equal 20%, are not reserved and within a 500m range from the user's current provided location or from an address specified.

[D24] **2 minutes wait**: time elapsed from the locking of the car and the moment in which the system checks whether the car has been plugged or not and computes the total amount of discounts and recharges

C.2. Acronyms

MSO : money saving option

PGS : Power Grid Station

REST (or ReST) : REST stands for **R**epresentational **S**tate **T**ransfer. It relies on a stateless, clientserver, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used. It is *an architecture style* for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

RMI : RMI stands for Remote Method Invocation and it is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

DBMS : Database Management System

DB : Database

JDBC : JDBC stands for Java Database Connectivity and it is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is part of the Java Standard Edition platform, from Oracle Corporation.

JVM : Java Virtual Machine

EJB : Enterprise Java Beans

JSP : Java Server Pages

D. Reference Documents

The RASD document, which has been produced in the previous stage of the project (about Requirements and Specification), is the only one document that here is referred at.

E. Document Structure

1. **INTRODUCTION** : this section contains a brief introduction to the Design Document, including a description of the purpose of this kind of document, the aim of the project and a glossary with the recurring terms and acronyms used during the project.
2. **ARCHITECTURAL DESIGN**
 - A. *Overview*: this part shows high level components and their interaction and how the system is divided in tiers.
 - B. *Component view* : component diagram that shows how the several components of the system are arranged
 - C. *Database logic view* : here we show a logic view of the database related to our system
 - D. *Deployment view* : deployment diagram that shows how and where the several components of the Power Enjoy system are placed
 - E. *Runtime view*: this chapter explains through different sequence diagrams how the system's component interact to accomplish the different tasks
 - F. *Component interfaces* : shows the interfaces of the components of the system
 - G. *Selected architectural styles and patterns*: contains explanation of the style and patterns adopted.
 - H. *Other design decisions*
3. **ALGORITHM DESIGN**: this chapter aims to specify some details about the system through the use of pseudocode that describes the flow of the events in certain particular situations
4. **USER INTERFACE DESIGN**: contains all the possible interaction between user and system in the website and mobile version of our software
5. **REQUIREMENTS TRACEABILITY**: here we map the requirements we've introduced in the RASD to the design elements of the DD

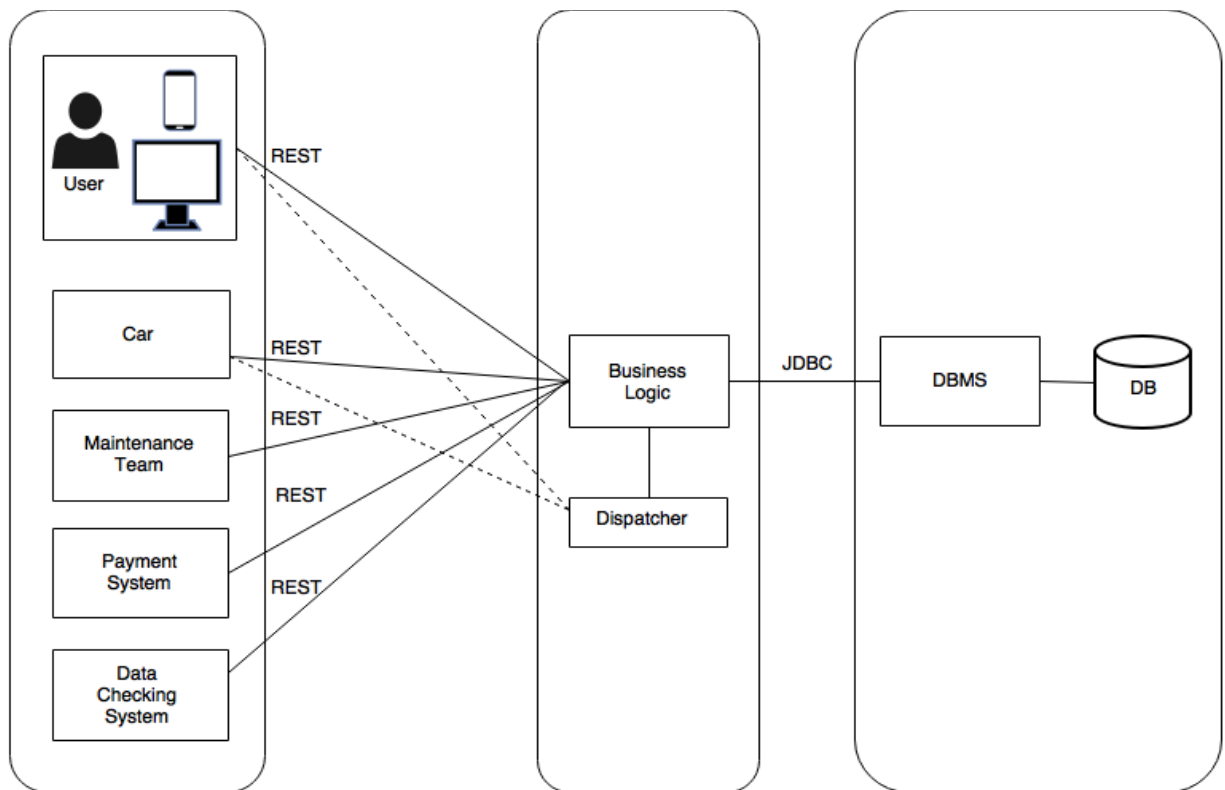
2. ARCHITECTURAL DESIGN

A. Overview

Going deeply into the details, the system has been thought as a client-server three-tier architecture where there are some event-based architecture features have been added.

The three tiers are:

- the client side logic, mainly composed of graphical interfaces for the (visitors and) users or software interfaces on the cars; not a great amount of logic here
- the application logic, placed inside the server, where all the required functionalities are managed; the whole logic is placed here
- the storage of the data useful for the system



Actually, the tiers would increase in number if there were considered the context having more payment methods because that would lead to have a tier manage the interaction with all the external systems for the payments.

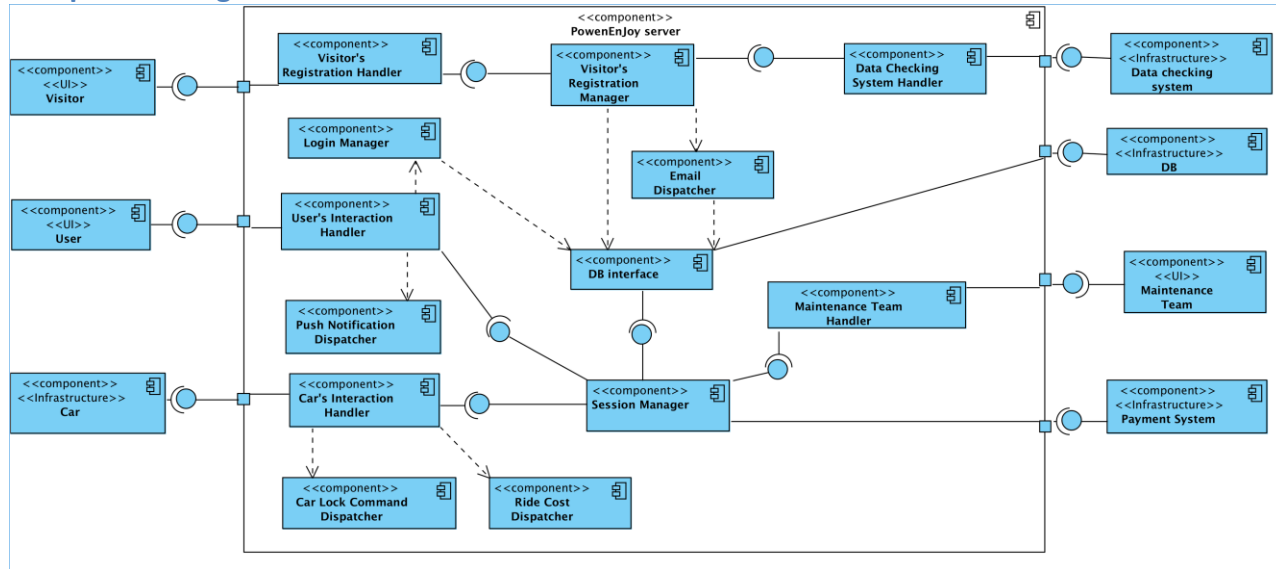
The *maintenance team*, *payment system* and *data checking system* (in the diagram above) are external systems seen by PowerEnjoy as servers and, since there should be also an asynchronous communication, those consider the PowerEnjoy system as a server too.

The server of the PowerEnjoy system hosts also the database and a dispatcher for cars and users. A dispatcher component has been added because cars and users are considered as clients and therefore they do not have a static address on the Internet and because, in certain situations, the system should send them some asynchronous messages.

B. Component view

The UML component diagram below shows how the several components of the system are arranged.

Component Diagram



The component named as *handlers* are both security component (as firewalls for example) for the incoming messages to the server and gateways for the outgoing messages to the clients (or other servers). The *user's interaction handler* is the first of these components and it manages to also send messages to the user without supposing a request-response procedure (session) ongoing; so does the *car's interaction handler*, obviously messaging the cars though.

Despite these two components are able to either send and receive messages, the *visitor's interaction handler* works only in the first way and the *maintenance team handler* in the second way only.

The only one of those *handler* elements that is slightly different is the *Payment handler*: indeed this component has some logic inside because, beside the standard role, can also handle an interaction with the *Payment system* triggering a request for the *DB Interface* component without passing through any Manager component.

Since, as just said, some messages are also sent from the server to the client without supposing a session ongoing, the system has been thought with an event-based architecture beside the main client-server one.

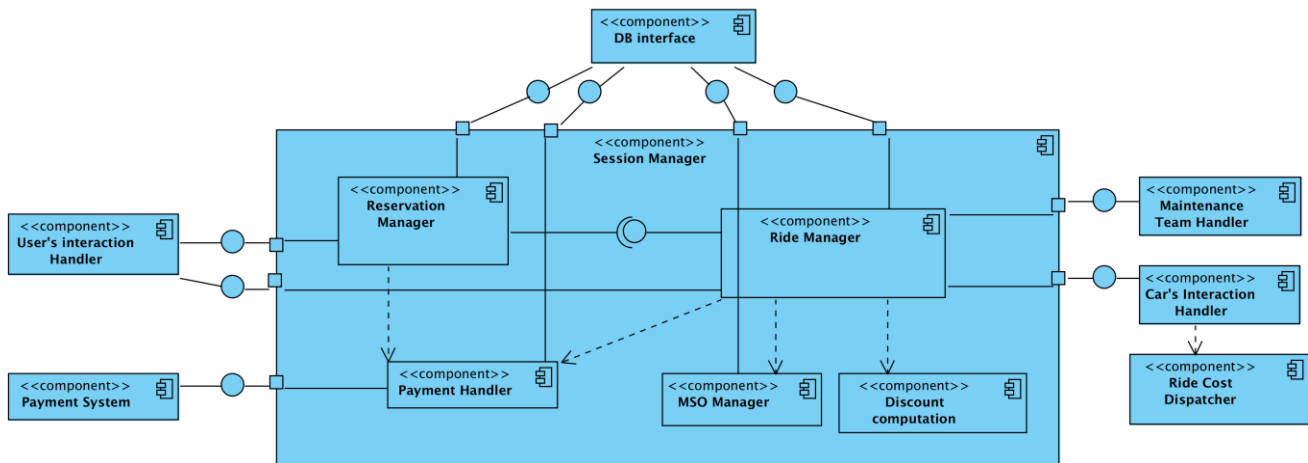
The *email dispatcher* is just a gateway for the email that should be sent to the users.

The interactions with the external systems are a server-server kind, which follows a client-server protocol.

The *Session manager* wraps up all the components needed to manage a session, starting from the reservation and going to the ride and its ending.

To go on into the details in a more formal way, here follows the component diagram drawing the explosion of the *Session manager* component.

Session Manager Explosion



List of Components

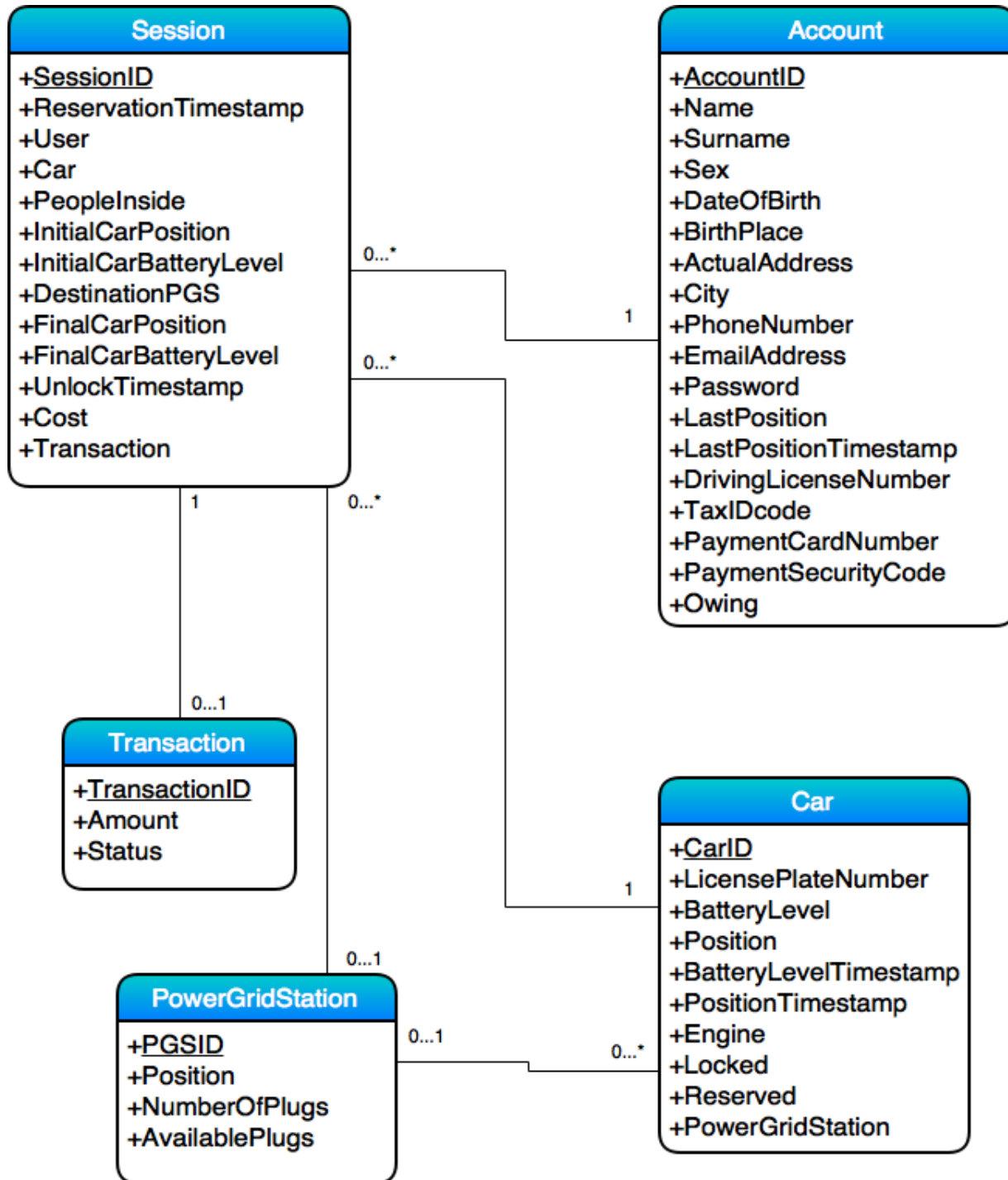
A list of all the components with a brief description is provided here:

- **Visitor's registration handler**
handles the interaction with the visitor (UI) submitting a request-response protocol, handling the registration requests of the visitors
- **Visitor's registration manager**
manages the visitor-registration requests, sending the data retrieved by the registration form to the Data Checking System and storing the data in the DataBase too. This component also manages to inform the user (by email through the **email dispatcher**) when its account is active (providing a password) or an error in the reservation is occurred (error message)
- **Data checking system handler**
handles the interaction within the Data Checking system and the PowerEnjoy system (properly the visitor's registration manager). The interactions are supposed to be either request-response (from the PowerEnjoy system to the Data checking system) style or using asynchronous messages (in the other way around)
- **DB interface**
manages all the interactions between the components and the DataBase, decoupling the system logic and the implementation technology of the database. The component also contains the elements of the model of the external environment
- **Email Dispatcher**
is supposed to dispatch the email to the users through an external email service.
- **User's interaction handler**
handles the interactions between the user (UI) and the PowerEnjoy system. The interactions from the user (UI) to the system are request-response based but the others in the other way are done through a event-based protocol (carried out by the **Push Notification Dispatcher**).

- **Login manager**
manages all the functionalities used during the login phase. The component just queries the DataBase through its interface and provide a feedback on whether the provided credentials are related to any account or not
- **Session manager**
Subsystem that manages all the functionalities used during the session phase (reservation + ride).
- **Reservation manager**
manages everything concerned the reservation of a car and therefore the storing of the data and the trigger of a payment request
- **Payment Handler**
interacts with the Payment system and changes the user status when a pending transaction is carried out
- **Car's interaction handler**
- interacts with the cars, receiving data and signals from them and sending that commands or information to be displayed to the user (carried out by the **Car Unlock Command Dispatcher** and **Ride cost dispatcher**).
- **Ride manager**
manages the logic behind the ride phase, receiving data, assessing the ride cost and locking the car under certain conditions
- **MSOManager**
provides the destination where the user should be lead at (submitting to the car uniform distribution constraint) when the MSO option is enabled.
- **Discount computation**
computes the cost of the ride according the initial and final status of the session
- **Maintenance team handler**
handles the interaction with the Maintenance Team, sending requests of intervention

C. Database logic view

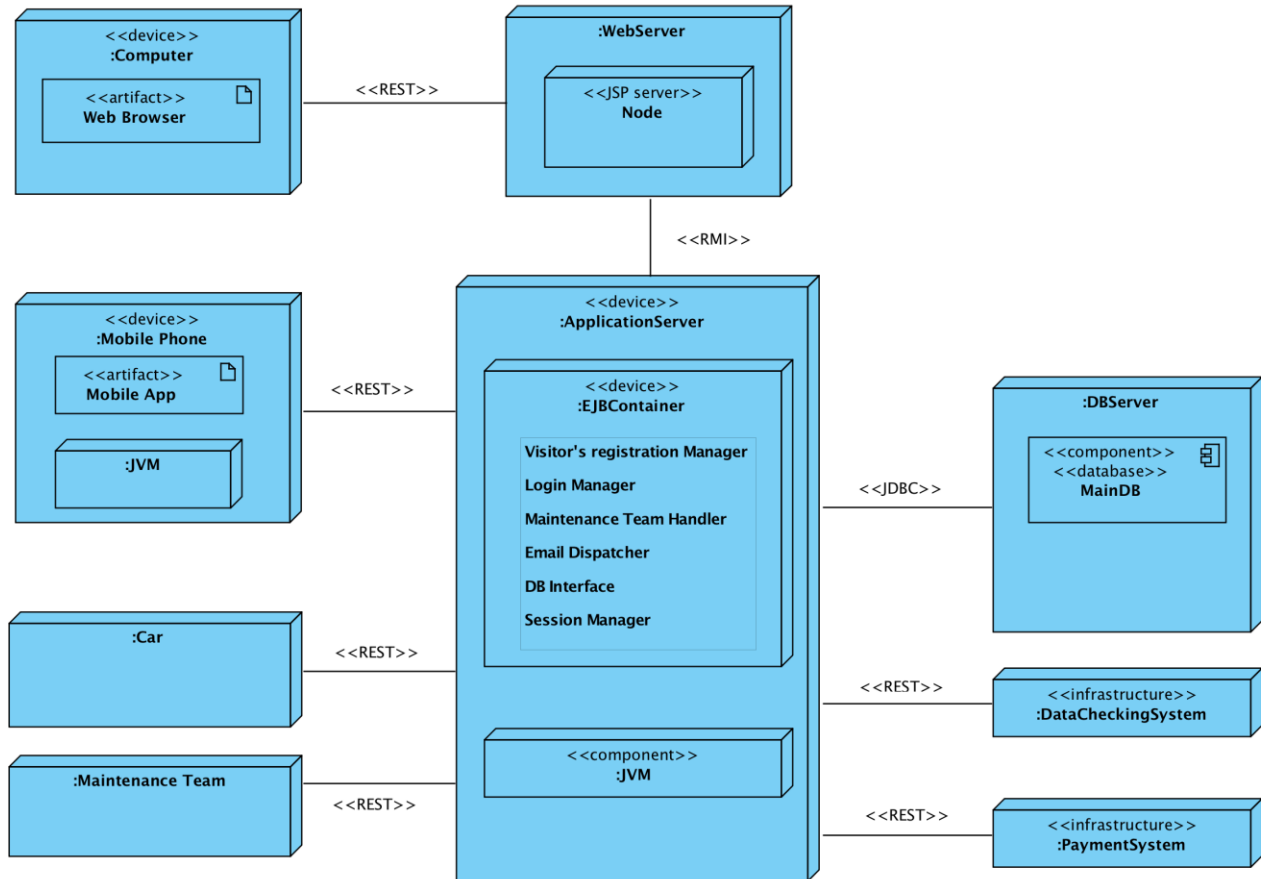
The following diagram shows the logic view of the database that, anyway, would be developed in a relational way although the ER diagram is omitted.



The various fields should be slightly straightforward but the ones labeled as “timestamp” that keep track of the time in which a data has been fetched, this allows the system to know how the data is up-to-date.

D. Deployment view

The following is a deployment UML diagram that shows how and where the several components of the Power Enjoy system are placed.



The diagram above shows how the system is actually deployed and a detailed description follows.

The **business logic** on the server side is developed in JEE language using the Oracle Glass Fish server.

The **data storing** on the server side is done through a persistence unit containing the relational database, with the MySQL RDBMS, linked to the business logic through the JDBC API.

Actually, besides JDBC there also is the JPA Object Relational Mapping that, as it is supposed to do, synchronizes the tables of the relational database with the entities of the corresponding classes in the server business logic.

Therefore, the **DBInterface** component also contains the several classes of the models of the database (shown in the logic view of the database and for this reason omitted in the component diagram).

Typical of an JEE implementation, the server holds an *EJBContainer* that contains all the enterprise beans (derived by the components shown in the proper UML diagram) and manages their stateless lifecycle. The beans are named mainly with the *handler* suffix or the *manager* with the aim that it should convey what functionalities are really going to provide inside the system. The *handler* components are supposed to fulfill the functionalities related to the security of the accesses and hide the complexity of the communication from/to the external part of the server. Actually, due to the JEE implementation, the EJBContainer manages all the first topics and therefore only the latter has to be implemented.

The client side has not any particular logic and therefore it is only an interface with the stakeholders toward a browser installed on the client PC or a mobile application on the Mobile Device of the stakeholders. The web interface (website) is implemented with Java Server Pages (and Servlets where the implementation would come up being too difficult in terms of UI) and the Mobile application would be implemented for the main Mobile Operating Systems (iOS and Android at least).

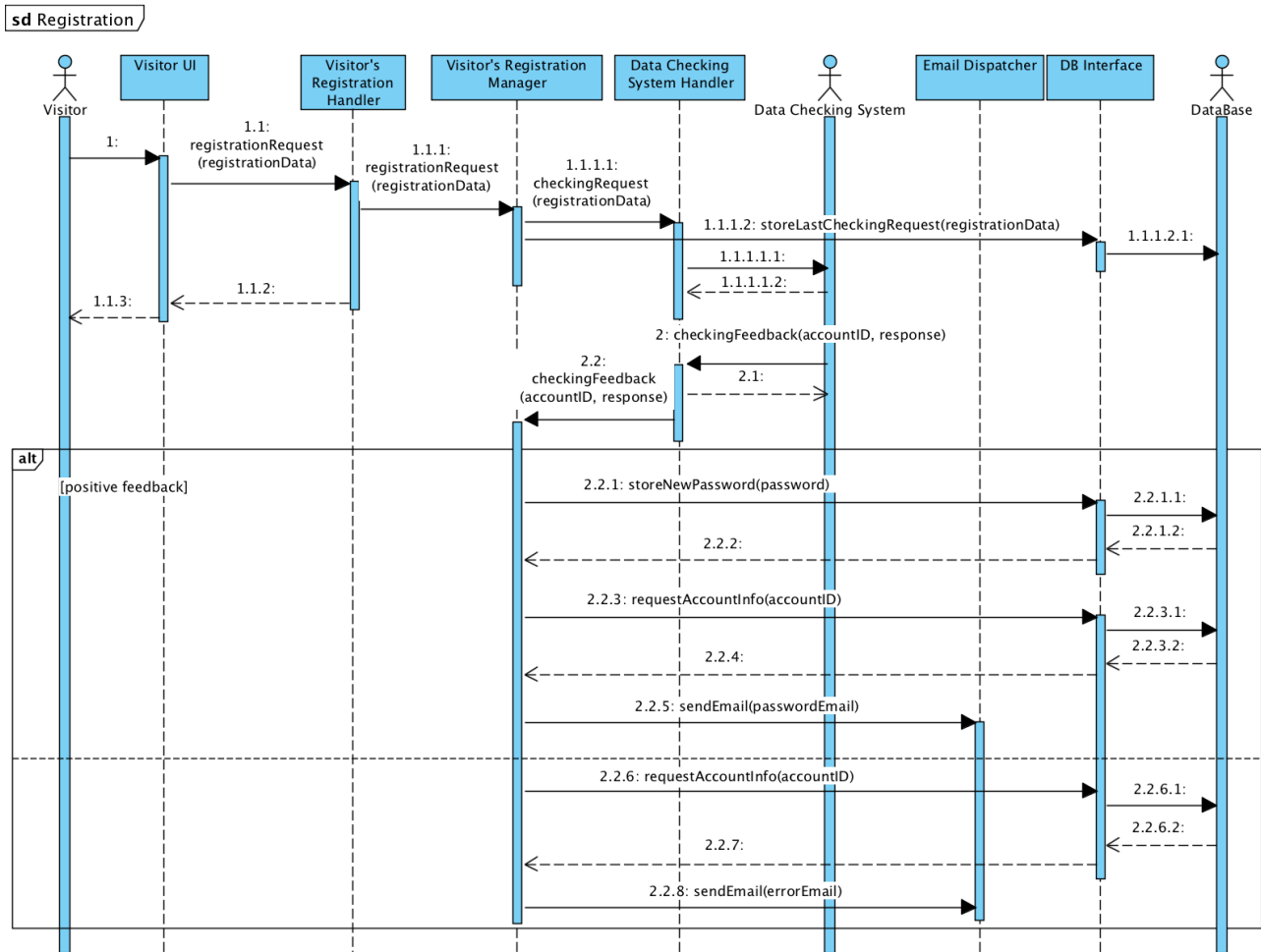
The client-server interactions and the server-server ones (with the external systems) are intended as carried out with the RESTful architecture and JSON response.

The Java Enterprise Edition allows the developer to implement the system not caring about the low level details. This also infers that scalability (mainly due to the stateless beans), reliability, availability and security are inherited by the JEE properties and the hardware deployment. Moreover, lying on a Java base, the system is highly portable. The JSP web implementation allows the developer to build a good user interface (improving usability) with an also complex controller (referring to an MVC model) with presumably less effort than using its servlet “version” or other ones.

E. Runtime view

In this section, all the relevant UML sequence diagrams that describe the flow of events in the system are shown.

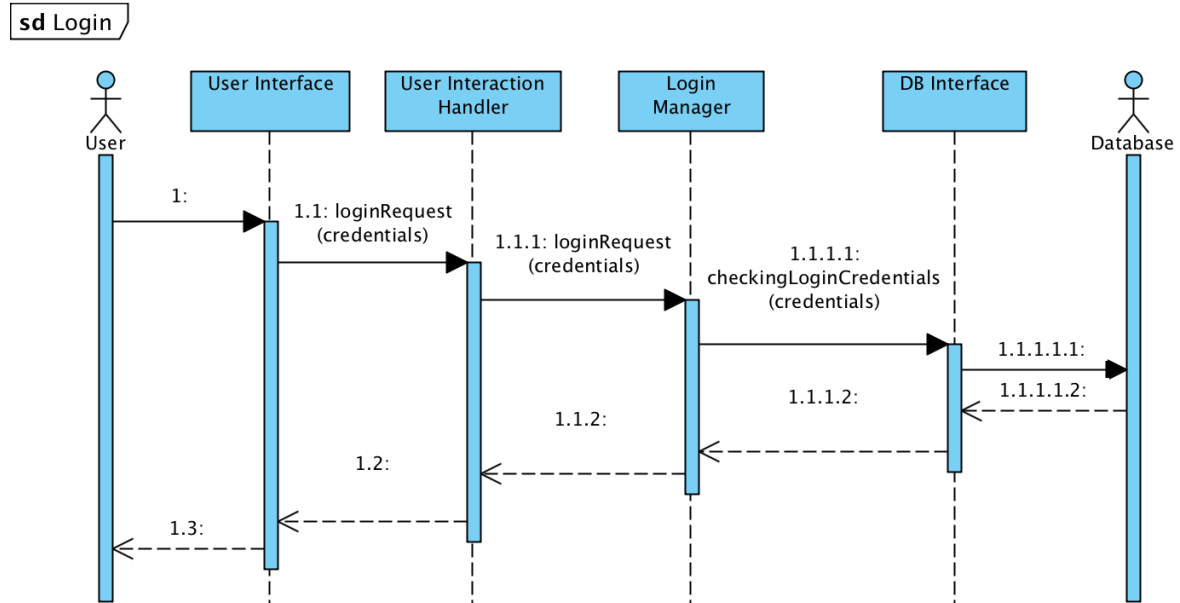
E.1. Registration



In the first part of the registration phase, the system expects that the user inserts his/her data and stores them in the database but with an “valid” account. The account will become “valid” as soon as the external payment system provides a positive feedback of the user’s data that has been sent just after stored in the database. If the feedback is positive the password is sent back to the user (by mail) and if the feedback is negative an error message is sent to the user (by mail) explaining what went wrong and providing a link to a web page that allows to modify the wrong provided data. The generation of the page has been omitted.

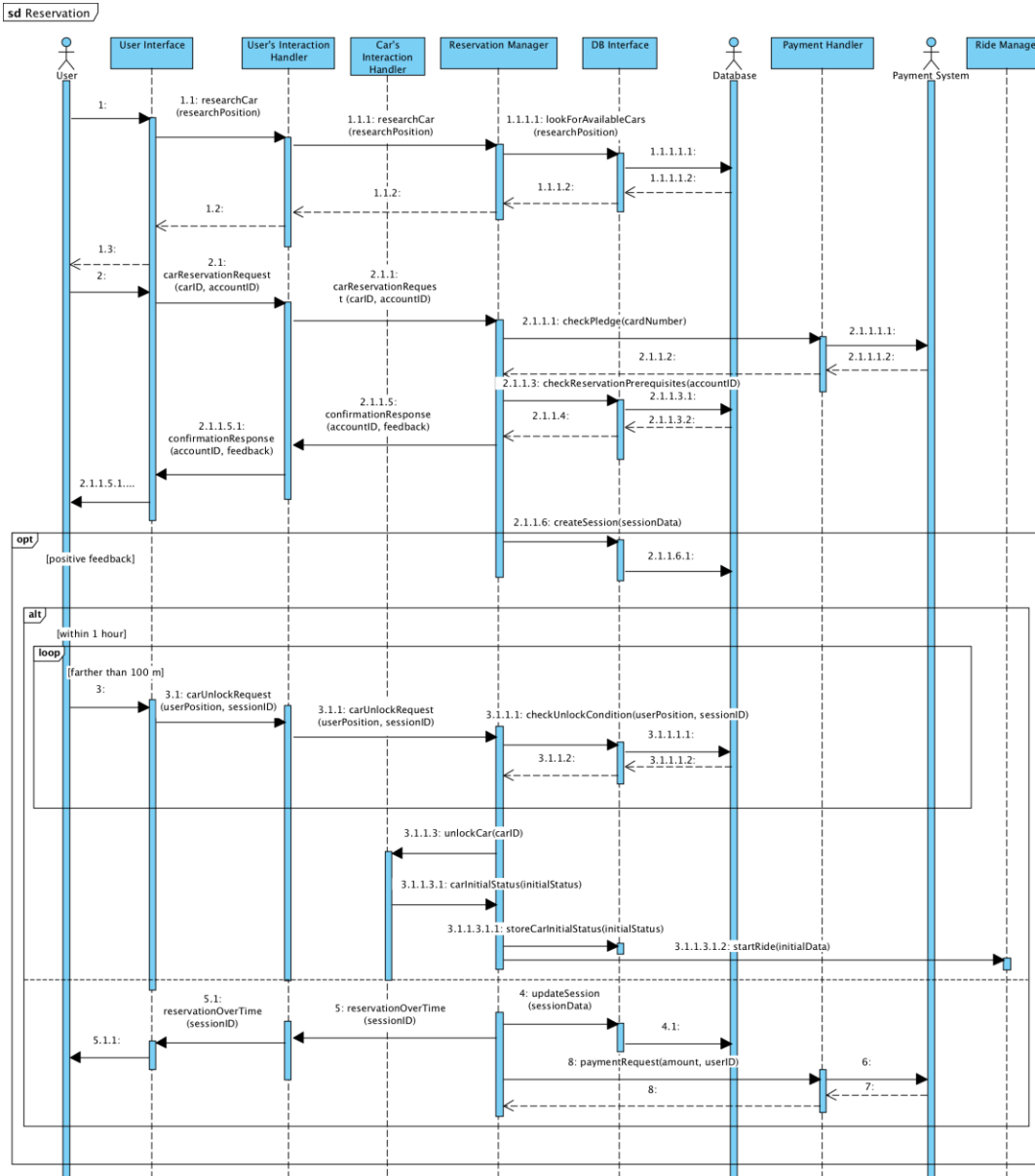
A “garbage collector” policy is supposed to exist but omitted here; perhaps the one that better fits here is an expiration date over the amount of time where an account is stored in the database but not valid.

E.2. Login



Being a trivial procedure, any detail about the login has been omitted.
The interaction is typical of a client-server architecture.

E.3. Reservation

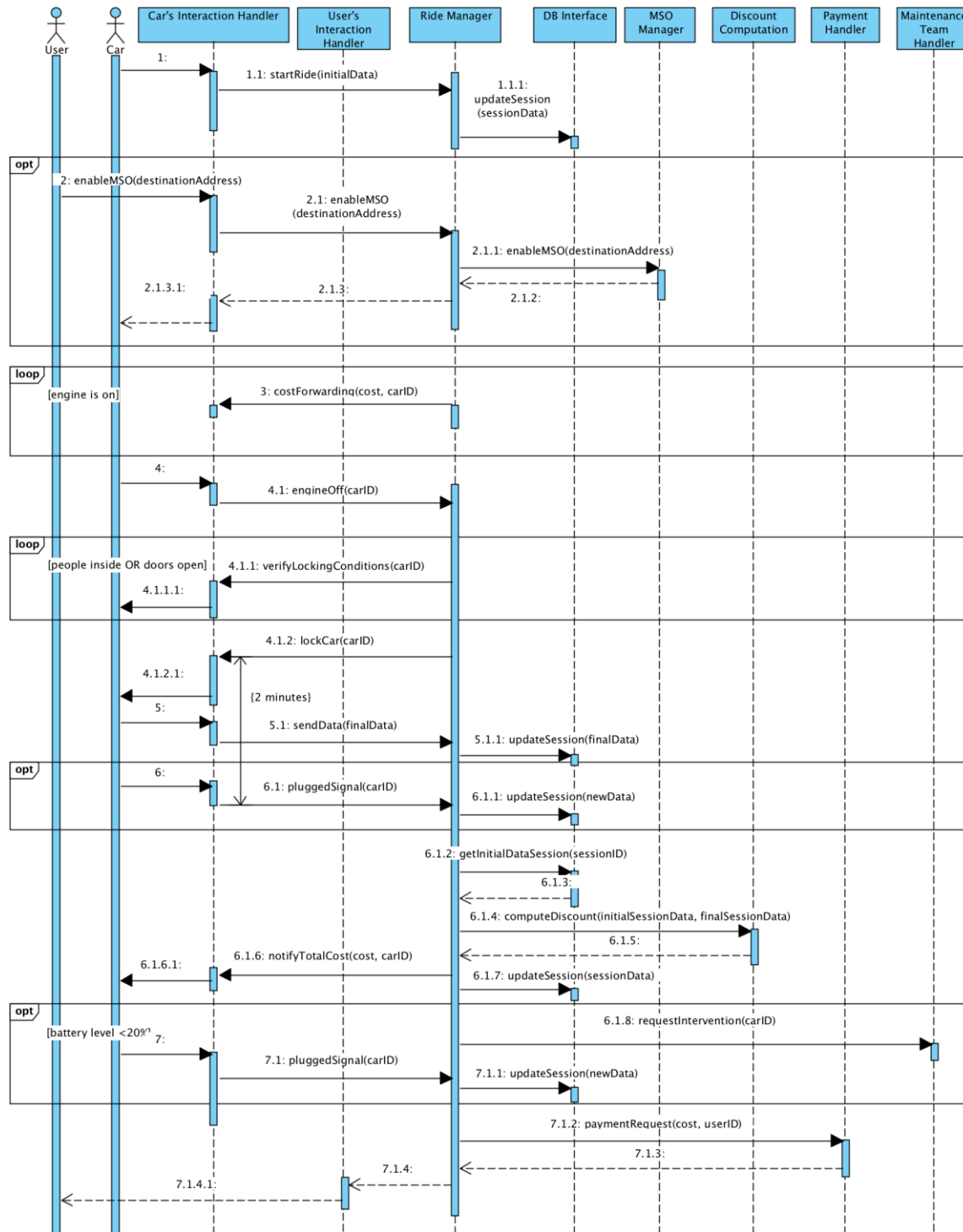


The reservation phase starts with the user that looks for a car, chooses one from the displayed ones, is checked for the money availability to afford a pledge and is checked for the satisfiability of reservation prerequisites (s(he) is not owing and has not any other session opened). If the two conditions are satisfied the system creates a new session, marks the car as “reserved”, informs the user and waits him/her to unlock the car. Up to now, all the interactions have been proper ones of a client-server architecture.

Now, if an user does not request to unlock the car within an hour from the signal of the acceptance of the reservation, the system closes the session, sets the car as “not reserved” anymore and triggers a request of payment for the user. In the other case system unlocks the car (through the handler/dispatcher Car’s Interaction Handler) and stores the initial session data.

E.4. Drive a car

sd Drive a car



The ride car phase starts from where the user turns the engine on and the car sends then all the data about the initial car status.

Here the user has the opportunity to enable the Money saving option where a power grid station would be suggested.

As long as the engine is on, the system keeps sending the current cost of the ride to the car.

Under certain conditions (no people inside and door closed), cyclically checked from when the engine has been turned off, the system locks the car.

The car sends the final data to the system that will be updated in case the car is plugged within two minutes from the locking of the car; if so the session data will be updated again (the car should be marked as “plugged”).

Here the cost of the ride is assessed, notified to the user and stored in the database.

If the car is left with less than 20% of battery level the intervention of the maintenance team is required. The scope of the intervention is to plug the car in a power grid station (the car notifies it to the Power Enjoy system by itself).

Up to the money saving option service (included) the interactions are like request-response.

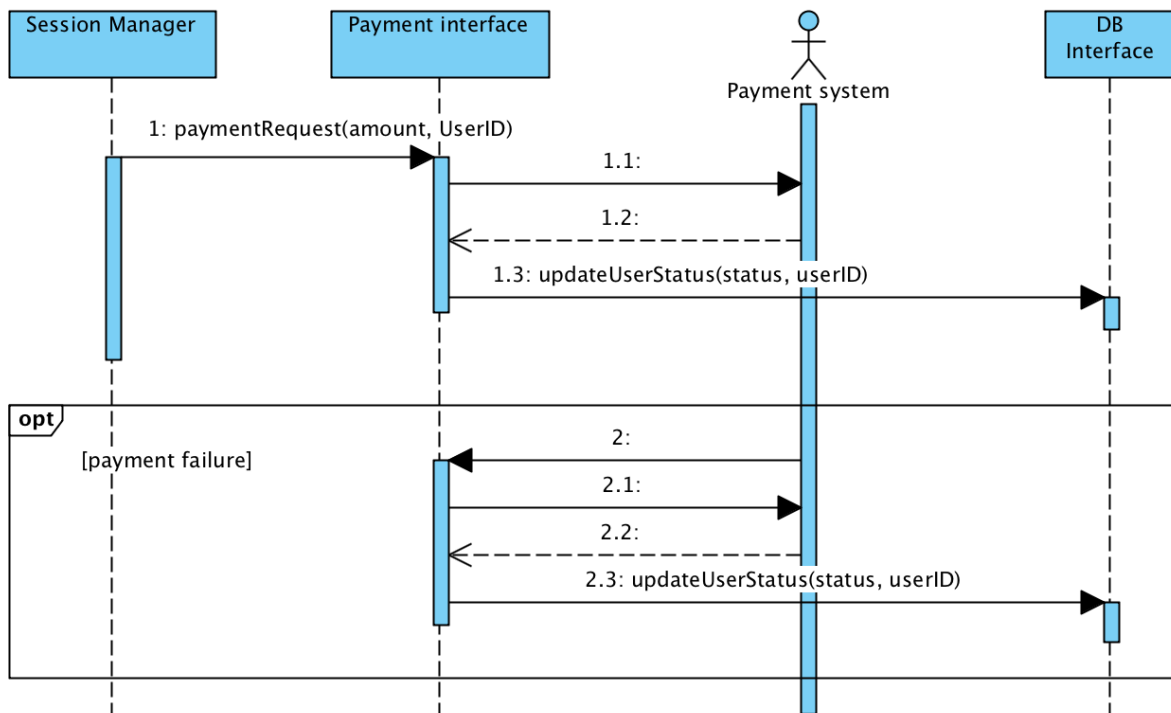
The notification of the ride's cost, the verification of the locking condition and the locking itself are done through the handler/dispatcher Car's Interaction Handler.

The final cost of the ride is notified to the user through the handler/dispatcher User's Interaction Handler.

The maintenance system is called through a request to its server.

E.5. Payment interaction

sd Payment system interactions



The Power Enjoy system sends a request and waits for a payment system's response and if negative marks the user as "owing". The user lies in that status until the payment system does not inform that the user is able to pay that amount and let the PowerEnjoy system do the payment request over. In this sequence diagram we chose to show the session manager as a package (instead of showing the single components).

F. Component interfaces

The following list shows the interfaces of the components of the system. In the leftmost column is specified where the functions of each component are used in the relative sequence diagram (Legend: Reg = Registration, Log = Login, Res = Reservation, Dri = Drive a car, Pay = Payment System Interaction).

Note: Here have been listed more interfaces than the ones drawn in the UML component diagram since here have been listed all the components interactions (considering also the subcomponents).

01. Visitor's registration handler:

+ bool : registrationRequest(data : registrationData) Reg 1.1

02. Visitor's registration manager:

+ bool : registrationRequest(data : registrationData) Reg 1.1.1
+ bool : checkingFeedback(int : accountID, bool : response) Reg 2.2

03. Data checking system handler:

+ bool : checkingRequest(registrationData) Reg 1.1.1.1
+ bool : checkingFeedback(int : accountID, bool : response) Reg 2

04. DB interface:

+ bool : storeLastCheckingRequest(data : registrationData) Reg 1.1.1.2
+ bool : storeNewPassword(string : password) Reg 2.2.1
+ data : requestAccountInfo(int : accountID) Reg 2.2.3 & Reg 2.2.6
+ bool : checkingLoginCredentials(data : credentials) Log 1.1.1.1
+ list : lookForAvailableCars(position : researchPosition) Res 1.1.1.1
+ bool : checkReservationPrerequisites(int : accountID) Res 2.1.1.3
Note: Prerequisites = Owing + Not other sessions opened
+ bool : createSession(data : sessionData) Res 2.1.1.6
+ bool : checkUnlockCondition(position : userPosition, int sessionID) Res 3.1.1.1
+ bool : storeCarInitialStatus(data : initialStatus) Res 3.1.1.3.1.1
+ bool : updateSession(data : sessionData) Res 4 & Dri 1.1.1 & Dri 5.1.1 & Dri 6.1.1 & Dri 6.1.7 & Dri 7.1.1
+ data : getInitialDataSession(sessionID) Dri 6.1.2
+ bool : updateUserStatus(data : status, int : userID) Pay 1.3 & Pay 2.3

05. Email Dispatcher:

+ bool : sendEmail(email : passwordEmail) Reg 2.2.5
+ bool : sendEmail(email : errorEmail) Reg 2.2.8

06. User's interaction handler:

+ bool : loginRequest(data : credentials) Log 1.1
+ list : researchCar(position : researchPosition) Res 1.1
+ int : carReservationRequest(int : accountID, int : carNumber) Res 2.1
+ bool : confirmationResponse(int : accountID, bool : feedback) Res 2.1.1.5
+ bool : carUnlockRequest(position : userPosition, int sessionID) Res 3.1
+ bool : reservationOverTime(int : sessionID) Res 5

07. Login manager:

+ bool : loginRequest(data : credentials)

Log 1.1.1

08. Session manager: (-> NOT USED ANYMORE)

...

09. Reservation manager:

+ list : researchCar(position : researchPosition)

Res 1.1.1

+ int : carReservationRequest(int : carID, int : accountID)

Res 2.1.1

+ bool : carUnlockRequest (position : userPosition, int sessionID)

Res 3.1.1

+ bool : carInitialStatus(data : initialStatus)

Res 3.1.1.3.1

+ bool : startRide(data : initialData)

Dri 3.1.1.3.1.2

10. Payment Handler:

+ bool : checkPledge(int : cardNumber)

Res 2.1.1.1

+ bool : paymentRequest(double : amount : cardNumber)

Res 6 & Dri 7.1.2 & Pay 1

11. Car's interaction handler:

+ bool : unlockCar(int : carID)

Res 3.1.1.3

+ address : enableMSO(address : destinationAddress)

Dri 2

+ bool : costForwarding(double : cost, int carID)

Dri 3

+ bool : verifyLockingConditions(int : carID)

Dri 4.1.1

+ bool : lockCar(int : carID)

Dri 4.1.2

+ bool : notifyTotalCost(int : cost, int : carID)

Dri 6.1.6

12. Ride manager:

+ bool : startRide(data : initialData)

Dri 1.1

+ bool : enableMSO(address : destinationAddress)

Dri 2.1

+ bool : engineOff(int : carID)

Dri 4.1

+ bool : sendData(data : finalData)

Dri 5.1

+ bool : pluggedSignal(int : carID)

Dri 6.1 & Dri 7.1

13. MSOManager:

+ address : enableMSO(address : destinationAddress)

Dri 2.1.1

14. Discount computation:

+ int : computeDiscount(data : initialSessionData, data : finalSessionData)

Dri 6.1.4

15. Maintenance team handler:

+ bool : requestIntervention(int : carID)

Dri 6.1.8

G. Selected architectural styles and patterns

The PowerEnjoy system has to manage the fleet of electric vehicles, provide an user friendly interface to the user and interact with external systems.

This has lead to devise the system over a client-server architecture with the help of some event-based architecture features. The choice of mainly using a client-server architecture is due to the fact that the requirements can be fulfilled almost entirely with that architecture despite of some functionalities (as the commands that should be sent to the cars) that has been covered by the use of dispatchers (proper of an event-based architecture). The idea of mixing the two architecture types up instead of using a pure event-based architecture is supported also by the fact that the client-server architecture is (likely) one of the most widely used and well known too and, therefore, more maintainable and supported.

H. Other design decision

Furthermore, it has been considered also to move some logic on the car too (like having the car figure out by itself the cost of the ride that has to be shown to the user) in order to decrease the flow of data between car and server but this way has been discarded in order not to have too many constraints on the hardware that controls the car.

When a user looks for a car, the system knows their status directly querying the database instead of sending a message to all the cars. The shown data are always up-to-date despite when a car is plugged to a power grid station; in this situation the status of the car (precisely the battery level) keeps changing if the battery level has not already soar the full recharge.

To cope with this issue, the system has a routine that periodically runs over the database, querying the data and looking for the plugged car in order to fetch the up-to-date battery level and update the value in the database. It helps to move the flow of data in background although the data could not be totally up-to-date but the other way, querying the car at runtime, could have delayed the response to the user.

3. ALGORITHM DESIGN

This chapter aims to specify some details about the system through the use of pseudocode that describes the flow of the events in certain particular situations. These situations are the ones considered not properly straightforward and, for this reason, the block of pseudocode is preceded by a brief description of the context and the goal of the code itself.

The **moneySavingOptionSuggestedPGS** function computes which is the right power grid station where the user should be led at when the money saving option is activated. This power grid station is assessed considering the destination inserted by the user and the number of available plugs in the power grid station whereby that destination in a way that the chosen power grid station is the nearest to the user destination but that be having a number of available plugs that is more or equal the half of the number of total plugs.

```
chosenPGS moneySavingOptionSuggestedPGS( userDestination )
{
    // Power grid station suggested by the system considering the user destination position
    declare chosenPGS = NULL

    // List of all the power grid stations stored in the Database
    declare PGSLIST

    fill PGSLIST with values from the DB

    // Two fields-per-element list, the two fields of each element are:
    //     pgs = a reference to an item of PGSLIST
    //     distance = distance of the indexed PGS from the user destination
    declare distanceList

    for( all distanceList items )
    {
        distanceList[i] = distance( PGSLIST[i].position, userDestination )
    }

    sort the distanceList by the distanceList.distance field (crescent order)

    for( all distanceList items && chosenPGS == NULL )
    {
        if( distanceList[i].pgs.availablePlugs >= (distanceList[i].pgs.totalPlugs)/2 )
        {
            chosenPGS = distanceList[i].pgs
        }
    }

    //Therefore the function would return:
    //     a reference to a power grid station in case it was found or
    //     a NULL value otherwise
    return chosenPGS
}
```

The **carNearbyAnyPGS** function figures out whether a certain car is within a specified range (in meters) from any power grid station or not.

```
bool carNearbyAnyPGS ( carFinalPosition, range )
{
    declare feedback = FALSE

    // List of all the power grid station stored in the Database
    declare PGSList

    fill PGSList with values from the DB

    for( all PGSList items && feedback == FALSE )
    {
        if( distance ( carFinalPosition, PGSList[i].position ) <= range )
            feedback = TRUE
    }

    return feedback
}
```

The **assessRideDiscount** function assesses the discount (positive or negative where the latter, being a negative discount, is considered as a charge) that should be applied to a certain ride considering the states in which the car has been left at the beginning and the end of the ride. The discount has an upper bound of 30%.

```
int assessRideDiscount( peopleInside, batteryLevelWhenLocked, carFinallyPlugged, car3KmNearAnyPGS )
{
    // Variables that keep track of the amount of discount and charge that should be applied to the ride cost
    declare discounts = 0
    declare charges = 0
    declare finalDiscount = 0

    if( peopleInside >= 3 )
        discounts+ = 10
    if( batteryLevelWhenLocked >= 50 )
        discounts+ = 20
    if( carFinallyPlugged == TRUE )
        discounts+ = 20
    if( car3KmNearAnyPGS == FALSE || batteryLevelWhenLocked < 20 )
        charges+ = 30

    return max( 30, discounts - charges )
}
```

The **finalSessionCostAssessment** function assesses the final cost of the ride considering the time elapsed between the beginning and the end of the ride itself and the chance that a discount (or charge) on the cost could be applied.

```
double finalSessionCostAssessment( peopleInside, batteryLevelWhenLocked, carFinallyPlugged, carFinalPosition,
rideDuration, costPerMinute)
{
    declare sessionCost = 0
    declare car3KmNearAnyPGS = carNearbyAnyPGS( carFinalPosition, PGSLIST, 3000 )

    sessionCost = rideDuration * costPerMinute

    sessionCost *= 1 - assessRideDiscount( peopleInside, batteryLevelWhenLocked, carFinallyPlugged,
car3KmNearAnyPGS )

    return sessionCost
}
```

The **carsStatusUpdate** function is a routine executed on the Database that, every fixed amount of time, queries the database to know the cars that should be monitored because plugged in some power outlet and therefore having a status that changes over time. The fetched list is then used to send messages to the cars asking for the current level of battery and update the Database forth.

```
void carsStatusUpdate( )
{
    carList = "SELECT CarID FROM Car WHERE Car.plugged = TRUE"

    query the cars belonging to the carList

    update the DB
}
```

The **assessThePGS** function assesses in which power grid station the car is plugged in (if so)

```
pgs assessThePGS( car )
{
    declare pgs = NULL

    // List of all the power grid station stored in the Database
    declare PGSList

    fill PGSList with values from the DB

    if( car.plugged == TRUE )
    {
        for( all PGSList items && pgs == NULL )
        {
            if( distance( car.position, PGSList[i].position ) < 50 )
            {
                pgs = PGSList[i]
            }
        }
    }

    return pgs
}
```

4. USER INTERFACE DESIGN

The user interface diagrams represent all the possible interaction between user and system in the website and mobile version of our software.

A. Mockups

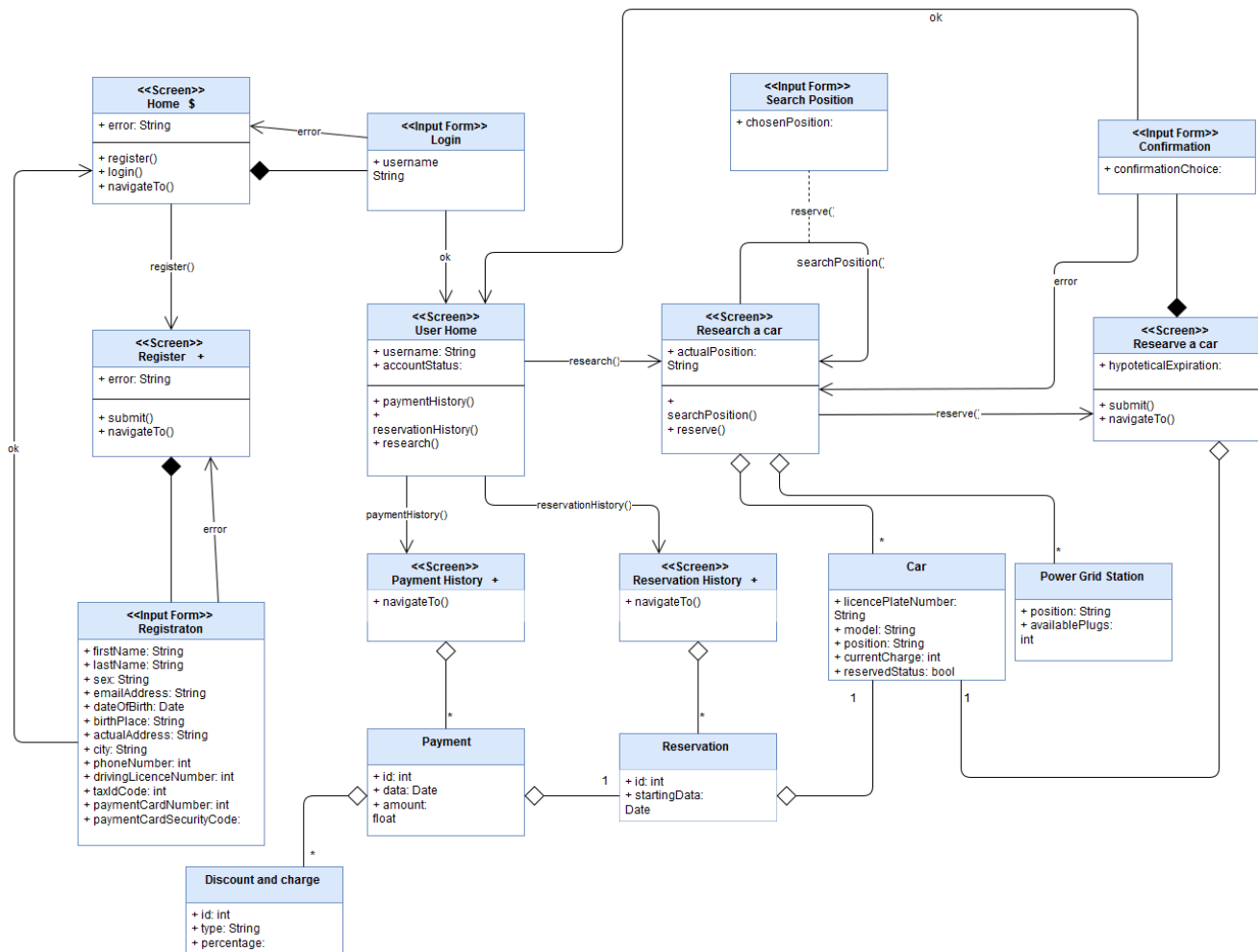
The mockups of the system had already been presented in the RASD document.

B. UX diagrams

UX diagrams are the representation of the interactions among all the various pages that the user can see during his/her navigation on the website or on the application. We've split the attributes into different blocks in order to give a better view also on the interactions present between the various entities of the system (e.g. between payment and reservation).

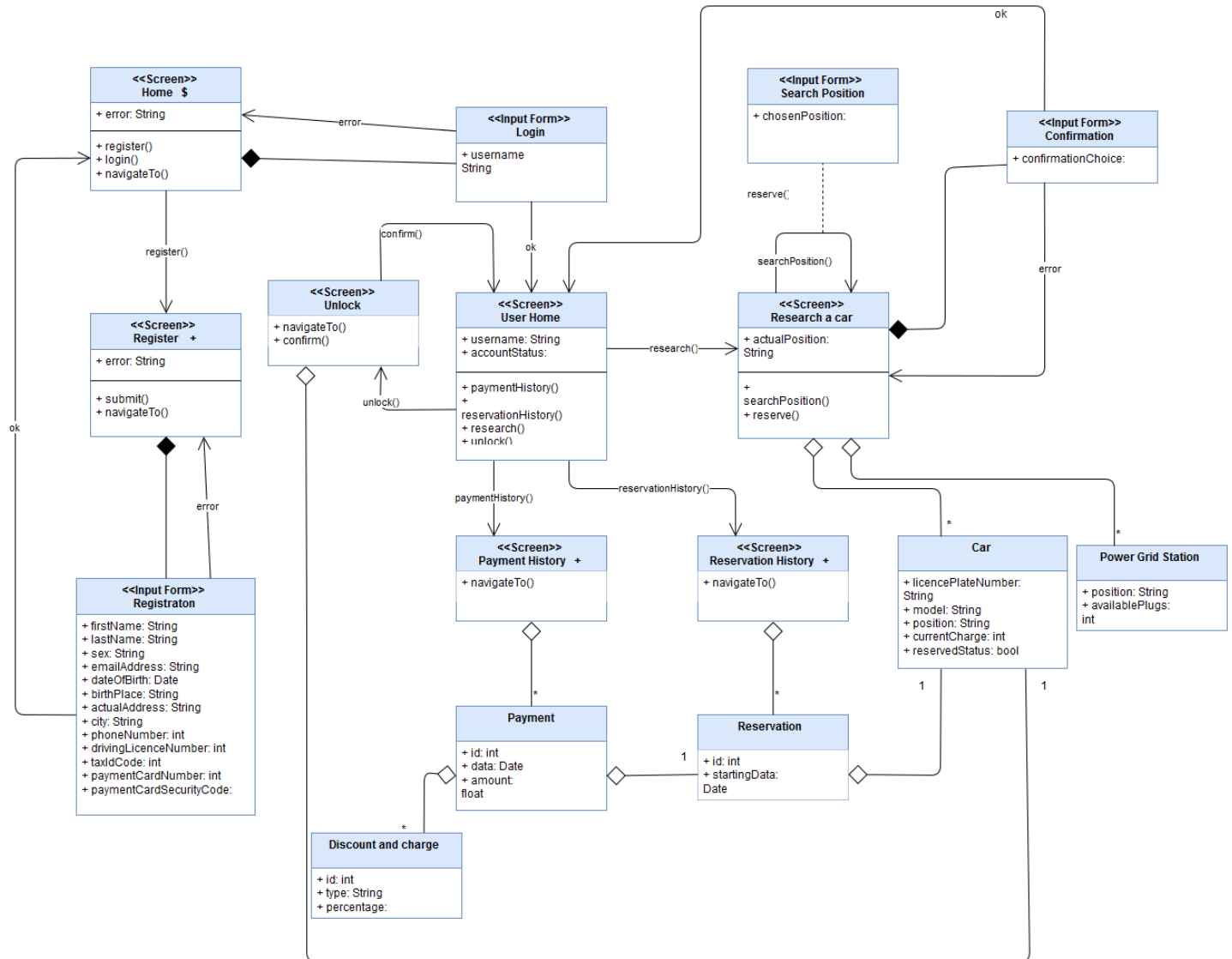
B.1. Desktop version

This is the UX diagram of the desktop website.



B.2. Mobile version

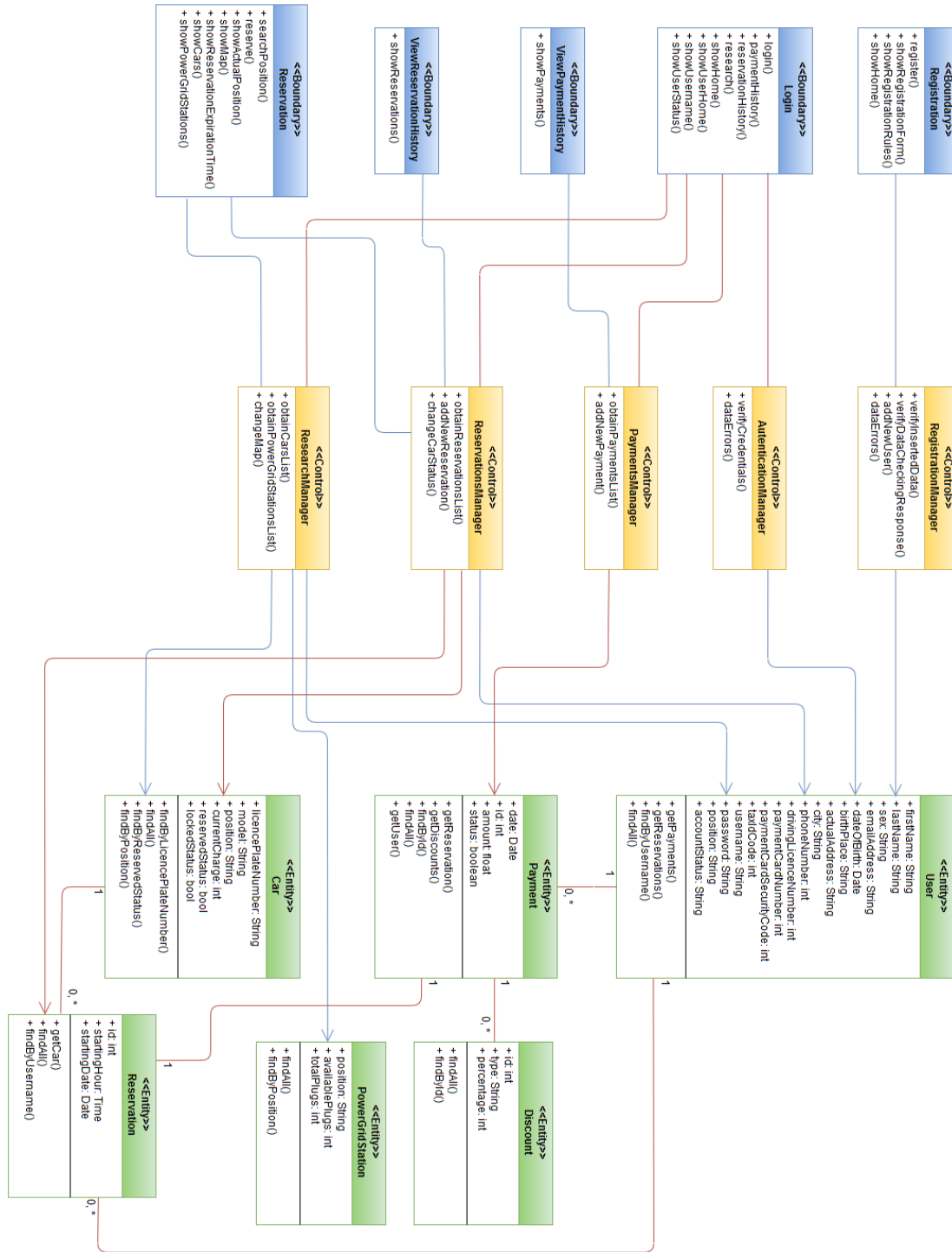
The mobile version of the UX model is different from the previous one mainly for the presence of the “Unlock” function (available only in the mobile version of the application).



C. BCE diagrams

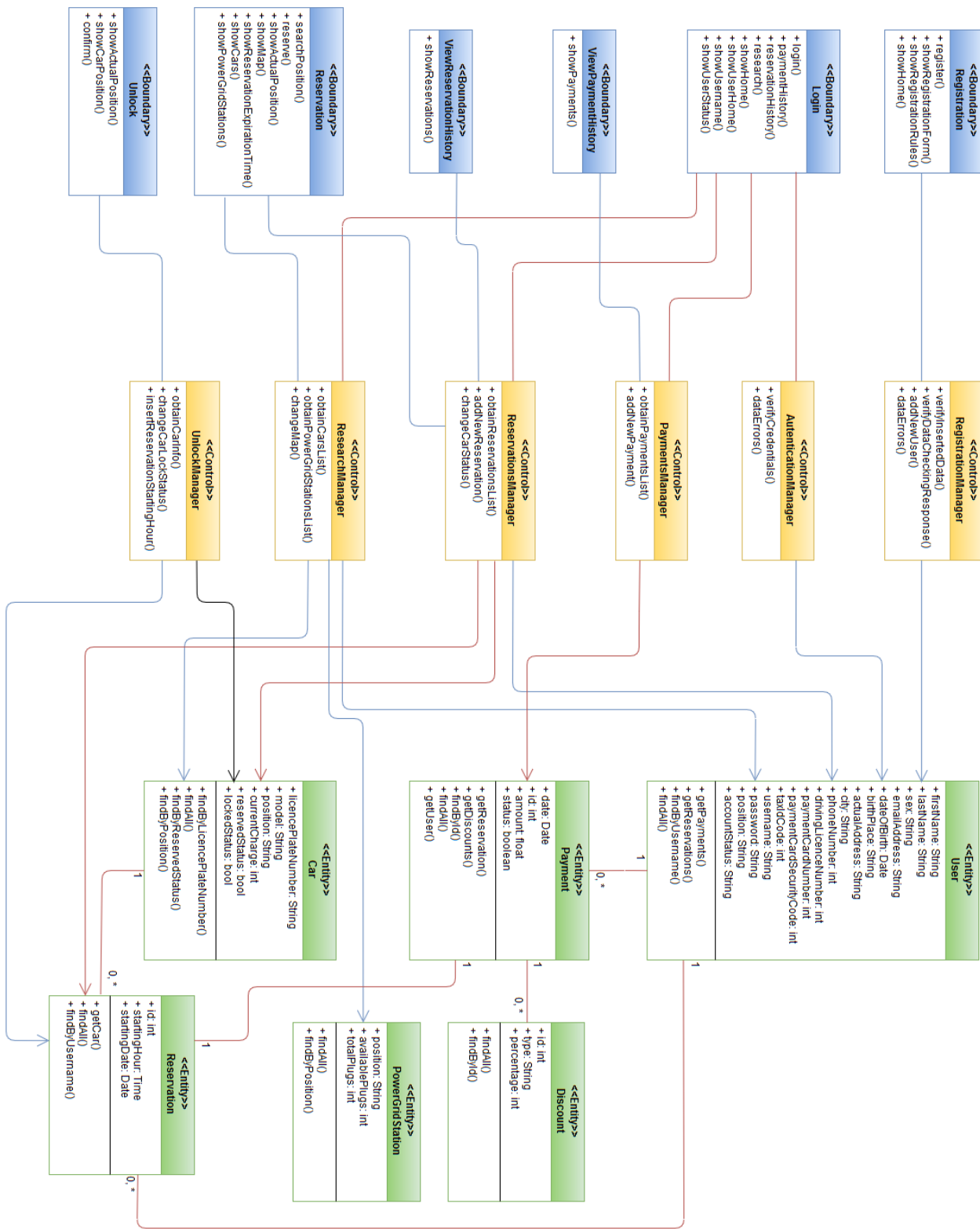
BCE diagrams represent the composition of the various entities of the system involved in the interface used by the user on website and mobile application.

C.1 Desktop version



C.2. Mobile version

As for the UX diagram, the mobile version of BCE is different from the desktop one mainly for the presence of the “unlock” function.

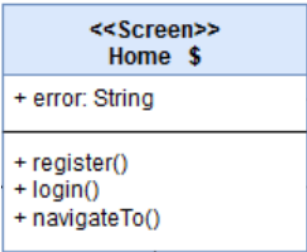
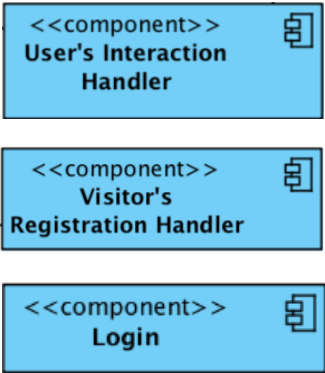


D. Ux diagram and component correspondence

Every *screen* in the UX diagram represents an interaction between different *components* of the system in order to guarantee the satisfaction of all users' request on website or application. Here we analyze these interaction showing the connection between each screen and the various components.

D.1. Primary Home

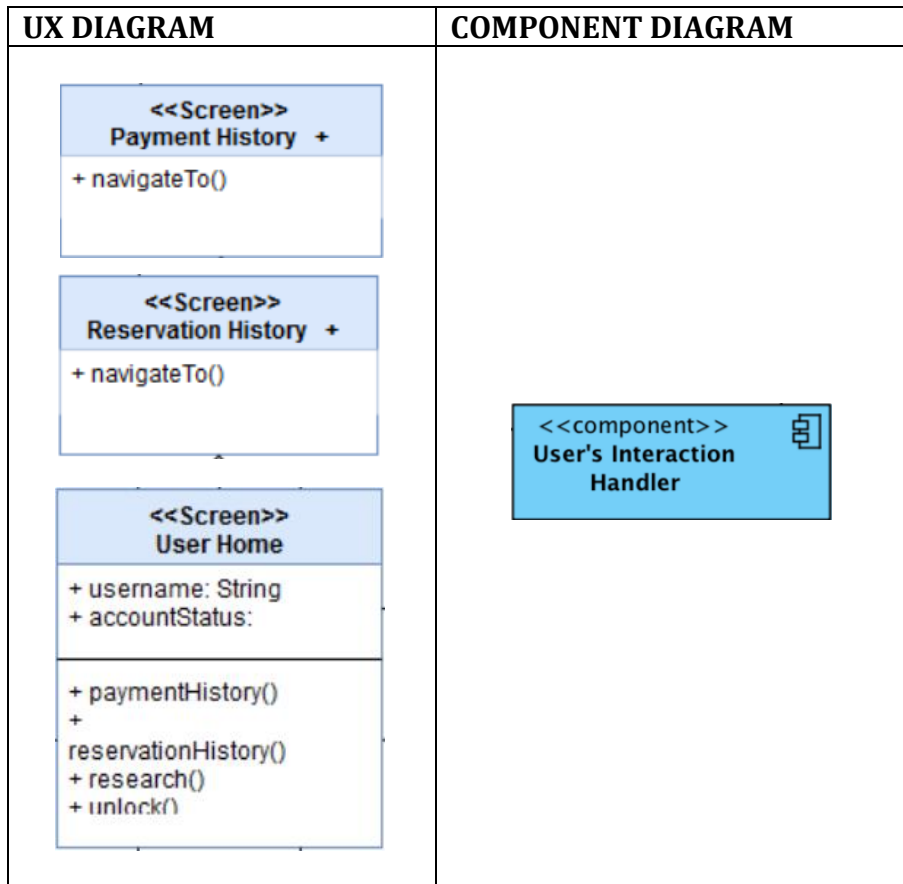
This is the homepage the user has to interact with when s(he) connects first to the mobile application/website: this homepage must provide a link to the registration screen and login functionality, so it must be able to communicate with the user's interaction handler, the visitor's registration handler and the database in order to allow the user to access his(her) account.

UX DIAGRAM	COMPONENT DIAGRAM
 <p>The UX diagram for the Home screen is a UML-like box. The top section is a light blue header with the text '<<Screen>>' and 'Home \$'. Below this is a white section containing the text '+ error: String'. The bottom section is a white box containing the text '+ register()', '+ login()', and '+ navigateTo()'.</p>	 <p>The component diagram for the Home screen shows three blue component boxes stacked vertically. Each box has a small icon in the top right corner. The top box is labeled '<<component>>' and 'User's Interaction Handler'. The middle box is labeled '<<component>>' and 'Visitor's Registration Handler'. The bottom box is labeled '<<component>>' and 'Login'.</p>

D.2. User Home/Payment History/Reservation History

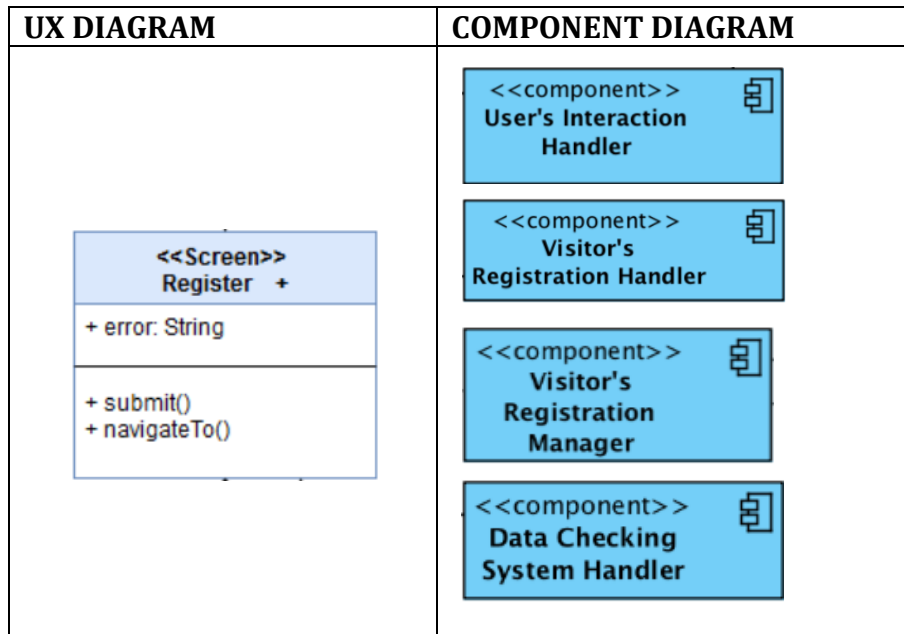
Here we have the homepage the user has to interact with when s(he) logs into the system: this homepage must provide the research, payment history and reservation history links, so it must be able to communicate only with the user's interaction handler that manages the user's requests.

Also the Payment and Reservation history screens must be able to communicate with the User's Interaction Handler because they don't have to modify anything in the system.



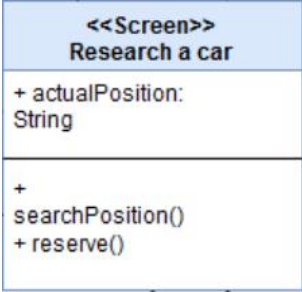
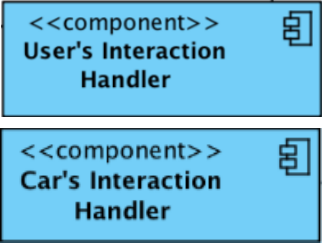

D.3. Registration

This page offers the registration form to the visitor, and sends all the data to the data checking system and communicate to the central database of Power Enjoy in order to create a new account associated to the visitor asking for registration.



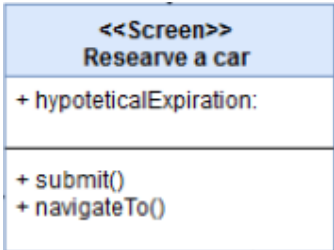
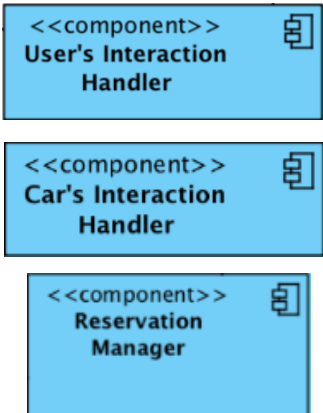
D.4. Research a car

This page must provide a map that shows all the nearby not reserved car and all the nearby available power grid stations after s(he) chose the position, and for this reason it must communicate with the car's interaction handler. The mobile version, in particular, offers the direct reservation from this page, so it's connected also to the session manager.

UX DIAGRAM (desktop)	COMPONENT DIAGRAM
	
UX DIAGRAM (mobile)	COMPONENT DIAGRAM
The same as above	The same as above plus 

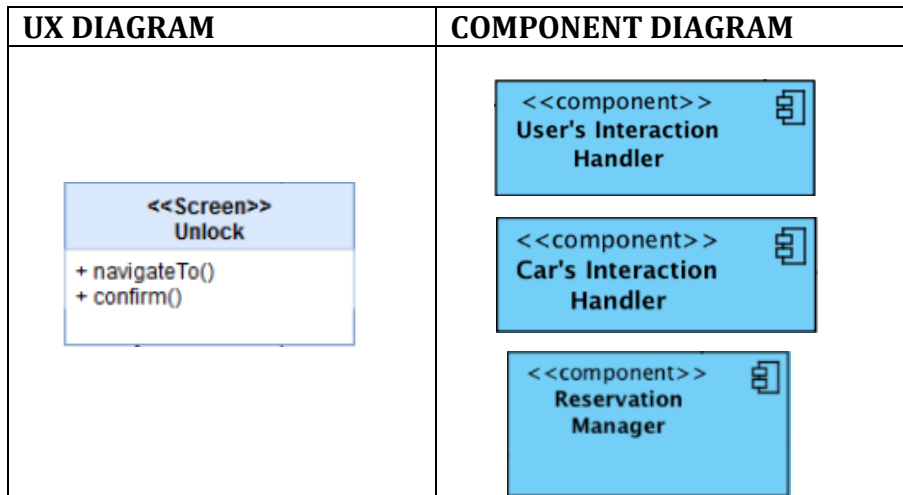
D.5. Reserve a car

This screen (available only in the desktop version) allows the user to reserve a specific car selected by a pin on the map. It summarizes all the characteristics of the car so it has to communicate with the car's interaction handler and with the session manager on consequence.

UX DIAGRAM	COMPONENT DIAGRAM
	

D.5. Unlock a car

This screen (available only with the mobile version) allows the user to unlock the reserved car when (s)he is nearby it. So, it has to communicate with the user's interaction handler, car's interaction handler and with the session manager on consequence.



5. REQUIREMENTS TRACEABILITY

Table legend:

- rows: requirements
- columns: components index

Components legend:

1. Visitor's registration handler
2. Visitor's registration manager
3. Data checking system handler
4. DB interface
5. Email Dispatcher
6. User's interaction handler
7. Login manager
8. Session manager (Not used anymore)
9. Reservation manager
10. Payment Handler
11. Car's interaction handler
12. Ride manager
13. MSOManager
14. Discount computation
15. Maintenance team handler

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
R01f	x	x		x											
R02f	x	x		x											
R03f		x	x	x	x										
R05f						x									
R06f				x		x	x								
R08f				x		x		x	x						
R11f				x		x		x	x	x					
R12f				x		x		x	x						
R14f				x		x		x	x	x					
R15f				x		x		x	x		x				
R17f				x				x			x	x			
R18f				x				x			x	x			
R19f				x				x			x	x			
R20f				x				x			x	x			
R21f				x				x			x	x		x	x
R22f											x	x	x		
R23f											x	x	x		
R25f											x	x			
R27f				x				x			x	x			
R29f				x		x		x	x	x		x			
R30f				x		x		x		x					
R32f				x				x			x	x			x

6. EFFORT SPENT

28/11/16	Marco	4	Component diagram and tier architecture
29/11/16	Paola	2	Component diagram review and formalization and tier architecture
	Marco	2	Component diagram review and formalization and tier architecture
	Marco	5	Deployment diagram and state diagrams
	Paola	5	Deployment diagram review and formalization
	Giulia	5	UX e BCE first trial
30/11/16	Giulia	5	UX diagrams
	Marco	2	Component diagram and deployment diagram review + sequence diagrams development
	Paola	2	Sequence diagrams formalization and fixing
01/12/16	Giulia	5	BCE diagrams (not finished yet)
	Marco	7	Sequence diagrams fixing and formalization + logical DB diagram
	Paola	7	Sequence diagrams fixing and formalization
02/12/16	Giulia	5	BCE diagrams
	Marco	2.5	Algorithms
	Paola	2.5	Overall consistency
	Paola	2+4	Overall consistency + rasd fixing + sequence diagrams layouts + dd structure
	Marco	2	Algorithms (details)
03/12/16	Paola	1	Database logic view
	Marco	7	Interfaces and sequence diagrams fixing
	Paola	4+1	various fixes
	Giulia	4	Fixed mockups problems and general review
04/12/16	Giulia	3	Fixing things and adding alloy models in rasd
	Giulia	3	Fixed UX and BCE and added them into DD
	Marco	1.5	Architecture choices explanation
	Marco	6	Requirements traceability, sequences' fixes ...
	Paola	6	Requirements traceability, sequences' fixes ...
	Paola	1.5	Sequence review and layout
05/12/16	Marco	5	Architecture explanation and diagrams layout
	Paola	5	Fixes on diagrams and introduction
	Marco	3	Architecture explanation + payment interaction diagram
06/12/16	Marco	2	Architecture explanation
07/12/16	Paola	2	Document review and minor fixes

7. REFERENCES

Used tools:

- Microsoft word: for document redaction
- Visual Paradigm: for the sequence, component and deployment diagram
- Draw.io: for ux, bce, database logic view and high level architecture
- Microsoft Excel: for hours counting