Figure 1: Politecnico di Milano

Design Document
Version 1.0

Emanuele Ricciardelli (mat. 875221)
Giorgio Tavecchia (mat. 874716)
Francesco Vetró (mat. 877593)

December 11, 2016

Contents

1 Introduction

1.1 Purpose

This document will provide a set of architectural and design details in order to drive software developent of the PowerEnjoy system-to-be. Specifically, the following will be now identified and described:

- High level architecture;
- Main component view;
- The runtime behaviour of components;
- Design patterns.

1.2 Scope

The system should be able to register new users with their credentials, such as name, surname, address and e-mail, their valid driving license number and some payment information. Once the registration process is completed and all entered data are veried, the system will send a password to the user: this password will be used by the user to access the system and its functionalities. The system provides the following common features:

- find an available car;
- reserve an available car one hour prior the pick up;
- unlock the car when the user is detected nearby the car he/she reserved;

In addition there will be incentives for virtuous behaviors of the user that will aect their fares: if an user helps the community by being responsible with the usage of the car, his/her fare will be reduced by a specied percentage; otherwise, by not following the guidelines of the company, the user will be charged more than the standard fare

1.3 Definitions, acronysm, abbreviations

- **RASD**: Requirements Analysis and Specifications Document;
- **Gateway**: tool to set the communication and the exchange of data between two or more networks with, often, different protocols;
- **Push notification**: notification sent to a smartphone via application installed on it;
- **Email**: mail service carried over the web;
- **API**: Application Programming Interface. It is a set of subroutine definitions, protocols, and tools for building application software;
- **Google Maps API**: APIs provided by Google to compute functionalities over the Google Maps environment;

- **MVC**: Model View Controller pattern;
- **Tier**: architectural level in which the system is decomposed;
- **Client**: program or part of a program which allows to exchange data with a server to perform computations;
- **Thin Client**: client dedicated to only graphical representation, which displays the results of a computational logic contained exclusively in server;
- **Application Server**: it provides the infrastructure and support functions, development and execution of applications in a distributed environment;
- **Database Server**: software system able to store and manipulate data contained in a database;
- **Execution Environment**: it represents a particular execution platform, such as an operating system or a database management system. Execution environments are used to describe the context in which the execution of a model takes place;
- **Controller**: software system that applies changes to the model and database;
- **UX**: user experience diagram;
- **BCE**: Boundary Control Entity diagram;
- **Ticket**: the term refers to a fine that a driver has received during a ride;
- **Record Maintenance**: record saved in the Database, used to save information concerning the maintenance processes of a car.

For all terms not specified, please refer to the glossary contained in the previous document (RASD).

1.4 Reference documents

- Specication document: Assignments AA 2016-2017.pdf;
- RASD previously version 1.0;

1.5 Document structure

- Introduction: this section introduces the design document, providing a brief indication of its utility and about the covered sections;
- Architecture design:

  - Overview: this sections explains the high level components and their interaction;

  - Component view: more detailed view of the components of the application;

- Deployment view: this section shows all components that must be deployed to have the application running completely and correctly;

- Runtime view: it describes how components interact with each other to accomplish specific tasks;

- Component interfaces: it describes the interfaces between the components;

- Selected architectural styles and patterns: it provides details related to patterns and architectural styles selected for the development;

- Other design decisions;

- Algorithms design: this section provides some algorithms useful to describe some functionalities of the system;
- User interface design: this section presents a detailed description of user experience explained with diagrams like UX and BCE;
- Requirements traceability: this section aims to explain how the decision taken in the RASD are linked to design elements.

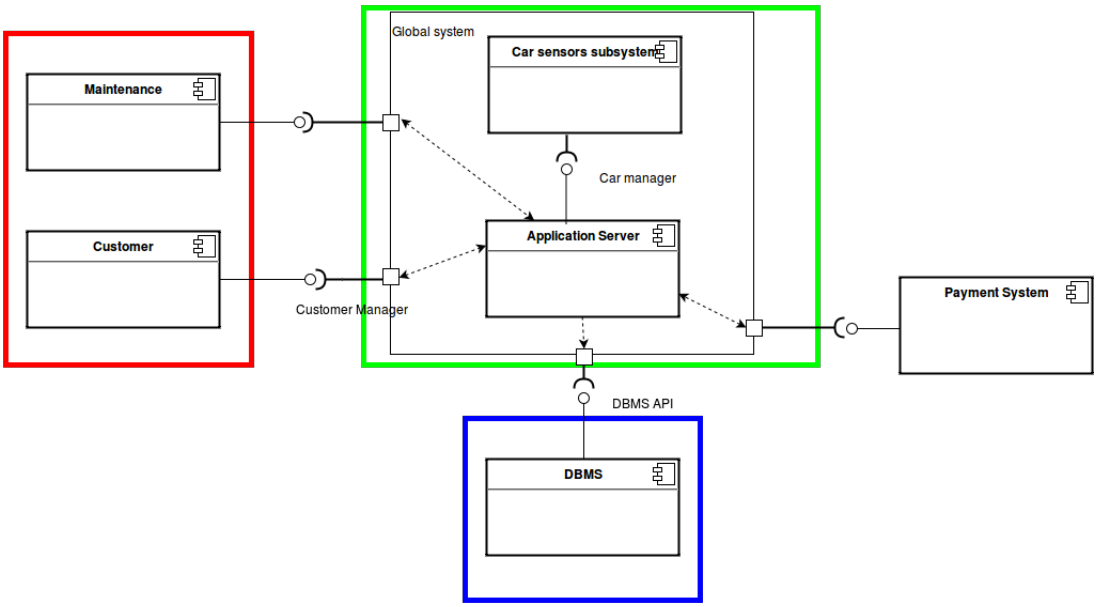

Figure 2: High level components

2 Architectural Design

2.1 Overview

The PowerEnJoy system will be shaped in a three tier architecture. This architectural style allows a further enhancement: the possibility to move the entire application in a dedicated cloud server with elastic infrastructure in order to get performances needed by the load on the application on demand and the consequent money saving. This choice will also face the coupling problem with respect either to the logic and data level and to the logic and presentation level. Nonetheless a performance evaluation might need to check whether the contract terms are satisfied. Important decision based on this architecture is the thin client characterization: the client can only send to the server a sequences of requests on what operation is intended to perform. This choice can possibly raise the expenses for the company. In the right part of Figure 2 is present the component named as Payment System: this component is intended to be a set of functionalities offered to the banks in order to ensure the correctness of all the payment operations. In Figure 3 we show the existence of firewalls in between the layers: this is due to the high security problems that this service arouses. Since our data level will store personal information such as identity, mail address, number of credit cards and even positions of a human being, a possible solution is to use different techniques to ensure that none will ever access private data.

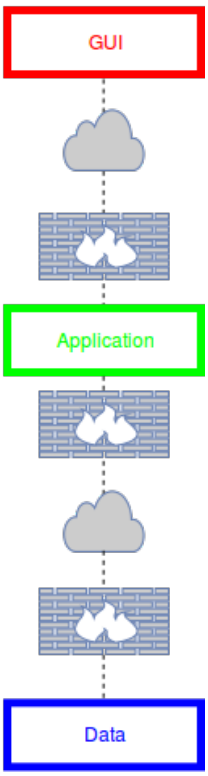

Figure 3: Tier levels

2.2 High level components and their interaction

The high level components architecture in Figure 2 is made of five parts. The main part is the *Global system*: it is a singleton that contains all the business logic. It is splitted into two sections:

- the *Car sensors subsystem*, as stated in the RASD, the cars are seen by the system as a set of sensors that only provide data. This part of the system-to-be is devoted to collect data from cars, convert them in a usable structure for the system and then send them to the business logic part named *Application server*;
- the *Application server* that will receive requests, will manage the information provided by the *Car sensors subsystem* and will be able to communicate with the *Payment system* in order to retrieve payments information.

The customer, via the mobile app or the services web page, can initiate a communication with the *Application server* that will complete the task requested and will answer back immediately with a message about the success or failure of the request and, if other information are needed, will send an email or a push notification providing the missing part (e.g. the client reserves a car, the request is successful and the system answers with a push notification/email containing the code to unlock the car). Then the maintenance service can communicate with the system through their own web application expressly developed in order to this purpose using our API's. Here the maintenance personnel can look at all the stored maintenance requests and change the status of this requests. (e.g. a request of a car left with less than 5% of the battery might be changed to *performing*, and after the car is put in charge its status will be changed in *complete*). In the right part of Figure 2 there is the *Payment system*. This system is supposed to interact with banks and the Application server in order to perform checks and updates all the information regarding payments. The banks will use this interface to communicate with our system, and the *Payment system* will act as a controller and will request the *Application server* to perform the needed changes to the database in order to maintain the correct status of payments and users. This is due to a previous decision stated in the RASD *(Table 13: Provide a payment)*. The *Application server* needs to store and perform queries in order to perform all the tasks and so communicates with a database.

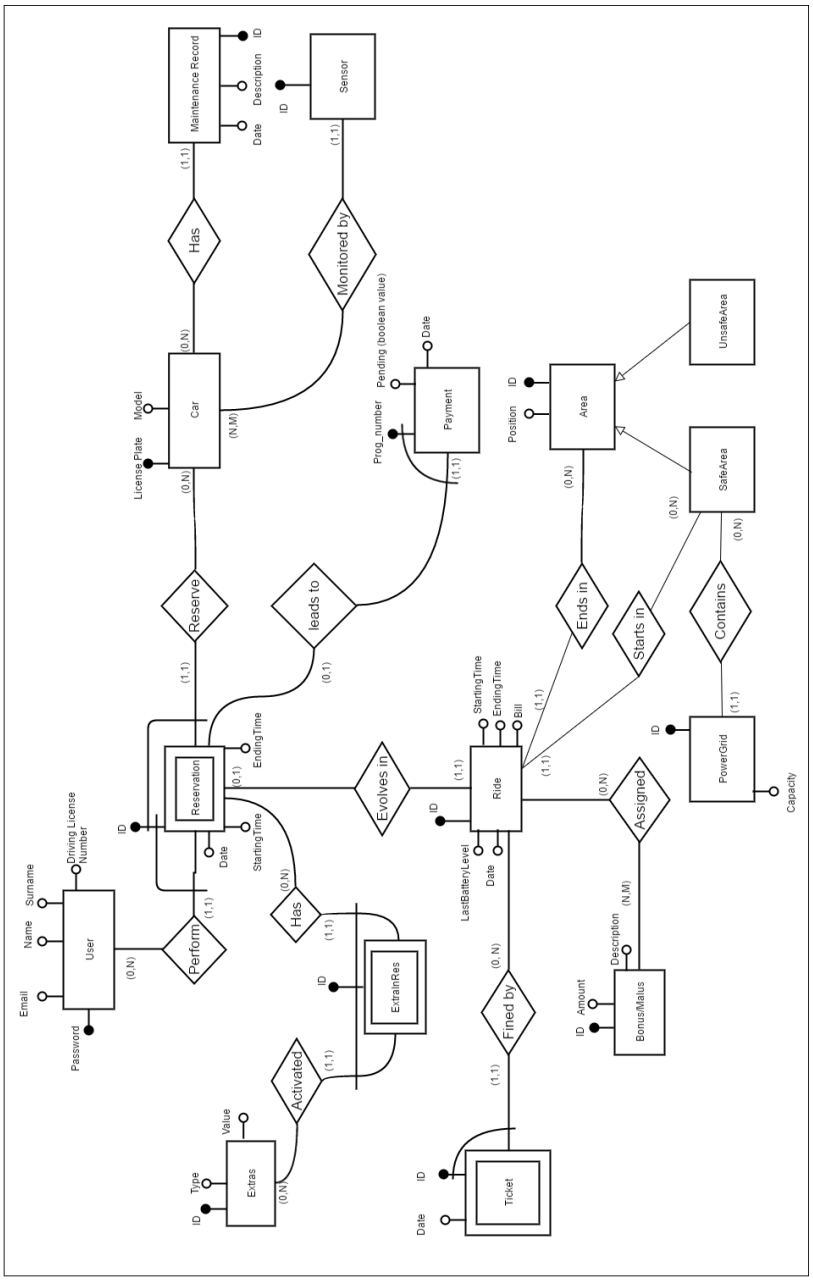

Figure 4: ER diagram

Observation: ID in *Reservation* entity, combined with the user and car's informations, will later be converted into QR Code through cryptography algorithms to enable the user to access the car that he has previously reserved. *ExtrainRes* is the set of all the operations that have been performed on a specific reservation that can involve an addition in the payment of the total cost, in our case just the *extension* of the reservation time and the deletion (*delete*) of the reservation itself.

2.3 Component view

The component view consists of the following components:

- **Client**: clients device used to connect to the service
- **Maintenance**: maintenance device used to interact with the system in order to provide maintenance features.
- **Payment system**: third party device used to provide information about payments, their completion and other relating information.
- **Request manager**: manager used to receive request from the outside and to route them to the correct controller.
- **Login/Registration controller**: manage login and registration aspects
- **Reservation controller**: manage reservation aspects.
- **Ride controller**: manage ride aspects.
- **Cars controller**: manage all informations related to cars, sensors and their interaction with the users.
- **Collector data**: manage the data collection from different sensors
- **Payment controller**: manage payment aspects like register debts or mark their completion.
- **Communication manager**: component used to manage information exchange between different controllers and between them and the model.
- **Notification helper**: manage notification aspects, routing them to the correct gateway
- **Push gateway**: manage the sending of push notification to mobile app.
- **Email gateway**: manage the sending of Email messages.
- **Sensor gateway**: handle the sending and receiving of data collected by the various sensors present on cars.
- **Maintenance gateway**: manage the sending of notification to the maintenance company
- **Payment gateway**: manage the sending of payment information to the third party payment service
- **Model**: model of the world, used by the controllers.
- **Database**: database used to store persistent data.
- **Google Maps API**: API service used by the ride controller to provide navigation information.

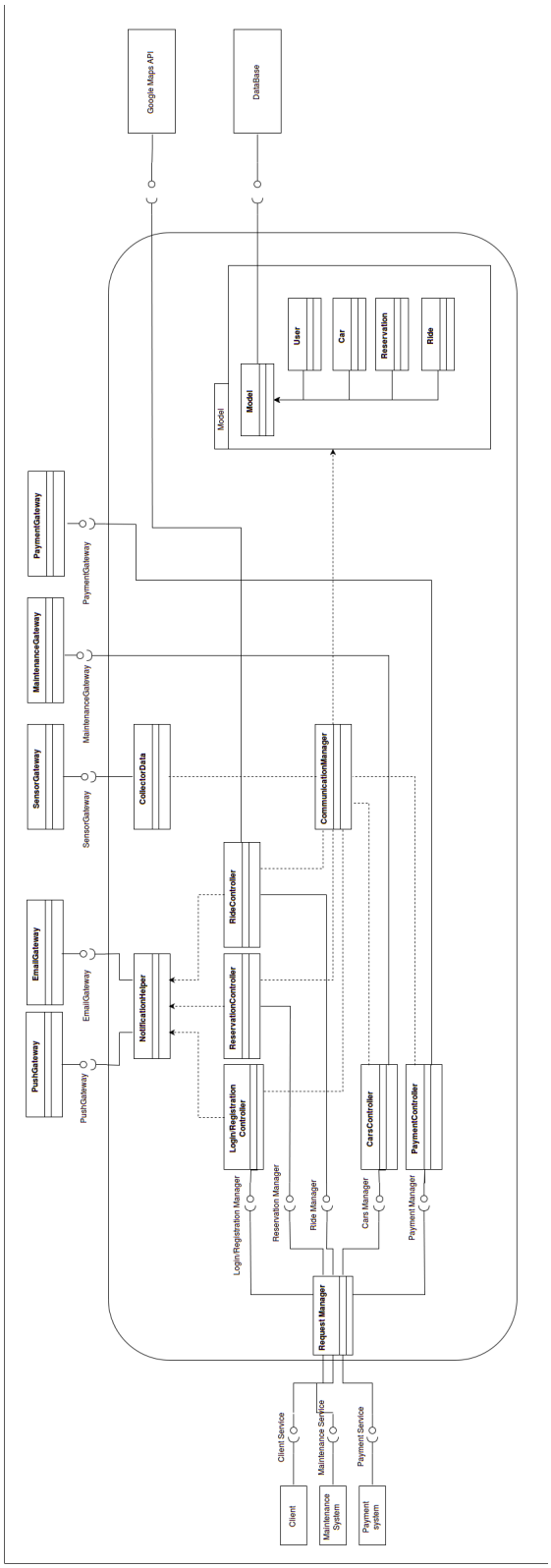

Figure 5: Component view

The interaction between the system and users is provided by a series of interfaces, one for each type of actor. All requests are handled by the single component *Request Manager* that will provide for their proper forwarding within the system. We have designed a system consisting of several controllers, each dedicated to a specific type of functionality. This allows us to maintain a high level of cohesion of the different components, allowing, in case of future expansions of the service, to update the individual components smoothly and with great flexibility. The division of functions in various components will also allow us to efficiently test the service provided. In order to maintain a high degree of flexibility, we decided to introduce a component *Communication manager* dedicated to the sorting of the information exchanged between the controllers themselves and the model, thus avoiding the definition of direct links between controllers that might obstruct and make more complex a future possible expansion of the service. In analogous way we introduced a component *Notification helper* dedicated to the proper addressing of the notifications to users, thus separating the functionality of sending of notifications from the logic of the underlying service. As shown in the diagram, the only access to the model and to the database it is via the *Communication Manager* that will carry out read and write functionality.

2.4 Deployment view

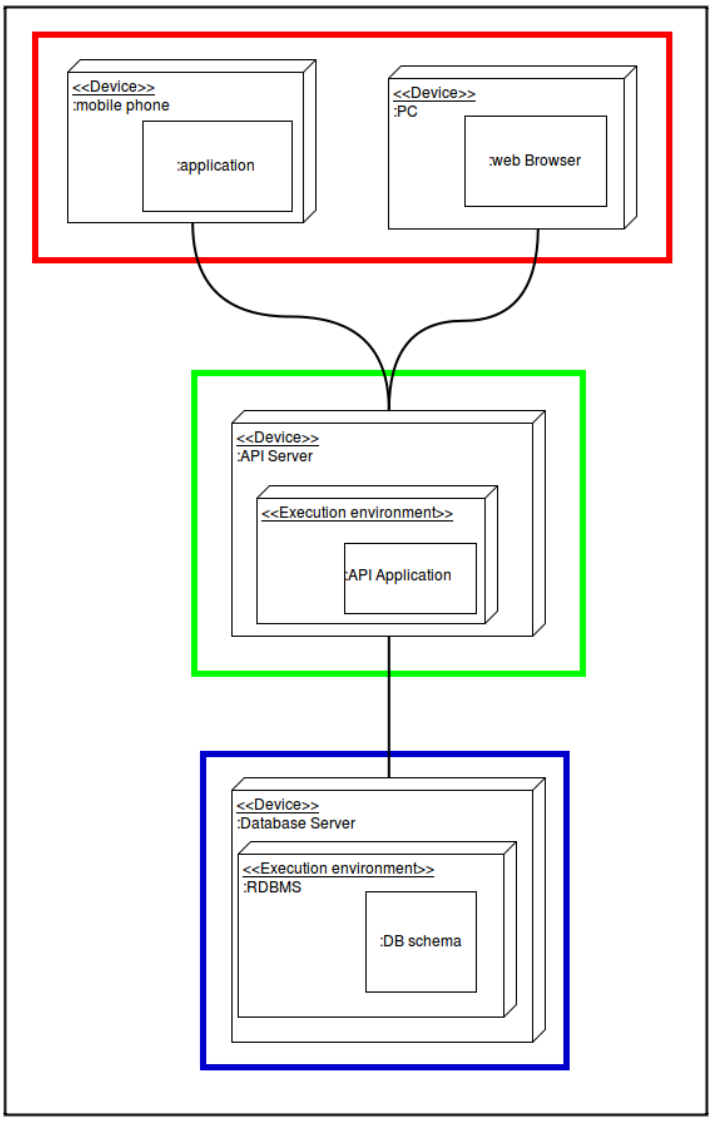

Figure 6: Deployment view

2.5 Runtime view

2.5.1 Login

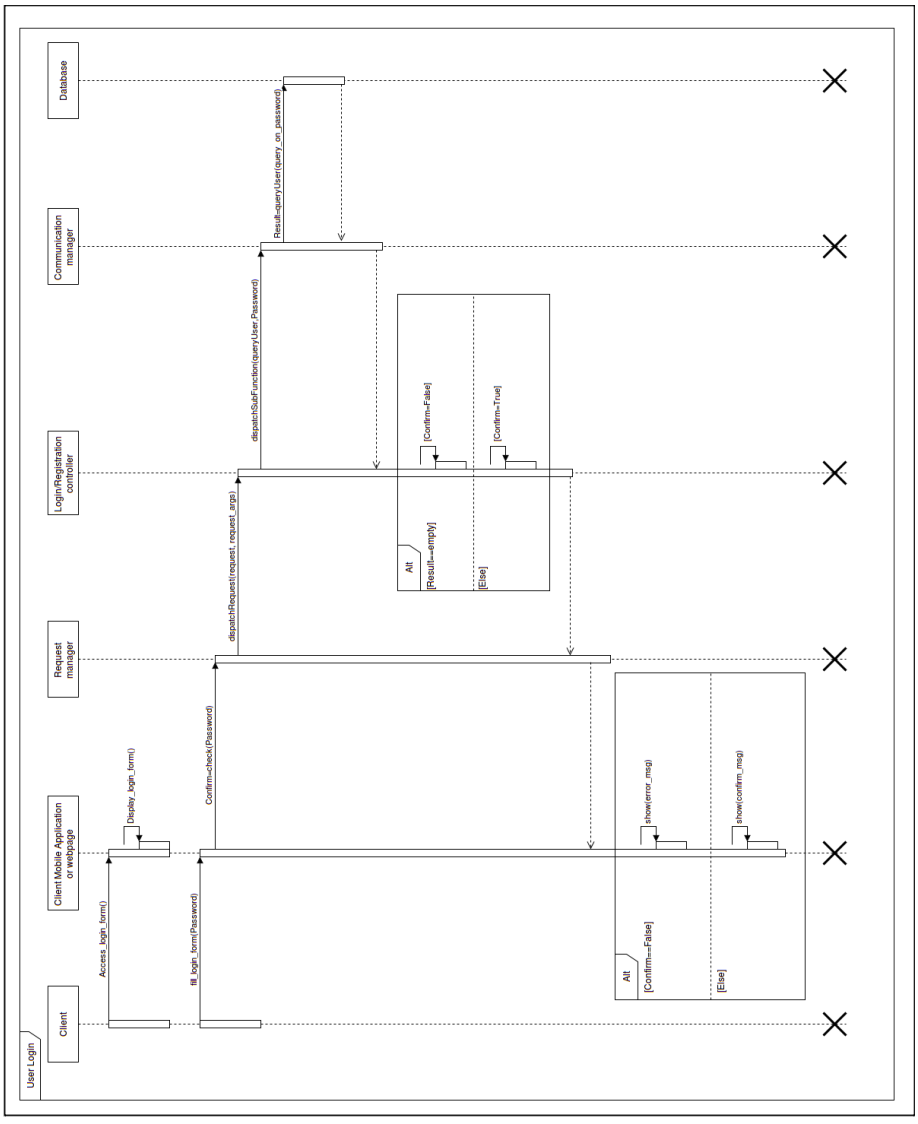

Figure 7: Sequence diagram - login

In this sequence diagram it is represented the interaction among the components of the system in order to verify the correctness of the password entered by a user. After completing the login form, the request is forwarded to the *Request manager* that, once identified, will forward it to the correct controller (in this case the*Login / Registration controller*). It, to deal with the request, will perform a query on the database of the system in order to verify the presence or absence of a user with corresponding password to the one supplied. The new

request is then forwarded to the *Communication manager* that ultimately will apply it to the DB. The query result will be interpreted by the *Login / Registration controller* that will send the confirmation, or not, of the login to the interface which started it. In this sequence diagram it is not represented the collection of all the information needed to display the user's homepage because it will be made only after effective login confirmation.

2.5.2 Report car issue

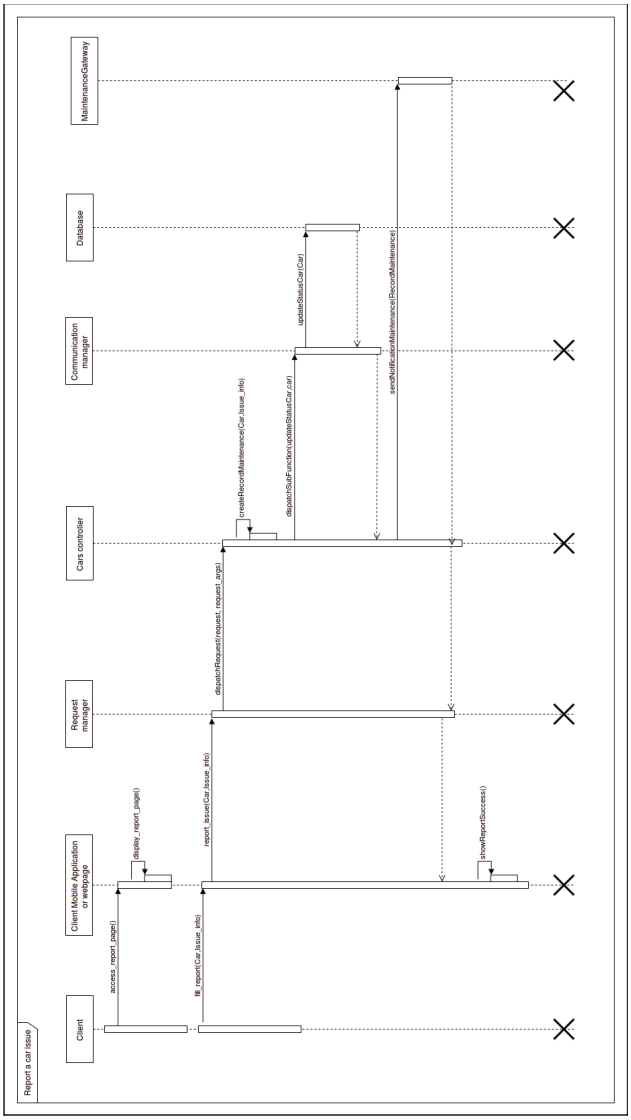

Figure 8: Sequence diagram - report car issue

In this sequence diagram it is shown the flow of information and instructions

exchanged between the components for reporting car issue. The report starts from a user, that, with his own application fills out a form with all the details regarding the car and the problem. The request is now forwarded to the *Car controller* that will seek to create a record for maintenance and will save it to the database, associating it to the corresponding car. Once this is done, the same *Car controller* will perform a notification to the maintenance company that later will have access to the report and provide for the resolution of the problem.

2.5.3 Money saving option

This sequence diagram shows the flow of instructions that allow the activation of the money-saving option during a ride. The user must provide his final destination in order to compute the shortest route to the nearest powegrid station to his destination. Once the driver has selected his destination, the system in the vehicle will interface to the PowerEnJoy server to provide the selected service: the request is received by the CollectorData which in turn will communicate to the RideController; the features of this controller are not limited only to the definition of rides and their stages, but will contain all the logic required for the above described functionality. The Ride Controller will perform the proper algorithm developed to find the right PowerGrid in order to reach the purpose of this functionality. Once that is done, the algorithm will have figured out the position information of the proper PowerGrid and then the controller will interface with the APIs provided by Google Maps to compute the road. Finally, it will send the result of this computation to the driver who asked it.

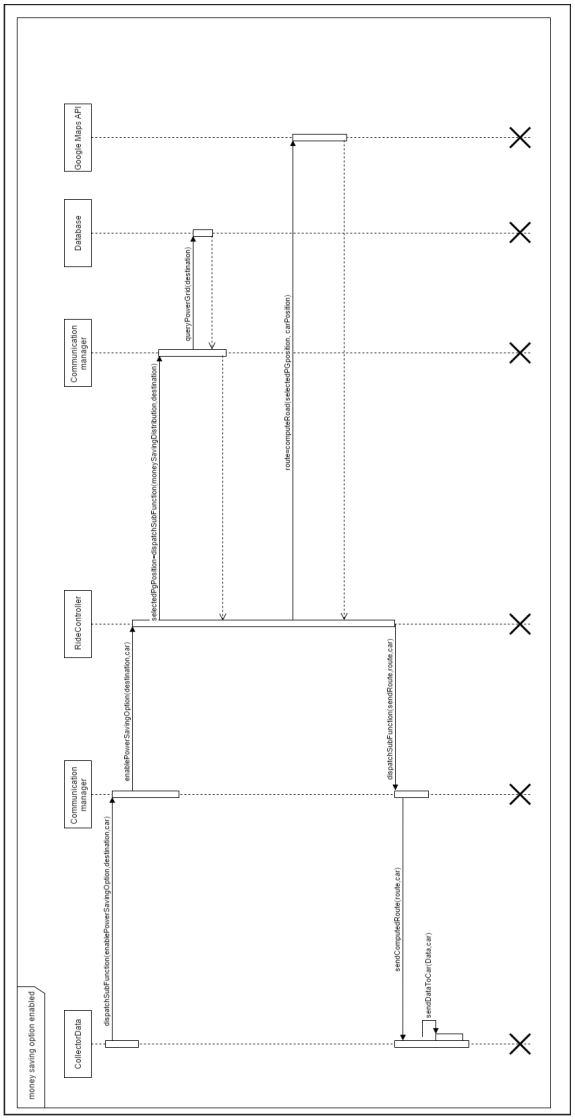

Figure 9: Sequence diagram - money saving option

Observation: *Communication manager* is here duplicated only for meanings of readability, in the system-to-be is only a single instance.

2.6 Component interfaces

In this section are explained the interfaces that the system provides to the applications that connect to it. For each of them it is described the methods that are made available and the parameters necessary for their proper execution.

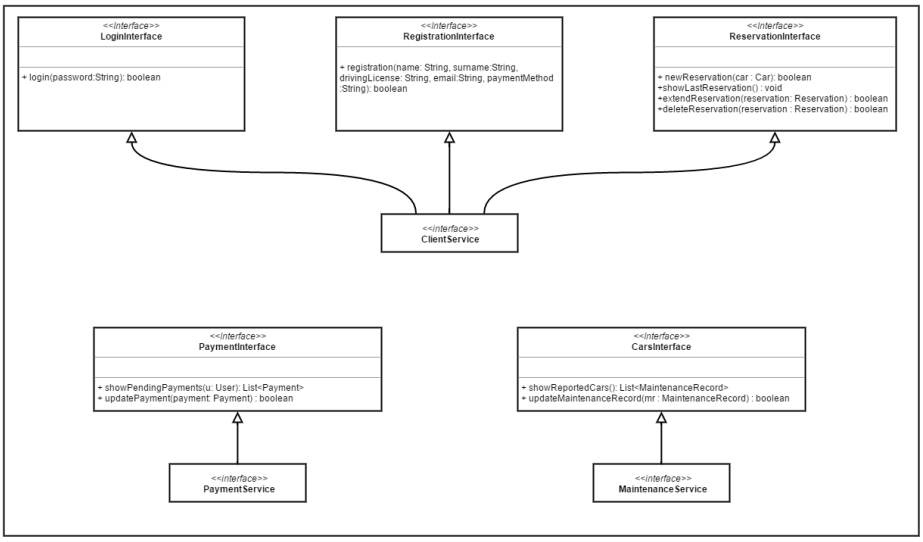

Figure 10: Component interfaces

2.7 Selected architectural styles and patterns

Reflecting on the architectural design of the system-to-be, we identified some styles and patterns that best reflect our idea of design, as well as provide us with a solid structure on which to base our development.

Client-Server

Since the first description of the system-to-be we identified in the model a distinction between the logic of the system and the users themselves; this type of functionality distinction is reflected in the architecture of client-server. In particular, we decided to develop an architecture based on 3-Tier where the client is of type thin. This allowed us to keep the client side as light as possible, so as to concentrate the computational power of the devices on only GUI aspects. On the other hand, the large computational load required by the servers will be guaranteed, both from a reliability and efficiency point of view, through the use of cloud computing, which will allow us to have an easily scalable service. A client server architecture of this type, in addition to the advantages mentioned above, allows us to guarantee a high level of security, because the clients do not directly access the data contained in the database as well as have no dependence on each other. Finally, since the clients use interfaces provided by the server,

and does not compute any functional activity, future service enhancement will be completely transparent to all clients themselves.

Adapter

As visible in the *Component View* (§2.3), we designed our system dividing the functions that must be provided in multiple controllers,each covering a single aspect of the system. This, as previously described, allows us to maintain a high degree of cohesion between the various components, as well as allowing us to manage them in a more elastic way. However, the breakdown of the more system controllers introduces a problem concerning the exchange of information between them. The system will not present direct links between these but, through the Communication Manager will implement the Adapter pattern. This will permit to separate the development of a component from the surrounding ones, to which ,using interfaces, will be fully transparent.

MVC

The Model-View-Controller pattern will cover an important role in our system. As described in the component view, we decide to separate the internal representations of data from the ways that information is provided to the users. The view is composed by all kind of clients that can connect and operate on the system-to-be. The controller part is our application server and it contains all the logic of the system. The model cover the rule of managing the data of the service itself. With the use of this pattern we do not permit any direct access to data from clients, without going through the controller section.

2.8 Other design decisions

Since that our service must ensure a high degree of efficiency, both in terms of response times and of the number of users connected simultaneously, we suggest the development of the web server on NginX. This type of web server is in fact able to ensure indexes of performance as mentioned above, basing its computation on an asynchronous approach to events. The server will be based on non-blocking operations, thus able to not slow down the overall operation in case of high level of parallelism, as it would happen in Apache web server.

3 Algorithms design

In this section we want to provide a short description of some useful code that will be implemented in the future system.

Discounts' assignation

This algorithm will manage to properly assign discount at the end of a ride:

```
public List<Discount> computeDiscounts(Ride ride){
    List<Discount> discounts=new ArrayList<Discount>();
    Car car=ride.getCar();
```

```
    if(car.isInCharge())
        discounts.add(new PowerGridDiscount());
    else{
        if(car.getPosition().distanceFromNearestPG()>3){//3km
            discounts.add(new MalusDistance3());
        }
        else{
            if(car.getCharge()<=20)    //20%
                discounts.add(new MalusBattery20());
        }
        if(car.getCharge()>=50)    //50%
            discounts.add(new Battery50Discount());
    }
    if(ride.numPassengers()>2)
        discounts.add(new PassengersDiscount());

    return discounts;
}
```

Ride's end

This algorithm will enclose all the functions required to terminate a ride. We consider the execution starting point as the moment at which it takes place the closure signal provided by the concerned machine.

```
public void endRide(Reservation res){
    Ride ride=res.getRide();
        if(ride==null) return;


    if(!ride.getCar().getPosition().isSafeArea())
        MaintenanceManager.report(ride.getCar());



    List<Discount> discounts=new ArrayList<Discount>();

    //Set informations
    ride.setFinalPosition(ride.getCar().getPosition());
    ride.setEndingTime(System.currentTime());

    //Computation price
    float price=res.calculatePrice();

    discounts=DiscountManager.computeDiscounts(ride);
    for(Discount d: discounts)
        price += d.getValue()*price;
```

```
    res.register(price);//Will notify the payment system
    res.close();//Storage on DB
}


public float calculatePrice(){
    float totalPrice=0;
    Ride ride=this.getRide()
    totalPrice+=ride.getBill();//Bill related to the only travel time
    List<Extras> extras=ExtraInRes.getAllExtras(this.getId());
    for(Extras e:extras){
        totalPrice+=e.getValue();
}
    return totalPrice;
}
```

ExtrainRes is the set of all the operations that have been performed on a specific reservation that can involve an addition in the payment of the total cost, in our case just the *extension* of the reservation time and the deletion (*delete*) of the reservation itself.

Reservation's extension

This code will be executed when a user want to extend his reservation.

```
public void extendReservation(Reservation res){
    res.setEndingTime(
        new Time(res.getEndingTime().getHour+1,
                 res.getEndingTime().getMinute(),
                 res.getEndingTime().getSecond());

    ExtraInRes.createNewExtra("extension", res.getId());
}
```

This algorithm, as previously mentioned in the RASD, will prevent a user carries out a number of reservations, without extending them, causing financial damage to the company.

```
public Reservation newReservation(User user,Car car){
    if(lastCarReserved(user).equals(car) &&
        timeLastExpiredReservation(user)<30)
        return null;
    else
        return new Reservation(user,car);
}
```

Money saving option - Car's distribution

This algorithm will implement the uniform distribution of cars when enabling the money-saving option.

```
public Position moneySavingDistribution(Position destination){

    List<PowerGrid> powerGrids=DBManager.getAllPowerGrid();
    List<PowerGrid> selected=new ArrayList<>();
    for(PowerGrid pg: powerGrids){
        if(pg.isReachable(destination))
            selected.add(pg);
    }

    HashMap<SafeArea,Integer> map=new HashMap<>();
    for(PowerGrid pg: selected){
        SafeArea s=DBManager.getAreaByID(pg.getAreaID());
                if(map.get(s)==null){
                        map.put(s,pg.getCapacity());
                }else{
                        int val=map.get(s);
                        val+=pg.getCapacity();
                        map.put(s,val);
                }
    }

    SafeArea safeWithMin=s;
    int min=DBManager.getNumberOfCars(s);
    for(SafeArea s2: map.keySet()){
        int cars=DBManager.getNumberOfCars(s2);
        if(cars<map.get(s2)){
                        if(cars<min){
                        safeWithMin=s2;
                                min=map.get(s2);
                }
        }
    }
    return safeWithMin.getPosition();
}
```

4 User interface design

4.1 Mockups

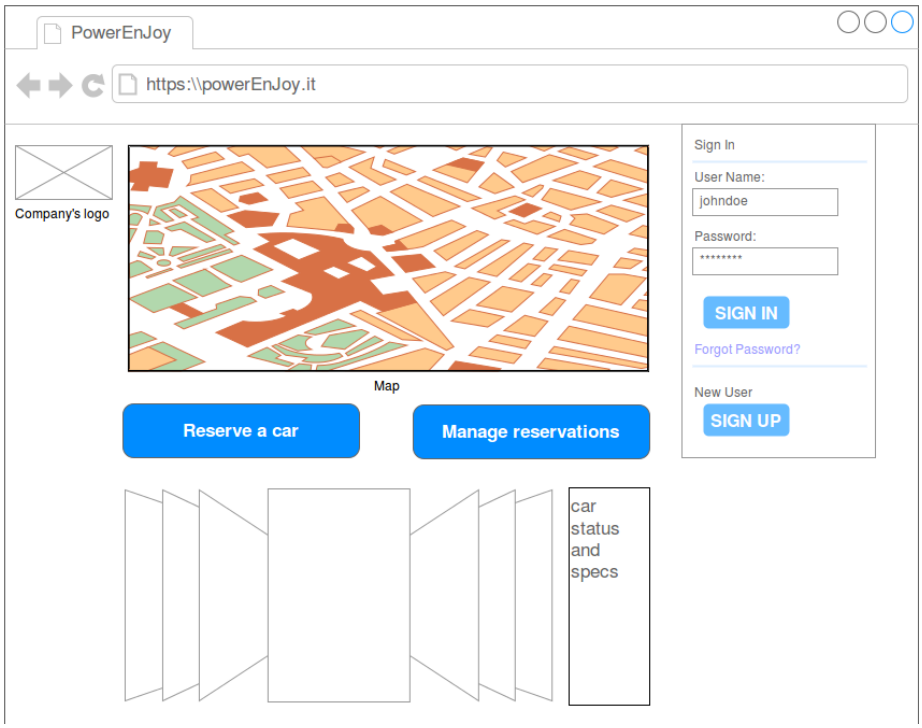

Figure 11: Mockup - Home page web application

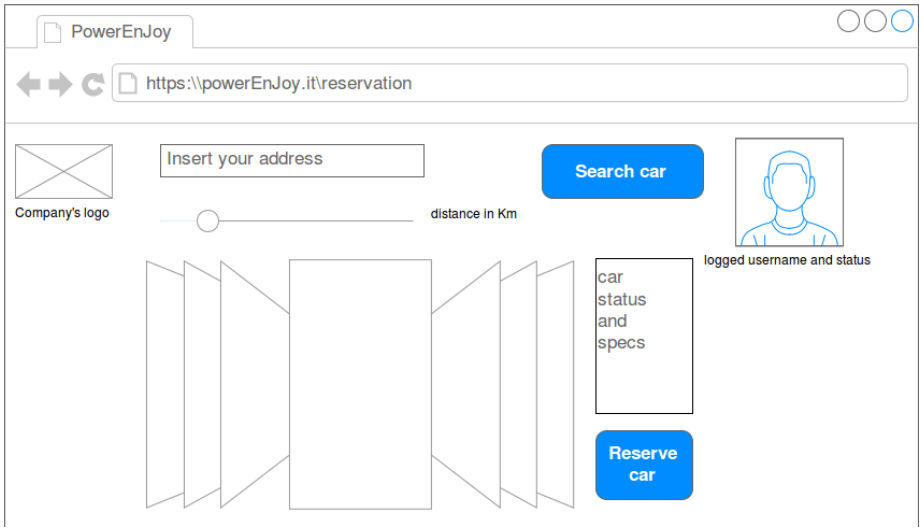

Figure 12: Mockup - Reservation page web application

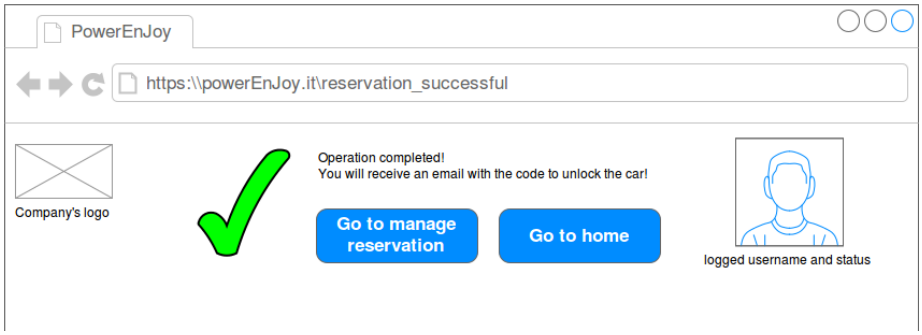

Figure 13: Mockup - Reservation successful page web application

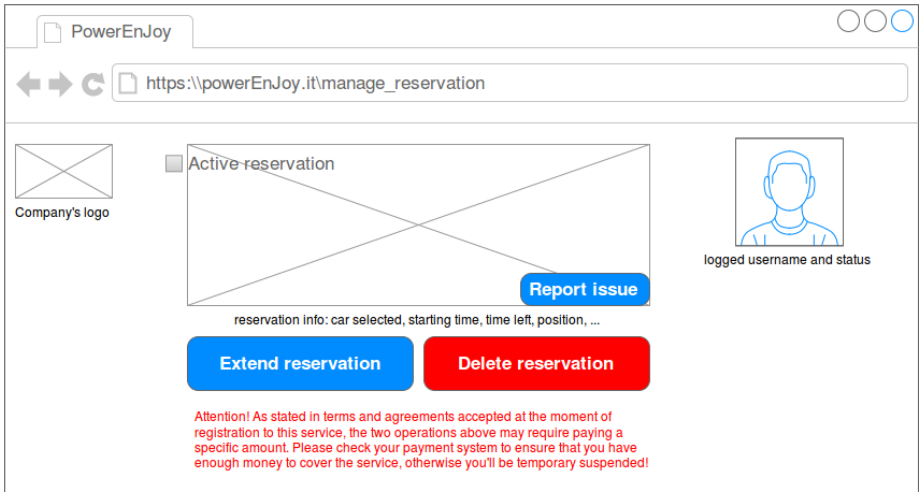

Figure 14: Mockup - Manage reservation page web application

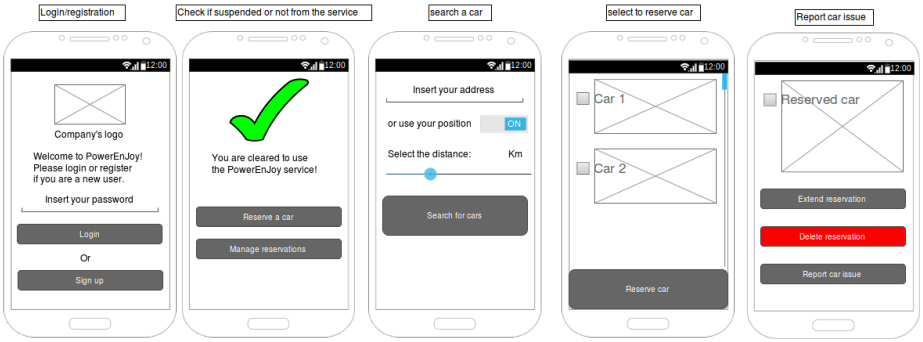

Figure 15: Mockup - Mobile app

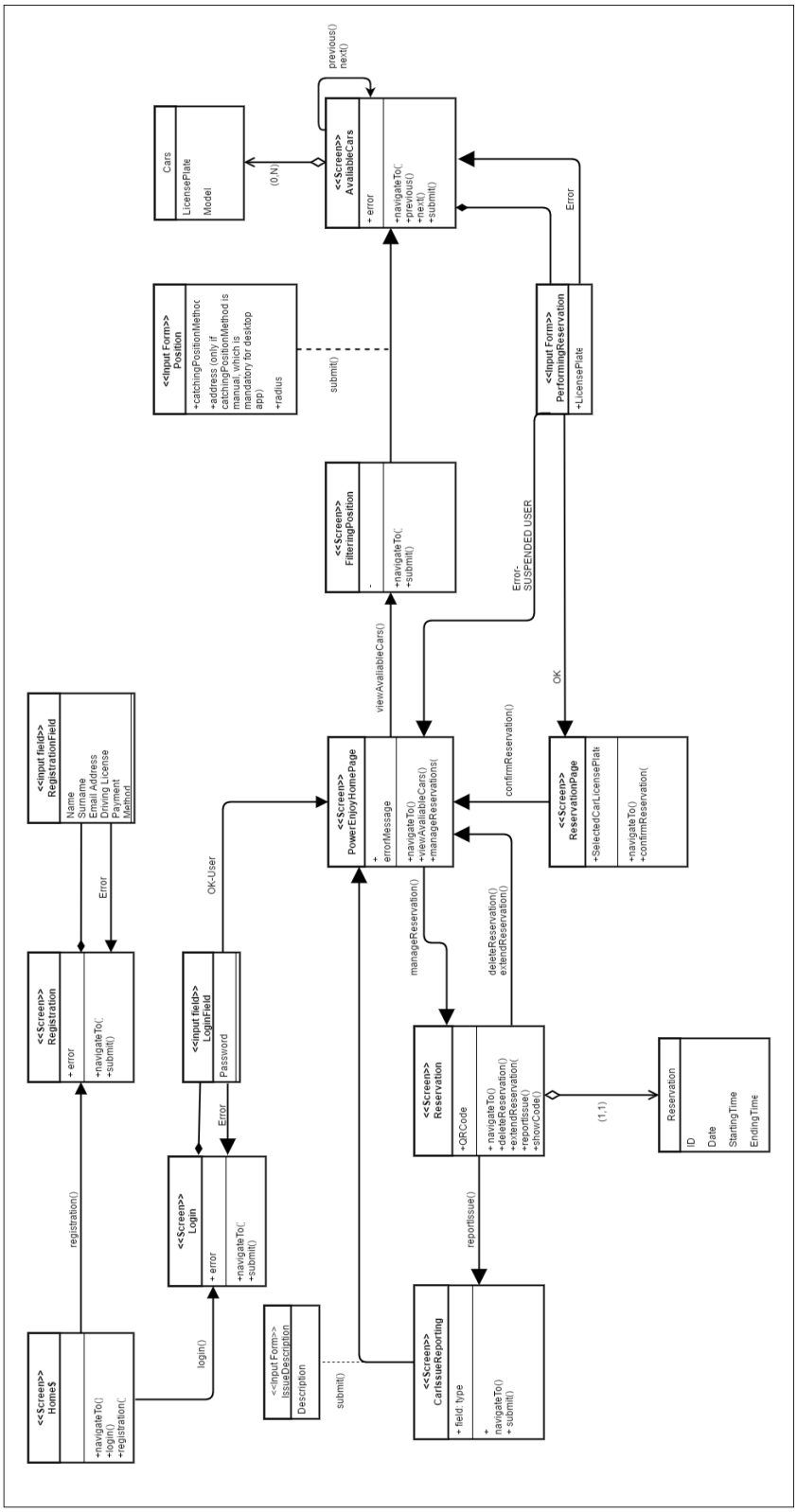
previous()
next()
Cars
LicensePlate
Model
(0,N)
<<Screen>>
AvailableCars
+ error
+navigateTo(
+previous()
+next()
+submit()
Error
<<Input Form>>
Position
+catchingPositionMethod
+address (only if catchingPositionMethod is manual, which is mandatory for desktop app)
+radius
submit()
<<Input Form>>
PerformingReservation
+LicensePlate
<<Screen>>
FilteringPosition
-
+navigateTo(
+submit()
Error-
SUSPENDED USER
viewAvailableCars()
OK
<<Input field>>
RegistrationField
Name
Surname
Email Address
Driving License
Payment
Method
<<Screen>>
PowerEnjoyHomePage
+
errorMessage
+navigateTo()
+viewAvailableCars()
+manageReservations(
confirmReservation()
<<Screen>>
ReservationPage
+SelectedCarLicensePlate
+navigateTo()
+confirmReservation(
<<Screen>>
Registration
+ error
+navigateTo(
+submit()
Error
OK-User
<<Input field>>
LoginField
Password
manageReservation()
deleteReservation()
extendReservation()
<<Screen>>
Reservation
+QRCode
+ navigateTo()
+deleteReservation()
+extendReservation(
+reportIssue()
+showCode()
(1,1)
Reservation
ID
Date
StartingTime
EndingTime
registration()
<<Screen>>
Login
+ error
+navigateTo(
+submit()
Error
reportIssue()
<<Screen>>
Home$
+navigateTo(
+login()
+registration(
login()
<<Input Form>>
IssueDescription
Description
submit()
<<Screen>>
CarIssueReporting
+ field: type
+
navigateTo()
+ submit()

Figure 16: UX

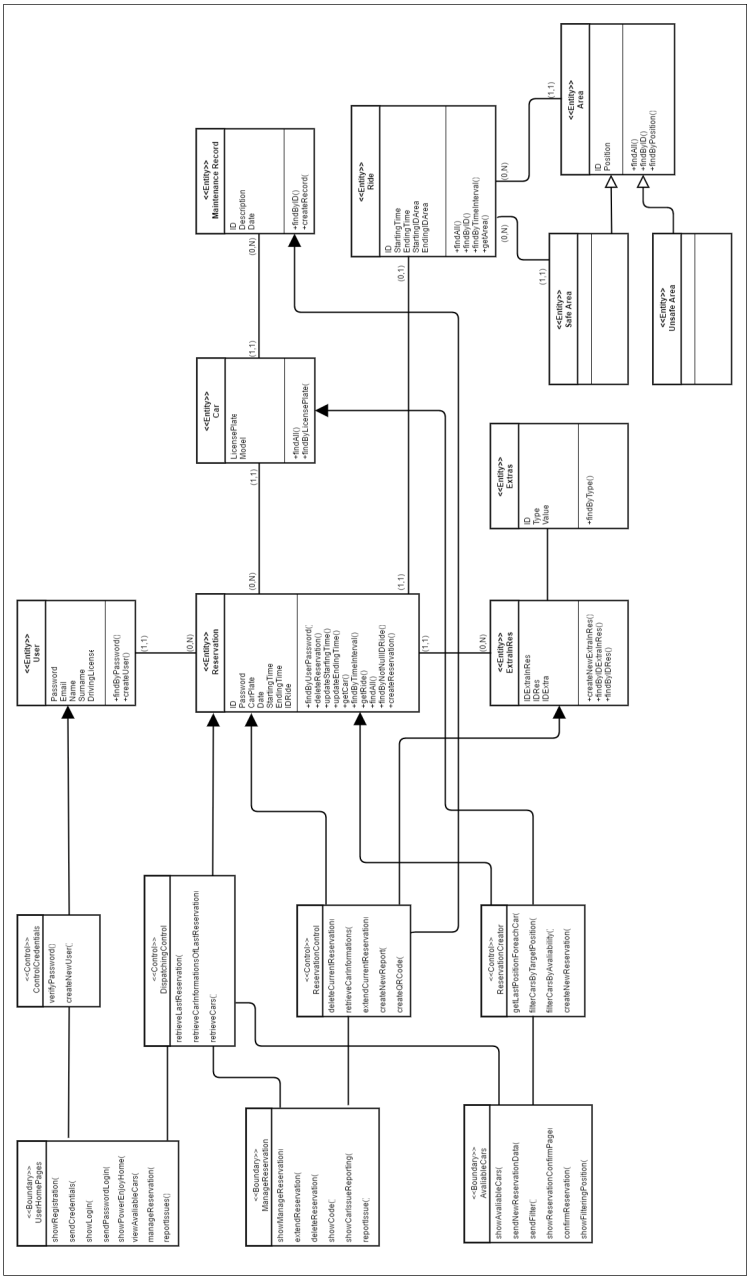

Figure 17: BCE - In this diagram the cardinality referred to the *Entity* are opposite with respect to the *ER diagram*.

4.2 APIs

4.2.1 Maintenance system

In order to keep all the cars working and having a more accessible service, as number of cars available for a user, we have decided to rely on the operations of an external company that will work on the maintenance of the cars. We decided to furnish the company with our API for car maintenance such that all the information about a *malfunctioning* car are handled by our partner. In this case, the term *malfunctioning* means all possible problems that our electric cars can face: for instance the battery charge is below the 5% or other problems that the *car sensors subsystem* stated in *high level components diagram* in Figure **??** at page **??** or even a user-reported problem. Our system-to-be will store the information regarding all the *malfunctions* and will interact with the *maintenance system* via our API. The maintenance company will be able to read the problems related to every car and will modify the status of the problem: e.g. a car has a battery charge below 5%, the maintenance personnel will mark as *Work In Progress (WIP)* the record of the service, the personnel will perform all operations needed in order to put the car in charge and then will mark the record as *completed* so that the car can be available again. The operations performed by the maintenance company are of no interest for our system: what matter is the fact that the car is out of service for a variable period of time and then back available again.

4.2.2 Payment system

In order to keep track of all performed payments and to ensure a *fair* use of the PowerEnJoy service, our system-to-be will be interfaced with different payment systems (right part of *high level components diagram* in Figure **??** at page **??**). The *fair* term, in this context, is based on the fact that an user can possibly have pending payments: in this case the system will not allow the insolvent user to request for other reservation. We will furnish all recognized payment systems with our API for money transfers that will send a request to the user-specified payment system of a bill that must be paid. The payment system will answer and, if the answer is positive and the transaction has been successful, the client is able to continue using our car sharing service. Otherwise, in a negative answer situation from the payment system, our system will temporary suspend the user from the service, until a new and affirmative notification from the payment system will be received.

5 Requirements traceability

This last section is devoted to the representation of how the goals and requirements identified and expressed in previous document (RASD) now find their concretization in the design choices made. Provided below is a list of goals previously identified and, for each of them, there are expressed the components and architectural aspects which allow their realization

[G1] Guest must be able to register himself in the system.

- Request Manager

- Login/Registration Controller
- Notification Helper
- Request Manager
- Login/Registration Controller
- Notification Helper

[G2] User must be able to log in and use the system, if registered.
- Request Manager
- Login/Registration Controller
- Reservation Controller
- Notification Helper
- Communication Manager

[G3] User must be able to insert a specified address or his current location to find available cars within a certain distance selected by the user himself.
- Request Manager
- Reservation Controller
- Communication Manager
- Cars Controller

[G4] User must be able to reserve a car for up one hour in advance.
- Reservation Controller
- Notification Helper

[G5] User must pay a fee of 1 euro if the reserved car is not picked up within an hour from the reservation.
- Reservation Controller
- Communication Manager
- Payment Controller
- Cars Controller

[G6] User must be able to extend his reservation with the payment of a further charge.
- Request Manager
- Reservation Controller
- Communication Manager
- Payment Controller
- Cars Controller

[G7] User must be able to notify to the system that he is nearby a reserved car in order to pick it up.
- Reservation Controller
- Cars Controller
- Communication Manager
- Collector Data

[G8] User must be able to visualize the current charge of his fare.

- Ride Controller
- Communication Manager
- Collector Data

[G8] As soon as the engine ignites, the charge is calculated based on the duration of the service.

- Reservation Controller
- Ride Controller
- Communication Manager
- Collector Data

[G9] User must be notified if his reserved car becomes out of service and it is no longer available.

- Cars Controller
- Communication Manager
- Request Manager
- Maintenance Gateway
- Notification Helper

[G10] User must be able to notify to the company if the reserved car is damaged or not properly working, during their ride or before to pick up the car.

- Request Manager
- Communication Manager
- Cars Controller

[G11] Users could achieve different discounts if they satisfy different conditions.

- Ride Controller
- Cars Controller
- Collector Data
- Communication Manager

[G12] It is applied a further 30% on the ride if the car is parked in one or both of this situations:

1. The car is left at more than 3Km from a power grid.
2. The car is left with less than 20% of the battery.

- Cars Controller
- Communication Manager
- Collector Data
- Ride Controller
- Payment Controller

[G13] User must be able to enable a money-saving option.

- Collector Data
- Communication Manager
- Ride Controller

When we refer to the *Request Manager* we implicitly also intend the use of the interfaces provided by our system to the various applications.

Used tools

- Github: for version control
- GoogleDoc: to write the document
- Draw.io: to create the diagrams
- LaTeX: to create the pdf

Work hours

- Emanuele Ricciardelli: $\sim$ 30 hrs.
- Giorgio Tavecchia: $\sim$ 30 hrs.
- Francesco Vetró: $\sim$ 30 hrs.