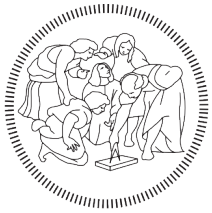


POLITECNICO
MILANO 1863

Design Document for PowerEnJoy

Daniele Riva* Marco Sartini†

February 7, 2017

version 1.1

*matr. 875154
†matr. 877979

Contents

1 Introduction

1.1 Purpose

This Design Document document has the purpose to describe in detailed way the architecture of the system, the data structures, the user interface, the components.

1.2 Scope

This document is produced in the *Power EnJoy* context, a project which aim is to implement an electric car sharing service supported by a mobile and a web application.

1.3 Definitions, acronyms, abbreviations

RASD requirements analysis and specification document;

DD design document;

UX User experience;

JPA Java Persistence API;

EJB Enterprise JavaBeans;

JSP JavaServer Pages;

HTTPS HyperText Transfer Protocol over Secure Socket Layer;

Apache Tomcat is a Java Servlet Container;

DBMS Data Base Management System;

1.4 Reference documents

RASD v1.0 available at https://github.com/marcosartini/PowerEnJoy/blob/master/releases/rasdPowerEnJoy.pdf

2 Architectural design

2.1 Overview

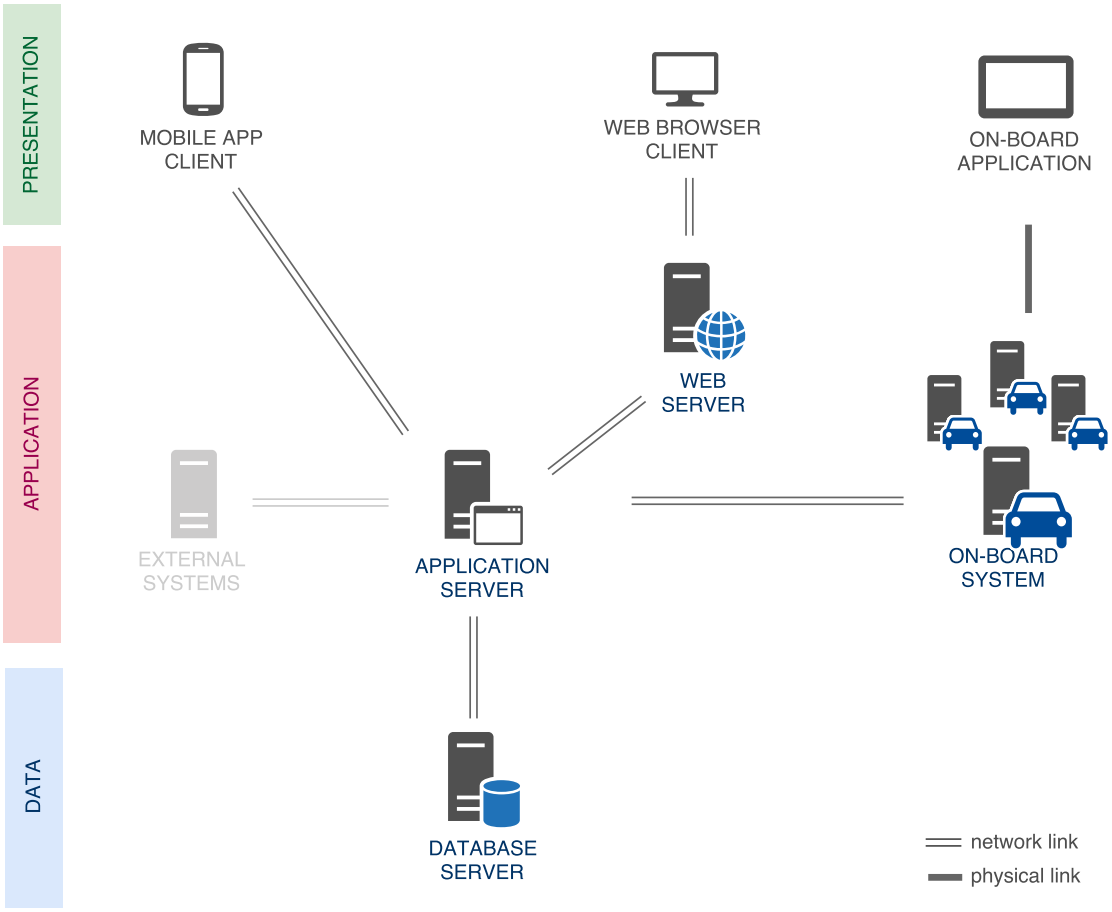


Figure 2.1: Overview of the system architecture: layer division and tier identification.

Analyzing the requirements of the system, we conclude that it is basically organized into three subsystems, and they refer to presentation layer, application layer and data layer.

- the *presentation layer* will carry out the graphical interface between the user and the system; we consider at this layer, as interfaces, the mobile application, the website pages and the screens on the car display;

- the *application layer* will carry out the logic of the system in the whole parts, that means it will react to user requests, it will compute the bills, it will manage the car status, it will execute the dynamic web pages of the website and whatever is is required to implement;
- the *data layer* will carry out to store and maintain the data needed by the system to work.

Each of the three layer is composed by a set of elements to have a more efficient and specialized contribute from each one. The elements are basically physical tiers; the requirements analysis suggest us to adopt the following architecture, that in fact is a multi-tier one.

- data layer will be implemented by a standalone server machine, to better physically separate the application machine from the data machine, and so guaranteeing not to overcharge the application server in the managing of the data; moreover, it will simplify, in a futuristic view, the scalability of the data management without affect the application server.
- each car will contain a small amount of logic, because it is better to have some functions to be directly executed on the cars than always inquired to the application server; furthermore, the car need to be unlocked, and we delegate the car to verify the matching between the user code and the one associated to the rental It will elaborates the location information form the sensor.
- to promptly support and run the website, we decide to adopt a dedicated web server, specifically intended to manage the interactions on that side.
- hearth of the system, the principal tier related to the application layer: the main logic server. This *Application server* will interact with the database server when needs to access the stored data or when it needs to deliver data to be stored; it will interact with the cars on-board systems and also with the mobile application and the website, to satisfy the computational tasks.

The *external systems* icon symbolizes and reminds that the system relies on also other systems to carry out its tasks, such as the interaction with the Payment Handler system and the third party maps service.

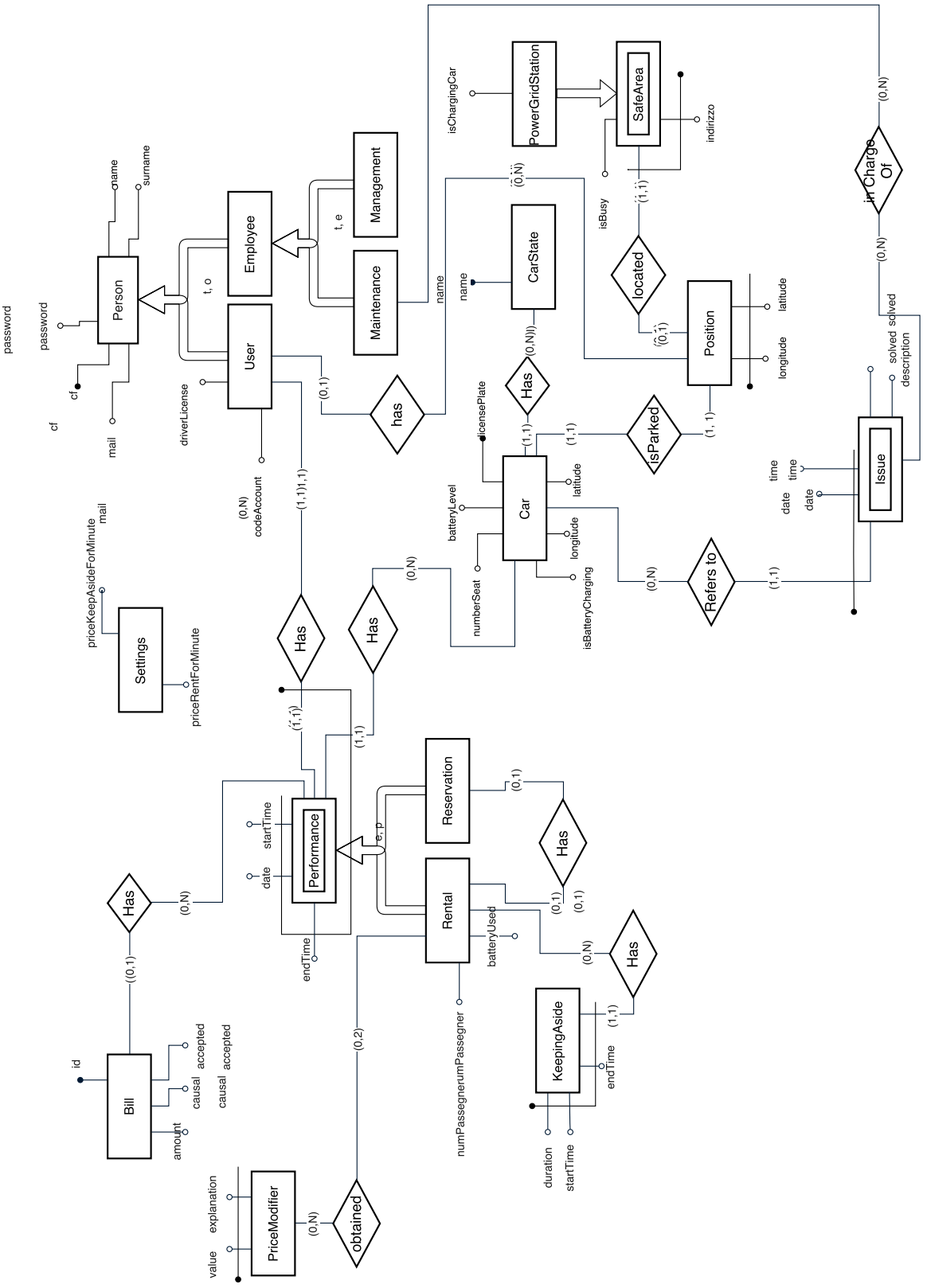
2.2 Component view

In this section the components will be presented individually to show up their behavior, their sub-components, how they interface each other.

2.2.1 Database

The database component will include a DBMS to properly manage the data and their handling. It will interact only with the Application Sever, using a secure interface which

guarantees the security requirements. Data in this component are stored with particular care on encryption. The logical data representation is shown in the E-R schema below. As evident from the schema, a relational database broadly meets the needs of the system.


password
password
cf
cf
Person
name
surname
mail
t, o
driverLicense
User
Employee
t, e
Management
Maintenance
has
name
(0,1)
(0,N)
codeAccount
(1,1)(1,1)
Has
(1,1)
(0,N)
priceKeepAsideForMinute
mail
Settings
priceRentForMinute
Has
(1,1)
(0,N)
Has
((0,1)
(0,N)
id
Bill
causal
accepted
causal
accepted
amount
explanation
value
PriceModifier
(0,N)
obtained
(0,2)
Performance
startTime
date
endTime
t, p
Reservation
Rental
batteryUsed
numPassegnerumPassegner
(0,1)
Has
(0,1)
(0,1)
(0,N)
Has
KeepingAside
(1,1)
endTime
duration
startTime
isChargingCar
PowerGridStation
SafeArea
indirizzo
(0,N)
isBusy
(1,1)
located
(0,1)
Position
latitude
longitude
CarState
name
(0,N))
Has
(1,1)
licensePlate
(1,1)
IsParked
(1, 1)
batteryLevel
Car
latitude
longitude
numberSeat
isBatteryCharging
(0,N)
Refers to
(1,1)
time
date
time
date
Issue
solved
solved
description
in Charge Of
(0,N)
(0,N)

2.2.2 Application server

The application server contains the main logic aspects of the system, except the ones performed by the *on-board system*; the features are implemented by the modules described below. The application server interfaces with the data layer using a dedicated connector responsible for the communication between the two components.

The application server is composed by:

UserManager this module will take care to handle login requests, registration requests and what concern with this operations such as adding users to the database, generating passwords, checking data consistency and so forth;

CarsManager this module will take care of car management regarding the status of the car with the respect to the possibility of usage, it will also handle the requests of check availability of a car and modify its status following a reservation or a rental and it will maintain updated the general state of the fleet;

ReservationManager this module will take care of the reservations, included setting them up, avoiding multiple reservation per same car and/or per same user simultaneously, implementing services to comply with timeout constraints. This module interacts with CarsManager module and PaymentManager module;

RentalManager this module will take care of the rental aspects, managing rental requests, generating proximity codes, timing the rental, interact with CarsManager, ReservationManager KeepAside Manager and PaymentManager;

KeepAsideManager this module will take care of the *keep aside* services, timing the keep aside requests, interacting with the RentalManager;

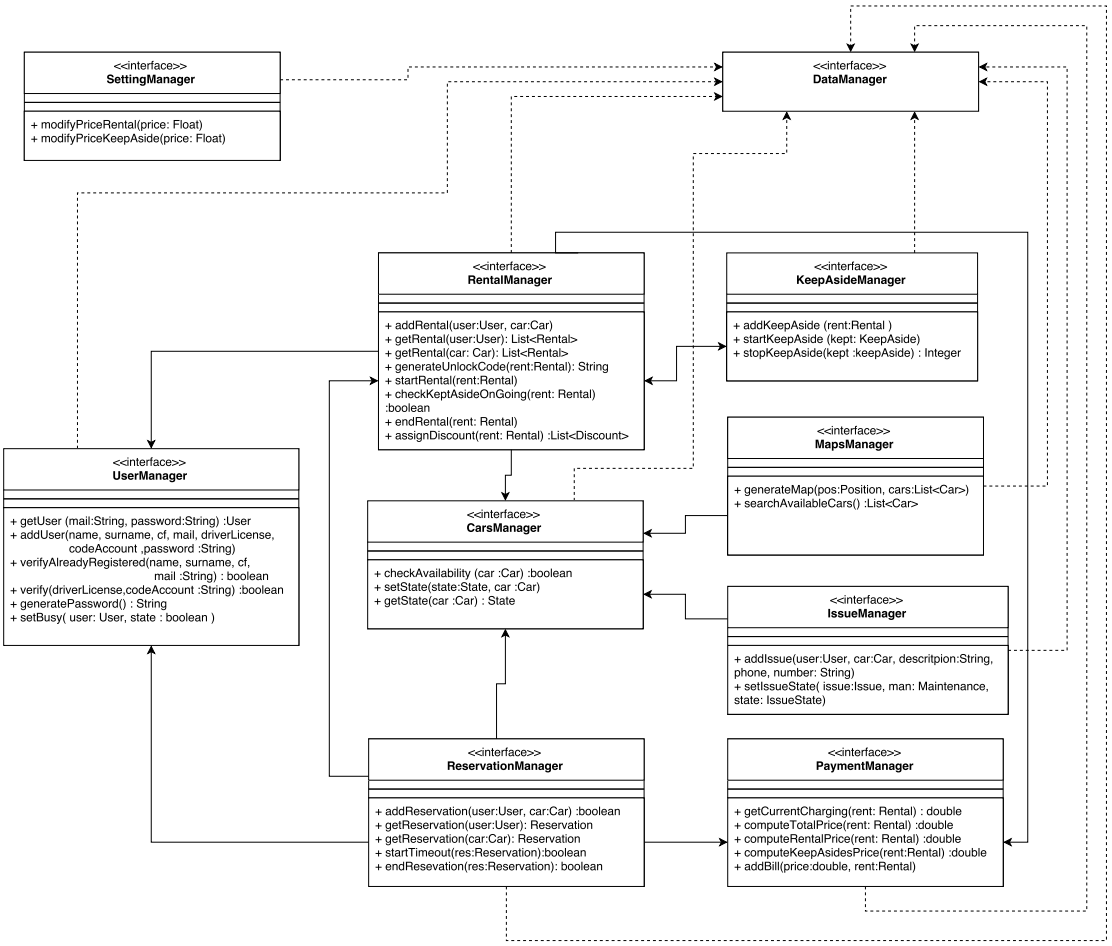
PaymentManager this module will take care of stating the amount of cash to debit to users based on the duration of benefited services, and will also interact with the payment handler (external system) to have the bills paid;

MapsManager this module will take care of composing the map view of the available cars and interact with the chosen maps API. This module interacts with the CarsManager;

IssueManager this module will take care of handling issue and their status;

SettingManager this module will take care of handling the possibility of edit the rental and keep aside cost per minute.

A graphical description of the modules is showed below:



There are two types of different arrows only to make more understable the graph ; they have the same meaning

2.2.3 On-board car system

This component is placed on the car, connected to the car control unit with a dedicated interface able to interpret signals provided by the control unit and also able to interact with it sending proper signals to actuate actuators. It act as a branch of the application server over each car, devoted to the physical control (that is more significant to be confined to the car than to be in charge of the main application server).

CarSystemManager: this module will be a bridge between the electronics of the car and the central system: it will manage actuation requests (such as lock/unlock doors) to interface with the car control unit and it will handle notifications from the car control unit to the central system, in addition to autonomously notify particular events to the central system.

Notice that this component only shows information to the user, but does not take inputs or request except the *unlock code*. This component is not designed to be an interaction component between the user and the application server, like the mobile app

does instead.

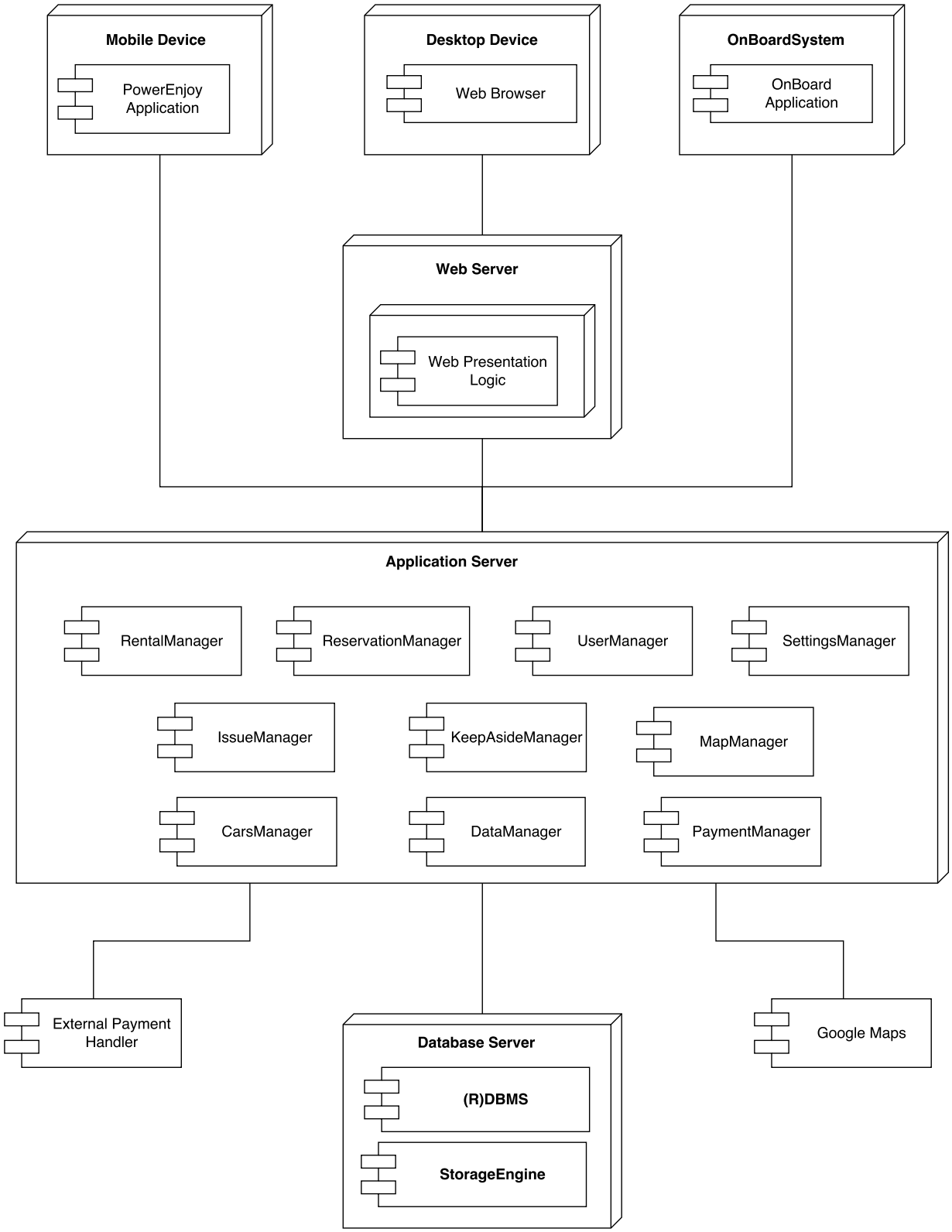
2.2.4 Mobile App client

The mobile app is probably the primary way the users adopt to interface with the *Power EnJoy* services. This component interact directly with the Application server due to a dedicated communication protocol. This component must provide a module to properly handle the geolocalization feature, which is able to get data from the smartphone GPS sensor.

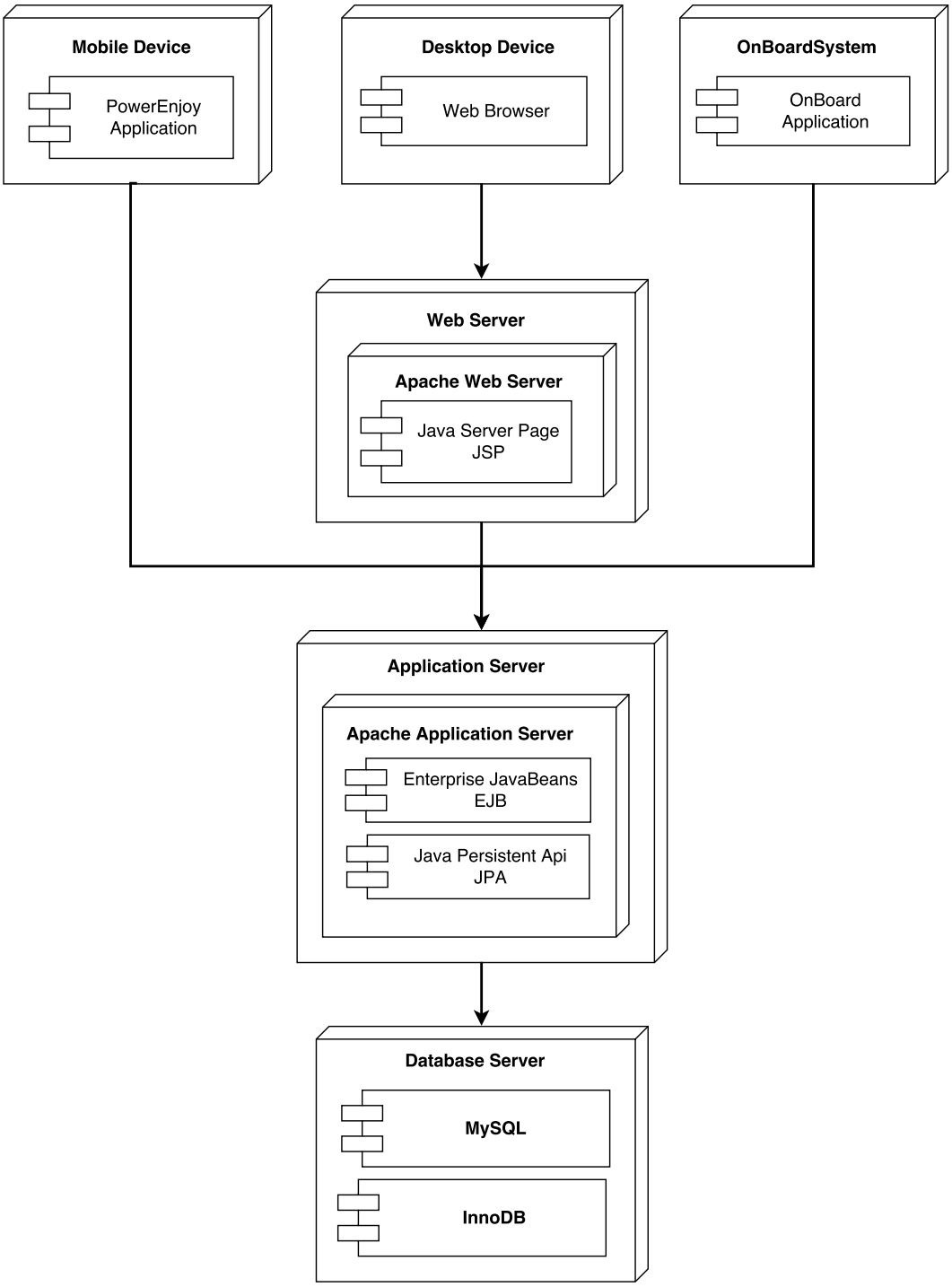
2.2.5 Web server

Since it must be available a web version of the application, it is essential to have a dedicated web server to handle the related requests and responses. The presentation layer of the website is entrusted to the user's web browser. The web server interfaces with the application server to proper get the information to include in the processing pages. It will also feed the application server with the data received from the user.

2.3 Deployment view



A possible choice of implementation could be the following:

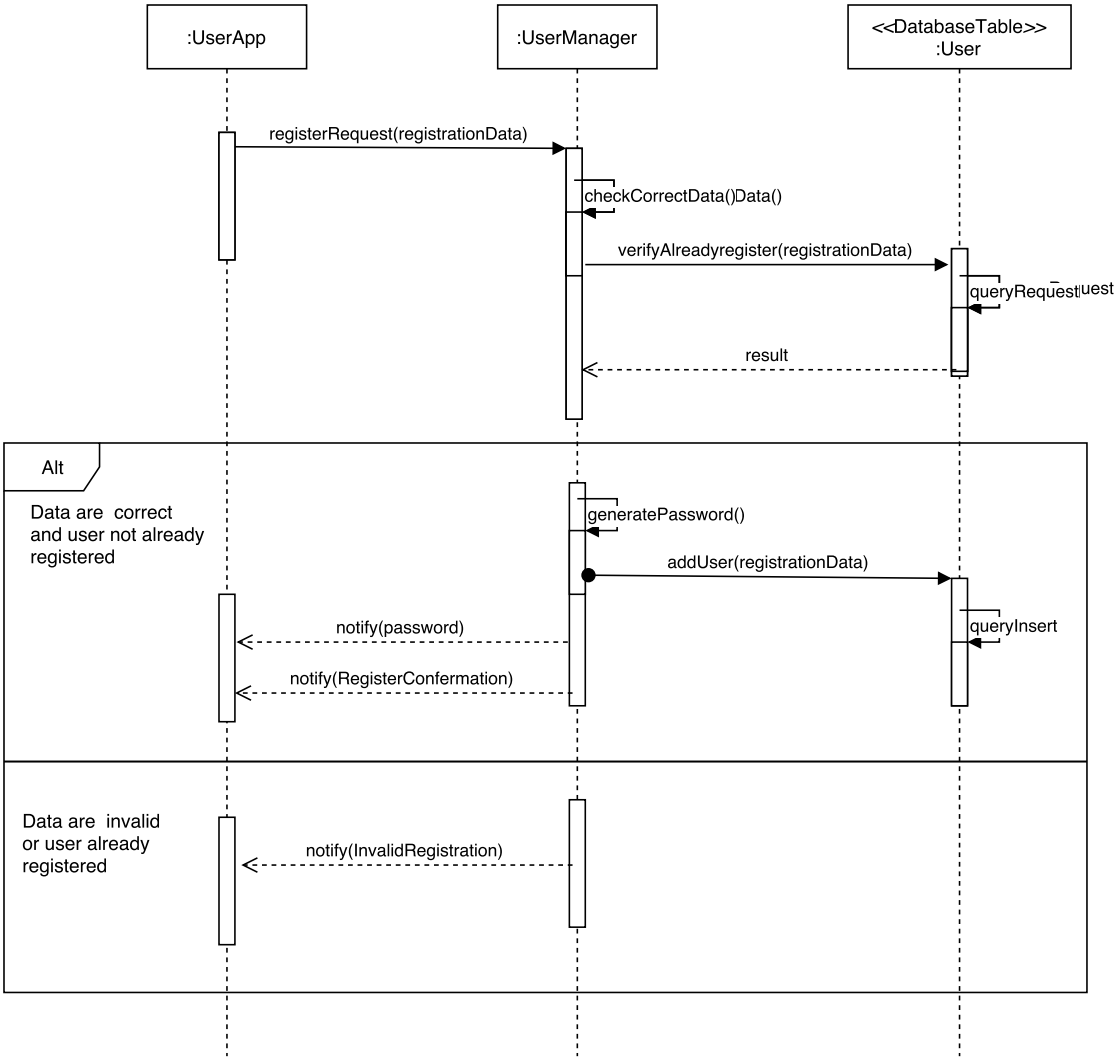


2.4 Runtime view

In this section are shown the sequence diagrams related to a bunch of functionalities implemented by the system, to better figure out the interactions between components and actors at runtime stage.

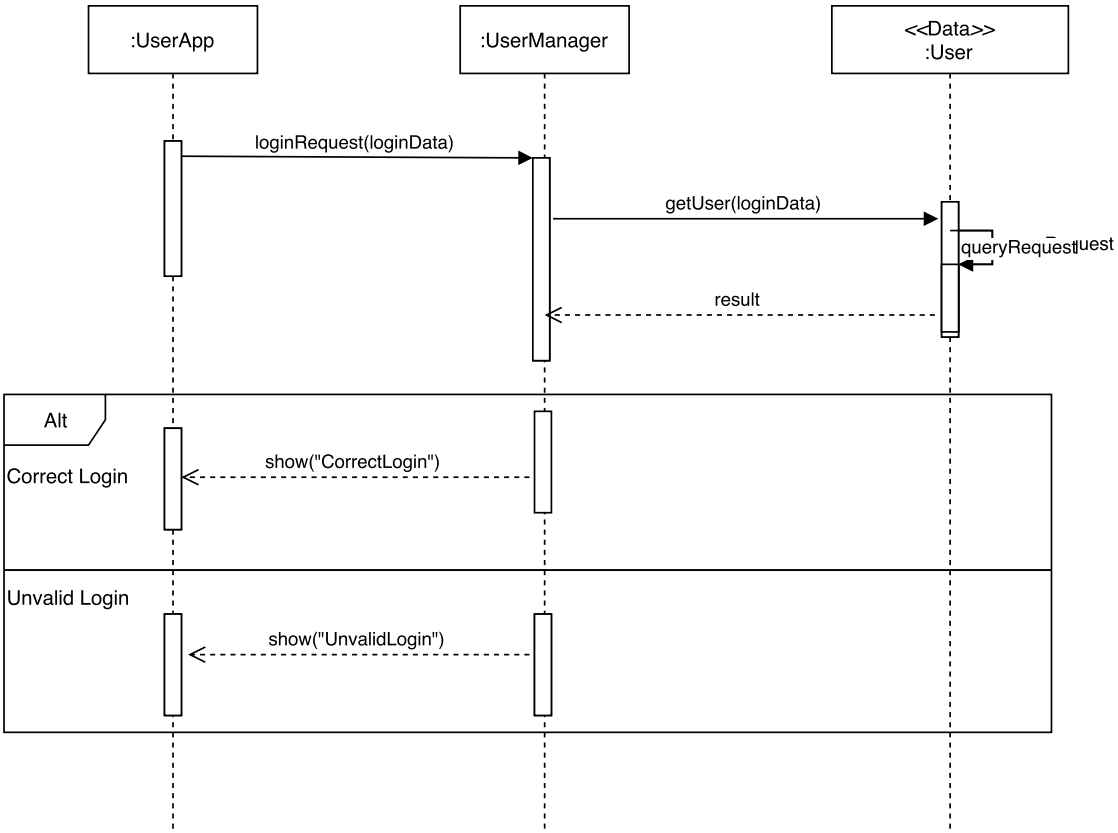
2.4.1 Registration

This sequence diagram exposes the registration process of a new user.



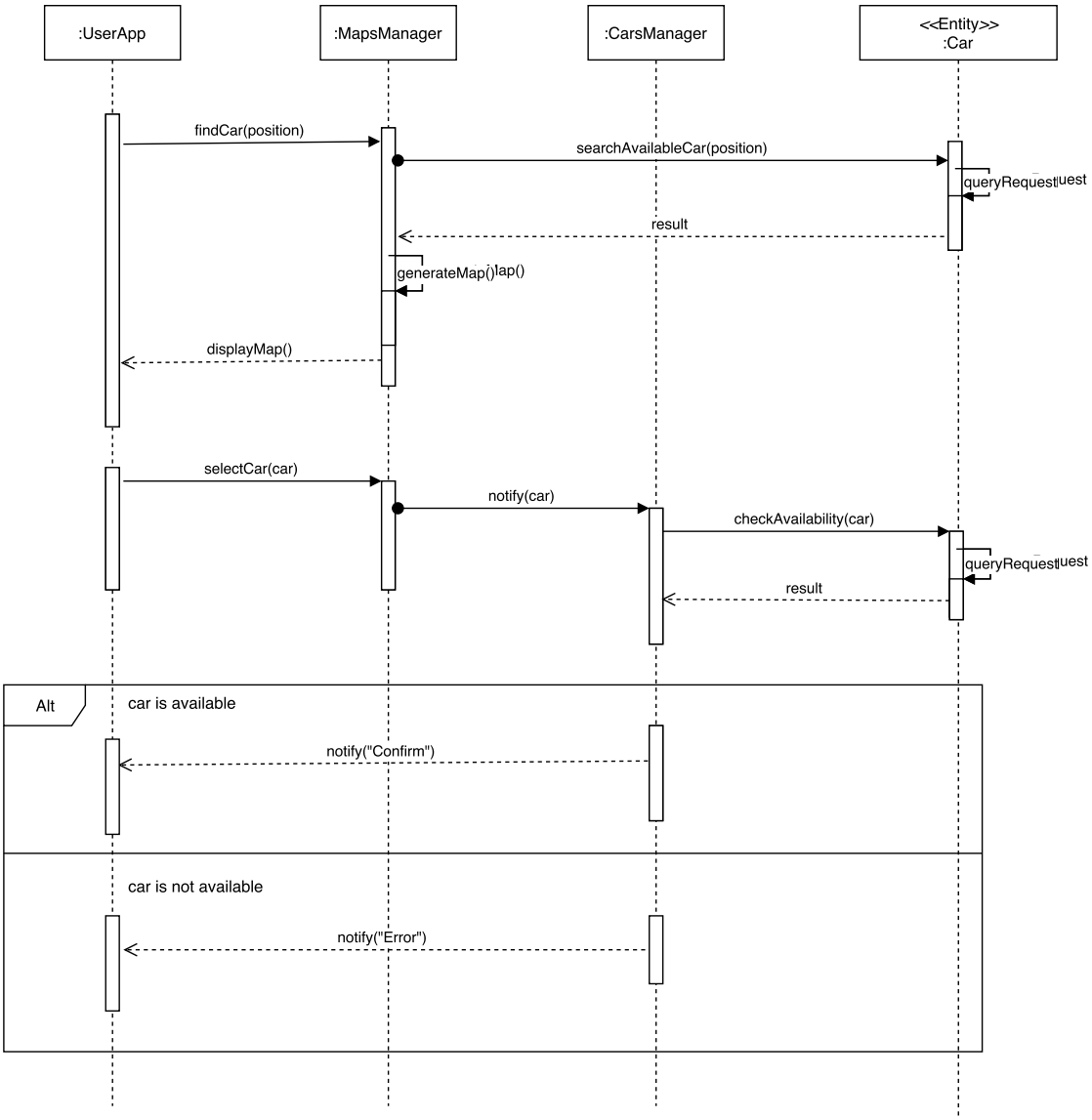
2.4.2 Login

This sequence diagram exposes the login process of a user or an employee.



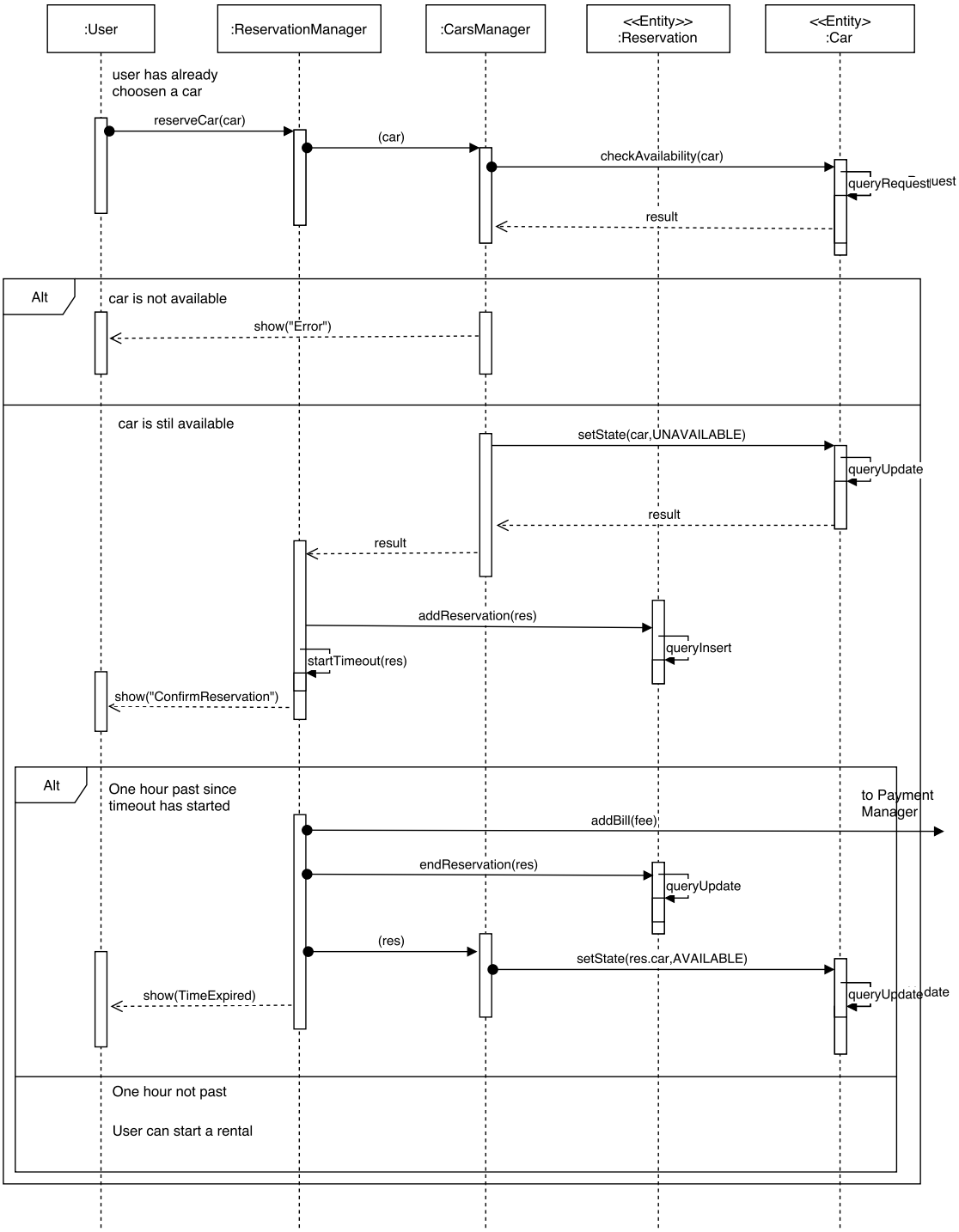
2.4.3 Car selection

This sequence diagram exposes the process related to the selection of a car by a user.



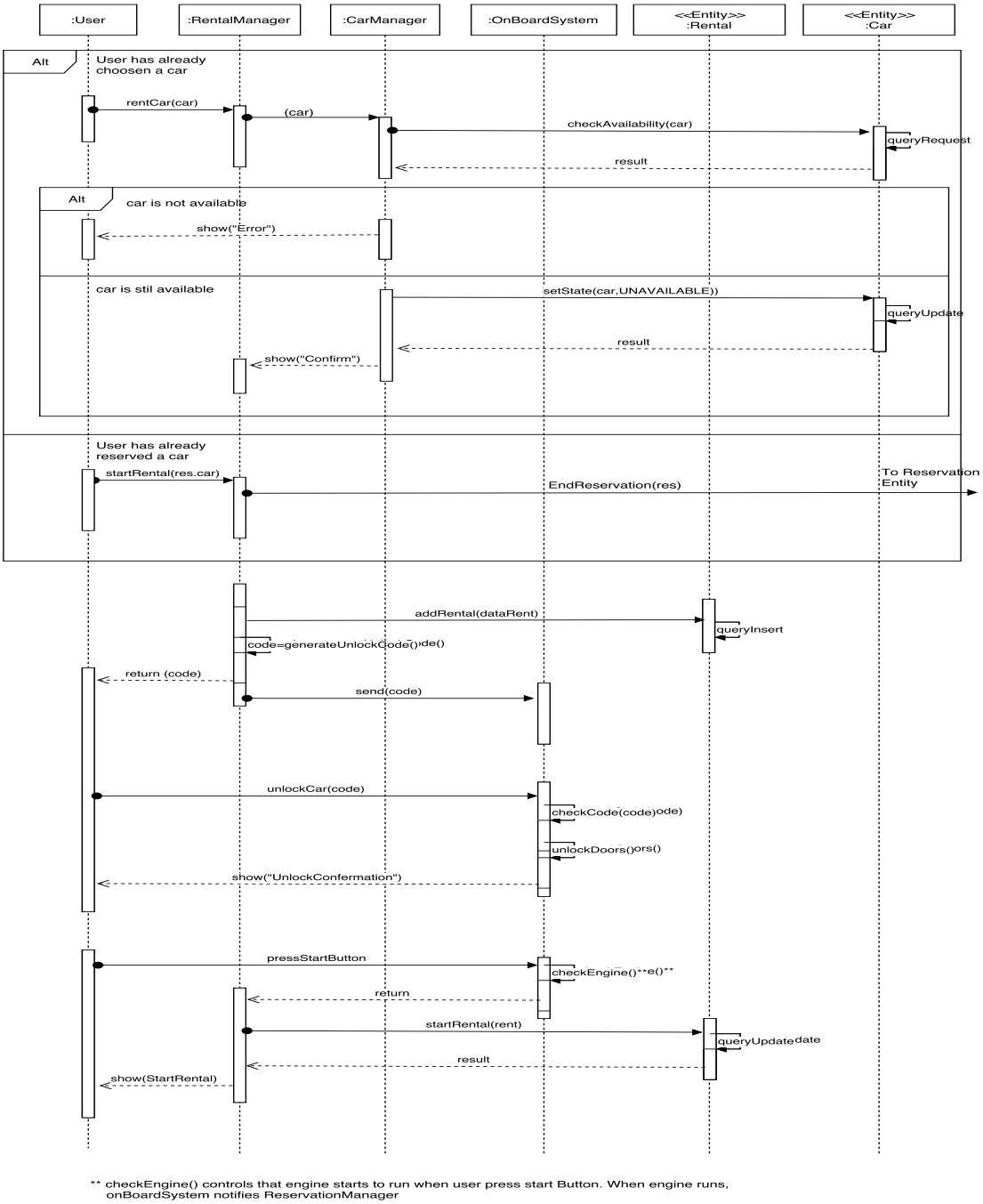
2.4.4 Car reservation

This sequence diagram exposes the car reservation process actuated by a user.



2.4.5 Car immediate rental

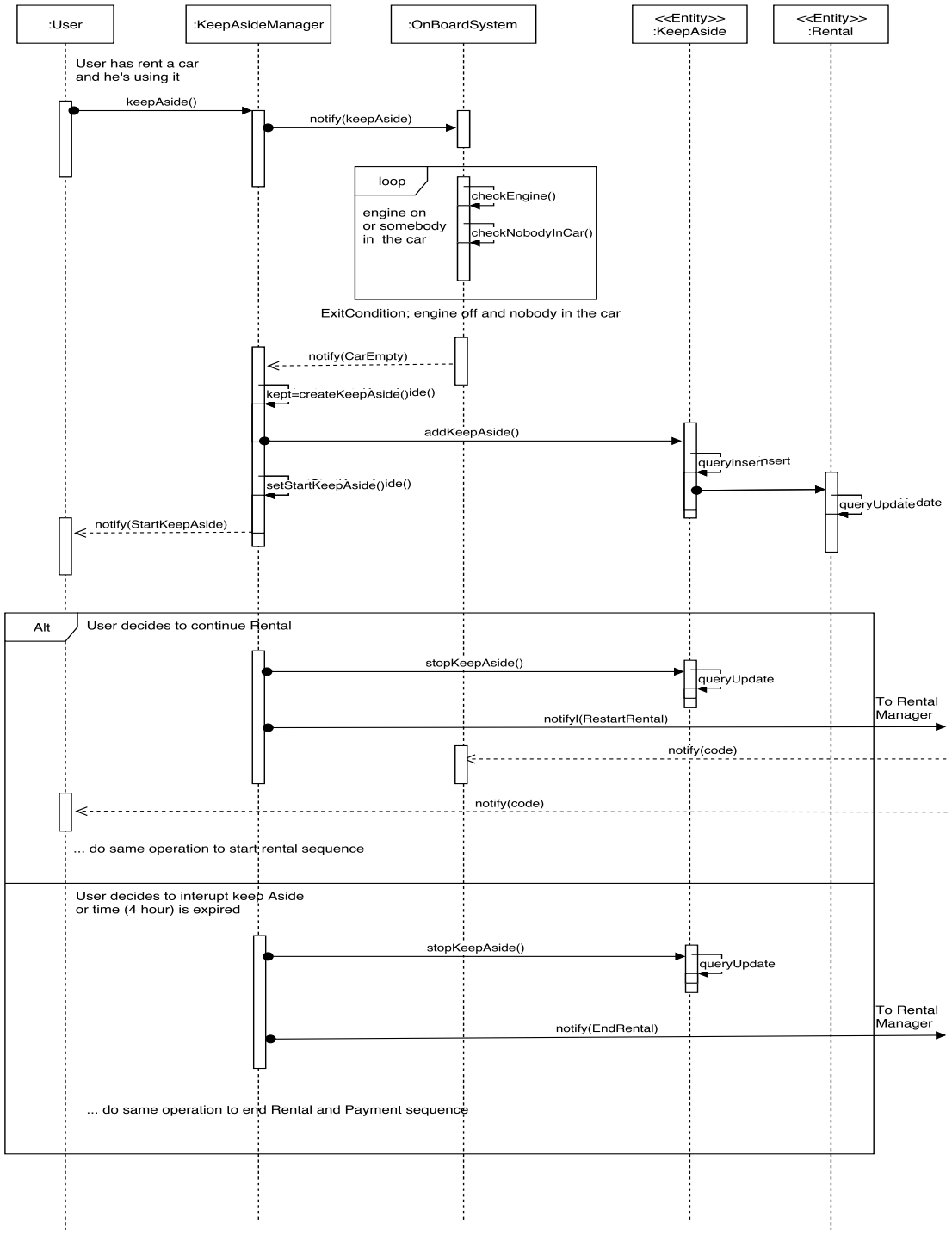
This sequence diagram exposes process related to a car rental without reservation actuated by a user.



** checkEngine() controls that engine starts to run when user press start Button. When engine runs, onBoardSystem notifies ReservationManager

2.4.6 Keep aside interaction

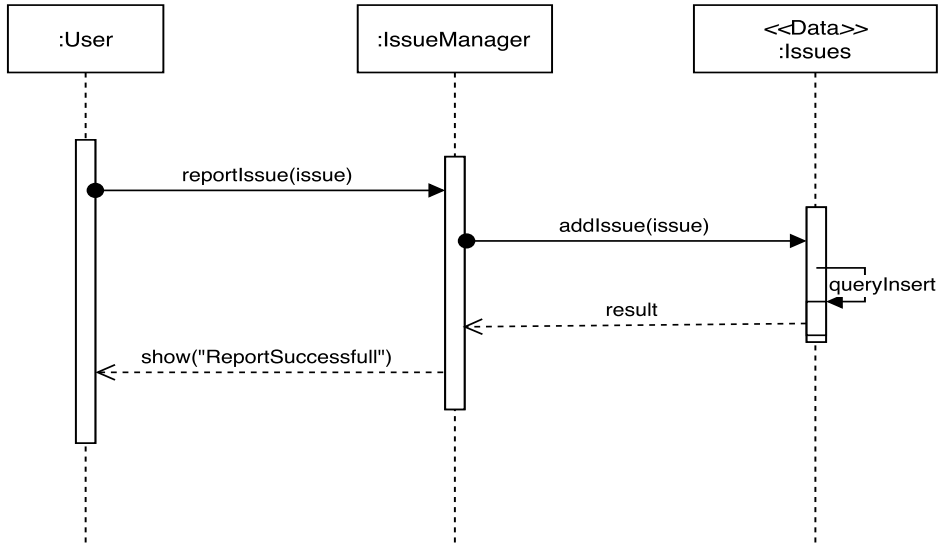
This sequence diagram exposes the *keep aside* process actuated by a user.



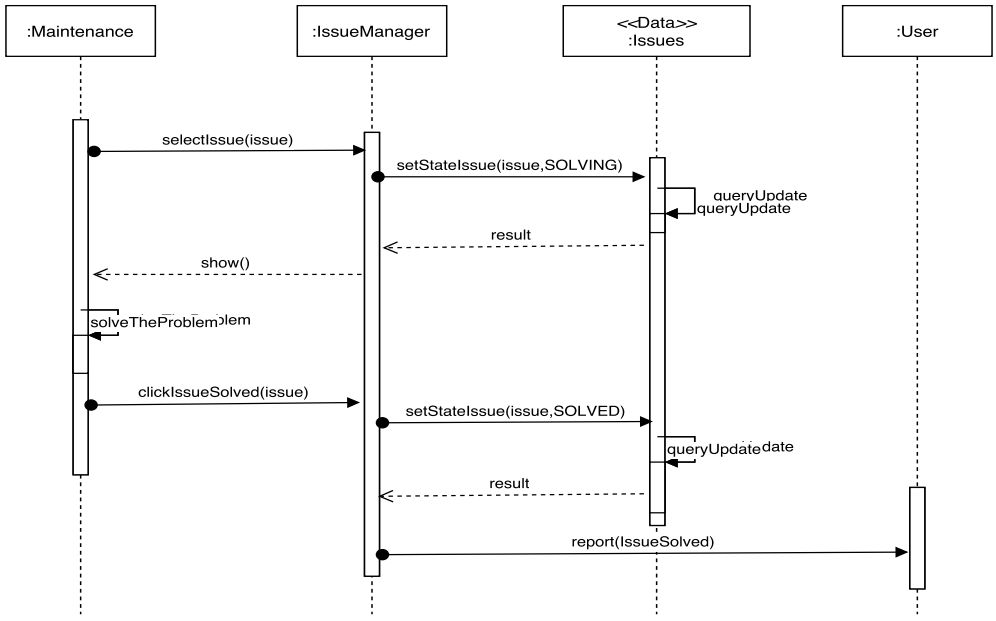
2.4.7 Reporting issue

These sequence diagrams expose the processes involved when an issue occurs and the user wants to report it.

Report Issues by User

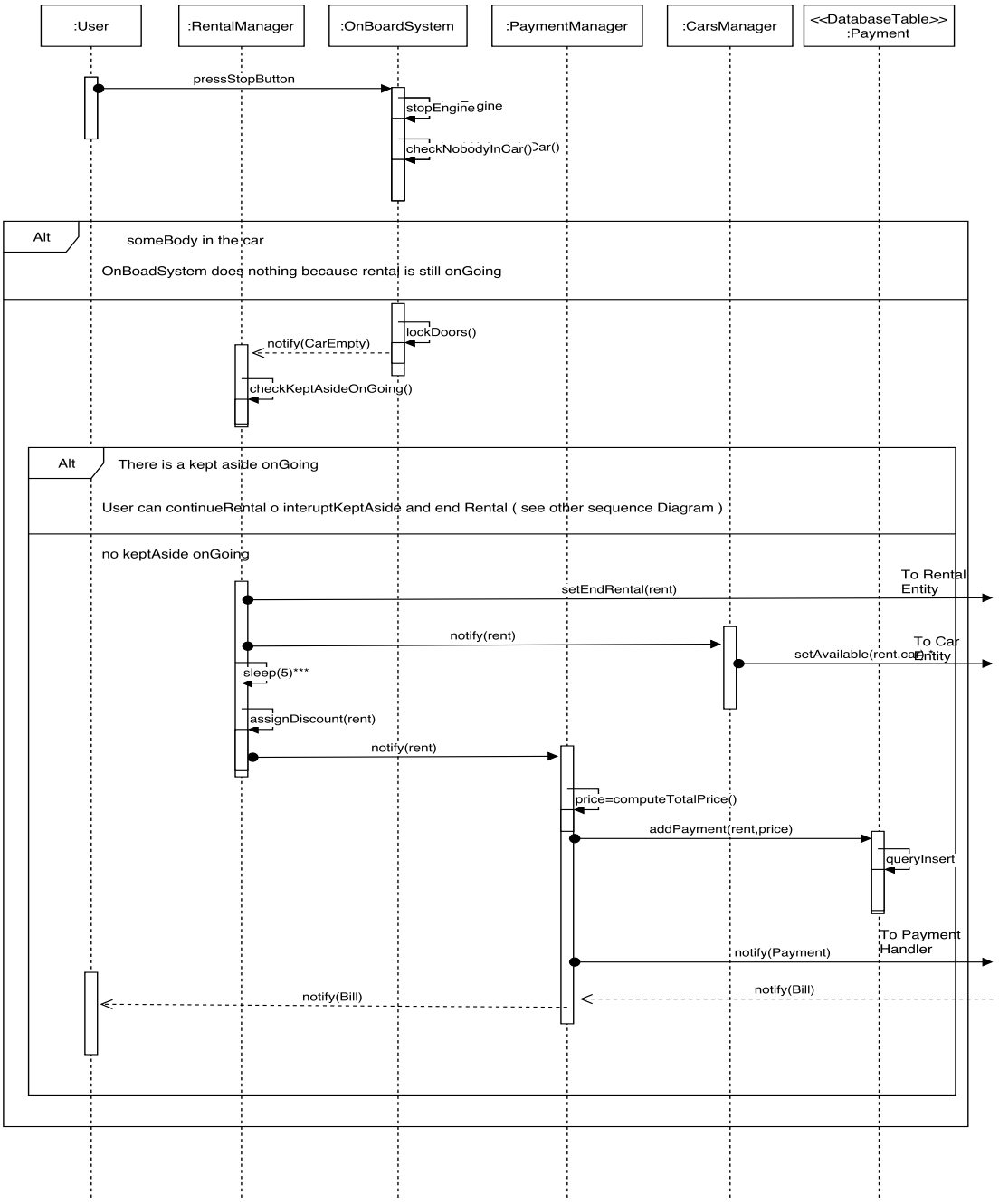


Solve Issues by Maintenance



2.4.8 End rental and payment

This sequence diagram exposes the how the end of a rental process works and how the payment is handled.

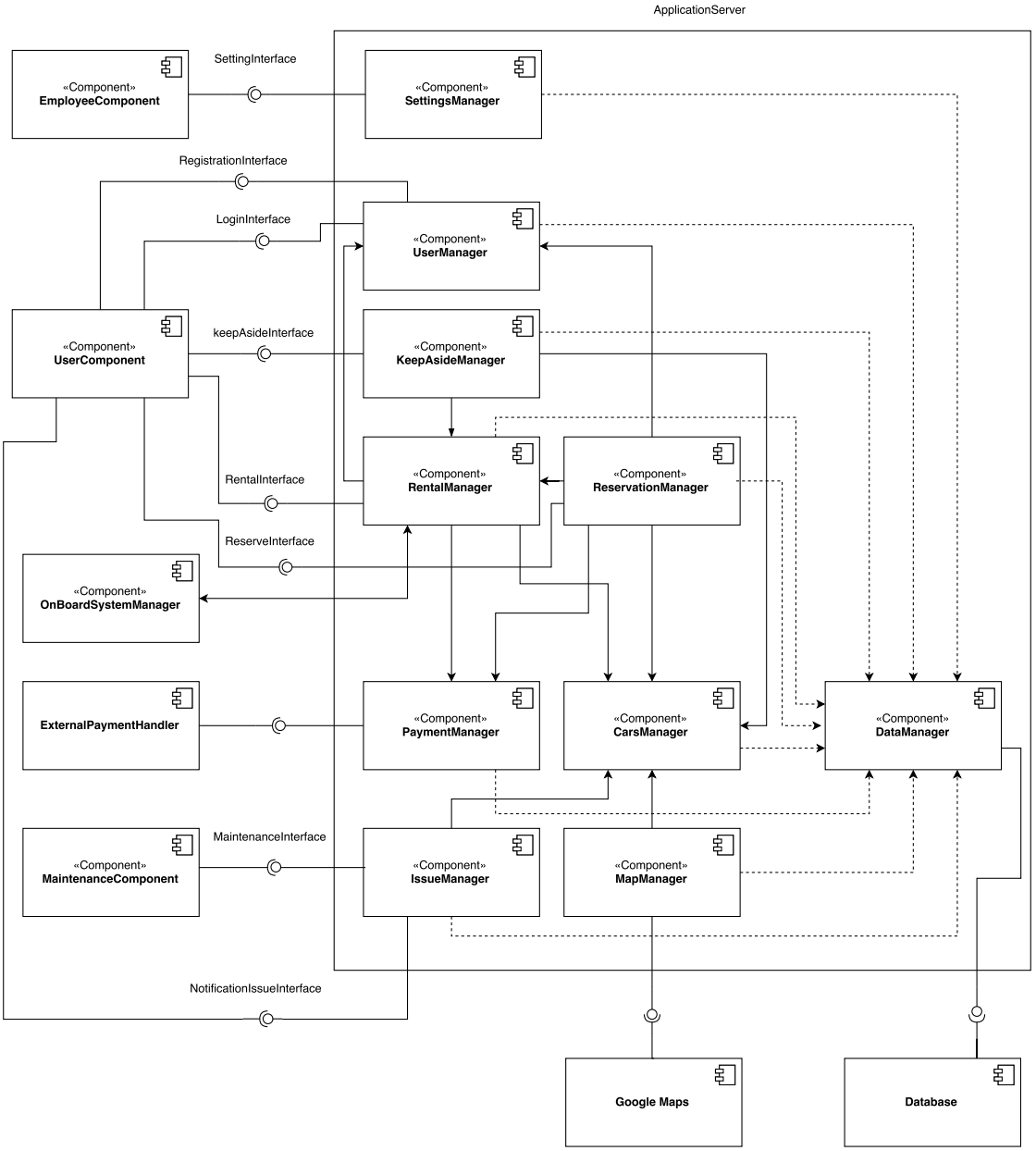


* setAvailable(rent.car) : before to set available the car again, system checks car battery level and if there are any issues. If battery level is low or there are some issues, system doesn't set available the car

*** Reservation waits five minutes before assign Discount; in this time user can connect car at a powerGridStation

2.5 Component interfaces

This section lists the interfaces between components.



2.5.1 Application and Database

The application server interact with the database server supported by the DataManager, which is a component intended to properly show an high level interface to the database.

2.5.2 Application and Web server

Direct communications with the web server are based over RESTful APIs served by the Application server. These APIs are implemented in JAX-RS.

2.5.3 Application and Clients

Direct communications with clients, as is the mobile application, are based over RESTful APIs served by the Application server. These APIs are implemented in JAX-RS.

2.5.4 Web browser and Web server

The browsers appointed to deal with the application will communicate with the web server using the HTTPS protocol. The server will reply with the same protocol.

2.5.5 Application server and external systems

The Application server (in particular the PaymentManager module) interfaces with the Payment Handler system using the APIs provided by the Payment Handler system.

2.5.6 Application, internal modules

CarsManager

checkAvailability(car:Car): boolean this procedure checks if a car is available to be used or not; it wants as parameter a car; as return value a boolean that indicates the result of the check.

setState(car:Car, state:State) this procedure sets the State of the car; it wants as parameters a car and it is a void procedure, it modifies the value of the car. The State of the car can assume the values indicate in the Car State Diagram.

getState(car:Car): State this method returns the current state of the car received as parameter.

ReservationManager

addReservation(car:Car, user:User) this procedure creates and adds a new reservation to the list of all reservations; before, the user must have already selected an available car. It wants as parameters the car to reserve and the user who has done the reservation; it's a void procedure, there is not a return value.

getReservation(user:User): List<Reservation> this procedure returns the list of reservation about the user passed as parameters; it wants as parameter a user and as return value a list of reservations;

getReservation(car:Car): List<Reservation> this procedure returns the list of reservations about the car passed as parameter; it wants as parameter a car and as return value a list of reservations;

startTimeout(res:Reservation) this procedure triggers the timer to compute one hour from when the reservation is started; it wants as parameter a reservation; it is a void procedure, there is not a return value.

endReservation(res:Reservation) this procedure sets the value of the attribute endTime to the current time; it wants as parameter a reservation; it is a void procedure, there is not a return value.

RentalManager

addRental(car:Car, user:User) this procedure creates and adds a new rental to the list of all rentals; before, the user must have already selected or reserved a car; then he can start a rental. It wants as parameters the car to be rented and the user who has done the rent; it is a void procedure, there is not a return value.

getRental(user:User): List<Rental> this procedure returns the list of rentals about the user passed as parameter; it wants as parameter a user and as return value a list of rentals;

getRental(car:Car): List<Rental> this procedure returns the list of rentals about the car passed as parameter; it wants as parameter a car and as return value a list of rentals;

generateUnlockCode(): String this procedure generates a code to unlock the car; the user will use this code when he will be nearby the car; it wants no parameters and as return value a String corresponding the unlock code.

startRental(rent:Rental) this procedure sets the value of startTime to the current time; it wants as parameter a rental; it is a void procedure, there is not a return value.

endRental(rent:Rental) this procedure sets the value of endTime to the current time; it wants as parameter a rental; it is a void procedure, there is not a return value.

checkKeepAsideOnGoing(rent:Rental):boolean this procedure checks if there are keepAsides ongoing in a rental: if duration of a keepaside is greater than 3 hours or the endTime is not NULL, keepaside is ended; it wants as parameter a rental and as return value a boolean indicates the result of the check.

assignDiscount(rent:Rental):List<Discount> this procedure checks which discounts and overcharges can be applied to a rental and adds them to a list; then it splits the list, holding only the minimum (best discount) and maximum value (worst overcharge). The procedure wants as parameter a rental and return value is a list containing all discounts and overcharges to be applied.

MapsManager

searchAvailableCars(): List<Car> this procedure searches which cars are available and puts them in a list; it wants no parameters and as return value a list of available cars.

generateMap(pos:Position, cars:List<Car>) this procedure generates a map using the position asked by the user and the list of available cars; it wants as parameters a position and a list of cars.

KeepAsideManager

addKeepAside(rent:Rental) this procedure creates and adds a new keepaside to a rental; before, the user must have already rented a car and the rental has not to be finished yet. It wants as parameter a rental; it is a void procedure, there is not a return value.

startKeepAside(kept:KeepAside) this procedure sets the value of startTime to the current time; it wants as parameter a keepAside; it is a void procedure, there is not a return value.

stopKeepAside(kept:KeepAside) : Integer this procedure sets the value of endTime to the current time and computes the duration of the keep aside; it wants as parameter a keepAside and as return value the duration of the keepAside.

OnBoardSystemManager

checkCode(userCode:String): boolean this procedure checks if the code generated early by application and the code used by user match. It wants as parameters a String corresponding the code and as return value a boolean indicates the result of the check.

setLockDoors(state: boolean) this procedure communicates with the control unit of the car, which locks the doors if the received state is true, and unlocks them if the state received is false.

checkNobodyInCar():boolean this procedure communicates with the control unit of the car which checks if there is somebody into the car. It has no parameters and as return value a boolean indicates result of the check.

checkEngine():boolean this procedure communicates with control unit of the car which checks if engine is running or not. As return value a boolean indicates result of the check. (True=engine on, False=engine off);

setUnlockCode(code: String) this method sets the unlock code on the car as the parameter code received.

UserManager

getUser(mail:String, password:String) :User this procedure checks if exists a user with mail and password used to login; it wants as parameters two Strings, one for mail and one for password; as return value the corresponding user or NULL if user with these credentials does not exist.

addUser(name, surname, cf, mail, driverLicense, codeAccount, password :String) this procedure creates and adds a new user; this procedure wants as parameters seven different Strings corresponding at name, surname, cf, mail, driver license,code account and password of the user. It is a void procedure then it has not a return value

generatePassword() :String this procedure generates a password for a user; password must be different for each user. It has no parameters and return value is a String corresponding the password generated.

verifyAlreadyRegistered (cf,mail: String) : boolean this procedure checks if some of user data (cf and mail) are different for each user otherwise user cannot be registered. This procedure wants as parameters two different Strings corresponding to user's cf and mail. The return value is a boolean which indicates the result of the check.

verify (driverLicense, codeAccount :String) : boolean this procedure checks if some of user data (driverLicense and codeAccount) are valid and correct otherwise user cannot be registered. This procedure wants as parameters two different Strings corresponding to the user's drivereLicense and codeAccount. The return value is a boolean which indicates the result of the check.

setBusy(user:User, state: boolean) this method sets the user state complying the received state parameter; the parameter must be true for set the user busy, false otherwise.

PaymentManager

computeRentalPrice(rent:Rental) :double this procedure computes the price of a over rental without keepaside nor discounst; it wants as parameter a rental and return value is the price computed.

computeKeepAsidesPrice(rent:Rental) :double this procedure computes the price of all keepAsides of a over rental: sum all duration of each keepAside and computes the total price. The procedure wants as parameter a rental and return value is the price computed.

computTotalPrice(rent:Rental) :double this procedure computes the total price of a over rental considering discounts and keepAsides: sum prices of rental and keepasides then applies discounts and overcharges. The procedure wants as parameter a rental and return value is the total price computed.

getCurrentCharging(rent:Rental) :double this procedure computes the total price of a ongoing rental (from the startTime to the current time) non considering discounts but keepAsides: sum prices of rental and keepasides. The procedure wants as parameter a rental and return value is the total price computed.

addBill(price: double, rent: Rental) this method generates a bill related to the rental received as parameter with the price received as parameter, and adds it to the database.

IssueManager

addIssue(user:User, car:Car, descritpion:String, phoneNumber: String) this procedure creates and adds a new issue to the list of all issues. It wants as parameters the car with the issue and the user who is reporting the issue, and also his phone number and a description of the issue; it's a void procedure, there isn't a return value;

setIssueState(issue:Issue,man:Maintenance, state: IssueState) this procedure modifies the issue status according to the one received as parameter. The procedure wants as parameters an issue and a maintenance man and a issue status. This is a void procedure, then no return value is defined;

SettingManager

modifyPriceRental(price:Float) this procedure modifies the current rental price to the value received as parameter. The parameter required is a float number complying the currency of the whole system. This procedure has return type void.

modifyPriceKeepAside(price:Float) this procedure modifies the current *keep aside* price to the value received as parameter. The parameter required is a float number complying the currency of the whole system. This procedure has return type void.

updateCar(car:Car, newCar: Car) this procedure update the current car information related to the received parameter car to the value received by the newCar parameter. The parameter required are a car and another car representing the same car as before with some attributes edited. This procedure has return type void.

updatePowerGridStation(pw:PowerGridStation, newPw: PowerGridStation) this procedure update the current power grid station information related to the received parameter pw to the value received by the newPw parameter. The parameter required are a power grid station and another power grid station representing the same power grid station as before with some attributes edited. This procedure has return type void.

updateSafeArea(sa:SafeArea, newSa: SafeArea) this procedure update the current safe area information related to the received parameter sa to the value received by the newSa parameter. The parameter required are a safe area and another

safe area representing the same power grid station as before with some attributes edited. This procedure has return type void.

2.6 Selected architectural styles and pattern

2.6.1 Client/server

This model is widely used in the system, in fact:

- the Application Server is the main brain of the system, and the users interact with it using a sort of silly client service as is the mobile application or the website;
- the Application Server act as a client with the respect to the Database Server;

2.6.2 Publisher/Subscriber

The Application Server and the on-board system act each other as a publisher/subscriber model, in fact the application server communicates data to the cars, and also the cars communicates data to the Application server but neither the on-board system nor the Application Server ask for that data.

2.6.3 Multi-tier architecture

The architecture is not centralized in a single component, but is composed by a number of different elements each one with a specific task. This improve efficiency and reliability of the system. This allows the physical delocalization of the elements, the possibility of component replication and so forth to ensure a high quality of service.

2.6.4 Thick client

We think that is better to delegate some of the Application Server logic to the single on-board systems, with the aim of reduce the number of request to the server which are basically related to the physical car and in a smaller part to the general system. The on-board system needs however to interact with the Application Server on questions in the context of the entire system.

2.6.5 Thin client

The mobile app and the website do not contain any kind of the system logic, and they need to forward all the request to the Application Server to have the job done.

2.6.6 Model-View-Control

This pattern is adopted to achieve a separation among the logic of the system, the presentation layer and the system model. This choice should favor the initial implementation and also future upgrades of the system.

2.7 Other design decision

2.7.1 Maps

To quickly and faithfully dispose of a complete map service, is noticeable the ranking of *Google Maps* as maps maker, addresses translator, maps updater and APIs provider to interface and take advantage of its functionalities. We think that now it is the best choice to adopt in this scope.

3 Algorithm design

This chapter includes some algorithms we want to describe. The code is described in the sections of the chapter.

```
// Controls for each station if the car is near at one of them

boolean noPowerStationNear(Position carPosition,
    Iterator<PowerGridStation> powerstations){
        PowerGridStation station;
        if(!powerstation.hasNext())
                return false;
        else{
                station=powerstation.next();
                if(isNear(carPosition,station))
                        return true;
                else noPowerGridStation(carPosition,powerstations)
        }
}

// check if a position is near a powerGridStation in a range of 3 Km

boolean isNear(Position pos,PowerGridStation station){
        if(
                ((pos.getLatitude() -
    station.getPosition().getLatitude())^2   +
                        (pos.getLongitude() -
    station.getPosition().getLongitude())^2) <=3
                )
                return true;
        else return false;
}

// From the list of all discounts and overcharges obtainable by a user,
// take only the minimum value corresponding at the maximum discount
// and the maximum value corresponding at maximum overcharges

List<PriceModifier> selectPriceModifier(List<PriceModifier>
    listAllpricemodifier){
```

```
        List<PriceModifier> discounts=new ArrayList<>();
        List<PriceModifier> orderList=
    listAllpricemodifier.stream().sorted((p1,p2)     ->
    Integer.compare(p1.getAmount(), p2.getAmount()))
    .collect(Collectors.toList());
        discounts.add(orderList.get(0));
        discounts.add(orderList.get(orderList.size()-1));
        return discounts;
}

// Example of assign discounts and overcharges (Not all cases)

List<PriceModifier> assignDiscount(Rental rent){
        List<PriceModifier> charges=new ArrayList<>();
        if(rent.getNumberOfPassenger()>=2)
                charges.add(new PriceModifier(-0.1,"Number of passenger
    major then 2"));
        if(rent.getBatteryUsed()<50)
                charges.add(new PriceModifier(-0.2,"Battery used minor
    then 50%"));
        if(rent.getCar().isChargirg())
                charges.add(new PriceModifier(-0.3,"Car connected at a
    powerGridStaion"));
        if(rent.getCar().getBatteryLevel()<20)
                charges.add(new PriceModifier(0.3,"Battery level under
    20%"));
        if(!noPowerGridStationNear(rent.getCar().getPosition(),
    safeareas.getPowerGridStation().iterator()))
                charges.add(new PriceModifier(0.3,"No powerGridStation
    near the car");
        return charges;
}

// Compute total Price of a rental

double computeTotalPrice(Rental rent){
        int rentalDuration,keepAsideDuration=0;
        double rentalPrice,totalPrice;
        forEach(kept:rent.getKeepAside()){
                keepAsideDuration+=kept.duration;
        }
        rentalDuration=rent.endTime-rent.startTime-keepAsideDuration;
        rentalPrice=rentalDuration * priceRentForMinutes +
    keepAsideDuration * priceKeepAsideForMinutes;
```

```
        totalPrice=rentalPrice;
        List<PriceModifier>
    discounts=selectPriceModifier(assignDiscount(rent));
        forEach(dis:discounts){
                totalPrice+=dis.getAmount()*rentalPrice();
        }
        return totalPrice;
}
```

3.1 Compute Discount and overcharges

For first and second discount, on board system is able to detect if discount conditions are satisfied thanks to cars sensor and notify it to the application. For third discount, application waits five minutes after user exits the car: if at the end of 5 minutes car results in charging, therefore system applies discount, otherwise no discount is applied. For overcharges : car battery Level is notified by on-board system to application, so application can evaluate to apply overcharge or not. If no station is found near the car (noPowerGridStationNear returns *false*), overcharges will be applied. Discount is saved as negative float between 0 and -1; overcharge is saved as positive float between 0 and 1. Application must hold only the minimum value (corresponding to the maximum discount obtained) and the maximum value (corresponding to the overcharge). At the end, application multiplies each element of this list for the total price computed before and sum them with total price.

3.2 Compute rental charge and total price

System computes rental charge and total price only after 5 minutes after user exits the car (system waits to understand if apply or not the third discount). After that application can compute total price as showed in the algorithm above.

4 User interface design

4.1 UX diagram

In this section we include a UX diagram which allows to clearly identify the screens and their interaction. Screens related to *employees* are available only in the website version.

4.2 User interface concepts

In this section are shown the user interfaces conceived for the system.

4.2.1 Mobile app

Here some screens to illustrate the mobile application appearance.

4.2.2 Website

The website version basically contains all the features painted in the mobile app mock-ups. In addition, the website implements the management section of the application, which is not available in the mobile app. Here some examples of the designed screens.

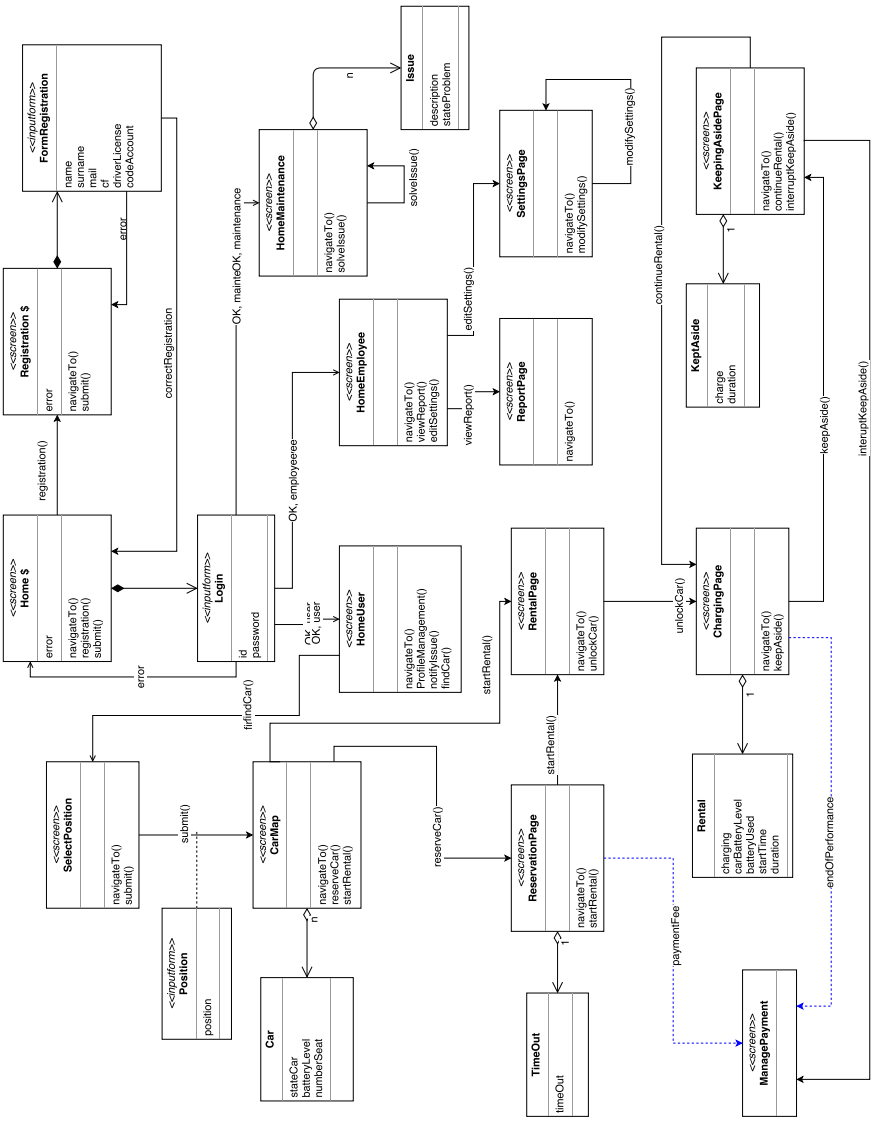
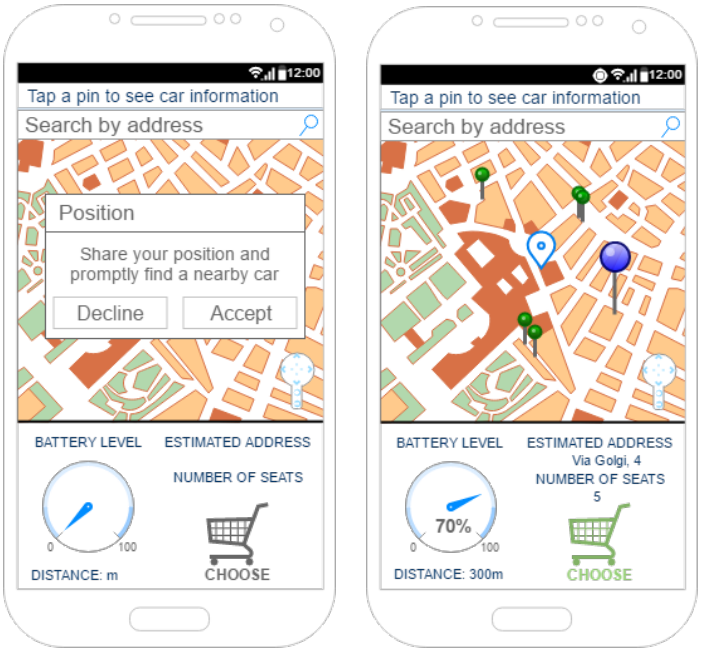


Figure 4.1: UX diagram. The dashed blue arrows describe actions that happen autonomously, after a timeout expiration or triggered by a specific event.

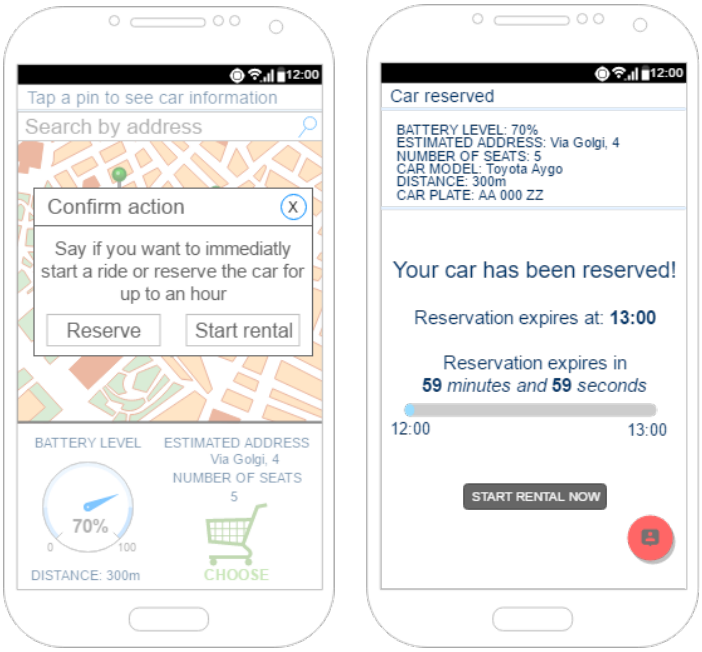
(a) *Registration.* (b) *Login.*

Figure 4.2: Mock-up



(a) *Car searching.* (b) *Available car.*

Figure 4.3: Mock-up: in (b), purple pin stands for the shown car

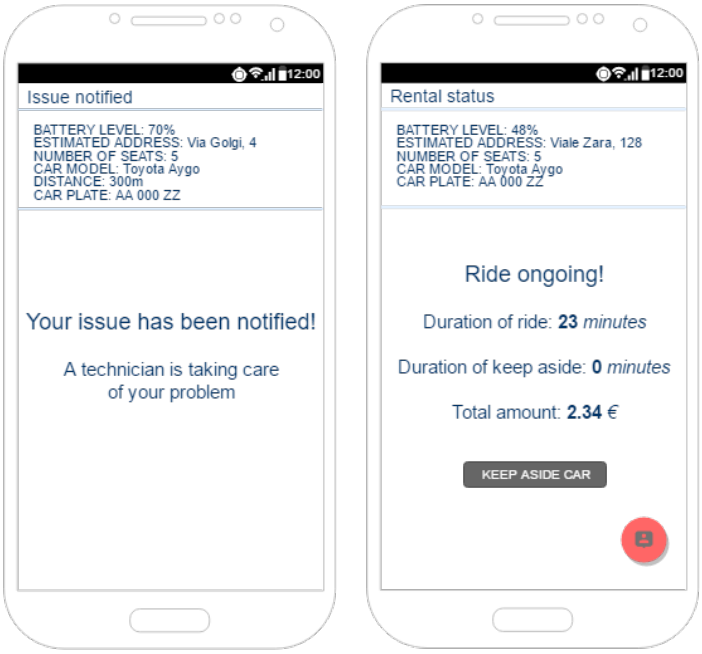


(a) *Confirm action.* (b) *Reservation confirm.*

Figure 4.4: Mock-up

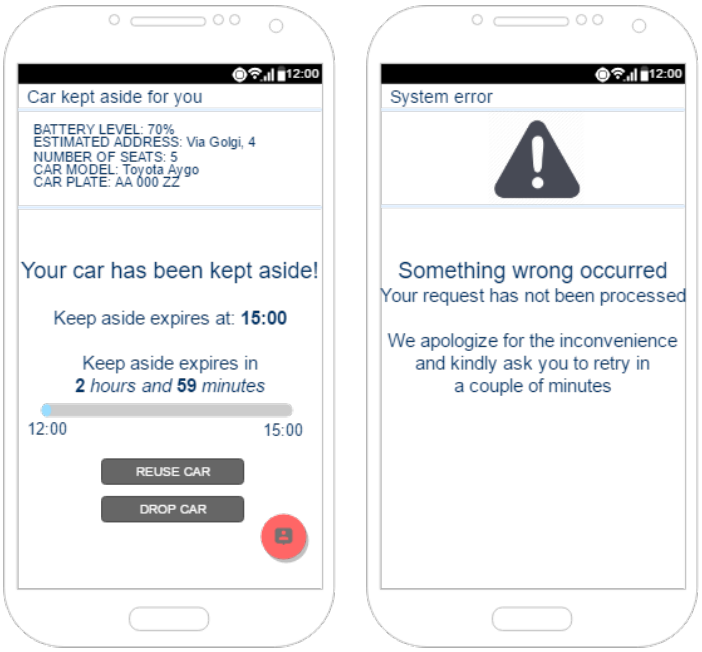
(a) *Unlock car.* (b) *Report issue.*

Figure 4.5: Mock-up



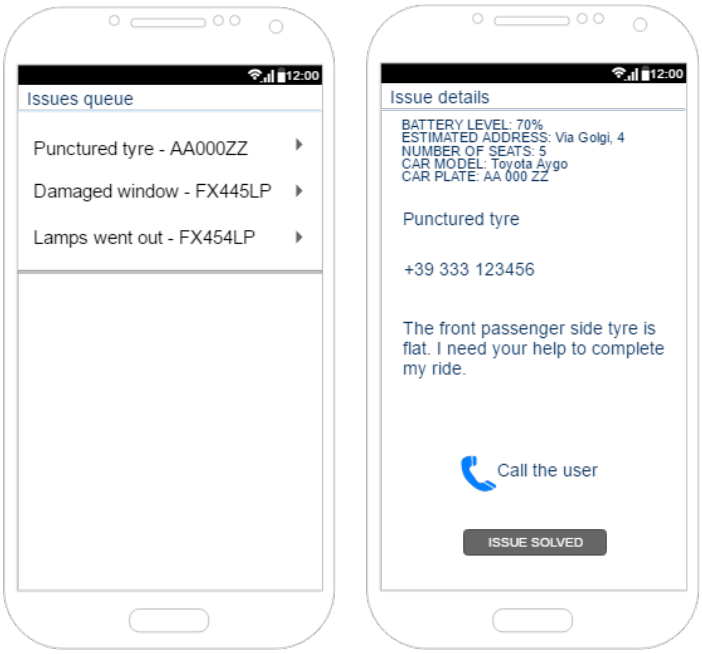
(a) *Issue confirm.* (b) *Rental status.*

Figure 4.6: Mock-up



(a) *Keep aside confirmation.* (b) *Error page.*

Figure 4.7: Mock-up



(a) *Maintenance issue queue.* (b) *Issue details.*

Figure 4.8: Mock-up: these screens are only in the maintenance employees' version

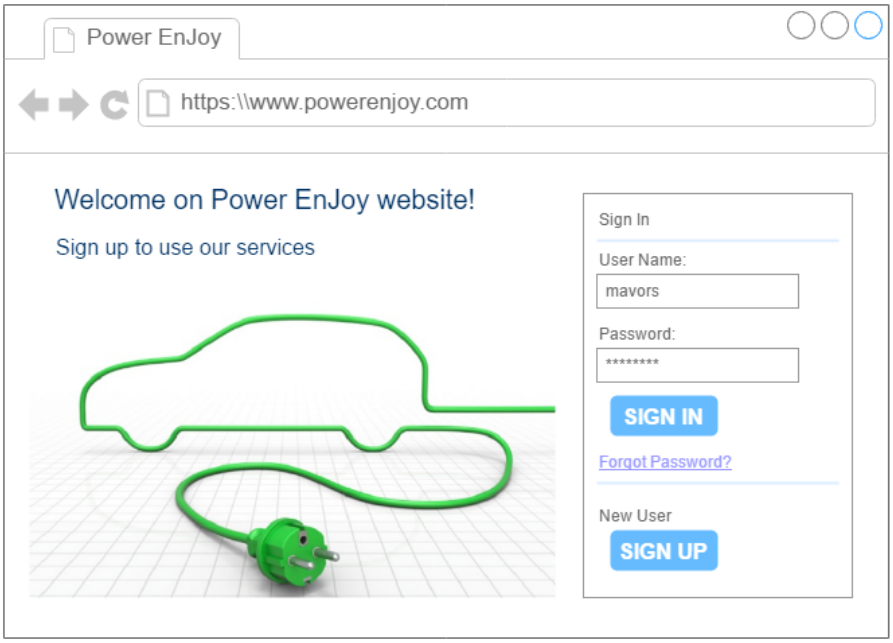


Figure 4.9: Mock-up: the user login form page on the website.

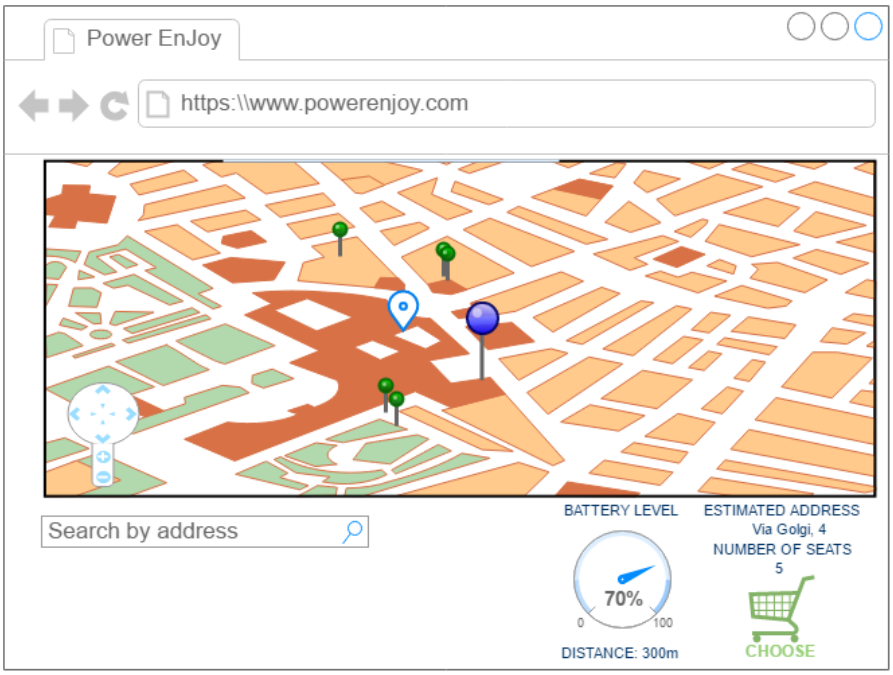


Figure 4.10: Mock-up: the car selection page on the website.

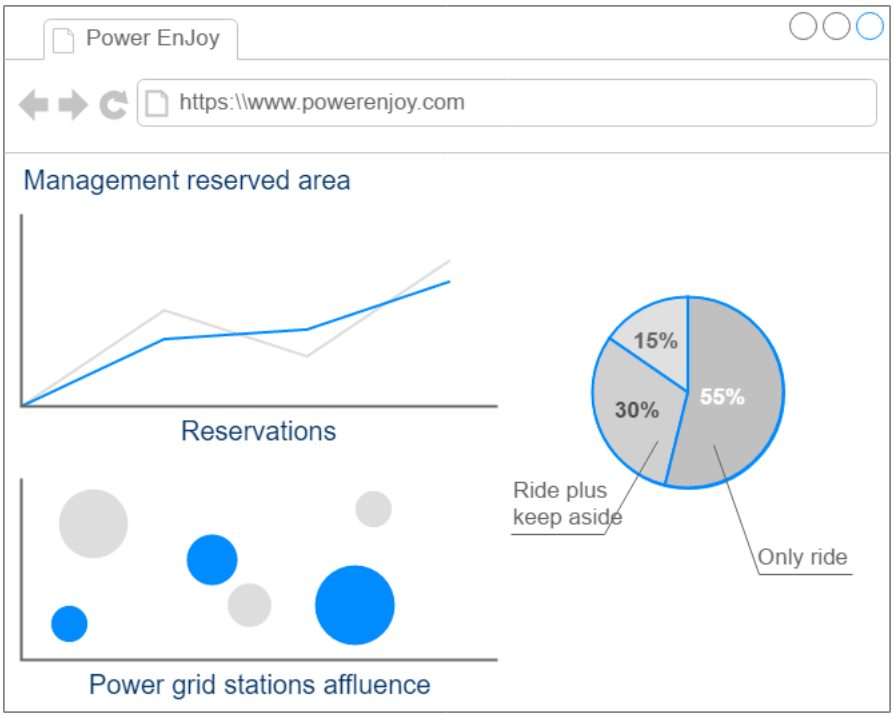


Figure 4.11: Mock-up: the management statistic page on the website.

5 Requirements traceability

G1. Allows potential users to sign up the system (information, credential and payments data):

- UserManager

G2. Allows users to sign in and use the system:

- UserManager

G3. Allows users to find the locations of available cars near their own current position or near a specified address:

- MapsManager
- CarsManager

G4. Allows users to book a car for up one hour in advance to pick it up:

- ReservationManager
- CarsManager

G5. Allows users to tell the system they are nearby the car:

- RentalManager
- OnBoardSystemManager

G6. Allows a user to see the amount of his current ride through a screen on the car:

- OnBoardSystemManager
- RentalManager

G7. Allows users to notify the system a problem related to the car:

- IssueManager

G8. Allows users to keep aside their car for a little time (max 3 hours, under specific payment):

- KeepAsideManager

G9. Allows to compute the amount of each user's ride [starts debit, stops debit, discount]:

- RentalManager

G10. Allows to manage the opening and closing of the cars:

- RentalManager
- OnBoardSystemManager

G11. Allows to stimulate the users' virtuous behaviors applying discounts on their last ride as and deter the malicious of the users applying extra fees as stated in the business rule section :

- RentalManager
- PaymentManager

G12. Allows to check the status of the cars as a prospectus/summary:

- CarsManager

G13. Allows the company to be notified if any on-board system detects a car failure:

- OnBoardSystemManager
- IssueManager

G14. Allows to edit settings such as prices per minute, discounts amount, "etc.":

- SettingsManager

G15. Allows the company to bill users for services they benefited:

- PaymentManager

G16. Allows to edit stored information as cars, Power Grid stations and Safe Areas:

- SettingsManager

6 Effort spent

Marco [h]	Daniele [h]
35	40

7 References

To draw up this document, we refer to the sample Design Documents provided in the lectures.

8 Change log

On 07/02/2017 we updated the deployment view diagram, the component interfaces diagram, the architecture description. We corrected the names of the method in the components accordingly with the edited names in the other documents.