Politecnico di Milano

Software Engineering II project

PowerEnJoy

Design Document

Authors:
Davide Piantella
Mario Scrocca
Moreno R. Vendra

Professor:
Luca Mottola

January 15, 2017

version 1.3

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose of this document

In this document we are going to describe software design and architecture of the PowerEnJoy system.
The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.[1]

1.2 Scope

PowerEnJoy is a car-sharing service that exclusively employs electric cars; we are going to develop a web-based software system that will provide the functionalities normally provided by car-sharing services, such as allowing the user to register to the system in order to access it, showing the cars available near a given location and allowing a user to reserve a car before picking it up. A screen located inside the car will show in real time the ride amount of money to the user. When the user reaches a predefined safe area and exits the car, the system will stop charging the user and will lock the car. The system will provide information about charging station location where the car can be plugged after the ride and incentivize virtuous behaviours of the users with discounts[2]

1.3 Glossary

The *PowerEnJoy: Requirements Analysis and Specification Document*[2] should be referenced for terms not defined in this section.

1.3.1 Definitions

Base cost: cost before any discount or fee application, obtained from rent time and time-based cost

1.3.2 Acronyms

RASD: Requirements Analysis and Specification Document

DD: Design Document

API: Application Programming Interface

GPS: Global Position System

DB: DataBase

DBMS: DataBase Management System

GIS: Geographic Information System

ER: Entity Relationship Model

XML: eXtensible Markup Language

REST API: REpresentational State Transfer API

JAX-RS: JAVA API for REST Web Services

ISP: Internet Service Provider

ARP: Address Resolution Protocol

1.3.3 Abbreviations

m: meters (with multiples and submultiples)

w.r.t.: with respect to

i.d.: id est

i.f.f.: if and only if

e.g.: exempli gratia

etc.: et cetera

1.4 Reference documents

Context, domain assumptions, goals, requirements and system interfaces are all described in the *PowerEnJoy: Requirements Analysis and Specification Document.*[2] Others references are:

- IEEE Std 1016-2009 Standard for Information Technology, Systems Design, Software Design Descriptions

1.5 Document overview

This document is structured as

1. **Introduction**: it provides an overview of the entire document
2. **Architectural design**: it describes different views of components and their interactions
3. **Algorithm design**: it focuses on the definition of the most relevant algorithmic part
4. **User interface design**: it provides an overview on how the user interfaces of our system will look like
5. **Requirements traceability**: it explains how the requirements we have defined in the RASD map to the design elements that we have defined in this document.
6. **Appendices**: it contains references, software and tools used and hours of work per each team member

2 Architectural design

2.1 Overview

2.1.1 Context viewpoint

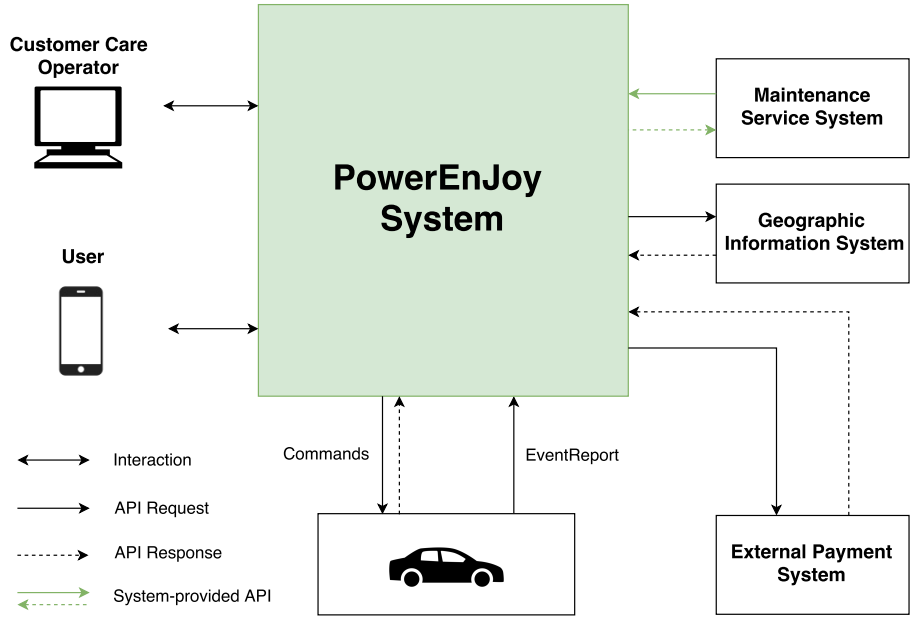

Figure 1: Context viewpoint

We need to design a system which allows communications with many agents such as cars, users, external systems, etc. Moreover we recognize that in most of the interactions the system is providing a service to agents so, after taking in consideration different alternatives, we decided to use a client-server architectural approach.

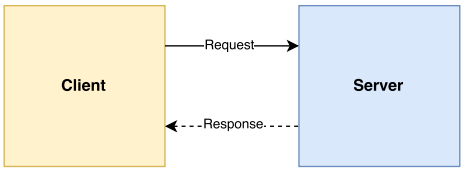

Figure 2: Client Server architecture

Cars offer to the system a set of primitives which allow it to interact with them: in this case it is clear that cars are providing the system services, so they can be identified as servers while the system acts as a client; on the other end the notification functionality offered by cars clearly yields to an event-based approach due to the asynchronous nature of such interactions, this led us to use a publish-subscribe paradigm for these specific interactions.

2.1.2 Composition viewpoint

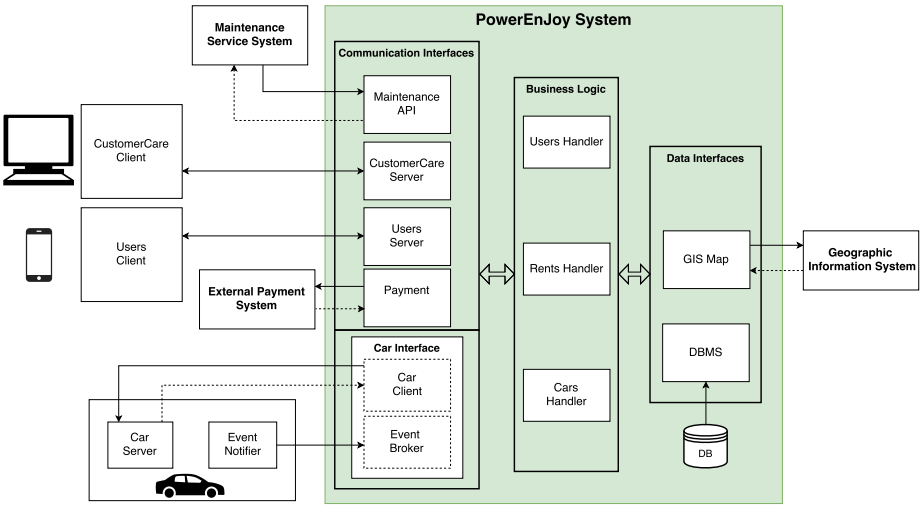

Figure 3: Composition viewpoint

Going deeper in the analysis of our system composition, we are able to identify some of the modules that will be required in order to provide the functionalities specified in the Requirement Analysis and Specification Document.

Communication Interfaces Since our system interacts with many external agents, it needs to have different *Communication Interfaces* in order to communicate with them.

- An API is needed to provide *Maintenance Service System* the information it needs to work with us
- A software module is needed in order to provide users functionalities of the system
- A software module is needed to provide the *Customer Care* the functionalities it needs
- An internal payment software module will deal with the communication with the *External Payment System*
- A set of modules will manage the communications between the cars and the system: a module to call primitives on cars through the provided API and another one to observe events triggered by the car. When the trigger method is called on *CarClient*, it enables the requested triggers on the car passed as a parameter. This allows the cars to notify to the system the events related to the aforementioned triggers; these events will be received and dispatched by the *EventBroker* to the subscribed components.

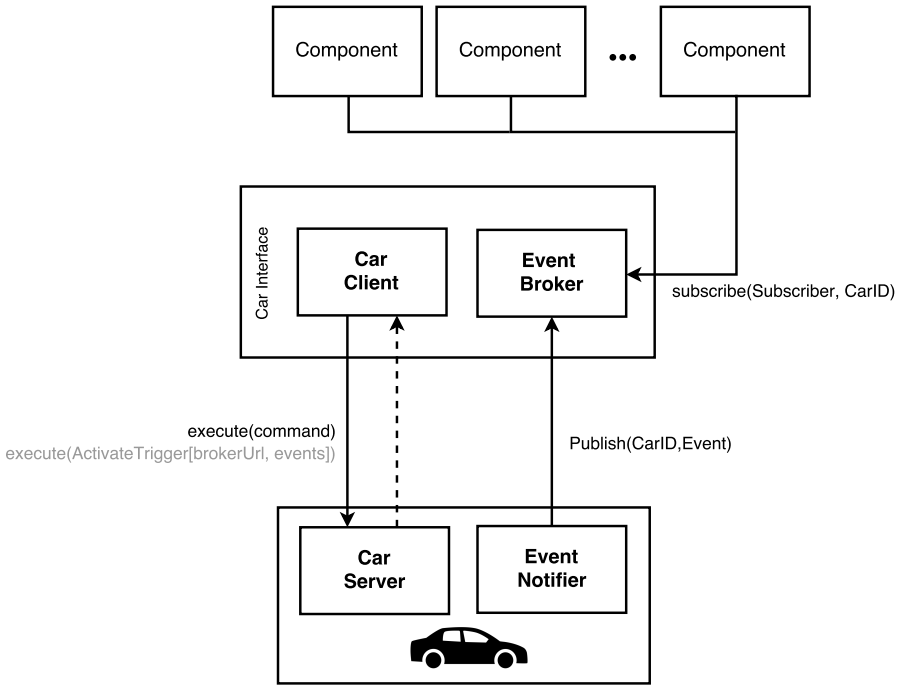

Figure 4: Car communication Interface

Business Logic The actual application logic of our system needs to manage the users information, the rents and the cars information; for each of these purposes several software modules are necessary; they will use communication interfaces to communicate with the agents and they will be able to retrieve data from the data interfaces.

Data Interface Our system needs a way to access and store the data it produces or retrieves from external resources, that is why *Data Interface* modules are needed. These modules allows interaction between the *Business Logic* modules and the System Databases; moreover they provide an interface to communicate with the GIS in order to allow the *Business Logic* modules to access its functionalities.

2.2 Component view

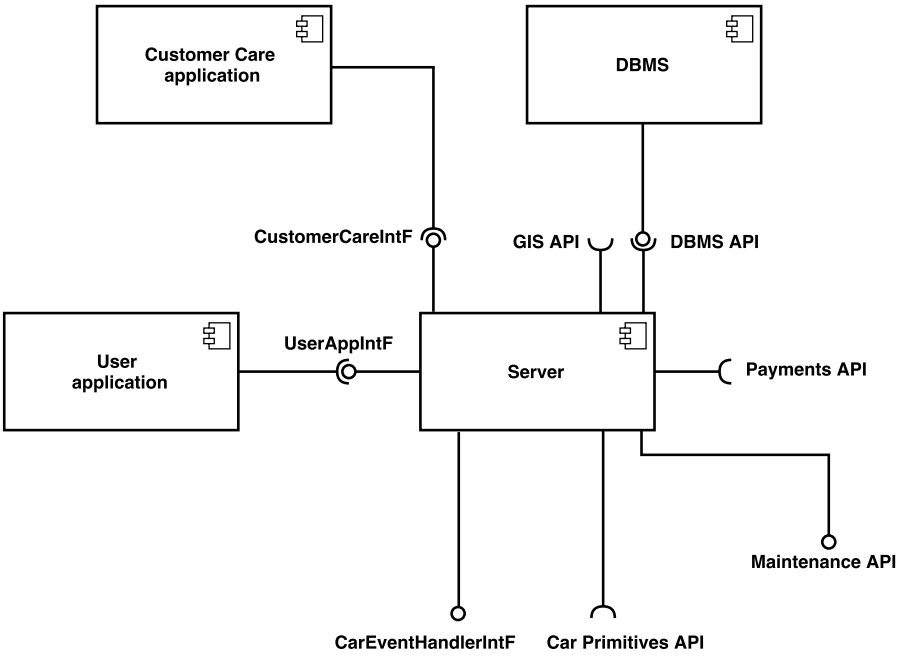

Figure 5: High-level components

Considering all the previous graphs, we have identified in Figure 5 the following high level components, interfaces and mapping of the functionality defined in the RASD:

- User application
  - Register
  - Login
  - View the map with the position of
    - himself
    - safe areas
    - available cars (with their battery level)
    - charging stations
  - Reserve a car
  - View customer care contacts
  - Unlock the reserved car
  - View and edit personal information
  - View rents and payments history
- Customer Care application
  - View each user profile, including personal information, progress of the current rent, rent and payment history

  - Mark and unmark users as banned
  - Mark cars as Not Available
- DBMS
  - Store and retrieve data
- GIS API
  - Retrieve a reference to an up-to-date map centered on a given position
  - Add pointers to the aforementioned map
  - Get the latitude and longitude of a given location and vice versa
- Payments API
  - Execute payment transactions
- Maintenance API
  - Expose the list of the cars tagged as Not Available with their GPS position, a brief description of the problem and a software key to access the car
  - Tag Not Available cars as Available
- Car Primitives API
  - Call car embedded system's primitives
- Car Event Handler Interface
  - Triggered by an event notification from cars

2.2.1 DB component

ER model In Figure 6 is represented the ER model of the system's database.

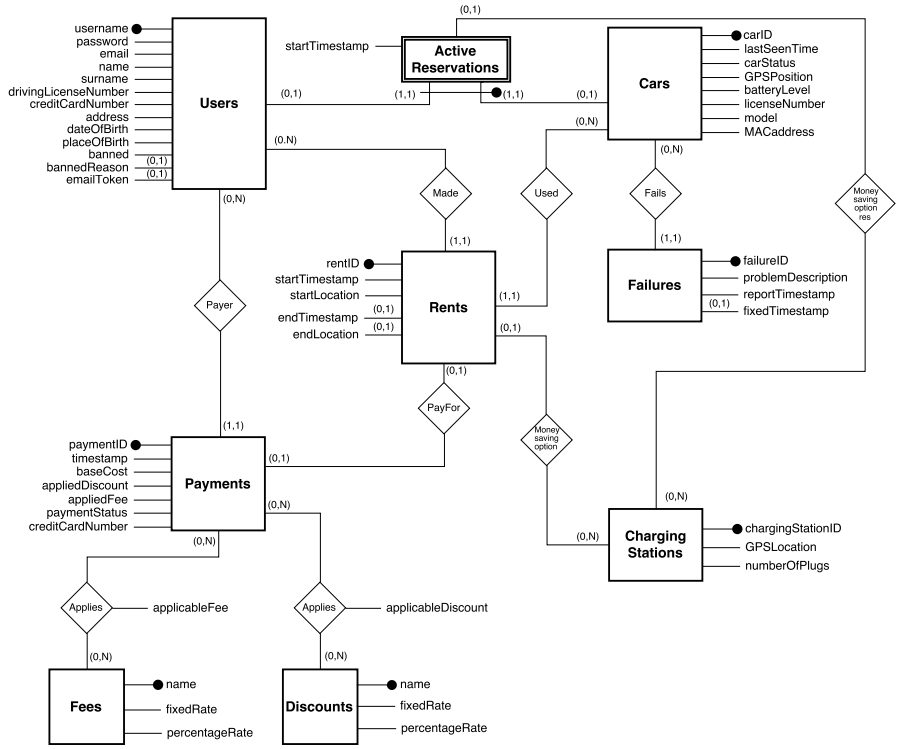

Figure 6: ER model

Users Beyond the primary key, the *email*, *drivingLicenseNumber* and *emailToken* attributes must be unique.

Fees and discounts Fees and discounts must be defined as sum of a fixed value and a percentage factor w.r.t. the base rent cost. The *appliedDiscount* and *appliedFee* payment attributes are determined, respectively, based upon the set of all *applicableDiscount* and *applicableFee* attributes associated to the rent. See payment algorithms for more details.

As particular example, the *OutOfSafeArea* fee, which is composed by a fixed amount plus a variable amount proportional to the *distance* of the car form the nearest safe area, can be expressed as a fixed number of fees clustered w.r.t. distance.

Cars Beyond the primary key, the *licenseNumber* and *MACaddress* attributes must be unique. The *lastSeen* attribute is set by default to the timestamp of the last car information update. The *carStatus* attribute represents the car statuses as defined in the RASD.

Payment status There are three possible payment status:

- *Pending*: a payment transaction request has been sent to the external payment system
- *Confirmed*: the payment transaction has been successfully executed
- *Rejected*: the payment transaction has failed

The default state for a payment is *Pending*.

Failures A failure is considered "open" (i.d. to be fixed) if the attribute *fixedTimestamp* is not present. A car can have at most one open failure.

Charging stations Charging stations are provided to the system as an XML file with the following DTD:

```
<!ELEMENT ChargingStations(ChargingStation+)>
<!ELEMENT ChargingStation(GPSLocation, NumberOfPlugs)>
<!ELEMENT GPSLocation(lat,long)>
<!ELEMENT lat(#CDATA)>
<!ELEMENT long(#CDATA)>
<!ELEMENT NumberOfPlugs(#CDATA)>
<!ATTLIST ChargingStation id ID #REQUIRED>
```

An example of such an XML file is

```
<ChargingStations>

  <ChargingStation id="1">
    <GPSLocation>
      <lat>45.477452</lat>
      <long>9.218617</long>
    </GPSLocation>
    <NumberOfPlugs>10</NumberOfPlugs>
  </ChargingStation>

  <ChargingStation id="2">
    <GPSLocation>
      <lat>45.476257</lat>
      <long>9.171926</long>
    </GPSLocation>
    <NumberOfPlugs>15</NumberOfPlugs>
  </ChargingStation>

</ChargingStations>
```

Safe areas Safe areas are also provided as XML file according to the *GPX GPS Exchange Format* [3] and they are composed by closed polygonal chains.

Safe areas and charging stations deployment The position of charging stations and safe areas are processed and sent to cars through the dedicated primitive only in case of:

- system initialization (sent to all cars)
- changes in XML files (sent to all cars)
- new car (sent to one car)

Security Users' passwords and all credit card numbers (associated to users and used for past payments) must be stored with proper and secure encryption.

2.2.2 Server component

To explain how the Server component manages interfaces, communication with external components and system functionalities we represent in Figure 7 its internal structure showing its main components and their interactions.

This white box representation shows the parts composing the *Server* component and their interactions by means of lollipop-socket notation. When designing this component's internal structure several concerns and requirements were taken into account:

- all of its required and provided interfaces had to be delegated to some part of its internal structure
- all of the functionalities related to the car reservation, unlock, use and rent payment had to be addressed and provided by some parts of this component
- it had to communicate in different ways with different components, so interface specific parts had to be designed
- associations between internal components needed to be clarified, so lollipop-socket notation was used to express the provided or required interfaces of each part

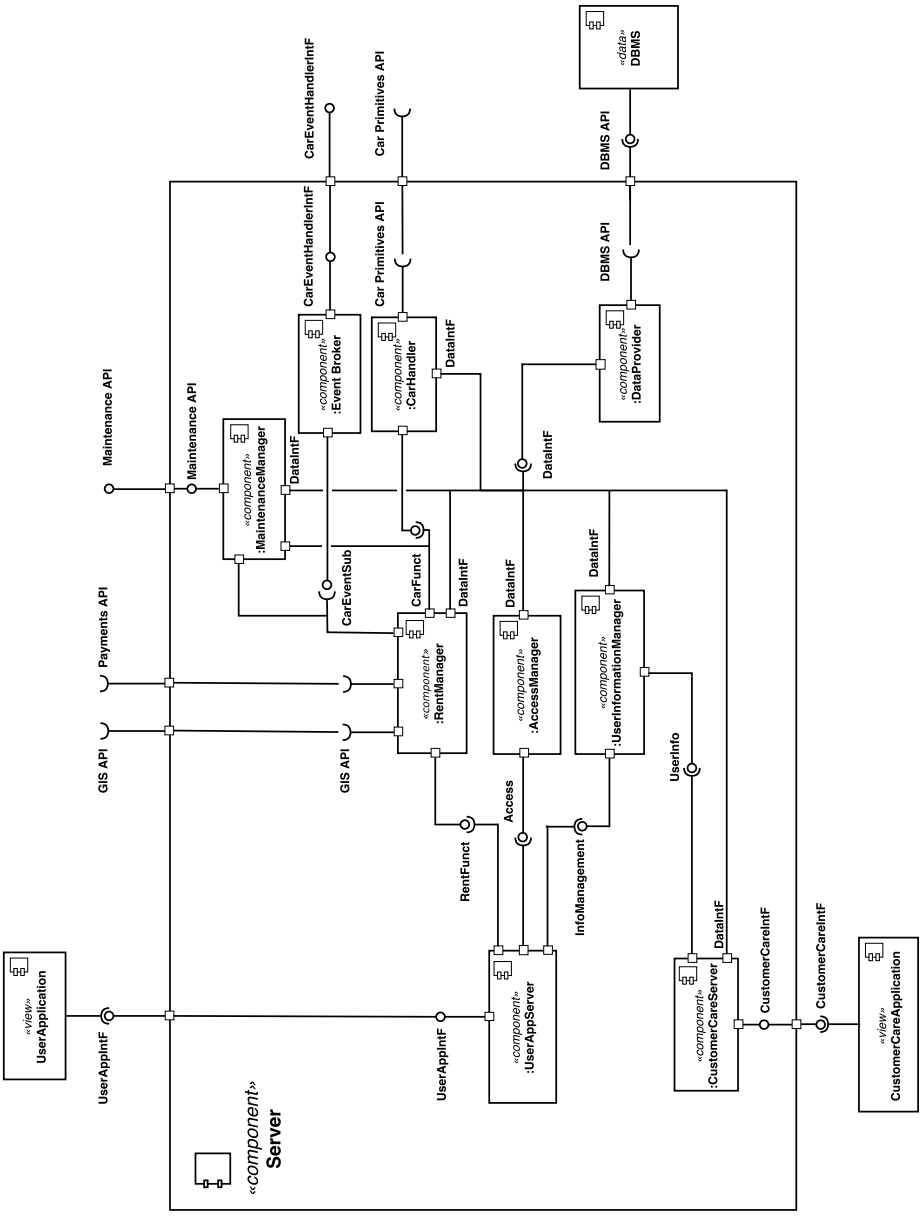

Figure 7: Server component

2.2.3 RentManager component

Figure 8 shows the internal structure of the Rent Manager component; an Object Diagram was used in order to show the objects realizing the component functionalities and the associations between the aforementioned objects. This diagram also shows the delegation associations between the provided and required interfaces of the components and the objects realizing it.
This component encapsulates most of the logic of the System, it manages the information coming from the cars and it creates and manages rents and payments.

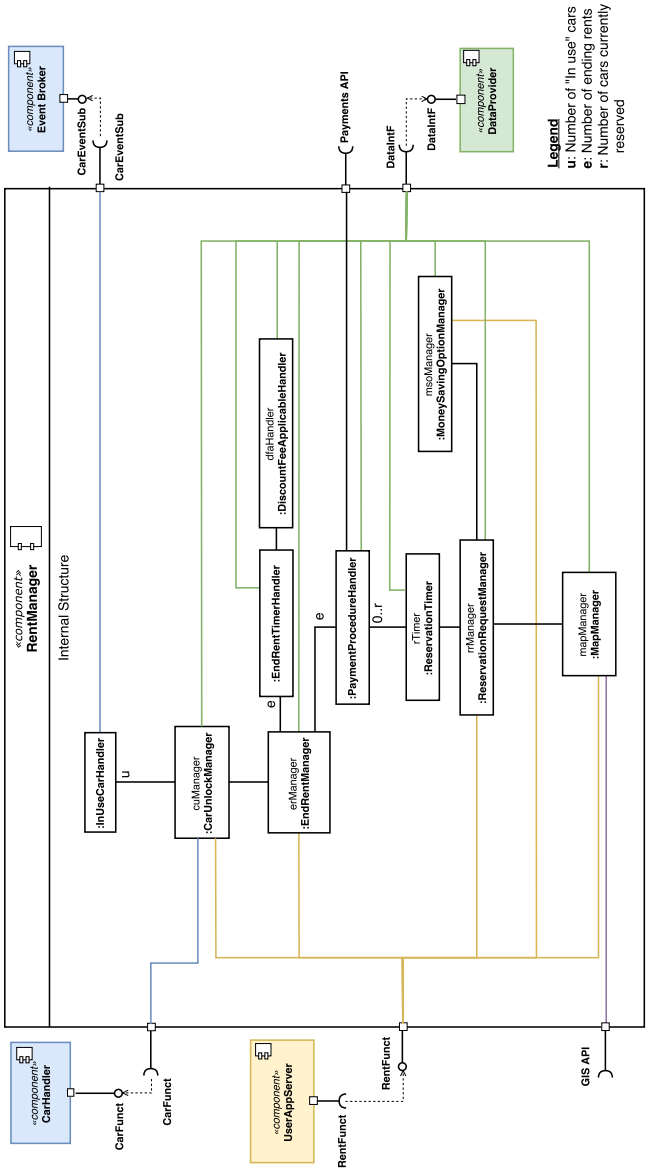

Figure 8: *RentManager* object diagram

InUseCarHandler: this part of the RentManager manages in use cars, it subscribes to the events of a specific car through the EventBroker component which enables asynchronous communication between the cars and the system.

CarUnlockManager: this part of the RentManager is the one that manages the unlock car requests, it checks, by accessing the DataProvider component, that the user who is requesting the unlock actually reserved the car and that he is less than 5 meters away from it; after that it activates the necessary triggers on the car and instantiate an InUseCarHandler object which will subscribe to them. When a rent ends the CarUnlockManager is notified by the corresponding InUseCarHandler, it will update the status of the car and it will notify the EndRentManager component in order to terminate the rent.

EndRentManager: this part of the RentManager is the one that manages the termination of the rent and that starts the payment procedure via the PaymentProcedureHandler object.

ReservationRequestManager: this part of the RentManager component is the one in charge of receiving and validating the reservation requests made by users.

PaymentProcedureHandler: this part of the RentManager allows other components to instantiate payments and interacts with the External Payment System in order to realize them.

ReservationTimer: this part of the RentManager component checks if any reservation expires and if so it instantiate a ProcedureHandeler in order to make the user pay the fee for the expiration.

MoneySavingOptionHandler: this component is used by the ReservationRequestManager when a user enables the Money Saving Option in his reservation; given the user's destination it calculates the most convenient Charging Station and it returns it to the ReservationRequestManager.

EndRentTimerHandler: this timer is used to let the user connect the car to a power plug after his rent has ended; if a car is connected to the power plug after this timer has expired, the user won't receive the related discount.

DiscountFeeApplicableHandler: this part of the RentManager component is used by the EndRentTimerHandler at the end of the rent and it checks which Discounts and which Fees are applicable to the rent.

2.2.4 MaintenanceManager component

Figure 9 shows the internal structure of the Maintenance Manager component; an Object Diagram was used in order to show the objects realizing the component functionalities and the associations between the aforementioned objects. This diagram also shows the delegation associations between the provided and required interfaces of the components and the objects realizing it.
This component provides the Maintenance Service with all the functionalities it needs, granting it access to the API and allowing it to see unavailable cars and tag fixed cars as available.

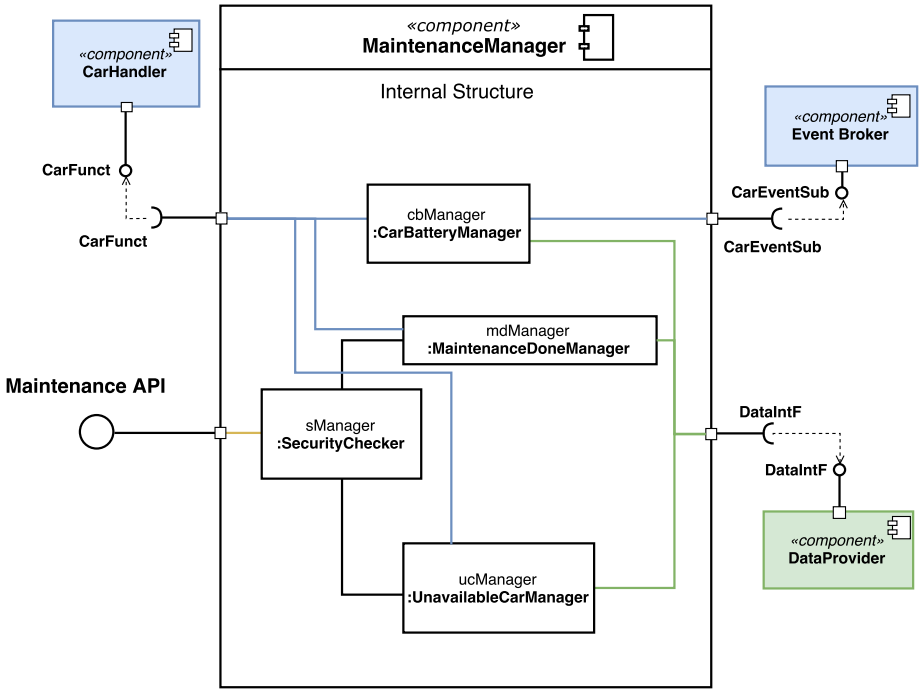

Figure 9: *MaintenanceManager* object diagram

CarBatteryManager: this part of the MaintenanceManager component can check the battery status of the cars and be notified by them when their battery level is critical.

SecurityChecker: this part of the MaintenanceManager componenent is in charge of ensuring that only the Maintenance Service is able to access the services offered by the API.

UnavailableCarManager: this part of the MaintenanceManager component builds the list of *Not available* cars from the database and for each car of the list generates software keys to let maintenance operators to access those cars.

MaintenanceDoneManager: this part of the MaintenanceManager component allows the Maintenance Service operators to tag a car as *Available* after fixing a problem.

2.2.5 UserInformationManager component

Figure 10 shows the internal structure of the UserInformationManager component; an Object Diagram was used in order to show the objects realizing the component functionalities and the associations between the aforementioned objects. This diagram also shows the delegation associations between the provided and required interfaces of the components and the objects realizing it. This component manages the user information present in the system, granting access to it to the Customer Care and to the User Application Server.

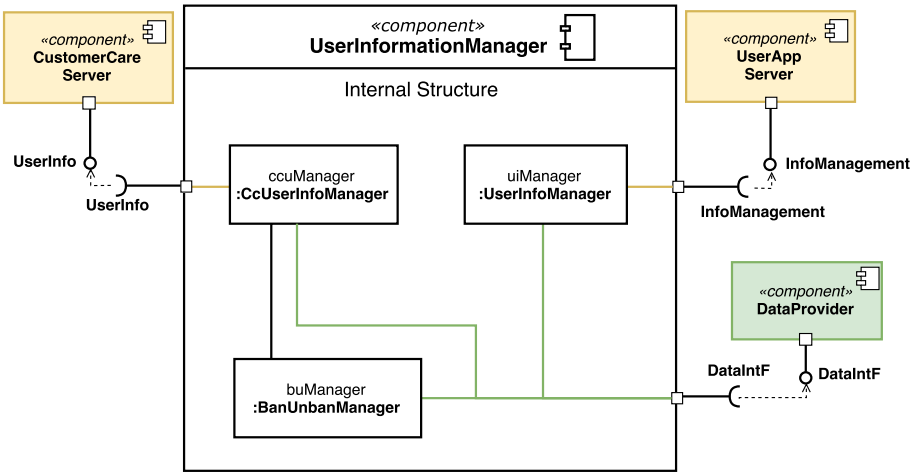

Figure 10: *UserInformationManager* object diagram

CcUserInfoManager: this part of the UserInformationManager component provides the Customer Care with the information it is allowed to access, and allows it access the BanUnbanManager.

UserInfoManager: this part of the UserInformationManafer component provides the User Application Server with the information it is allowed to access.

BanUnbanManager: allows the Customer Care to ban or unban Users.

2.2.6 AccessManager component

This component manages the registration and login phase of users, it encapsulates the logic that allows the users to register and login. Having a component for this specific purpose enhances decoupling in the system and ensures that these functionalities will not be affected by changes in other components of the system.

2.2.7 UserAppServer component

This component acts as an interface between the user and the application logic of the system; it produces the html pages shown to the user by his web browser and it receives the commands the users inputs on the web page. This component solves the concerns related to the handling of the requests sent by the users: these connection requests and the work related to generating the web pages will be handled by this component, without causing load problems on the application logic. It also allows the decoupling of the application logic from the presentation logic, which has to be realized in a four tier client-server architecture.

2.2.8 CustomerCareServer component

This component provides the CustomerCareApplication client with the information it needs to show and with the functionalities it needs to access; this component was decoupled from the UserAppServer component because it offers different functionalities and is only used and accessible by a specific client application.

2.2.9 DataProvider component

This component is in charge of providing a unified interface to access the different data sources of the System, both the DBMS and the files contained in the File System of the OS hosting the application logic. This component allows a better decoupling between the application logic components and the underlying data layer.

2.2.10 CarHandler and EventBroker components

These 2 components are responsible for the communication with the cars embedded system.

Car Handler exposes to the PowerEnJoy system all of the primitives provided by the car embedded system; it does that via a client server communication in which the cars embedded system is the server, offering its primitives, and the CarHandler component is the client, requesting the primitives.

EventBroker instead is notified by the cars embedded system when some specific events take place in the car system; this communication is realized via a publish subscribe paradigm in which the cars embedded system is the publisher, publishing events on the EventBroker component, and the internal components of the PowerEnJoy system are the subscribers which are interested to the events taking place in the cars.

2.3 Deployment view

2.3.1 Four tier architecture

Taking into account that:

- the *Composition viewpoint* diagram shows the need of database decoupling from the actual system
- in the *Server component view* we can clearly distinguish modules who take care of presentation and communication with the client
- in the *Server component view* we can clearly distinguish modules who take care of the specific application logic

we decided to design the system on a four tier architecture pattern (see also

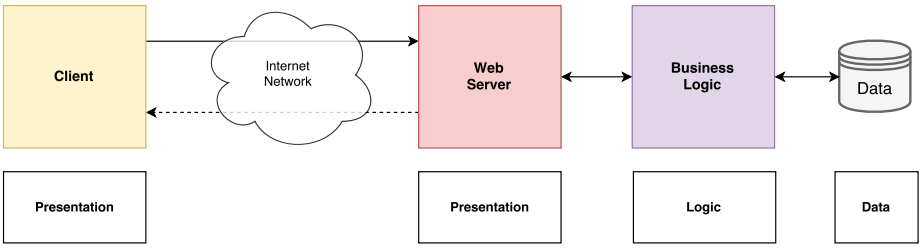

Figure 11: Four tier architecture with internet layer

Mapping of Server components on architecture: to clarify at a finer level how the Server components are mapped in the four tier layered architecture the following diagram represents the Server components highlighted in the same color of the tier in the previous diagram:

- Client: components used by users in order to access the functionalities offered by the system
  - UserApplication
  - CustomerCareApplication
- Web Server: components which provides interfaces to clients in order to allow them to use functionalities offered by the system
  - UserAppServer
  - CustomerCareServer
- Business Logic: components which realizes the functionalities offered by the system
  - RentManager
  - AccessManager
  - UserInformationManager
  - MaintenanceManager

  - EventBroker
  - CarHandler
  - DataProvider

- Data: components which store and manage the access to the data produced and needed by the Business Logic
  - DBMS

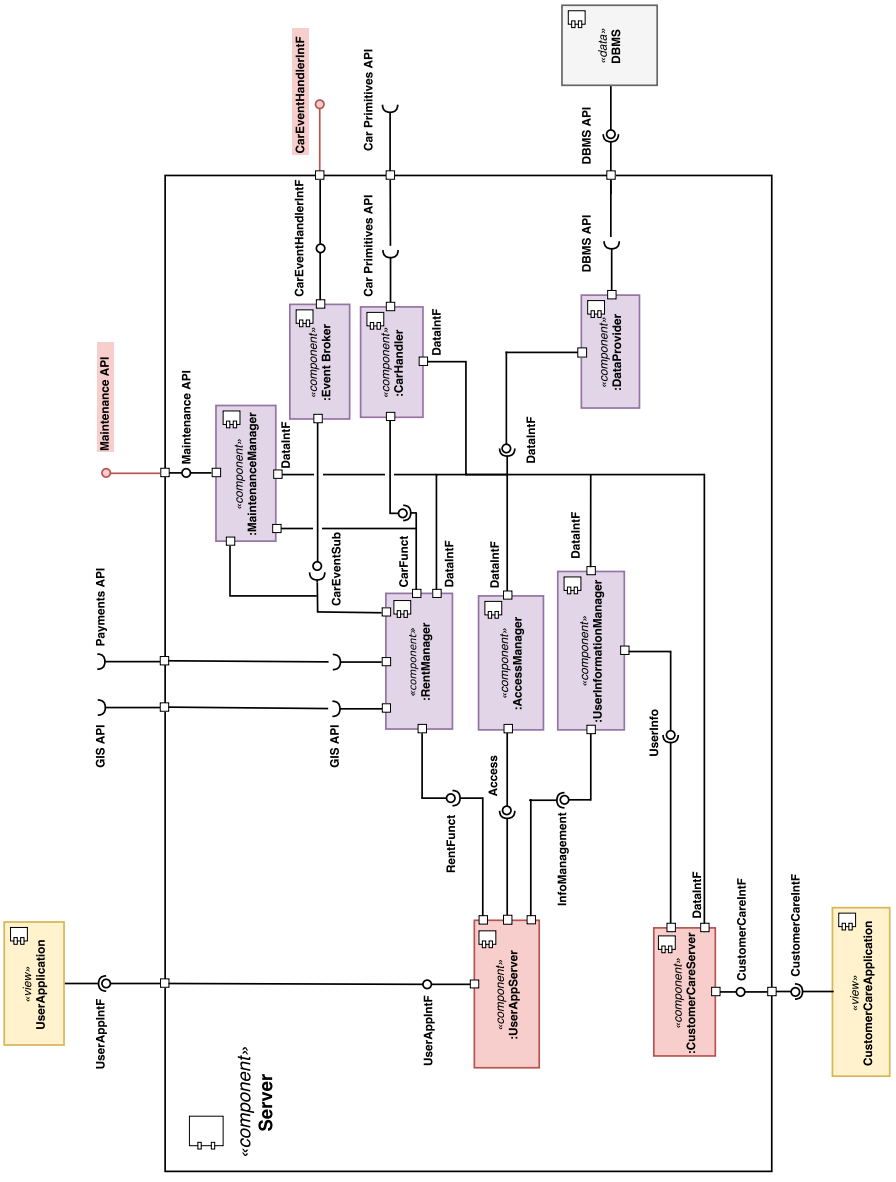

Figure 12: Mapping server component on architecture

2.3.2 Implementation choices

The following technologies were chosen for the implementation of the PowerEnJoy system. J2EE was the main choice we made because it offers a wide range of functionalities that makes the development of scalable multi tiered web based application much easier than with other technologies.

Web Pages: JSP was chosen over Servlets because the content of our web pages has few dynamic parts so writing small chunks of Java code in html pages is a more suitable solution

Application Logic: EJB were used to implement the Application Logic components since our application is developed using J2EE

Application Server and Web Server: GlassFish 4.1 was chosen since it offers containers both for the EJB and the JSP pages

Provided APIs: all the API provided by this system are compliant with the REST paradigm so JAX-RS was used

Data - Application Logic Communication: JPA over JDBC was chosen as a mean to enable communication between the Application Logic and the DBMS

Web Server - Application Logic Communication: JNDI was chosen as a naming and directory service to allow the web servers to access the functionalities offered by the Application Logic

DBMS: MySQL was chosen as DBMS since it is the most popular and widespread and it has a lot of documentation and a big community of users; in combination with it we chose InnoDB which allows the usage of foreign keys

2.3.3 Decision Tree

Figure 13 represents a summary of the main design and implementation decisions that were made during the development of the architecture of the PowerEnJoy system.

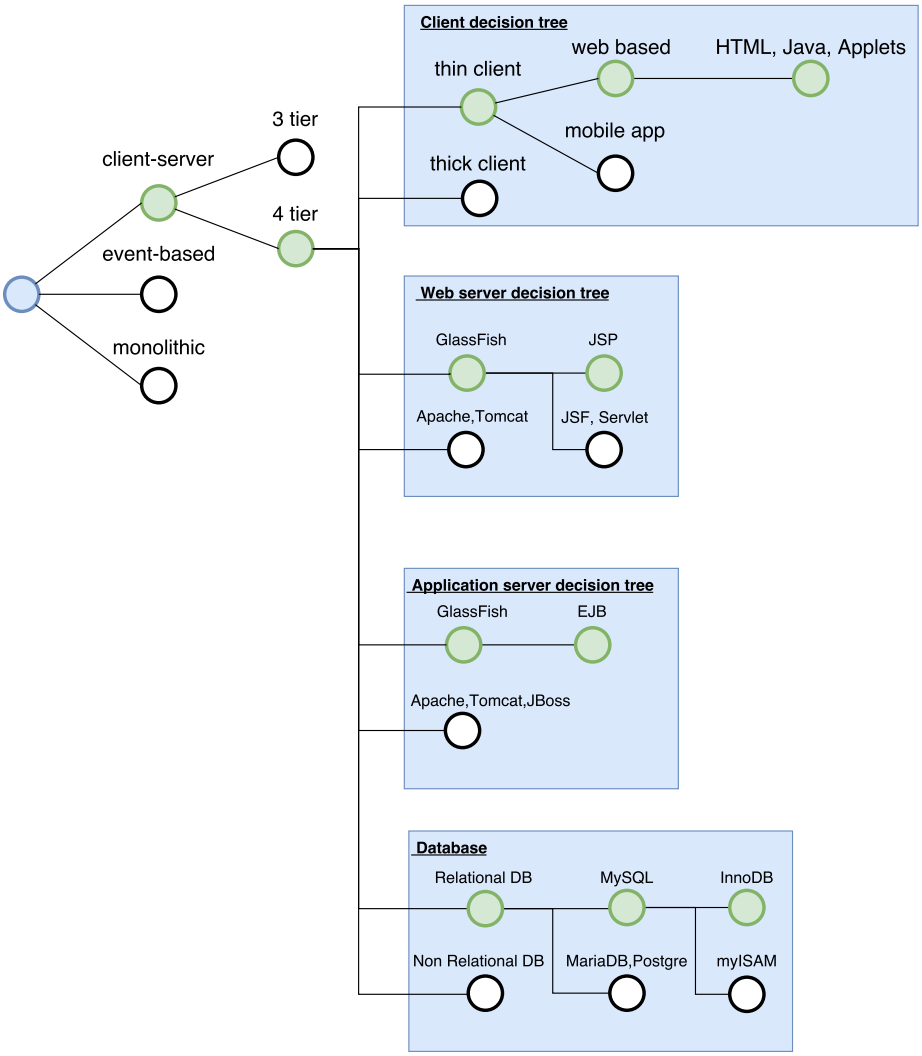

Figure 13: Decision Tree

2.3.4 Deployment diagram

The diagram in Figure 14 represents the mapping of the software components depicted in Figure 12 and Figure 5 on the devices that will run them. Many design concerns were considered while developing this solution.

Security: security is ensured in different points of the architecture, in particular the *UserAppServer* uses HTTPS as communication protocol to communicate with the users; the devices running the *CustomerCareApplication* component are in a VPN with the device running the *CustomerCareServer*.

Scalability: this model of deployment is scalable in the sense that the system administrators will be able to add more devices and deploy more instances of the needed components when and where performance issues will arise, in order to maintain a minimum level of performance even with loads increase.

Decoupling: decoupling in this architecture is present at different levels; in the deployment diagram it is clear that the each of the four tier runs on different devices, moreover the *UserAppServer* component runs on a different device than the *CustomerCareServer*, the Maintenance API and the *CarEventHandlerIntF* interface.

Redundancy: in this iteration of the architecture no redundancy of components or devices is present, but it is allowed in prevision of future expansions of the system's infrastructure.

Fault Tolerance: deploying different components on different machines allows the system to be easier to recover in case of a problem on one of the machines; for an example the Web Server running the *UserAppServer* component goes down, it can be replaced with another machine and in the mean time the *ApplicationServer* would still be up and running and still be able to provide the *CustomerCareApplication* with its functionalities.

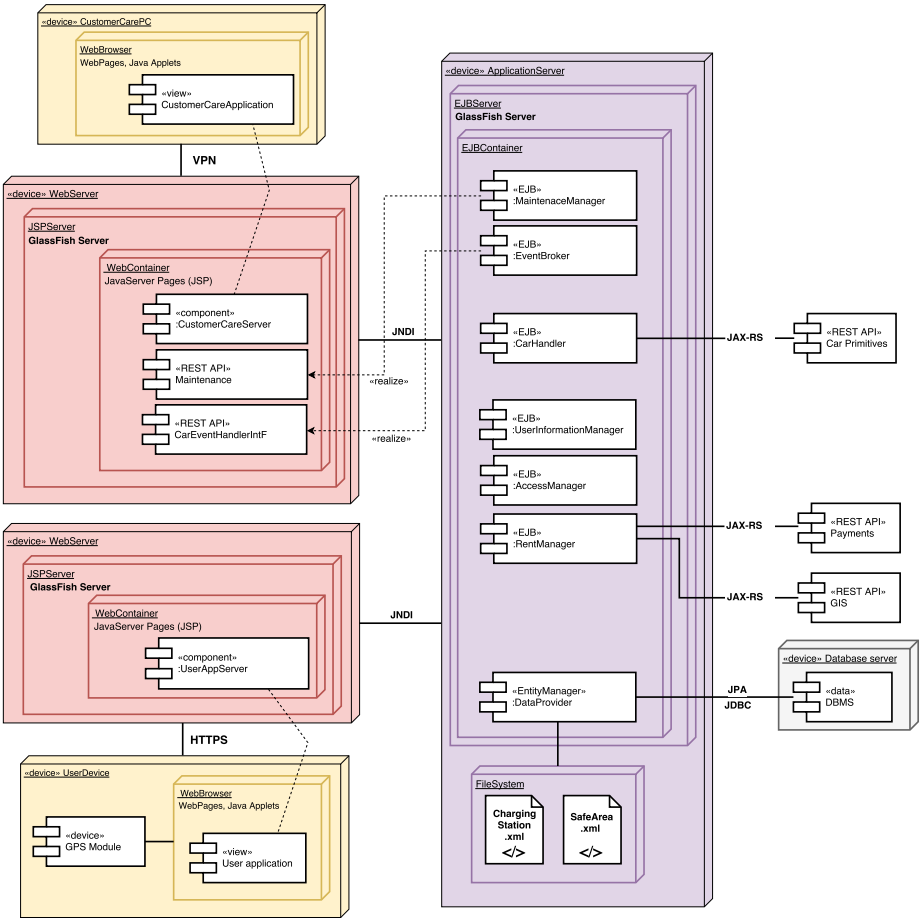

Figure 14: Deployment diagram

2.4 Runtime view

2.4.1 Map

The Figure 15 shows how the map is combined with safe areas, charging stations and available cars nearby a location given by the user. The user interface allows the user to input a specific location or use the GPS position of his device; the GIS will then provide a reference to a map centred in that location and populated as specified.

The *MapManager* component takes care of updating the battery level information of the car shown on the map which the system has not interacted with for the last two hours.

Notes The self-message *setParameters* is used to represent CarHandler operations about retrieving from cars the updated battery level and update the cars JPA.

Interactions not represented If the user inserts a specific location as address, the *UserAppServer* will send it as a string to the *MapManager* which will call a GIS API method to resolve it as a GPS position.

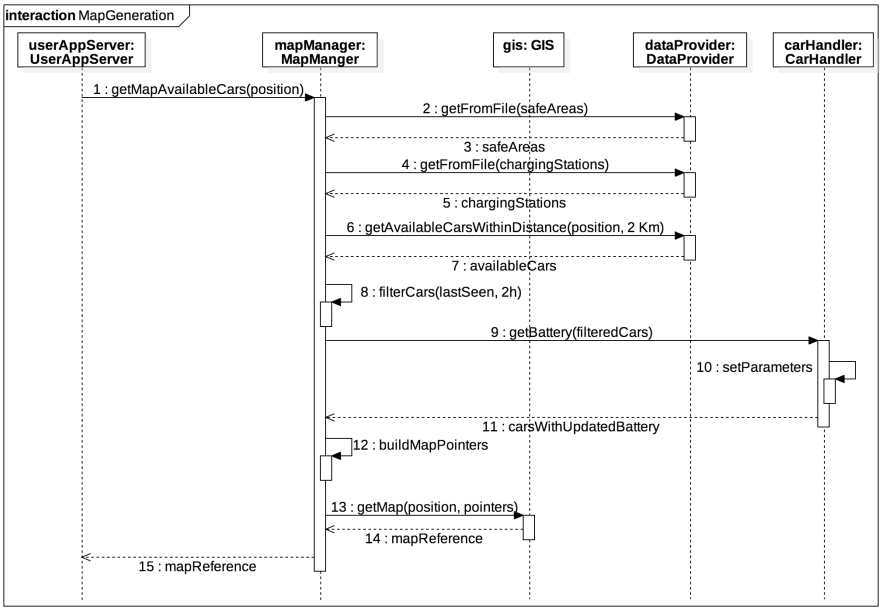

Figure 15: *Map generation* sequence diagram

2.4.2 Car reservation

The diagram in Figure 16 shows the car reservation process.
The *MoneySavingOptionManager* component determines the appropriate charging station for the money saving option given the user destination using the algorithm described in the proper section.

Interactions not represented

- The *createReservationJPA* method of *DataProvider* checks if the car is *Available* before creating the reservation, the eventual exception flow is not represented.
- The *ReservationRequestManager* need an interaction with the external GIS API in order to convert into latitude and longitude the destination inserted by the user for the money saving option.
- To improve redeability of the diagram some interactions between *MSOManager* and *DataProvider* are omitted.

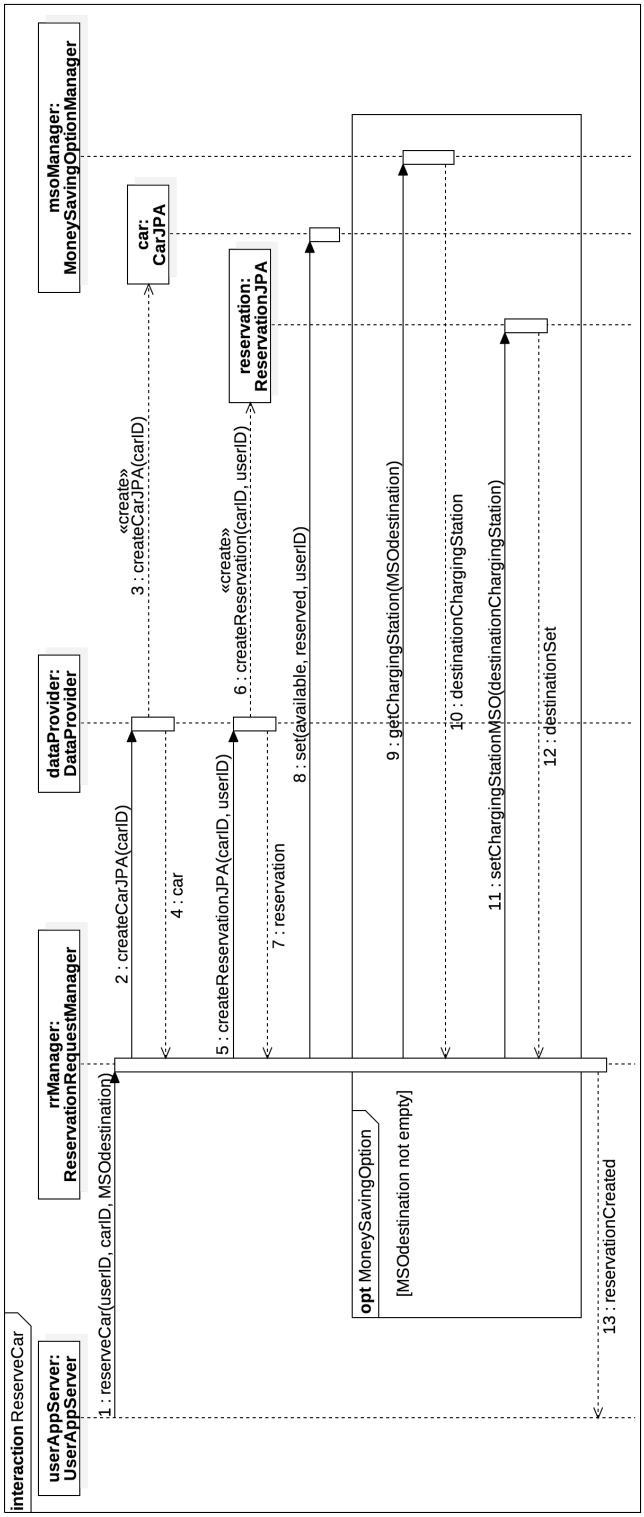

Figure 16: *Car reservation* sequence diagram

2.4.3 Performing a rent

The following sequence diagrams show the interactions between the Rent Manager and other system components in different scenarios:

- Figure 17: a user unlocks the car he reserved and starts the rent
- Figure 18: a in use car notifies the system that the engine is off, there are no passengers in the car and the doors are closed
- Figure 19: the rent just ended and the system initialize the payment procedure

Interactions not represented

- During the rent, at predetermined regular time intervals, *CarUnlockManager* calls a primitive on the car through the *CarHandler* component in order to retrieve the GPS position
- When an attribute is modified in a JPA object the changes are reflected into the *DataProviderComponent* in order to keep the database updated
- When the trigger method is called on *CarHandler*, it uses the carJPA passed as parameter in order to enable on the correspondent car the triggers passed as parameters; this happens as specified in previous sections.
- The *EventBroker* component calls the notify method on *InUseCarHandler* when a notification is received by a car, as specified in previous sections.
- In Figure 18 the *setParameters* method refers to a set of calls to setter methods of the carJPA in order to updated parameters retrieved from cars

Notes The timestamp saved in the rent record in the DB is the one of interaction *15: startRent* while the interaction *17: startRentTimer* is used to start the timer on the car, just to show the user the current cost of its ride based upon the time-based cost provided to the car using a specific primitive.

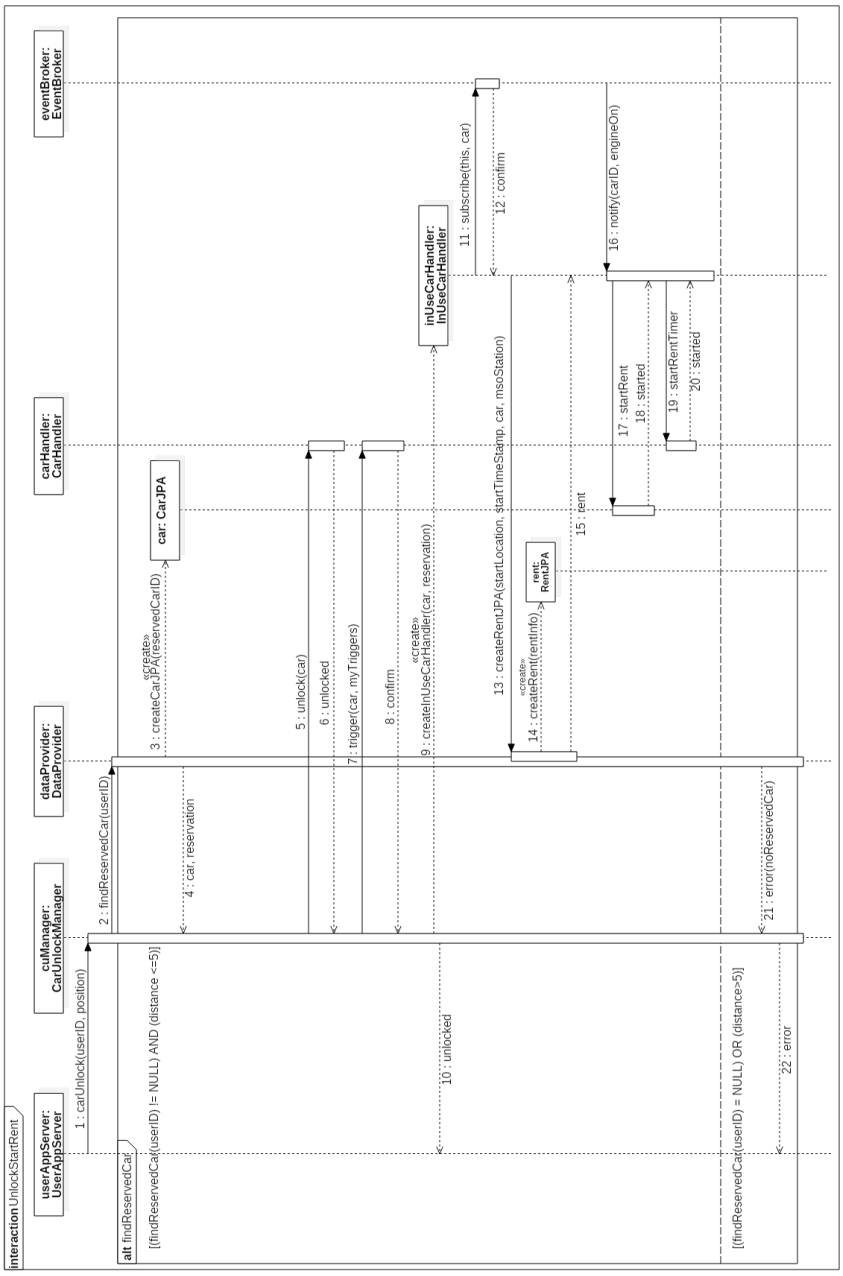

Figure 17: *Car unlock and start rent* sequence diagram

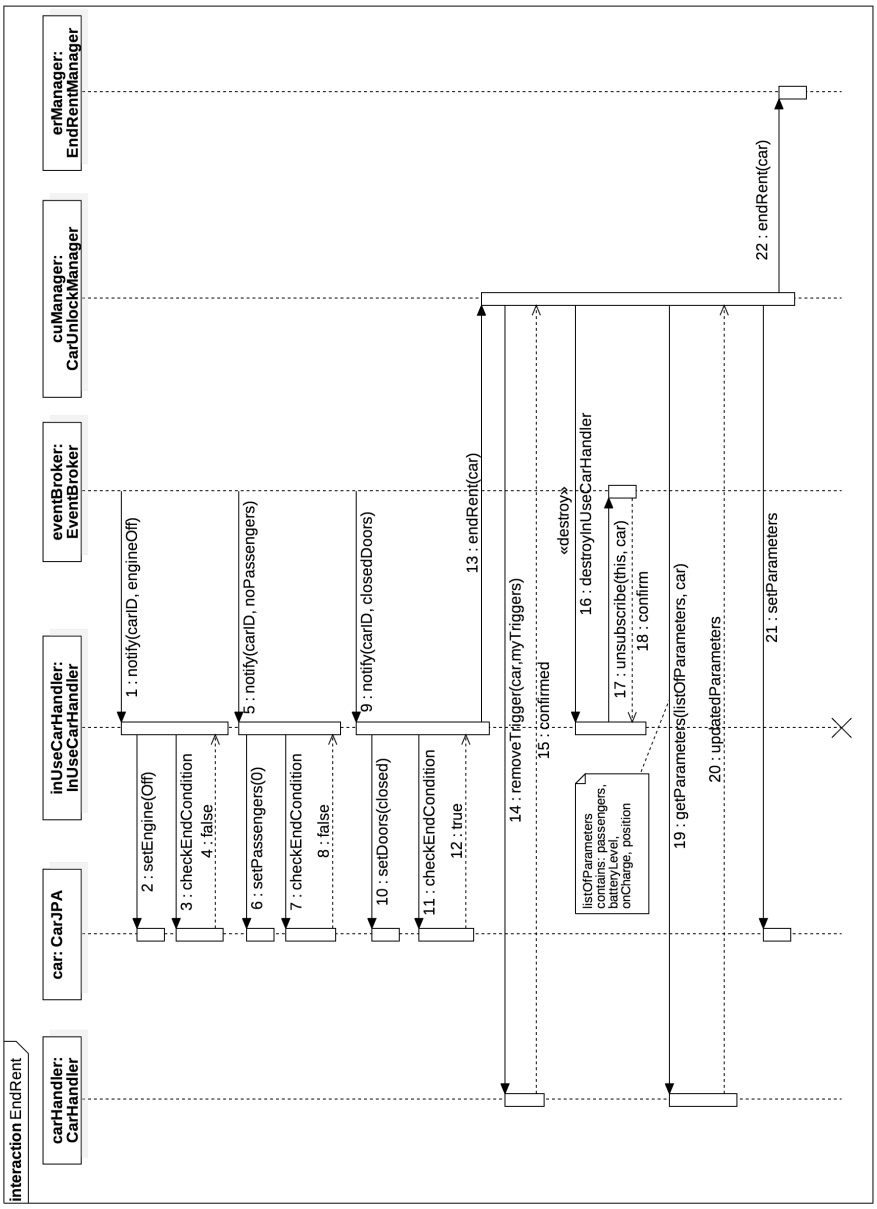

Figure 18: *End rent* sequence diagram (part 1)

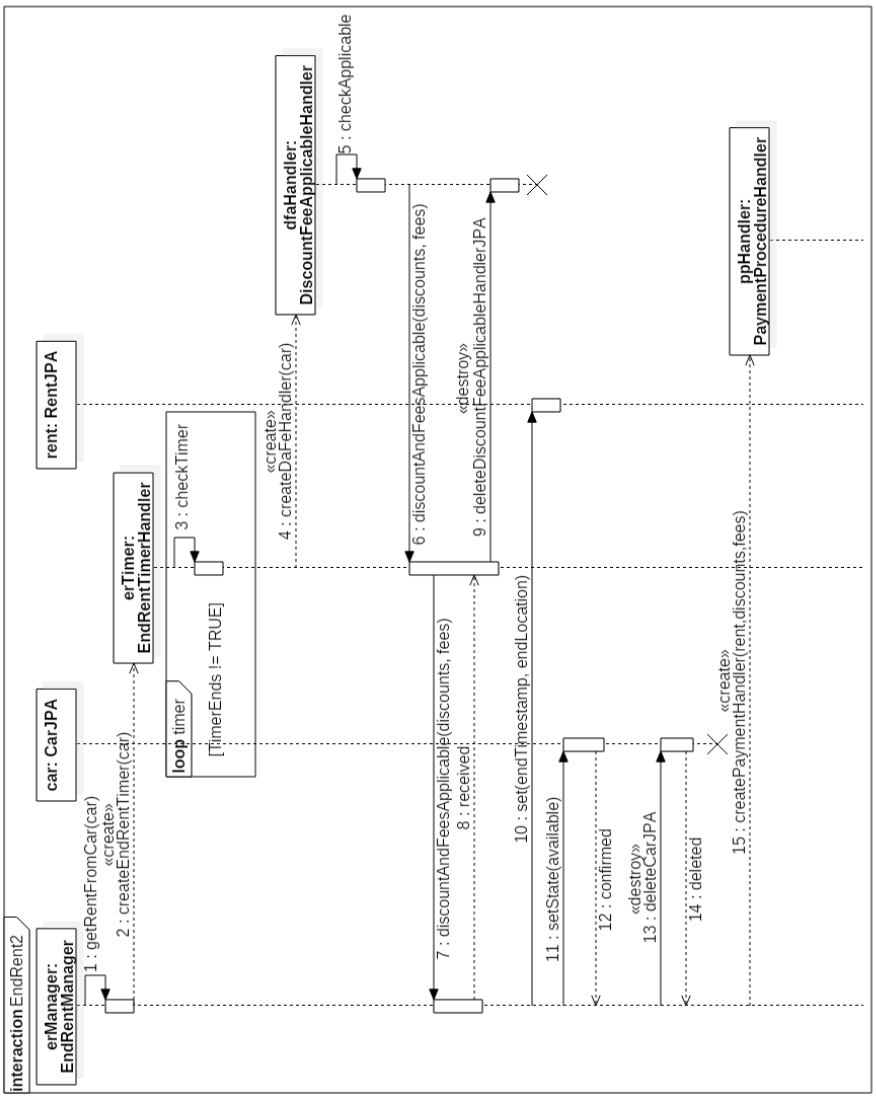

Figure 19: *End rent* sequence diagram (part 2)

2.4.4 Payment

In the Figure 20 is shown a rent payment initiated by the *EndRentManager* component. This diagram is the continuation of the diagram in Figure 19.

Interactions not represented An interaction by the *EndRentManager* with the *DataProvider* component is required in order to ban a user if the payment is rejected.

Notes In this sequence diagram and some of the following ones the usage of the *«create» message* to create a JPA object implies the usage of the DataProvider component, which is the only component able to create JPA objects; we omitted this interaction to improve the readability of the diagrams.

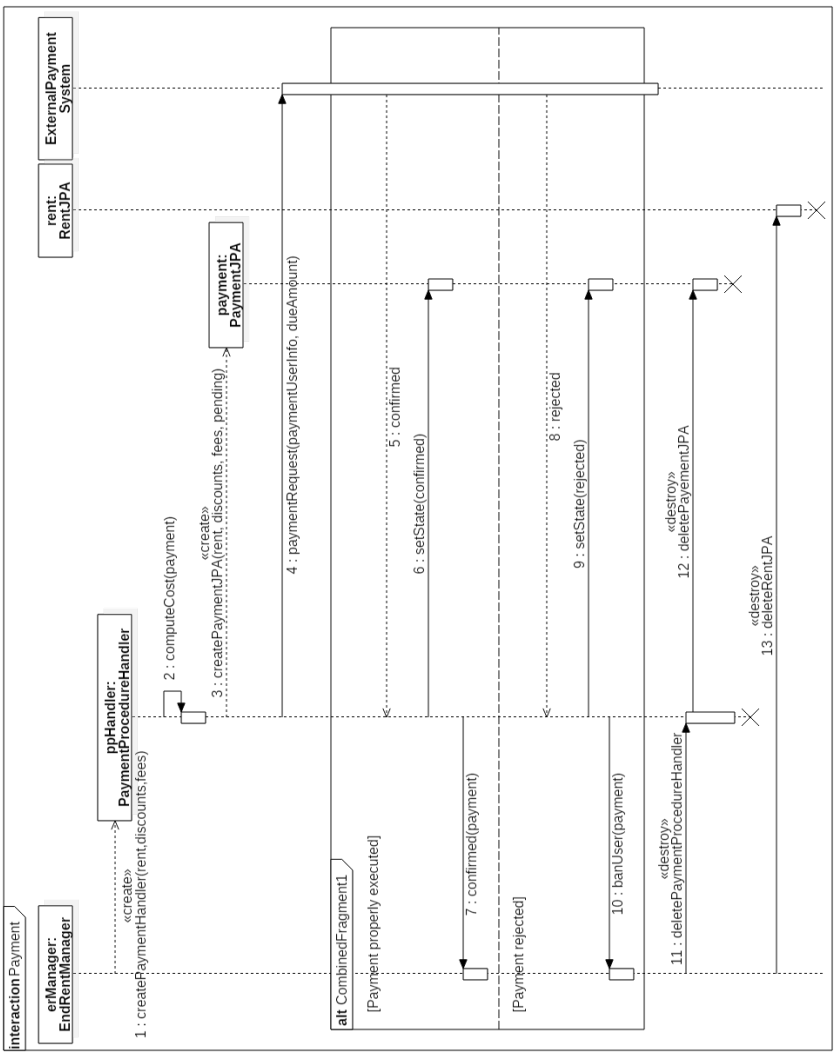

Figure 20: *Payment* sequence diagram

2.4.5 Reservation expiring

To ensure a maximum reservation time of one hour, the *ReservationTimer* component shown in Figure 21 checks every minute the presence of expired reservations in the database.

When a reservation expired is found, this component takes care of creating a payment request for the relative fee and deletes the reservation.

Note The extra time that could be granted to a reservation due to the timer interval and the system's latency is considered acceptable.

Interactions not represented An interaction by the *ReservationTimer* with the *DataProvider* component is required in order to delete a reservation and to ban a user if the payment is rejected.

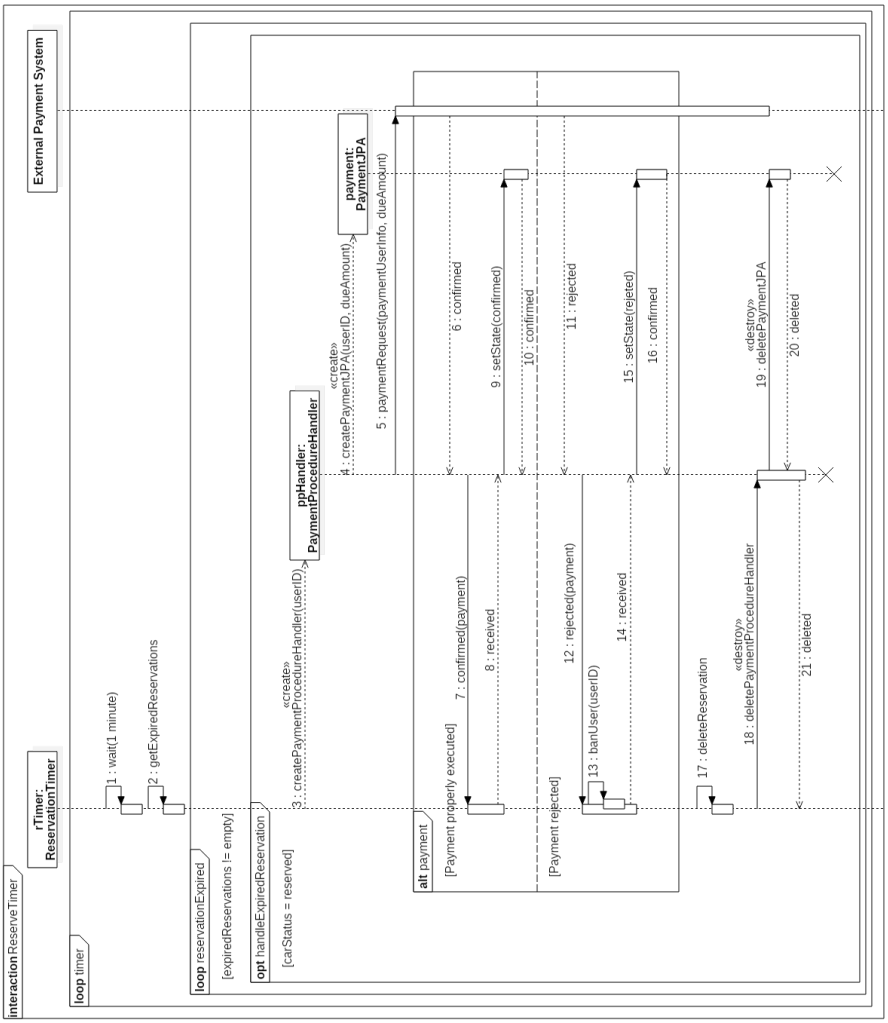

Figure 21: *Reservation expiring* sequence diagram

2.4.6 Maintenance API and failures occurrence

Our *Maintenance API* is used by maintenance operators to obtain a list of car failures and to tag the cars back as Available when the failure has been fixed. The execution of these two commands is shown in Figure 22.

Interactions not represented When a car has to be tagged back as Available the *MaintenanceDoneManager* takes care of verifying via *DataProvider* and *CarHandler* components if the car is associated with an open failure, if the doors are locked and all the others checks defined in the RASD[2].

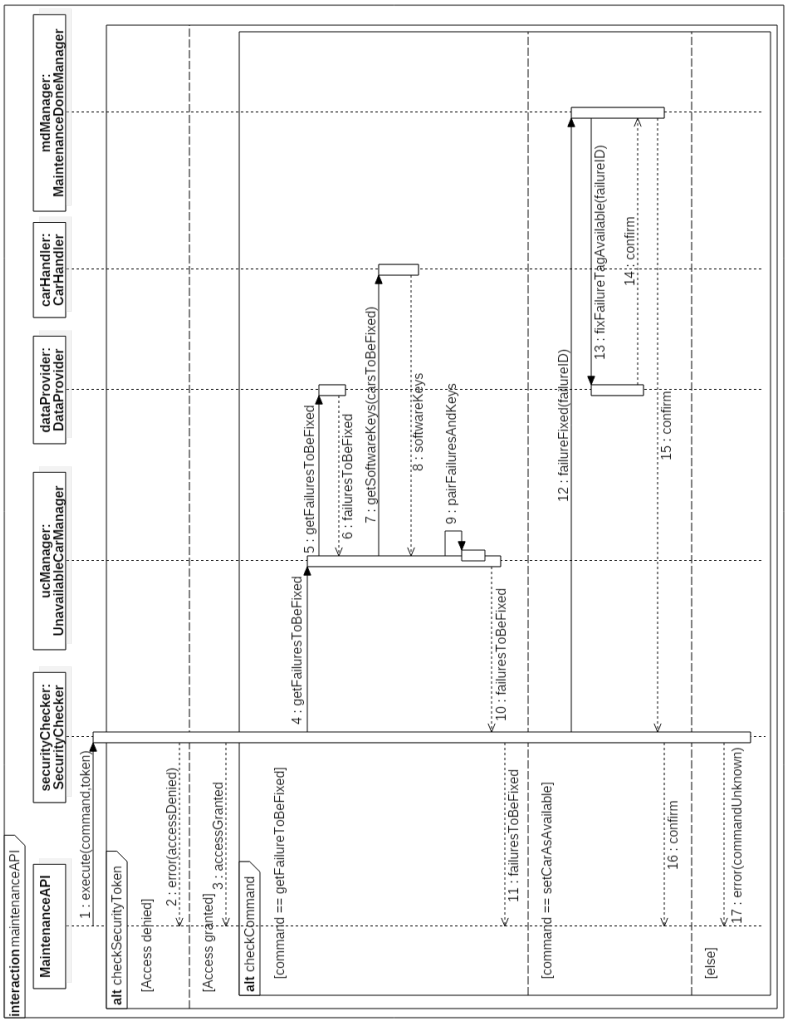

Figure 22: *MaintenanceAPI* sequence diagram

Critical battery level In Figure 23 is represented the case in which a car battery reaches its critical level.

Note The *CarBatteryManager* component must be subscribed via the *EventBroker* component to all cars and it must enable via the *CarHandler* component the trigger for the critical battery level event. We can suppose that these operations are done during the initialization phase.

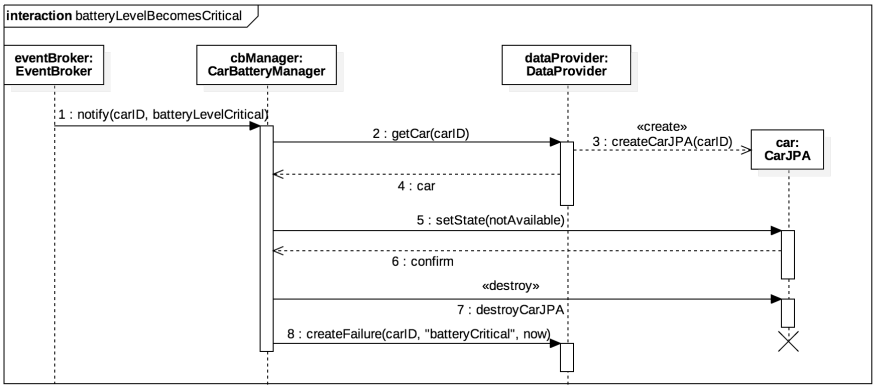

Figure 23: *Battery level critical trigger* sequence diagram

2.5 Component interfaces

2.5.1 Car-server interfaces

All cars are provided with a module that grants TCP/IP connectivity over a dedicated subnet offered by a specific ISP. In order to ensure better security for the communications, in the same subnet is also located the server with the *CarEventHandler* interface and the API used by Maintenance.

The *CarHandler* component communicates to cars via the *CarPrimitives* external API provided by cars (using the JAX-RS API) which executes functions directly on their modules.

A specific car is reach by the system using its IP address or via ARP protocol using the *MACaddress* database attribute when the IP is unknown.

2.5.2 Maintenance API interface

The Maintenance API must be realized using the REST paradigm. The external maintenance system should query the Maintenance API at predefined time intervals in order to retrieve an updated list of car failures paired with proper software keys in order to unlock the doors.

Considering the importance of the operations performed, these security measures must be taken into consideration:

- The API must be accessible only over the system dedicated subnet
- A security token must be provided in order to perform any action through the API

The maintenance operators have been provided a smartphone which is connected in the same dedicated subnet used for cars and for the server with the Maintenance API. Security tokens must be negotiated by our system and the external maintenance service.

2.5.3 User Application interface

The *UserAppIntF* interface is responsible for communications between the user application and the user application server.

Because of the nature of exchanged data (credit card number, password, and privacy-related information), the protocol to be used is HTTPS.

2.5.4 Customer Care interface

Via the *CustomerCareIntF* interface is responsible for communications between the customer care application and the CustomerCare server.

All computers used by the customer care service are connected in the same dedicated subnet as the CustomerCare server and this web server is accessible only from the aforementioned subnet.

A VPN is used to achieve what mentioned above. Because of corporate data exchange, the protocol to be used id HTTPS

2.5.5 GIS API

Using this external API of a *Geographical Information System* our system is able to:

- retrieve a reference to an up-to-date map centered on a given position
- add pointers to the aforementioned map
- get the latitude and longitude of a given location and vice versa

The *UserApplication* component loads and shows the map using its reference.

2.5.6 DBMS API

Trough the DBMS API the system can retrieve and write data into the database. Note that the *DataProvider* component is the only one that access directly this interface.

2.5.7 Payments API

Payment transactions are processed using this external API of a *Web based electronic payment system.*
Information required from this API are in order to complete a payment transaction (fetched from the database):

- name and surname of the payer
- due amount
- credit card number

Data encryption must be taken into consideration.
The API response must be the payment outcome in JSON format (succeded or rejected).

3 Algorithm design

3.1 Payments

3.1.1 Discount assignment

Let $d_i = [d_{iFixed}, d_{iPercentage}]$ be a general discount with its fixed and percentage components.
Let c be the base cost of the rent.
Let $D = \{d_1, d_2, \ldots, d_n\}$ be the set of all applicable discounts for a specific rent.
Then the final applied discount is

$$d = \max_{d_i \in D}\{d_{iFixed} + d_{iPercentage} \cdot c\}$$

3.1.2 Fee assignment

Let $f_i = [f_{iFixed}, f_{iPercentage}]$ be a general fee with its fixed and percentage components.
Let c be the base cost of the rent.
Let $F = \{f_1, f_2, \ldots, f_n\}$ be the set of all applicable fees for a specific rent.
Then the final applied fee is

$$f = \sum_{f_i \in F}(f_{iFixed} + f_{iPercentage} \cdot c)$$

3.2 Money saving option

The Money Saving Option algorithm returns a charging station with available plugs near the destination inserted by the user to ensure a better distribution of cars.

Each charging station is assigned a score considering the distance from the desired destination, the number of cars nearby and its plugs availability.

The charging station with the highest score is returned.

```
BEGIN MoneySavingOption(dest)

    //dest: GPS position of the user inserted destination

    DATACS := READ "ChargingStations.xml"
    //array of charging station IDs
    C  := retrieveCs(DATACS);
    P,AP,D,NC,S := empty array;
    j:= radius distance constant;

    FOR i:=0 TO C.length DO
        availablePlug :=
            retrieveCsPlugsNumb(C[i], DATACS)
                - carsInCs(C[i]);
        IF availablePlug > 0 THEN
            //AP[i] contains available plugs
            //of C[i] charging station if
            //that number is greater than zero
            AP[i]:= availablePlug;
```

```
20          ELSE
21              //removes the i-th element shifting other
22              //elements respecting the precedent order
23              removeFromArray(C[i]);
24          END-IF
25
26      END-FOR
27
28      FOR i:=0 TO C.length DO
29          //P[i] contains GPS position
30          //of C[i] charging station
31          P[i]:= retrieveCsPosition(C[i], DATACS);
32
33          //D[i] contains the distance
34          //between C[i] charging station
35          //and the distance inserted by the user
36          D[i]:= calculateDistance(P[i], dest)
37
38          //NC[i] contains the number of
39          //cars within a radius of j km from
40          //the C[i] charging station normalized
41          //w.r.t the total number of cars
42          NC[i]:= getNumbCarsNearbyNormalized(C[i],j);
43      END-FOR
44
45      maxDistance := maxValue(D);
46      minDistance := minValue(D);
47
48      //Distances are normalized
49      IF maxDistance != minDistance THEN
50          FOR i:=0 TO C.length DO
51              D[i]:= (D[i] - minDistance)
52                          /( maxDistance - minDistance);
53          END-FOR
54      ELSE
55          FOR i:=0 TO C.length DO
56              D[i]:= 0;
57          END-FOR
58      END-IF
59
60      k := multiplicative weight of distance variable
61      t := multiplicative weight of availability of plugs
62      q := multiplicative weight of cars nearby
63
64      FOR i:=0 TO C.length DO
65          //Score assigned to the C[i] charging station
66              S[i]:=  D[i]*k + AP[i]*t - NC[i]*q;
67      END-FOR
68
69      //if two scores are equal the charging station closer to
70      //the destination inserted by the user is chosen
71      RETURN C[arrayKey(maxScore(S))];
72
73  END
```

4 User interface design

In this section are presented some mockups of the main features and related user interfaces the system is supposed to offer to the user and to the customer care through the proper web-based application.

The presented mockups have been designed based on both the *PowerEnJoy: Requirements Analysis and Specification Document*[2] and on the architectural design decisions and component interactions presented on this document.
In particular mockups show how the user interface is supposed to offer to the user the possibility to interact and make request to the system (obviously such user's interactions will result in a client-server communication of the user/customer care app view with the related server component based on the protocol chosen for that communication).

The main goal of our mockups design process is to build an interface that clearly distinguish functionalities offered by the system taking into account the architectural decoupling offered by the taken design choices.

4.1 User app

The user app must have a charming and intuitive user interface in order to provide a easy-to-use experience to the user. The user app is supposed to be a web-based application, so the interface must be optimized for mobile devices even if the application is accessible and must be usable from every web browser on different size devices.

4.1.1 Login page

Simple initial page for the application to allow the user to authenticate to the system through username and password. A new user can access the registration process through the *New User* button.
As specified in the requirements document, this page also allows a guest user or a *banned* registered user to access to customer care contact information through the *Contact us* button.
If a user is not recognized or is banned an error alert is shown when credential are submitted.

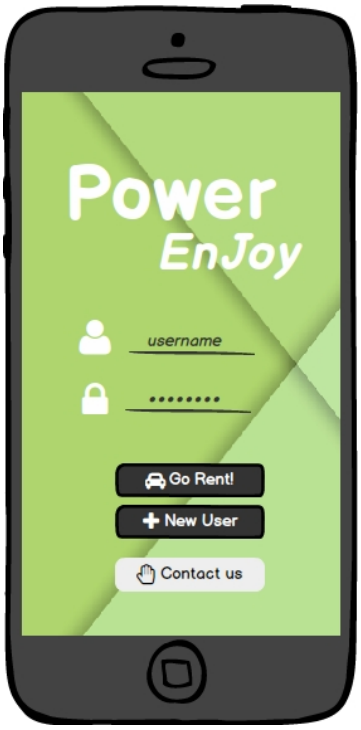

Figure 24: *Login page* mockup

4.1.2 Home page

The home page shows to a logged user all the possible functionalities provided from the app:

- See available cars and reserve one of them (eventually with the *money saving option*)
- Unlock a car (disabled button in the mockups, it would be active only if there is an active reservation for the registered user logged in)
- See user's information and edit them
- See user's rent history
- See user's payment history
- Show customer care contact information

This page shows also the user's name to ensure to the user he has been correctly recognized by the system and to make the interface more customized.

The icon in the right corner, from this page, brings the user to the functionality of see available cars; on all other pages (without the orange position icon) it brings the user to this page: the home page.

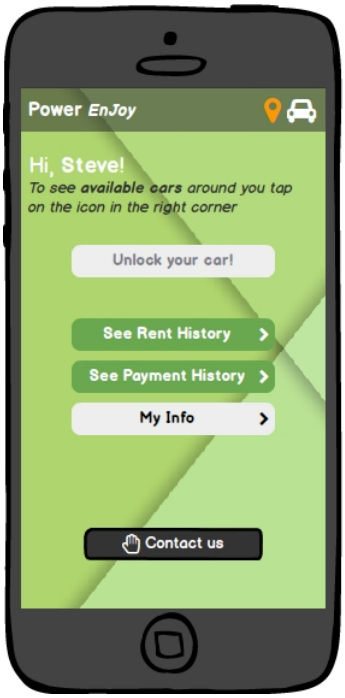

Figure 25: *Home page* mockup

4.1.3 See available cars

Accessing to the see available cars functionality, as described in the requirements document, the user app asks to the user if he wants to search car nearby his actual position (using GPS position of user's device) or he wants to insert a different position from which starting the research.

The *Use GPS* allows the user to choose the first option skipping other interactions on this page.

If the user chooses the second option, he is supposed to insert an address location (e.g. 34 Maria Victoria Lane, London) before pushing the *Search* button; the system will resolve that address as a GPS location displaying an error message if it could not done it.

The *Cancel* button allows the user to return to the home page.

The system searches for available cars (nearby the position given by the user or retrieved by the GPS) and displays a map with available cars on their actual position, charging station positions (green spot) and safe areas (black areas). Blue circled cars are actually plugged on a charging station, all others car are green circled.

The map is interactive and the user can move around the actual position.

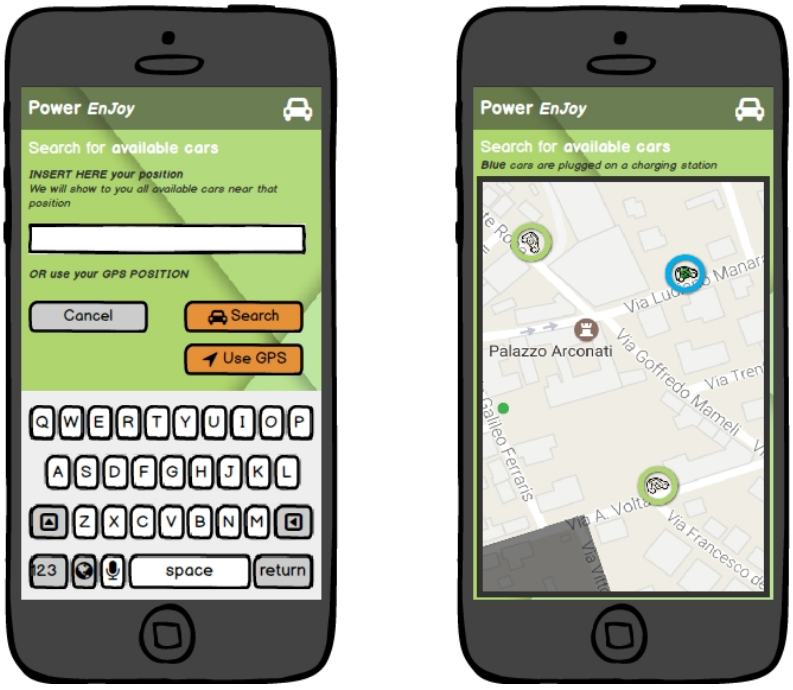

Figure 26: *See available cars* mockup

4.1.4 Reserve a car

From the page showing available cars on the map, a user tapping on a car could access information about it, in particular:

- Model of the car
- License number of the car
- The battery percentage level of the car

When the user taps on a car the circle around it becomes orange, to give a feedback on the tap to the user, and a box appears on the screen. Through it the user can access the aforementioned info about the car and reserve the selected car.

Before clicking the *Reserve it!* button the user has the possibility to choice if he wants or not to enable the *money saving option* through an on/off toggle.

If a user has already an active reservation or the selected car has been reserved while the user navigates on the map an error message is displayed.

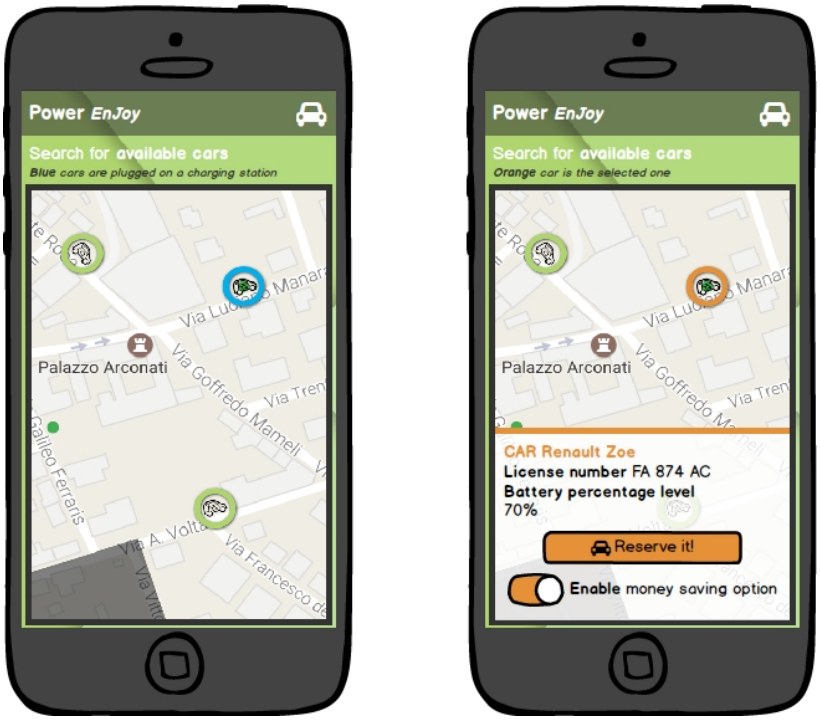

Figure 27: *Reserve a car* mockup

4.1.5 Money saving option

If an user enables the *money saving option* while reserving a car, the user app shows him a dedicated page to accomplish the reservation with the option. On this page the user must insert the planned destination of his rent in order to give to the system the possibility to calculate the charging station the user must leave the car plugged in to get the discount.

A brief description of the *money saving option* is offered to the user to clarify why the user app is asking him his planned destination.

The user is supposed to insert an address location (e.g. 34 Maria Victoria Lane, London) before pushing the *Confirm* button; the system will resolve that address as a GPS location displaying an error message if it could not done it.

If the address inserted by the user is correctly processed the system notifies the user with the charging station (number and address) the user must leave the car plugged in to get the discount.

The *Cancel* button allows the user to go back on the map, for example to make the reservation without enabling the *money saving option*.

Note Note that if the user chooses to enable the *money saving option* the car reservation request is sent to the server only when the user inserts also the destination.

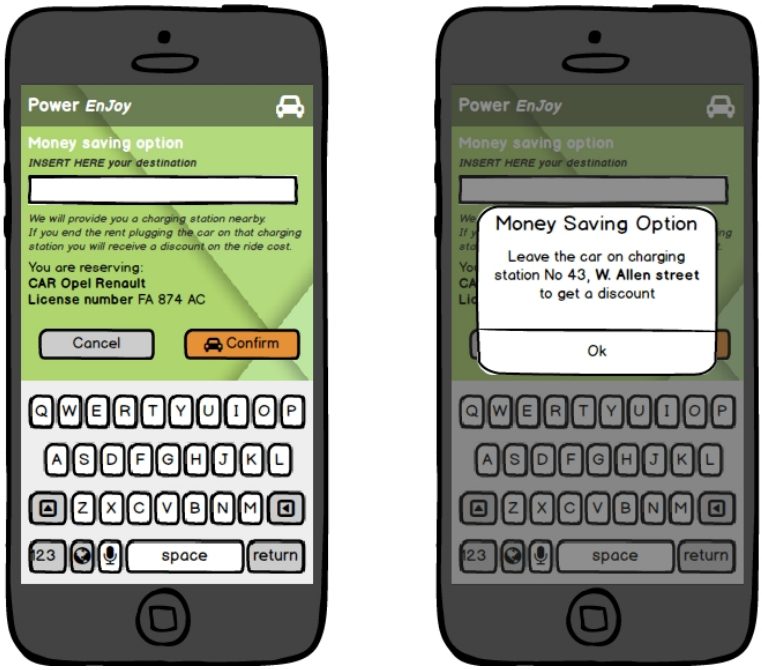

Figure 28: *Money saving option* mockup

4.1.6 Unlock a car

From the home page, if the user has an active reservation he can access to the *Unlock car* functionality through the dedicated button.

On this page the user can see data about his reservation:

- model and license number of the car he has reserved
- time until the reservation expires
- (*Optional*) charging station related to money saving option
- current car position (clicking on *see it on the map* link the user app looks for an installed program on the user device to open the GPS coordinates, if it can not find any program it only shows the GPS coordinates)

The user could ask the system to unlock the car through the *Unlock your car* button. If the user GPS position is not 5m away from the car an error message is displayed to ask the user to reach the car before trying to unlock the car.

If the system manages to process the request to unlock the car a message is displayed to notify the user the car has been unlocked and he could start the rent turning on the engine.

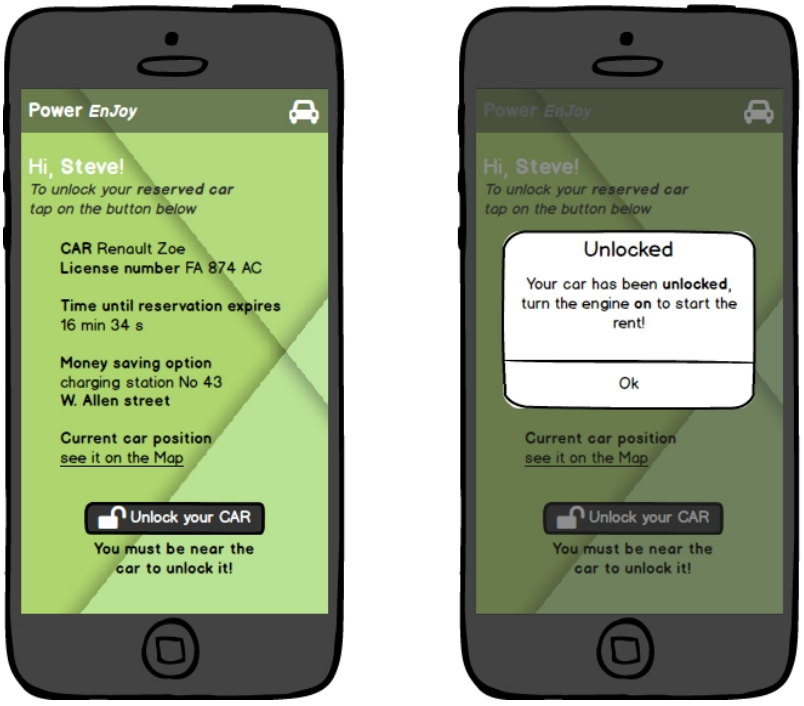

Figure 29: *Unlock a car* mockup

4.1.7 Payment history

From the home page, the user can access his own payment history through the *See Payment History* button.

On this page the user app shows to the user all made payments in chronological order, from the more recent to the latest ones as shown in Figure 30.

Payment not related to a rent, for example fee related to an expired reservation, are shown with a different color to clearly distinguish it.

Through this page a user can only see if a payment is related or not to a rent and date and hour of each payment. The user could access all payments details clicking on one of the rows shown, or can rapidly access to customer care contact information if for example it finds out some payment he is not aware of.

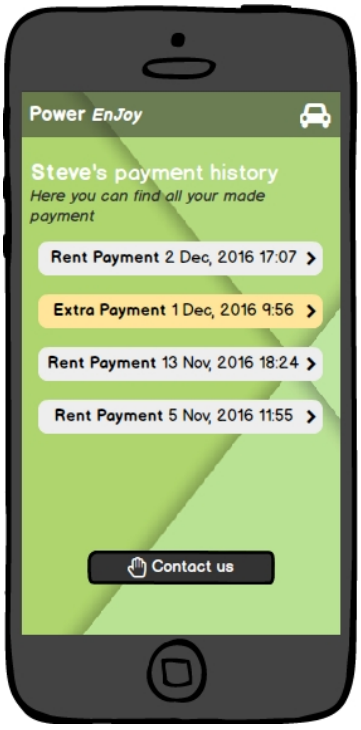

Figure 30: *Payment history* mockup

In Figure 31 are shown two examples of payments history's single record. The one on the right shows a payment related to a rent, instead the one on the left shows a payment related to a reservation expired fee.

For each payment record the user can see information about:

- payment ID
- time of the payment
- base cost (in case of rent it's calculated as *rent time* x *time based rate*)
- discount amount applied on the rent
- fee amount applied on the rent
- total paid amount
- payment used method (for security reason only the last four numbers are shown)
- (*optional*) rent associated with the payment

If the payment is associated with a rent, the user can access the rent record through the link in the *rent* section.

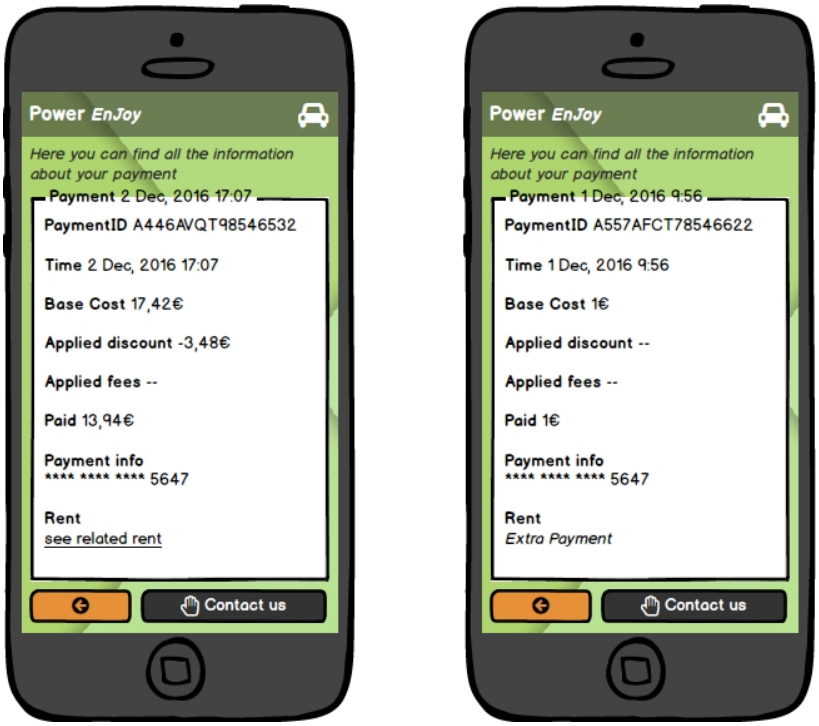

Figure 31: *Single payment records* mockup

4.1.8 Rent history

From the home page, the user can access his own rent history through the *See Rent History* button.

On this page the user app shows to the user all made rents in chronological order, from the more recent to the latest ones as shown in Figure 32. Through this page a user can only see date and hour of each rent. The user could access all rents details clicking on one of the rows shown.

For each rent record the user can see information about:

- rent ID
- start/end time and location of the rent
- all discount applicable to the rent
- all fee applicable to the rent
- payment related to the rent

The user can access the payment record related to the rent through the link in the *payment* section.

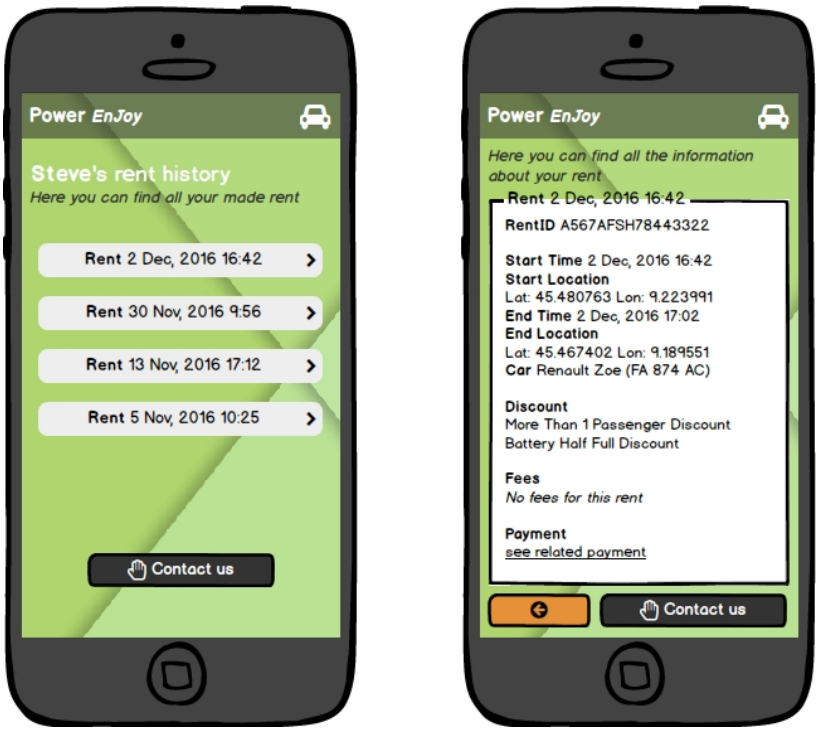

Figure 32: *Rent history* mockup

4.2 Customer care app

The customer care app has a simple interface in order to provide features to customer care operator in a clear and simple way. This interface must be optimized for a desktop monitor size.

4.2.1 Home page

All main features are accessible directly from the home page to provide a rapid and intuitive access to them.
From the home page the operator can:

- Access information about a user through the user's username or email address (see user's information section)
- Mark/Unmark user as banned through its username (if the username inserted is found by the system, in order to fulfil this task a new page is shown otherwise a new error message is displayed. The new page shows to the operator the current state of the user and if the operator wants to mark the user as *banned* asks the operator a brief description of the reasons)
- Retrieve basic data for each user (username, name, surname, email)
- Tag a car as *Not Available* (if the car license number inserted is found by the system, in order to fulfil this task a new page is shown otherwise a new error message is displayed. The new page asks the operator a brief description of the reasons why he wants to mark the car as Not Available)

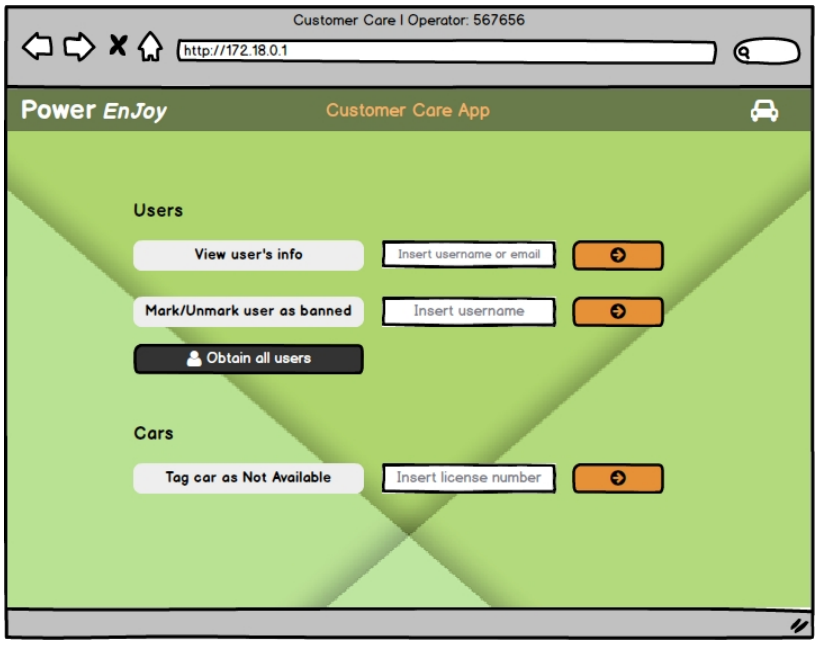

Figure 33: *Customer care home page* mockup

4.2.2 User's information

From the home page the operator could access information about a specific user. In this page the customer care app shows to the operator all user's information unless the sensitive ones (as password and credit car number).

From this page the operator could also access user's rent and payment history through specific buttons.

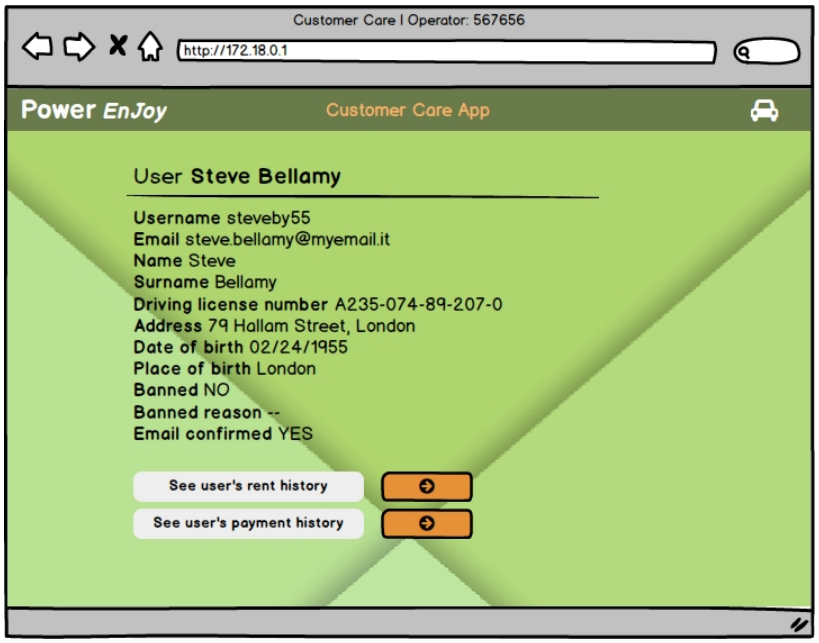

Figure 34: *Customer care user's information page* mockup

5 Requirements traceability

5.1 Functional requirements

In the Table 1 is presented a mapping correspondence between the requirements defined in the RASD related to each goal and the components identified in the server component diagram.

Requirements of goal	Components
G1 Allow guest users to register to the system	UserAppServer AccessManager DataProvider UserInformationManager
G2 Allow registered users to authenticate to the system	UserAppServer AccessManager DataProvider
G3 Provide logged users with the position of available cars	UserAppServer RentManager DataProvider
G4 Notify maintenance service with a list of not available cars	MaintenanceManager DataProvider EventBroker CarHandler
G5 Provide the maintenance service with a way to notify the system when a car is available again	MaintenanceManager DataProvider
G6 Provide users with a way to report a damaged car	UserAppServer CustomerCareServer
G7 Provide a way to show to each user his rents and payments history	UserAppServer UserInformationManager DataProvider
G8 Provide customer service with a way to ban registered users in order to prevent them from reserving or using other cars, and enable them to use the service again	CustomerCareServer UserInformationManager DataProvider
G9 Allow a logged user to reserve a car, if available, and hold that reservation for an hour	UserAppServer RentManager DataProvider
G10 Charge a user for 1€ in case he hasn't used the car he reserved after an hour from such reservation	UserAppServer RentManager DataProvider

G11 Allow a logged user to perform a complete rent: reserving a car, using it and leaving it terminating the rent, accomplishing the payment procedure related to aforementioned rent	UserAppServer RentManager EventBroker CarHandler DataProvider
G12 Calculate and charge the user for the correct amount of money he has to pay for his last ride, also considering the various discounts and fees applicable based on the ride	RentMaganer CarHandler DataProvider
G13 Allow a logged user to enable a money saving option which provides him with a charging station as destination of the ride to get a discount on the cost of the aforementioned ride	UserAppServer RentManager DataProvider

Table 1: Mapping goals on components

5.2 Non functional requirements

Performance requirements To guarantee a short response time we have tried to decouple components in order to enable an instance pooling management of components by the EJB container and so an as much as possible concurrent management of requests.
During all the design process we also have kept in mind the scalability of the software w.r.t. the number of cars trying to keep a linear complexity factor and to reduce car-dependent activities where it is possible.

Availability To ensure availability requirements server will be running 24 hours for day. The architecture is designed with the purpose of enabling a redundancy architecture if availability constraints would make it necessary.

Security Security protocols are used for transmission and storage of sensitive data. Maintenance API is only accessible through token mechanisms. Customer Care Server is accessed over a VPN to ensure security.

Portability Users access the system through a web-based application that enables the portability and a cross-operative system compatibility.

Appendices

A Software and tools used

For the development of this document we used

- LaTeX as document preparation system
- Git & GitHub as version control system
- Draw.io for graphs
- StarUML for sequence diagrams
- Balsamiq Mockups for user interface mockups

B Hours of work

This is the amount of time spent to redact this document:

- Davide Piantella: $\sim$ 48 hours
- Mario Scrocca: $\sim$ 52 hours
- Moreno R. Vendra: $\sim$ 51 hours

C Changelog

- **v1.0** December 11, 2016
- **v1.1** January 3, 2017
  - Add notes to the *User interface design* section clarifying reason of mockups design in relation with architectural design choices presented in the document
  - Add diagram and comments to the Car Communication Interface and the non represented interactions in the relative sequence diagrams.
  - Add an attribute in the DB to better specify how our system connects with cars (Car Server Interfaces)
- **v1.2** January 8, 2017
  - Add notes to clearify some sequence diagrams
  - Fix Figure 6 attributes names about Failures
  - Add notes to *money saving option* mockup
- **v1.3** January 15, 2017
  - Update Map running view section, related mockups and GIS API specifications improving description about how the map is generated, referenced and shown to users

References

[1] Bass, Clements, Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering, Addison-Wesley, 2003

[2] D. Piantella, M. Scrocca, M.R. Vendra, *PowerEnJoy: Requirements Analysis and Specification Document*, Politecnico di Milano - Software Engineering II Project, 2016

[3] GPX open standard, The GPS Exchange Format, GPX website