***PowerEnJoy***



**Design Document**

Authors:

Emanuele Chilà

Giorgio Lazzarinetti

December 10th 2016

SUMMARY

[1. INTRODUCTION 3](#_Toc469067693)

[1.1. PURPOSE 3](#_Toc469067694)

[1.2. SCOPE 3](#_Toc469067695)

[1.3. DEFINITION, ACRONYSM, ABBREVIATIONS 3](#_Toc469067696)

[1.4. REFERENCE DOCUMENT 3](#_Toc469067697)

[1.5. DOCUMENT STRUCTURE 4](#_Toc469067698)

[2. ARCHITECTURAL DESIGN 4](#_Toc469067699)

[2.1. OVERVIEW 4](#_Toc469067700)

[2.2. COMPONENT VIEW 5](#_Toc469067701)

[2.2.1. HIGH LEVEL COMPONENT 5](#_Toc469067702)

[2.2.2. SERVER: OBJECT WRAPPING DATA STRUCTURE 7](#_Toc469067703)

[2.3. DEPLOYMENT VIEW 8](#_Toc469067704)

[2.4. RUNTIME VIEW 9](#_Toc469067705)

[2.5. COMPONENT INTERFACES 15](#_Toc469067706)

[2.6. SELECTED ARCHITECTURAL STYLE AND PATTERNS 16](#_Toc469067707)

[2.6.1. OVERALL ARCHITECTURE 16](#_Toc469067708)

[2.6.2. DESIGN PATTERNS 16](#_Toc469067709)

[2.7. OTHER DESIGN DECISION 16](#_Toc469067710)

[3. ALGORITHM DESIGN DECISION 17](#_Toc469067711)

[3.1. TRIP COST CALCULATION 17](#_Toc469067712)

[3.2. AVAILABLE CAR SEARCH 18](#_Toc469067713)

[4. USER INTERFACE DESIGN 19](#_Toc469067714)

[5. REQUIREMENTS TRACEABILITY 22](#_Toc469067715)

[6. EFFORT SPENT 24](#_Toc469067716)

[7. REFERENCES 24](#_Toc469067717)

## INTRODUCTION

### PURPOSE

The purpose of this document is to give more technical details about PowerEnJoy system.

We want to establish all the subsystem that make up the PowerEnJoy software architecture and their purposes/behaviors.

### SCOPE

The software aims to provide a reliable and safe car sharing service in which the customer can either use a mobile application or a web page (accessible via browser) to search for available car given their position or a selected one, and then rent a selected electric car.

The system includes extra services and functionalities in order to allow the call center workers to handle extraordinary situation that may occur to the customers (car accident, battery low ecc...).

### DEFINITION, ACRONYSM, ABBREVIATIONS

* **RASD**: requirement analysis and specification document;
* **DD**: design document;
* **API**: application programming interface, common way to communicate with another system (for example we use this kind of interface to communicate with the google map services);
* **MVC**: Model View Controller, it’s a common design pattern that define the software architecture of a system at a very high level of abstraction;
* **Car**: we mean, of course, an electric car;
* **Reservation**: It’s a booking made by a user and paid to the use of a certain selected car;
* **Safe Area**: We mean the areas where the user can park the rented car;
* **Special Parking Area**: We mean the areas where the user can both park the rented car and recharge it (all the special parking area are safe area, but not vice versa);
* **Extraordinary situation**: We mean extraordinary events that can happen to the users such as car accident, car’s battery running low etc;

### REFERENCE DOCUMENT

* RASD produced before 2.0
* Specification Document: Assignment 1 and 2 (RASD and DD).pdf

### DOCUMENT STRUCTURE

• **Introduction**: this section introduces the design document. It contains a justiﬁcation of its utility and indications on which parts are covered in this document that are not covered by RASD.

• **Architecture Design**:

1. **Overview**: describes the division in tiers of our application;
2. **High level component view**: gives a global view of the components of the application and how they communicate;
3. **Object wrapping data structure**: represents the database’s data structure seen from the server point of view;
4. **Deployment view**: Describes the various physical nodes and the allocation of tasks (from the Process View) to the physical nodes;
5. **Runtime view**: sequence diagrams are represented in this section to show the course of the different tasks of the application and the way components interact to each other;
6. **Component interfaces**: presents the interfaces between components. As we will see, the methods considered in these interfaces are the same used in the sequence diagram in order to let the DD be more consistent;
7. **Selected architectural style and patterns**: explains the architectural choices taken
8. **Other design decision:** contains all the other decisions which are not included in the previous paragraph;

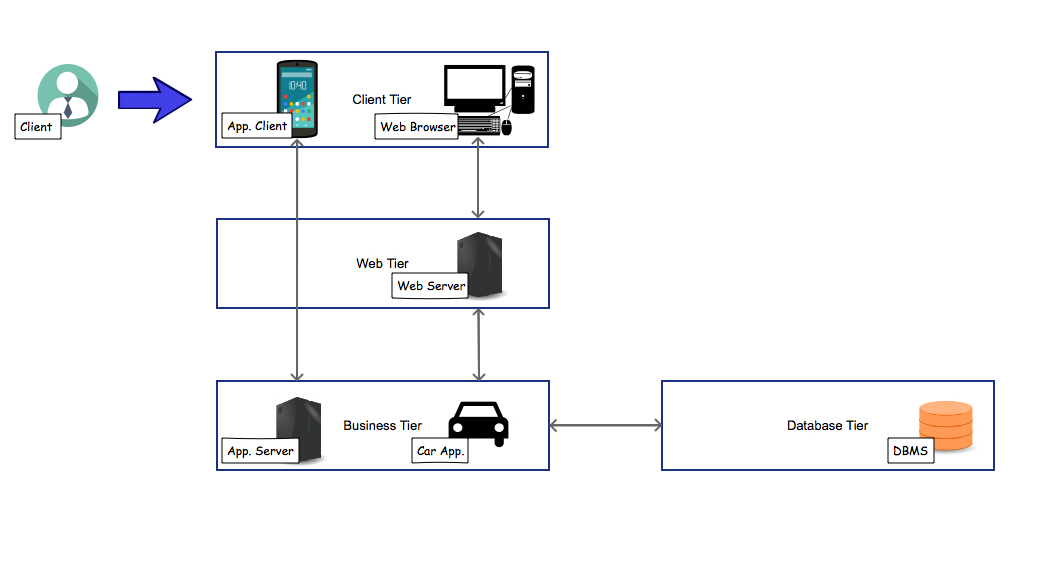
• **Algorithms Design**: this section describes the most critical parts via some algorithms. Pseudo code is used in order to hide unnecessary implementation details in order to focus on the most important parts.

• **User Interface Design**: this section presents mockups and user experience explained via UX diagrams.

• **Requirements Traceability**: this section aims to explain how the decisions taken in the RASD are linked to design elements.

## ARCHITECTURAL DESIGN

### OVERVIEW



PowerEnJoy will be developed as a Web application, based on a four-tier architecture. The tiers involved are:

* **Client tier**: includes the Application Clients and the Web Browser which interact directly with the users and so they cover the presentation layer of our system. The application Client interacts directly with the Business tier, while the Web browser interact with the Web tier;
* **Web tier**: it’s composed by an interface and the user could access it via Web Browser. It elaborates the request given by the Client tier (Web Browser) interacting with the Business tier;
* **Business tier**: contains the business logic of the application and allows all the functionalities of the system include that referring to the car application. It communicates with the Data Base tier in order to access the information and modify them;
* **Database tier**: it’s composed of a relational Database that assures the data persistency.

The Web tier is necessary for the Web Browser in order to let the user access the initial interface which allows him to interacts with the entire system. This simplifies the distribution, deployment and update of the system.

The Application client (which runs directly on the user’s machine) doesn’t need to interact with the Web tier because the Web application contains all the information it needs to communicate directly with the business tier.

The Database tier is based on a relational database structure because the system needs a persistent way to treat critical data related to the user such as payment information, reservation and so on. Using a NoSQL database would have allowed the system to be more flexible in terms of scalability of the database, however it wouldn’t have guaranteed the same level of persistence of data, which is more important than scalability for this kind of data.

### COMPONENT VIEW

#### **2.2.1. HIGH LEVEL COMPONENT**

The high level component diagram is composed of six different elements. There is a central Server that receives the requests of the other components, which can access the server through the provided interfaces. Among the components, a DBMS presents an interface through which the server can access the DBMS.

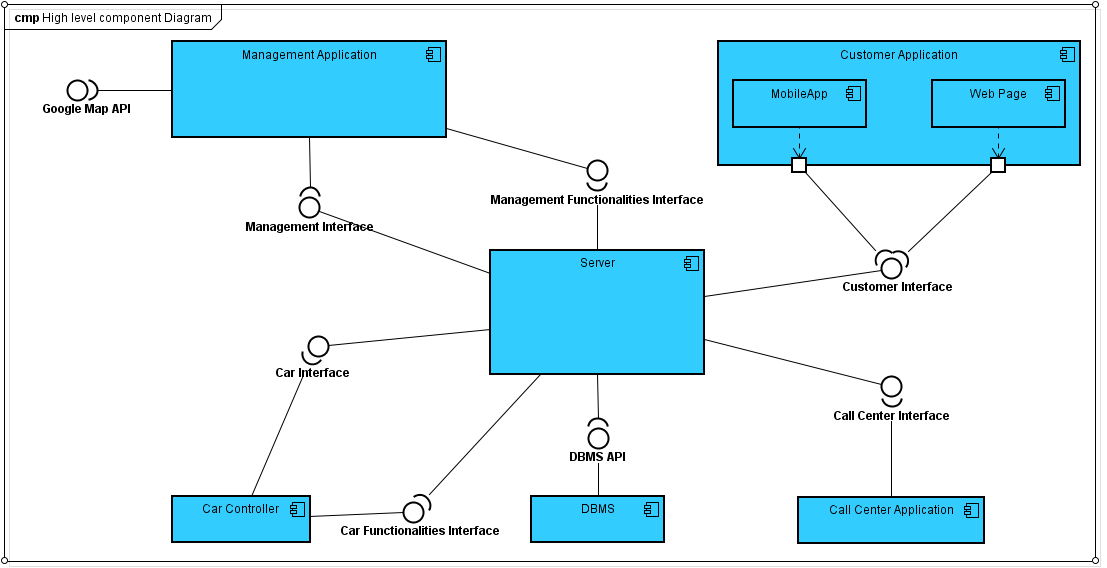
The Customer Application is composed of two components: the Mobile App and the Web Page. The client can communicate with the server both via Mobile App and via Web page, using the customer interface.

The Car Controller can communicate with the server w.r.t. the state of the car (e.g. number of passengers, cost of the trip, position of the car etc) by using the car interface.

The Call Center application, which provides additional management functionalities to the system operators, communicates with the server by using the Call Center interface.

Finally, the Management Application covers all the main management functionalities of the system explained in the RASD document and communicate with the Google map server via the Google Map API.

There are other 2 interfaces in the diagram: the Management Functionalities Interface and the Car Functionalities interface. These interfaces are provided by the Server respectively to the Management Application and to the Car Application in order to let these components make some kinds of requests to the Server. We will see in the next paragraphs how these interfaces are used by each components to interact with the others.

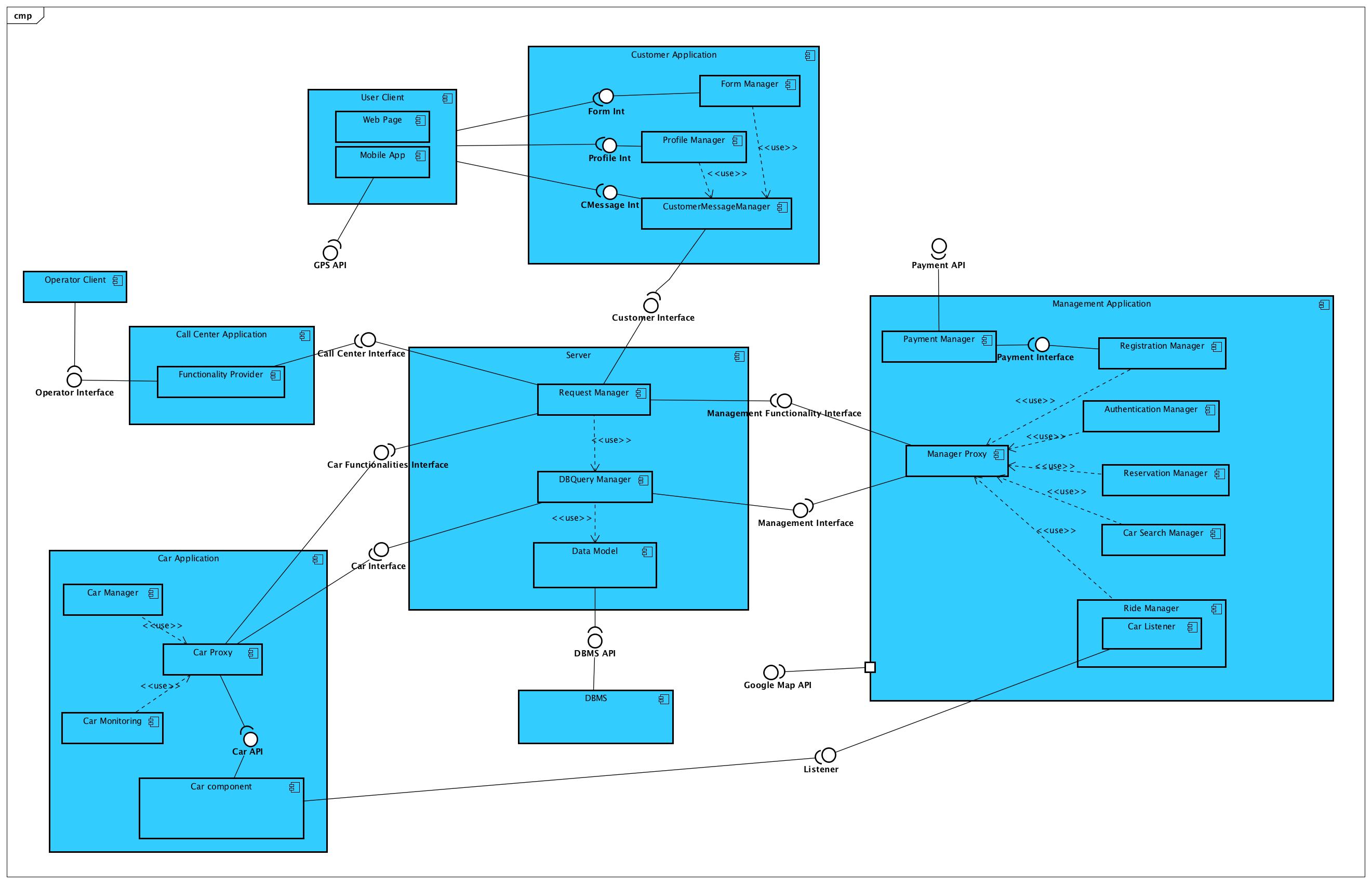


The provided components cover the following functionalities:

* Customer Application:
  + Sign Up functionality
  + Log In functionality
* Management Application
  + Credit Card check of validity
  + Localization functionality
  + Car status management
  + Interact with Google Map services
  + Handle reservation (and expiration time)
  + Calculate the cost of the trip and charge the user
* Car Controller
  + Opening/locking the car
  + Send information acquired by the sensor to the system (including position)
  + Showing the system message information on the car’s terminal
* Call Center Application
  + Manage extraordinary situation
  + Change the car status
  + Charge the user
* DBMS
  + Storage and management of data (users’ personal data, reservation and payments, cars’ status and position and so on)
* Server
  + Route all the request from a component to another one that can fulfil the request made.
  + Communicate with the DBMS to extract the data needed by the other component,
  + Keep the data stored in the object wrapping data structure consistent with the data in the DB

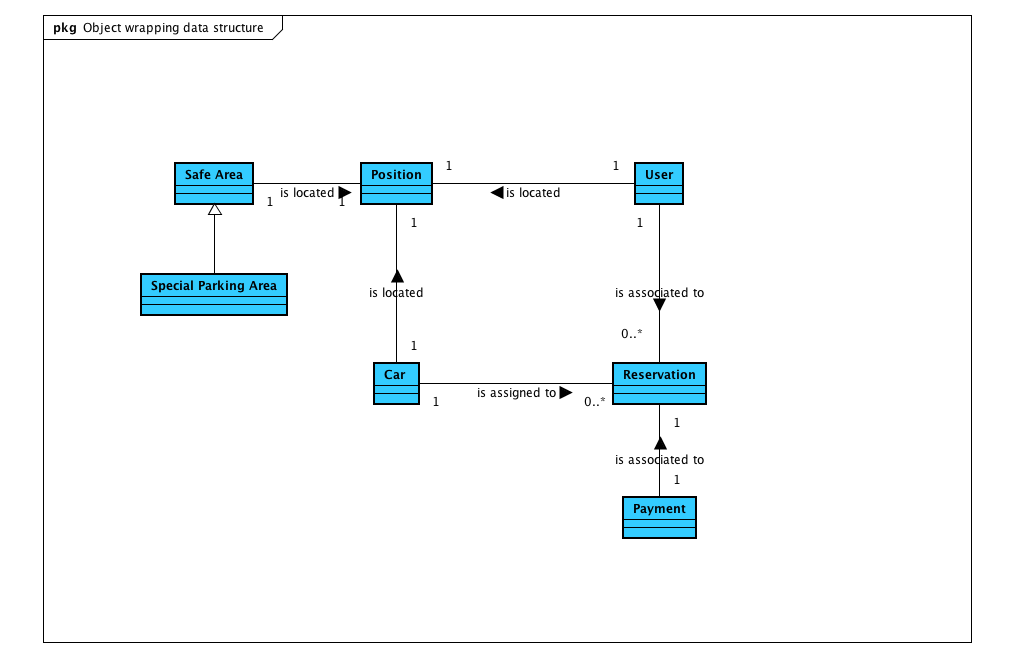
To better understand how the components map all the functionalities described above we provide a more detailed component diagram. In this diagram, each components presented in the previous one is expanded in other components. Let’s see in detail what each components newly presented does.

* Customer Application:
  + Form Manager: is responsible to show to the asking user the various form (such as the signup and login one);
  + Profile Manager: is responsible to show the user page with the various buttons on which the user can click. However, once the user ask for a functionality through this component the execution of this functionality is shifted to the server and processed by other components;
  + Customer Message Manager: is responsible to transfer the requests to the server;
* Management Application:
  + Registration Manager and Payment Manager: they are responsible to manage the functionality of registration, i.e. the Sign-up functionality, checking the payment credentials, in fact the payment manager implement the Payment API;
  + Authentication Manager: manages the Log-in functionality;
  + Reservation Manager: manages the operations for the reservation of a car;
  + Car Search Manager: manages the operations for searching the car;
  + Ride Manager and Car Listener: is responsible to manage the ride as far as Lock/Unlock options and changing car status are concerned;
  + Manager Proxy: is a proxy server that manages the request coming from the Reservation Manager and forward them to the right component; this is useful to simplify the operations;
* Car Application:
  + Car Manager: operates over the car in order to show on the monitor the various information that the user could be interested in (expiration time, car battery, safe are and special parking are position, cost of the trip and so on);
  + Car Monitoring: sends information about the car to the server;
  + Car Proxy: is a proxy server that separates the requests coming from the request manager and the car Manager. This is useful to simplify the operations;
  + Car Component: represent the car and the interface Car API to which the system can operate;
* Server:
  + Request Manager: is responsible to routing the requests to the exact component that has to operate over that;
  + DBQuery Manager: is responsible to query the data structure;
  + Data Model: wraps the data structure into objects efficiently usable by the server;
* Call Center Application
  + Functionality Provider: it shows all the functionality available for the Call Center Operator;



#### **2.2.2. SERVER: OBJECT WRAPPING DATA STRUCTURE**

The diagram presented in this paragraph represents the object wrapping data structure. Instead of producing the data structure of the database, we have thought that it would have been better to show the data structure used by the server in the application. These objects have of course been constructed on the supposed data structure of the relational database. As we can see, the main object are represented by the User, the Car and the Reservation. The User and the Car have their position but in the relational database the only entity with a position is the Car: in fact it would be impossible for the application to keep the position of the user always updated while, as far as the car position is concerned, we will see that the application updates the position just in some relevant cases which are when the car status is changed into “available”. The User must have his position according to the server because his position is used by the application (for example for the functionalities of locking and unlocking the car). His position, however, is taken from the GPS installed on his mobile. Another important aspect to be noticed is the Reservation object. As we can see this is associated both to a user and a car and to a payment. This wraps the exact data structure of the database and is used by the server when a rental instance is activated by the user. The cardinalities of the relations “is associated to” from User and Reservation and “is assigned to” from Car and Reservation are explained as follow: we considered the Reservation with a proper code so that to each Reservation can be associated just 1 User and 1 Car, but a User and a Car can be assigned to different Reservation of course not at the same time ad just if the status and the payment have the correct values as seen in the RASD (par. 6: Alloy Modeling). Finally we can see that the payment is not associated to the User but to the reservation. We supposed that in the database, when a User makes a reservation, this is update in the User tables. When the reservation is then paid the attribute Payment of the Reservation linked to the User is update as “solved”. In the logic of the application this is not relevant because, as we will see, the object Reservation is used by the Management Application for all the operations and the Management Application update the database with the consistent data. These data will then be queried when the user asks to make another reservation. So for the Server is not important that the object Payment is linked directly to the user, but has to be linked to the Reservation in order to correctly storage the data in the database.

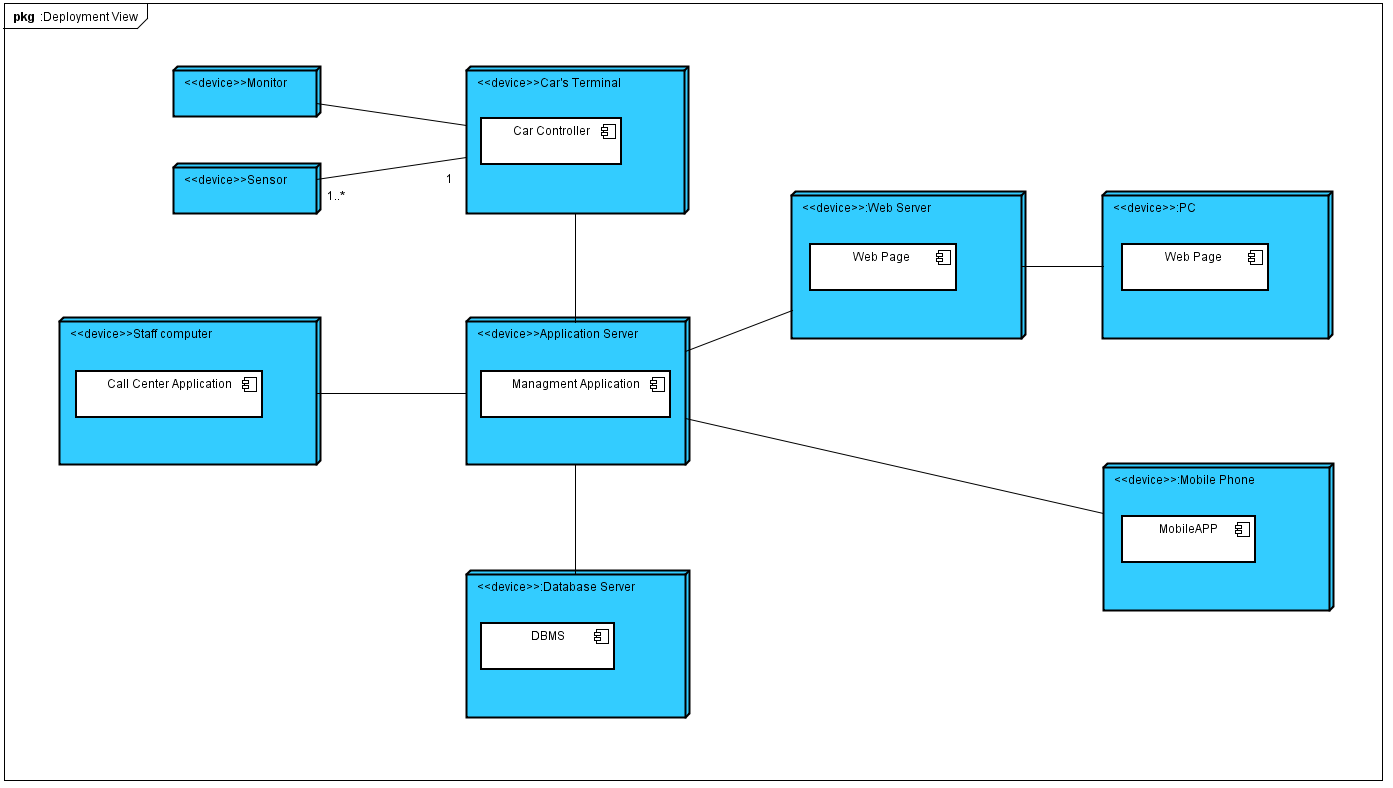


### DEPLOYMENT VIEW

We describe now with the deployment view how we distribute the software component with respect to the hardware one.

Pay particular attention on the following facts:

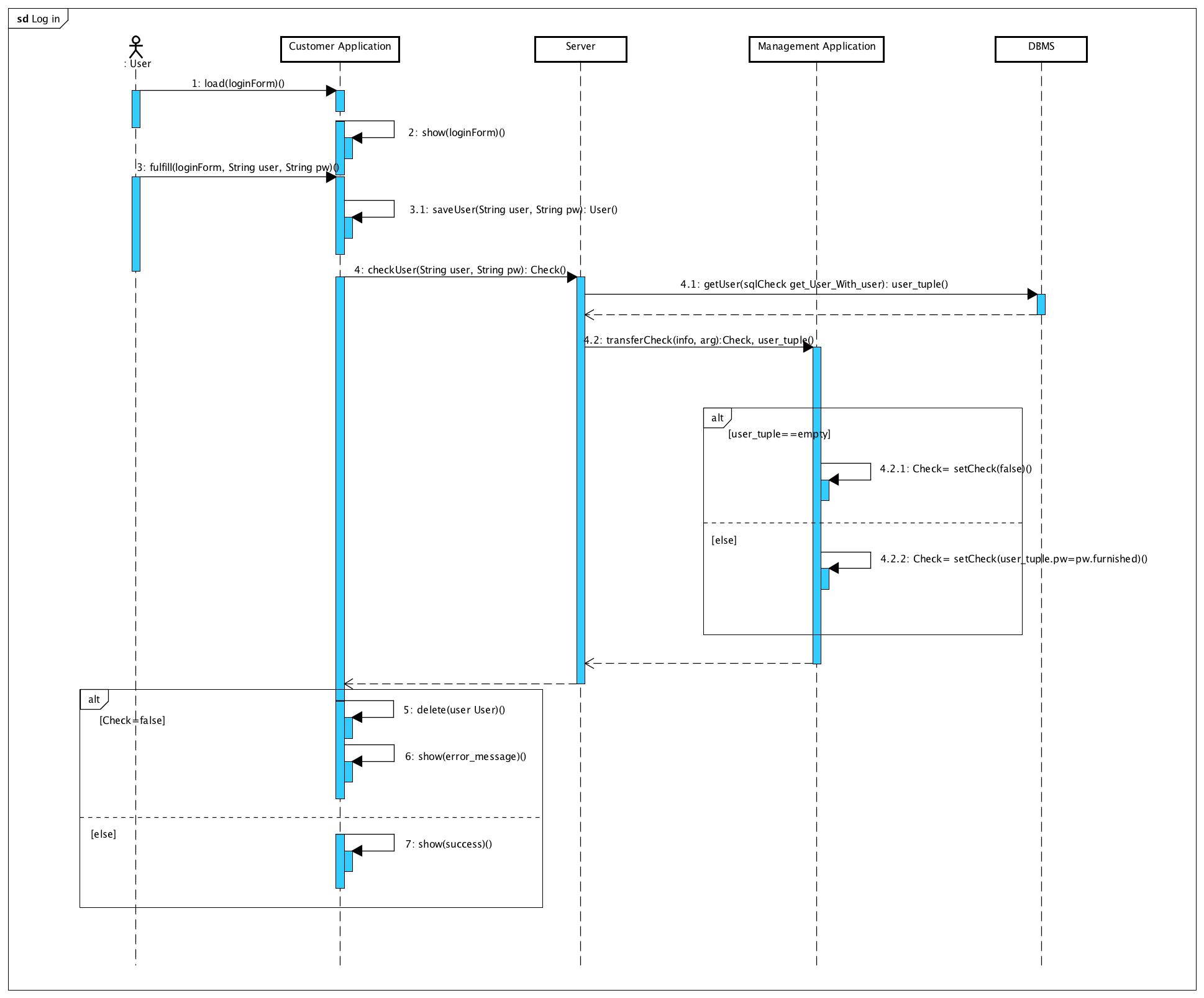
* The devices Web Server and PC run the same software component (Web Page) because when a client want to use the service via his PC, he uses the web page provided by the same Web Server.
* The device Car’s Terminal is a terminal installed on each car owned by the PowerEnJoy company.
* The device Staff computer are the computers with which the workers of the call centers handle the extraordinary situations.



### RUNTIME VIEW

In this section we are going to show how components are connected together using sequence diagram. We show just the main operations to be offered by components in order to understand how components interact.

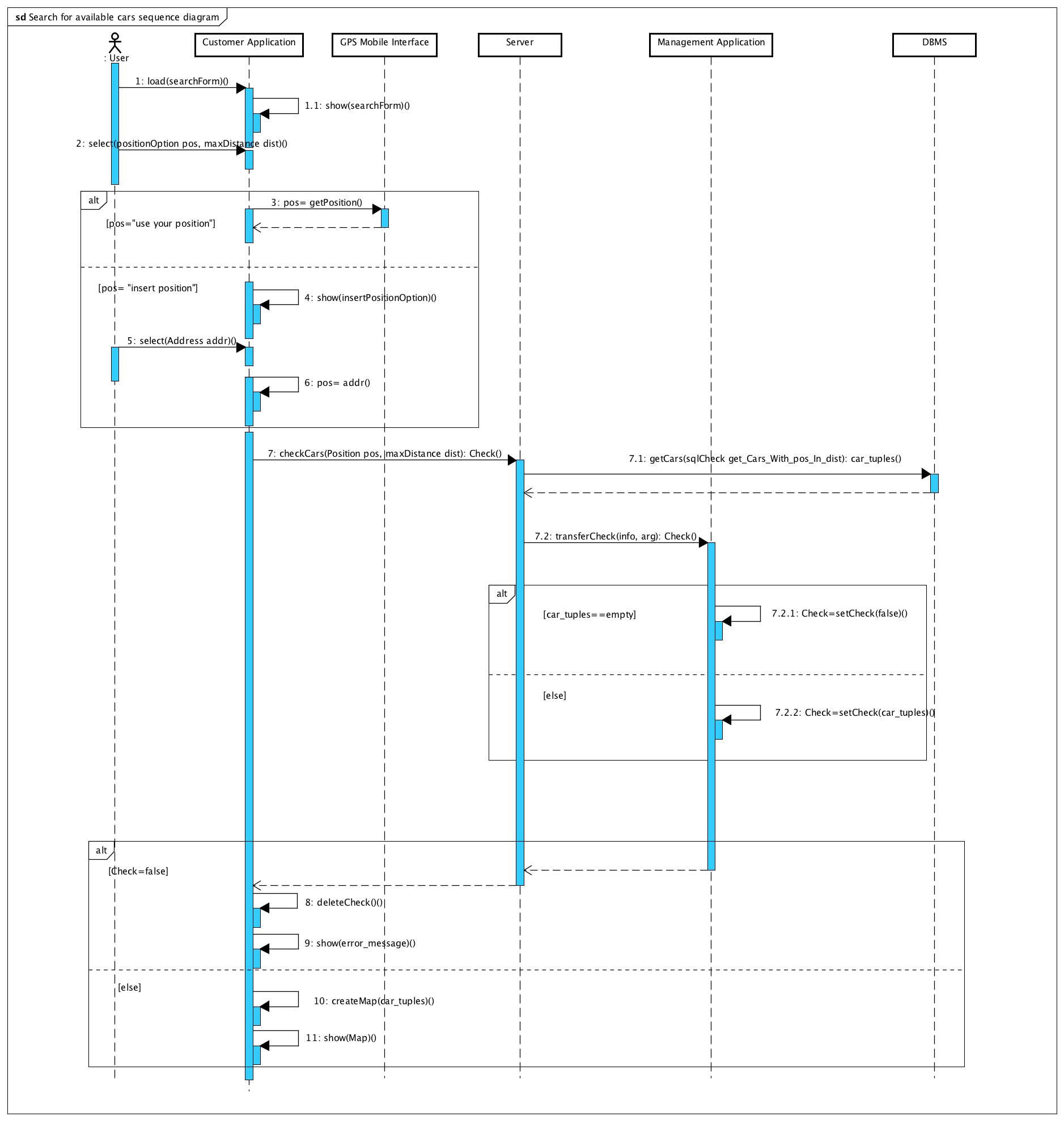
In the first sequence diagram, the Log in sequence diagram, we can see how the user has to input his login information in order to access the system. The user can interact with the other part of the system using the Customer Application. This could be a Mobile App or a Web Page (though it is not specified in the diagram). Once the user has input his login information, these are sent to the Server and the Server routes the request to the Management Application. The Server queries the DBMS and, once it has the information it wanted, it sends them back to the Management Application, and the Server sends them back to the Customer Application. Now the Customer Application can process the information and, in case of success, shows the success of the operation to the User.



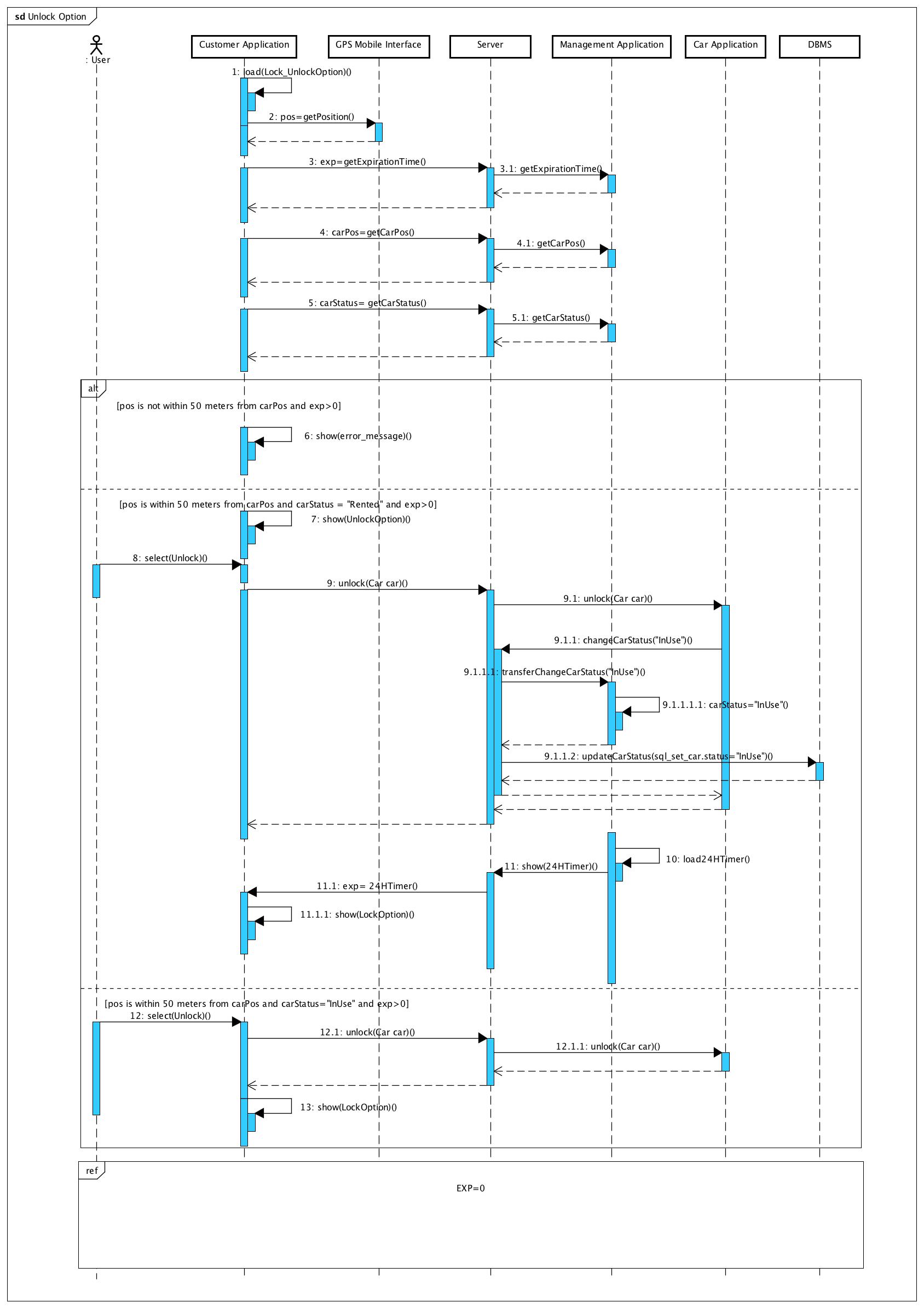
The second sequence diagram shows how the components interact when the user asks the system to search for available cars.

In the first part we can see how the user interacts with the Customer Application in order to insert the necessary information to search for cars. This sequence diagram shows in a more accurate way, w.r.t. the sequence diagrams in section 5.3.3. of the RASD, how the procedure of searching is distributed through components. We can see that once the Customer Application has all the information it needs to activate the search, it sends the information to the Server which after having extracted from the DBMS the car\_tuple i.e. a tables which contains all the car with certain characteristics (position and distance within the position), routes the information to the Management Application that check the validity of these tuples. Then these information, if they exist, are send back to the Customer Application.

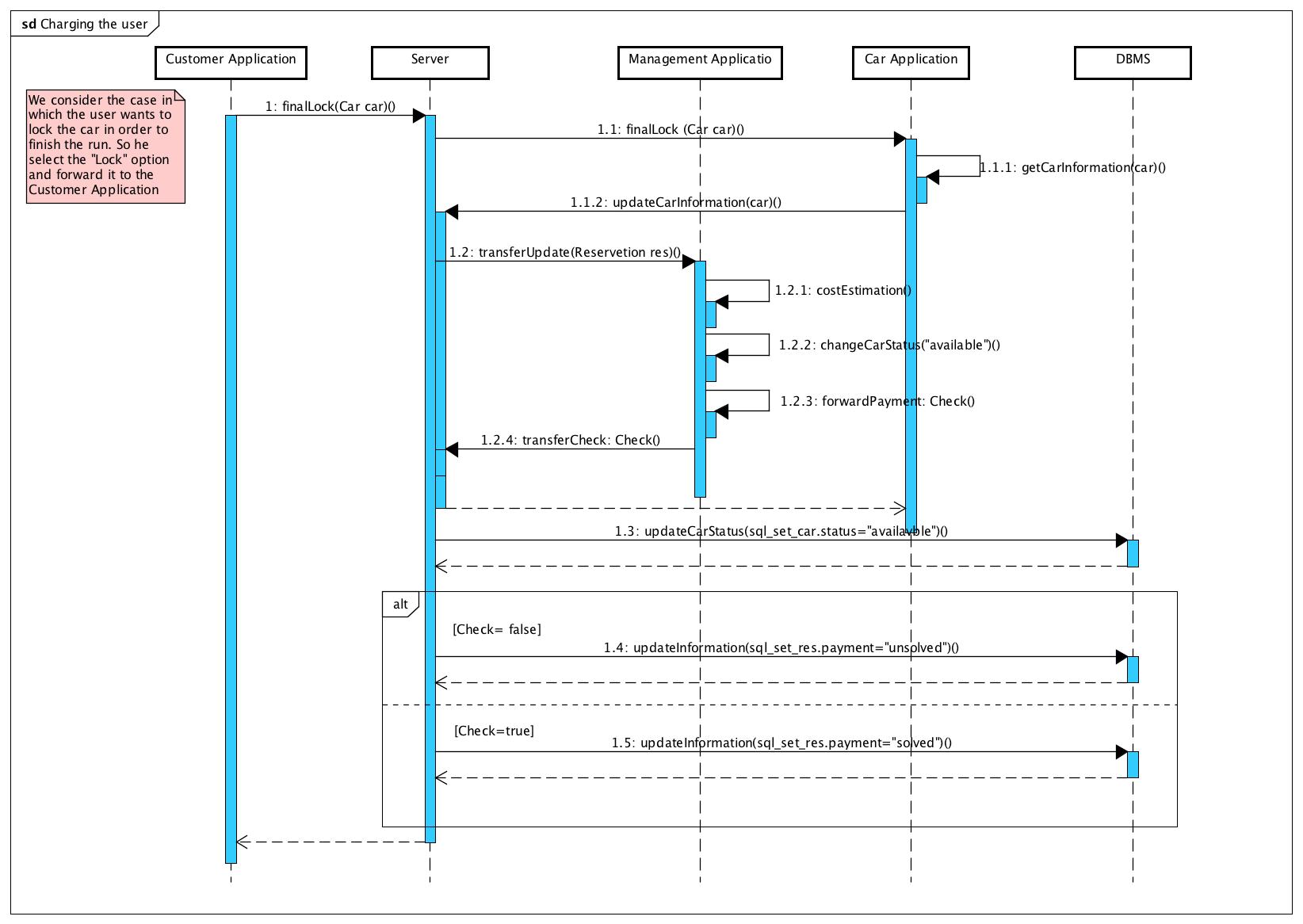
In the second part the Customer Application check the results of the search: if the search has produced no results, it shows an error message, else it create a map with the car\_tuples provided and show it to the user.



Let’s consider now how the rental instance could be created. Once the Map has been created the User can select a car from the map on the Customer Application. At this point, the Customer Application, through the Server, contacts the Management Application in order to modify the database, changing the status of the car from “available” to “rented”. Before doing this, the Management Application check if the user has a rental instance connected to unsolved payment. This is done querying the database through the Server. If it is not so, a rental instance is created from the Management Application and a new reservation linked to the user is added at the database. This rental instance contains all the information about the car position (obtained querying the database), the car status and the expiration time. In the next diagrams we can see how the functionalities of locking and unlocking works. Of course, we firstly have to consider that a rental instance has been activated.

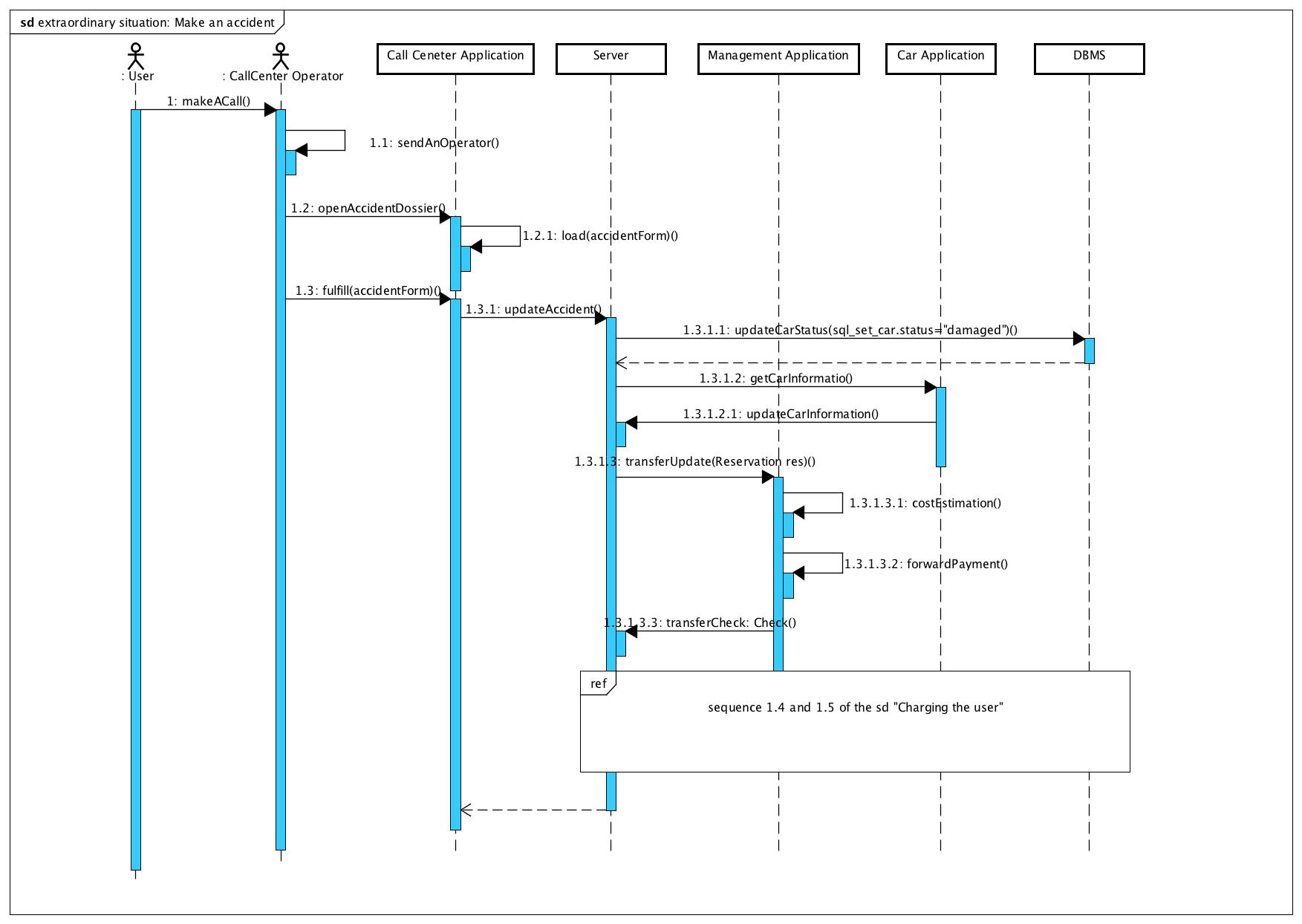


In this diagram we can see how the Unlock works. The Customer Application loads the Lock/Unlock Option. This is linked to the rental instance previously created by the user. In fact the Customer Application loads all the information related to the car rented (expiration timer, position, status) and to the user position. Then there are three case in which the exp is greater than 0. In the first case the Customer Application checks if the user is within 50 meters from the car. If it is not so, the Unlock option is not given to the user. If the user is within 50 meters, the Customer Application checks if the car status is “Rented” or “InUse”. In the first case the user wants to unlock the car for the first time. So the Customer application interacts with the Car Application and the Management Application in order to unlock the car and change the status of the car both in the database and in the rental instance. In the second case, the user wants to unlock the car after having already used that, so the status of the car isn’t to be changed and so the Customer Application interacts just with the Car Application in order to open it. At the end of this diagram there is a reference to the diagram that considers the case in which the expiration time is 0. We are not going to provide this diagram, but it would consider the case in which the exp is the first timer created at the moment of the rental and the case in which the exp is the 24 hours timer created when the car is unlocked for the first time. In both cases the status of the car is changed with the same procedure of the previous sequence diagram and the user is charged of the 1€ fee in the first case, and of the cost of the trip in the second. The procedure of charging is presented in the next sequence diagram.



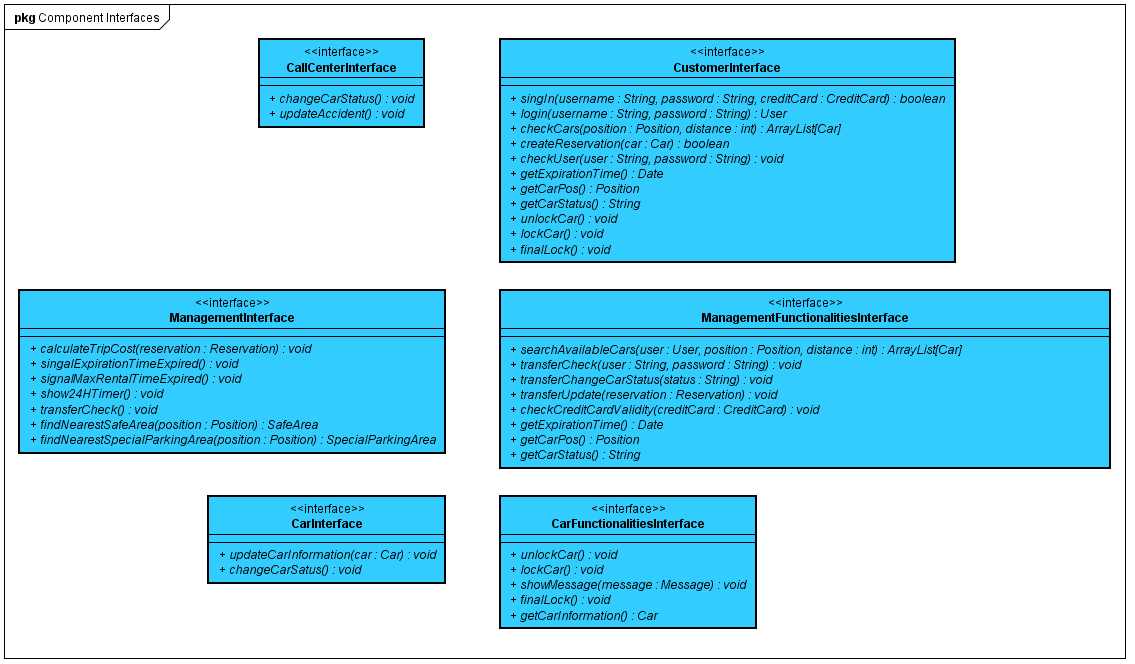
As we can see, the procedure of charging the user is demanded to the method “forwardPayment” of the Management Application (this method is better explained in the sequence diagram 5.3.6. of the RASD). However when the payment is forwarded the Management Application asks the Server to change the information related to the user in the database according to the status of the payment (solved or unsolved). In the database the reservation of the user is saved with the status of the payment so that next time the user wants to reserve a car the Management Application can check if the user is allowed to do this.

Let’s see now, to conclude this paragraph, the last sequence diagram to see how the Call Center interacts with the system. The Call Center Application too communicates with the Server in order to complete all the operation. Then the sequence of operation are similar to those of the previous sequence diagram.



### COMPONENT INTERFACES

We present now the interfaces exposed by each component. The *ManagementFuncitonalitiesInterface* and the *CarFunctionalitiesInterface,* as we can see from the high level component schema, are exposed to the server respectively by the Manager Application and the Car Controller. All the other interfaces are exposed by the Server to the other components. All the methods that have the prefix “transfer” are method that are called by the server to which the server has had a redirection of the requested function to a particular component (for example the WebApp asks the server to check for a password and a username in order to make a login by calling CheckUser(…); the server redirect this request to the Manager Application by calling the transferCheck(…) method). This is a direct consequence of the fact that the component can only communicate with the server, and one of the functionality of the server is routing requests to the correct component.



The Google Map API and DBMS API are not included because they are provided respectively by Google Map and the company in which PowerEnJoy will refer for the database.

### SELECTED ARCHITECTURAL STYLE AND PATTERNS

#### **OVERALL ARCHITECTURE**

Our application is divided in 4 tiers:

1. Database tier for the Data Layer;
2. Business tier for the Business Layer;
3. Client tier and Web tier for the Presentation Layer;

#### **DESIGN PATTERNS**

* **MVC**: Our application design structure is based on a Model-View-Controller pattern
* **Client-Server**: The application is bases on a Client-Server communication model.

We decide to create a thin client for this reasons:

* + To simplify the data synchronization (especially for the management of critical data such as payment or reservation)
  + By moving the computation on the server we reduce the minimum requirement to run our application getting an overall improved availability of the system

### OTHER DESIGN DECISION

We decided that the map services are going to be provided by google map in order to increase the reliability and the scalability of the system and because it allows us to calculate more accurately the costs of this service.

## ALGORITHM DESIGN DECISION

The Classes that we are going to describe in order to define some principal algorithm are not complete and may be more complex, we just define the least number of attributes and methods needed for the correct execution of the algorithm.

### TRIP COST CALCULATION

**public** **class** Reservation {

**private** Date startRentalTime;

**private** Date endRentalTime;

**private** ArrayList<Integer> discount;

**private** ArrayList<Integer> charges;

//Calculate the difference in minute between the 2 date, the function getTime return the milliseconds passed from the 1/1/1970 to the object date

**public** **long** getRentalTimeInMinute(){

**return** ((startRentalTime.getTime()- endRentalTime.getTime())/(1000\*60));

}

**public** ArrayList<Integer> getDiscounts(){

**return** (ArrayList<Integer>) discount.clone();

}

**public** ArrayList<Integer> getCharges(){

**return** (ArrayList<Integer>) charges.clone();

}

}

**public long CalculateTripCost(Reservation reservation)**{

**long** tripCost=reservation.getRentalTimeInMinute();

ArrayList<Integer> discounts=reservation.getDiscounts();

ArrayList<Integer> charges=reservation.getCharges();

//Since the ArrayList is made of normal integer we don’t need any particular Comparator

discounts.sort(**null**);

charges.sort(**null**);

**for**(**int** i=0;i<discounts.size();i++){

tripCost-=(tripCost\*discounts.get(i));

}

**for**(**int** i=0;i<charges.size();i++){

tripCost+=(tripCost\*discounts.get(i));

}

**return** tripCost;

}

### AVAILABLE CAR SEARCH

//The Position class is meant to be immutable, if it is not we must pay more attention on the method getPosition() of the Car and User class.

**public** **class** Position {

**private** **double** longitude;

**private** **double** latitude;

**public** Position(**double** longitude,**double** latitude){

**this**.longitude=longitude;

**this**.latitude=latitude;

}

**public** **double** getLongitude(){

**return** longitude;

}

**public** **double** getLatitude(){

**return** latitude;

}

}

**public** **class** Car{

**private** String status;

**private** Position position;

**public** String getStatus(){

**return** status;

}

**public** Position getPosition(){

**return** position;

}

}

**public** **class** User {

**private** Position position;

**public** Position getPosition(){

**return** position;

}

}

**public ArrayList<Car> searchAvailableCars(User user,**

**double maxDistance,ArrayList<Car> allCars){**

ArrayList<Car> availableCars=**new** ArrayList<Car>();

**for**(Car car:allCars){ **if**((distance(car.getPosition(),user.getPosition())<maxDistance)

&& (car.getStatus().equals("available")))

{availableCars.add(car);}

}

**return** availableCars;

}

**public double distance(Position p1,Position p2)**{

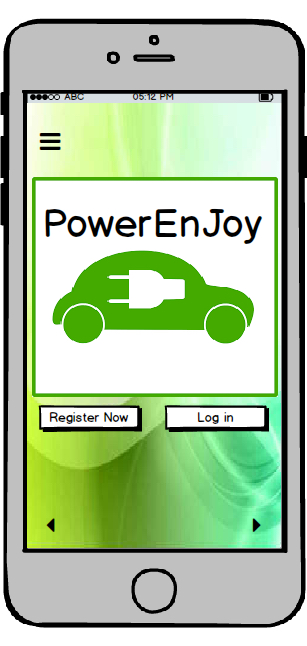
**return** Math.*sqrt*(Math.*pow*((p1.getLatitude()-p2.getLatitude()),2)+Math.*pow*((p1.getLongitude()-p2.getLongitude()),2));

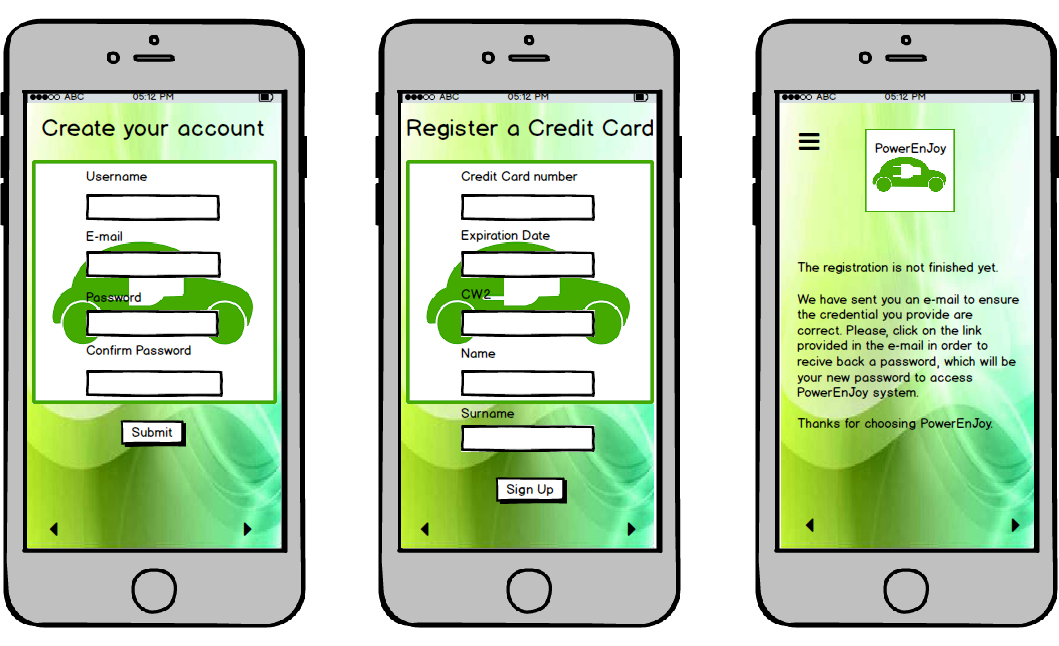
}

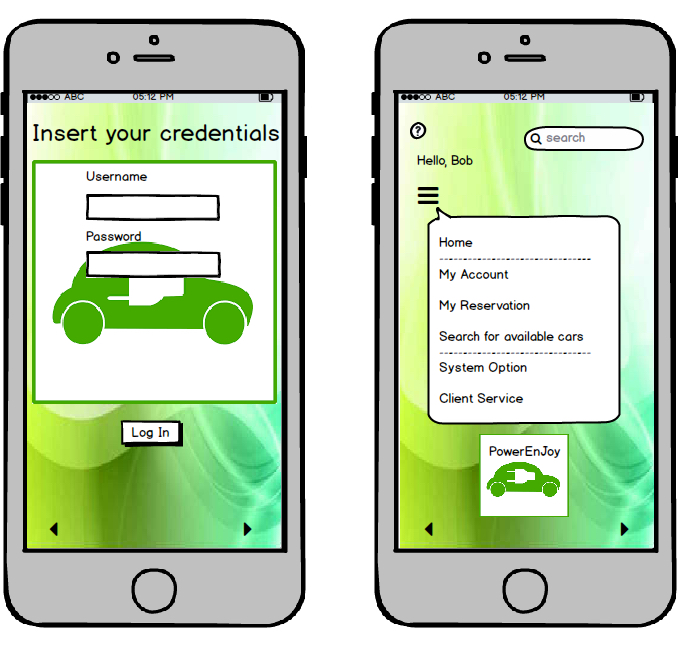
## USER INTERFACE DESIGN

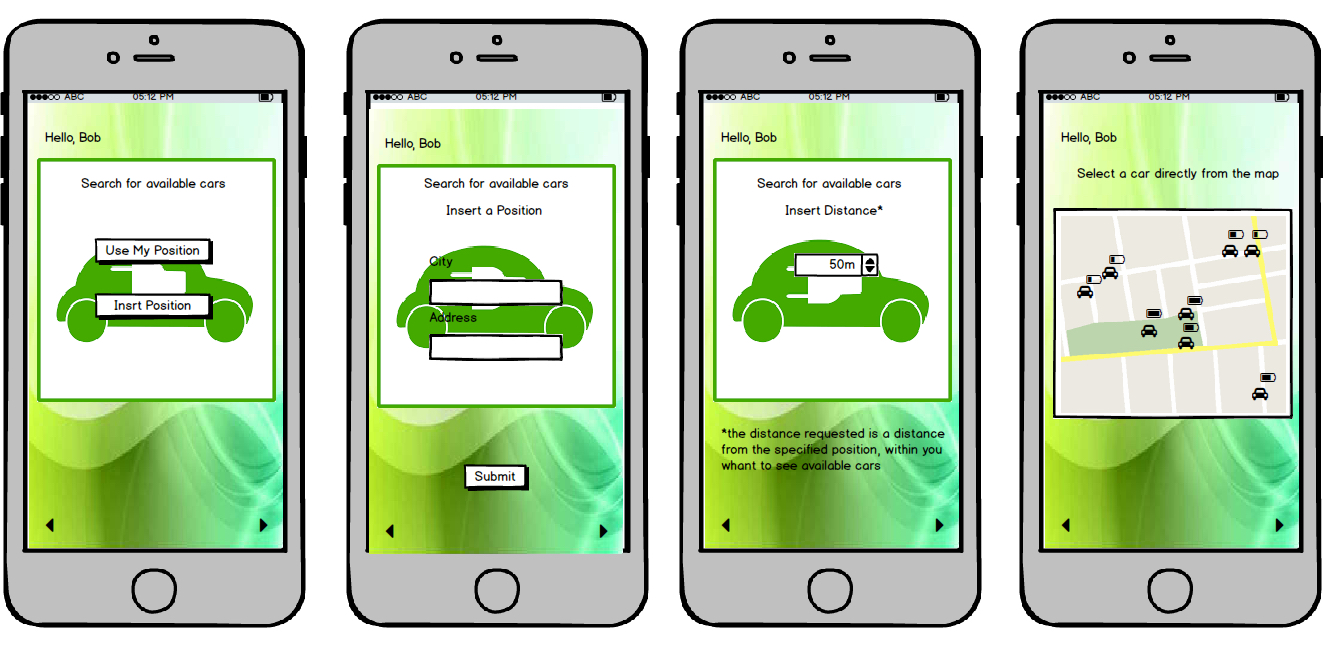
### MOCKUPS

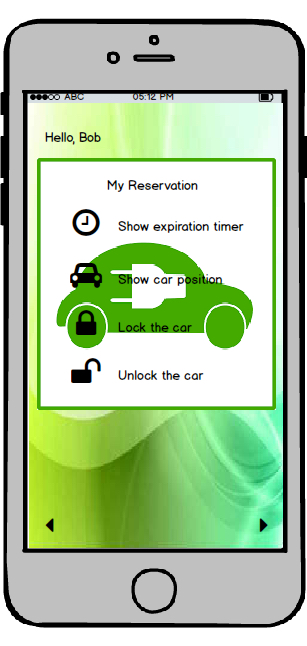
In this section we purpose some interface design. First of all there is the initial interface of the system in which the visitor can register and the user can log in. Then we can see the functionalities of sign up, log in, the home page of the user with a zoom on the icon that contains the user account with all his information, the information about the reservation, the functionalities to search for a car, the home page, the system option and the client service, the functionalities of searching a car and the page with the resume of the reservation. To better understand how the user could use these we will present, in the next paragraph, the user experience diagram.











### UX DIAGRAMS

In this paragraph we will describe three main User Experience(UX) diagrams that our system gives to his final users. We used a Class Diagram with different stereotypes:

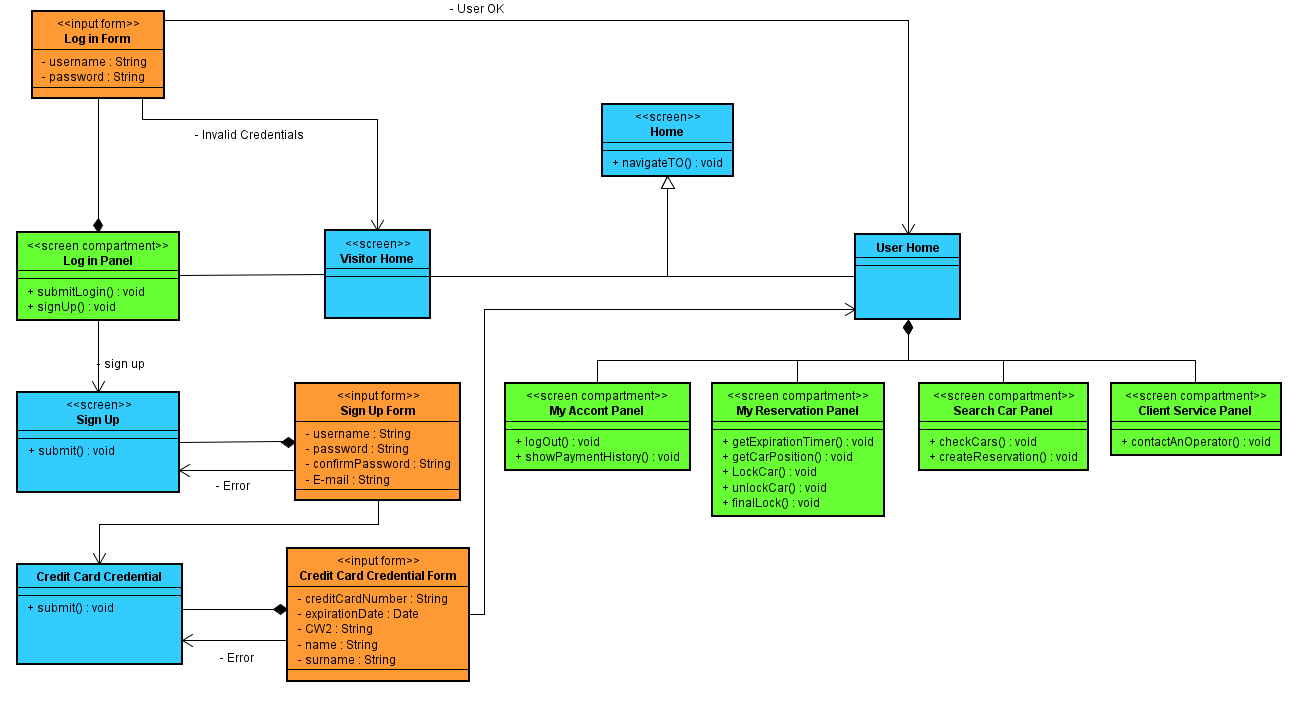
* <<screen>>: in order to represent a “screen” of the Application;
* <<screen compartment>>: in order to represent a component of a page (on the same page can appear more component);
* <<input form>>: in order to represent some fields that a user can fulfill and then send to the system with a button;

This is the Diagram that represents the different home pages in the system.

From a Visitor Home we can log into the system using the Log In Panel and, if there are no error, we will be redirected to our home page. In case of errors, the application will show us again the Visitor Home in order to retry the log in.

In the Visitor Home we will find the Visitor Menu, that allows a generic user to Sign Up into the system by fulfilling a form with his/her data and submit the registration to the system. In the User home we will find different panel with different functionalities:

* **My Account Panel**: allows the user to logout from the system or see the history of his payments;
* **My Reservation Panel**: allows the user to lock/unlock/get the position/lock to conclude the reservation the rented car and see the expiration timer;
* **Search Car Panel**:allows the user to search for available car and rent a selected one;
* **Client Service Panel**: allows the user to contact an operator of the call center;



## REQUIREMENTS TRACEABILITY

**Registration of a person into the system:**

* + - The system has to provide a signup functionality.
* Customer Application;
* Management Application;
  + - The system has to provide a Credit Card check of validity.
* Management Application;

**Log of a person into the system:**

* + - The system has to provide a login functionality.
* Customer Application;
* Management Application;

**Allow Users to search for available cars:**

* + - The system has to provide localization functionalities.
* Customer Application;
* Management Application;
* Car Application;
  + - The system has to provide functionalities to keep track about the status of each car.
* Management Application;
* Car Application;
  + - The system has to provide functionalities in order to interact with the Google Map service.
* Google Map API;

**Allow Users to reserve a single car:**

* + - The system must keep track about the user and the car status.
* Management Application;
* Car Application;
  + - The system must provide functionalities to handle a reservation, in particular the date and the expiration time.
* Customer Application;
* Management Application;
  + - The system has to provide functionalities in order to remotely control a certain car in particular opening/locking the doors.
* Customer Application;
* Management Application;
* Car Application;

**Correctly charge a user that uses the service:**

* + - The system has to provide functionalities in order to remotely acquire information from the car’s sensor so the system can define how much the user has to pay and if the user is or not entitled to discounts.
* Management Application;
* Car Application;
  + - The system has to manage the payments records solved and unsolved and be able to charge the cost of the trip to the user credit card.
* Management Application;

**Make sure that the user behaves well:**

* + - The system has to provide functionalities in order to charge more the user if he doesn’t park the car in a safe zone or doesn’t use the car in the way it is supposed to be used.
* Management Application;
* Car Application;
* Call Center Application;
  + - The system has to manage extraordinary event in particular accident, car maintenance and recovery.
* Management Application;
* Call Center Application;

## EFFORT SPENT

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Time spent [hours]** | |  |
| **Activities** | **Emanuele Chilà** | **Giorgio Lazzarinetti** | **Total** |
| Introductions | 2 | 1 | 3 |
| Architectural Design | 8 | 8 | 16 |
| Algorithm design decision | 3 | 2 | 5 |
| User interface design | 3 | 5 | 8 |
| Requirement Traceability | 1 | 1 | 2 |
| Overall document formatting | 1 | 1 | 2 |
|  | **Total DD** | | **36** |

## REFERENCES

The used tool used to create the DD are:

* + Astah professional: to write UML diagrams;
  + Microsoft Word 2011: to write the document;
  + Balsamiq Mockups 3: to design the interfaces;
  + Pencil: to draw the overview architecture;