POLITECNICO
MILANO 1863

POWER ENJOY

Design Document

Lorenzo Casalino

Tommaso Castagna

Document version 1.0

Contents

1 INTRODUCTION

1.1 Purpose

Purpose of the Software Design Document is the support of development team during the implementation phase, describing what develop and how is expected to work the final result. This goal is achieved through an exhaustive description of the whole software system architecture, focusing on the main aspects that are needed to develop correctly the described system in the Requirements Analysis and Specification Document.

1.2 Scope

The system to be developed aims to support the services offered by the car sharing service through a set of funcionalities. The hereby Design Document present the software system architecture through a set of descriptions that try to cover all the relevant aspects from different points of view.

The main focus is given to the identification of the functional components composing the software system, namely the software units with the task of provide the functionalities individuated in the Requirements Analysis and Specification Document. The discussion is completed with a description of the component's interaction, their deployment and the interfaces that they expose and require.

Enough room is reserved for the description of the design choices and other kind of choices chosen to support the architecture in terms of performances and non-functional requirements, pointing out how they integrate with the architecture and the main reason for their use.

This Design Document takes in account the most relevant algorithms for the software system, showing an overview of their structure, and the design of the user interfaces, extending and completing the description done through the user interfaces mockups in the Requirements Analysis and Specification Document.

1.3 Definitions, acronyms and abbreviations

1.3.1 Definitions

- **Software Architecture**: structure of a software system that is the result of a set of architectural choices.
- **Architectural Choice**: choice made to meet one or more design choices.
- **Design Choice**: choice made to meet one or more design issue. It is selected from a set of design option solving the design issue.

- **Design Issue**: any issue raised by functional requirements and non-functional requirements. Usually, a design issue has one or more design option that solve it.
- **Design Option**: a potential design choice that meet a specific design issue.
- **Layer**: logical level of separation used to spot and define the boundary of responsibilities of the content.
- **Tier**: physical level of separation used to spot the physical unit where the content will be deployed.
- **Demilitarized zone**: physical or logic area of a network where services and resources accessible from outside are placed.
- **Firewall**: device involved into network traffic analysis and network protection.

1.3.2 Acronyms

- **DMZ**: Demilitarized Zone.
- **SDD**: Software Design Document.
- **DD**: Design Document.
- **RASD**: Requirements Analysis and Specifications Document.
- **UX**: User eXperience.

1.3.3 Abbreviations

- **System**: software system.
- **Architecture**: software system architecture.
- **Design Document**: software design document.

1.4 Reference Documents

For the development of this Design Document, the following texts has been used as a base, guidelines and source of inspiration.

- **Project assignment: assigments number 2**
- **PowerEnjoy's Requirements Analysis and Specifications Document**
- **Elastic Load Balancer @ cloudcomputingpatterns.com**

- **Elastic Infrastructure @ cloudcomputingpatterns.com**
- **Elastic Environment @ cloudcomputingpatterns.com**
- **MyTaxiService's SDD sample**

1.5 Document Structure

1. **Introduction**: an introduction to the document and its content is presented, describing the purpose and the scope of the Design Document. Other informations useful for the understaing of the text, such as definitions and acronyms, are shown and explained.
2. **Architectural Design**: a functional description of the system is given, indentifing the main component that satisfies the functional requirements described in the RASD, describing their interactions, the interfaces they require and provides and how they will be deployed. Architectural styles and patterns used to satisfy the performances and non-functional requirements are described briefly, along with other design decisions.
3. **Algorithm Design**: the most relevant algorithms supporting the functionalities of the system are presented and described.
4. **User Interface Design**: a complete description of the design of user interfaces are shown, extending and completing the work did in the RASD.
5. **Requirements Traceability**: a description about how the architectural design choices map and satisfy the requirements, both functionals and non-functionals, identified in the RASD is presented.
6. **Effort Spent**: here the effort spent by each member of the group in term of hours is shown.
7. **References**: this section contains a more detailed description of the main documents used to draw up this Design Document.

2 ARCHITECTURAL DESIGN

2.1 Overview

In the following, a brief overview of the system's architecture is given in order to introduce the reader to the general structure of the system, highlighting the tasks separation.

The logic architecture of the system is composed of **three layer**. Each layer represents a distinct aspect of the system. These distinct aspects are the main tasks for which the system is design for. Each task is accomplished by a set of logically related functionalities, contained into one and only one of the logic layer. The communication between adjacent layers ties the logic architecture of the system.

- **Presentation layer**: where the datas are used, processed and converted into a usable form.
- **Logic layer**: where the whole logic of the system take place.
- **Data layer**: where the functionalities for data management live.

From a physical point of view, the system is layered in **four levels**, or *tiers*. Each tier represents a physical computational node where the system's components are placed, according to their functionalities. Each tier is separated from the other and can communicate between themselves.

- **Client tier**: layer containing the software components usable from the client to access to the system's functionalities and use the data requested.
- **Web server tier**: layer containing the components design for the processing of requests and responses.
- **Application tier**: layer containing the components created to implement and support the business logic.
- **Data tier**: layer containing software components for access and management of data.

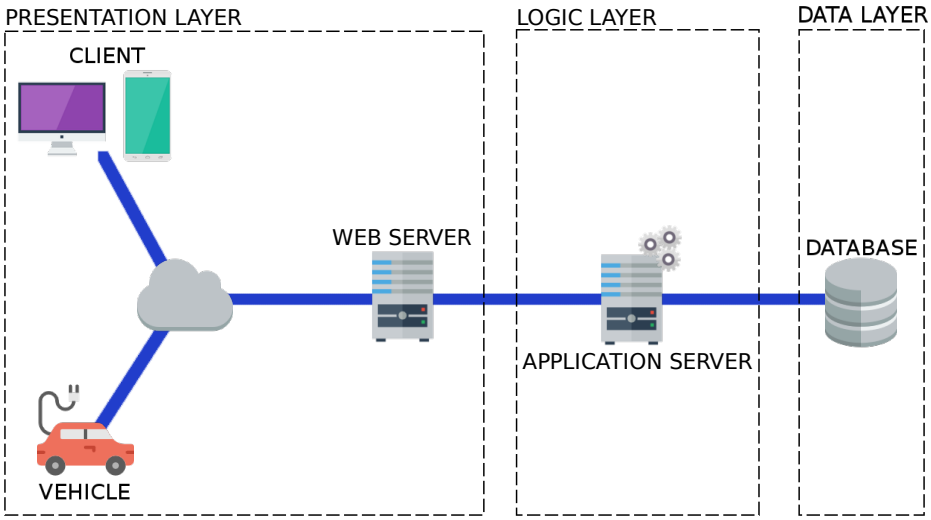

Figure 1: Architecture's division in layers

In order to design and, in the end, develop a highly scalable, flexible, easy to expande and easy to maintain system some design principles has been taken in consideration during the process of identification of the functional components composing the system to be:

- **High Cohesion.**
- **Loose Coupling.**
- **High level of abstraction.**
- **Separation of concerns.**

Since the environment where the system will be deployed is a highly concurrent context, with requests coming from a large and heterogeneous set of devices, and the workload generated is generally unpredictable, some design choices oriented to increase the performance of the system has been considered. Such choices are deeply discussed in *Selected architectural styles and patterns* section, describing the motivation for their choice and how they are used.

Design choices not regarding performances, i.e. System's security, are shown and explained in *Other design decisions*.

2.2 High-level components and their interaction

The main software components forming the system, the interfaces provided and required from the software units, along with the relationship of necessity between components, are shown in the component diagram below [figure 2].

Note that not all the components represented, like the browser, are actually software units of the final system, but are part of the environment with which the system will interact.

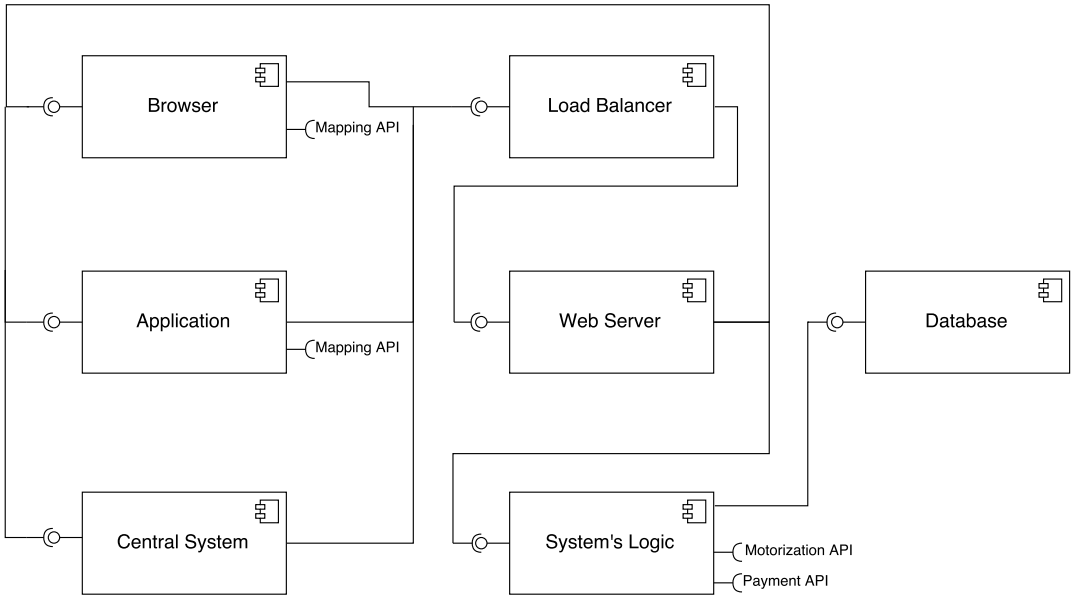

Figure 2: High-level component diagram

- **Browser**: The software used by the user to access to the service.
- **Mobile application**: the application installed on mobile devices and used by the user to access to the service.
- **Central system**: also called Car's Central System. It's the software responsible for car's management and interfacing with the system.
- **Load balancer**: the software used to distribute the workload to the servers.
- **Web server**: the software used to elaborate requests and send back responses.
- **System's logic**: the software responsible for the whole system's logic.
- **Database**: the software unit responsible for the management of queries and data.

2.3 Component view

This section represents the whole system and shows the interactions between each part of it. This section is also split into different subsections to see more in details how the system is composed.

2.3.1 Overall system view

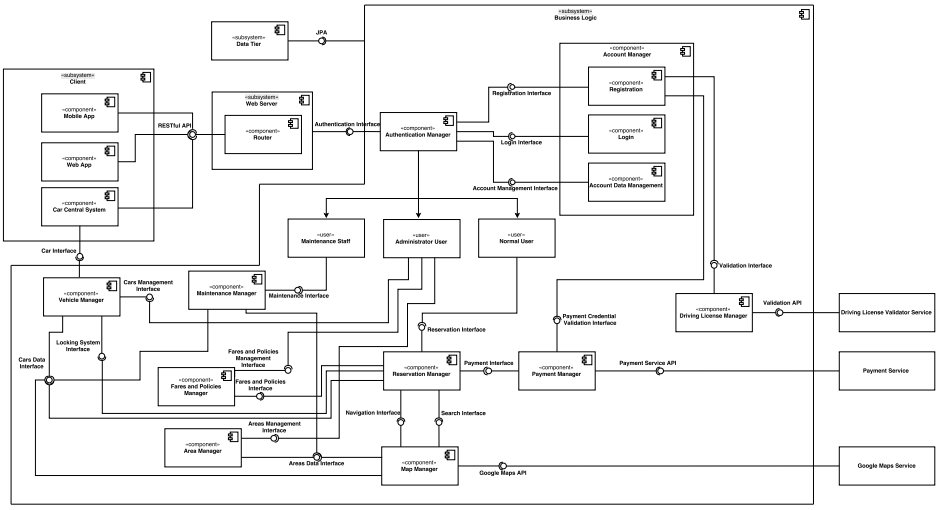

Figure 3: System component view

2.4 Deployment view

2.5 Runtime view

The main runtime interaction between the components of the system are briefly described and shown through the use of sequence diagrams.

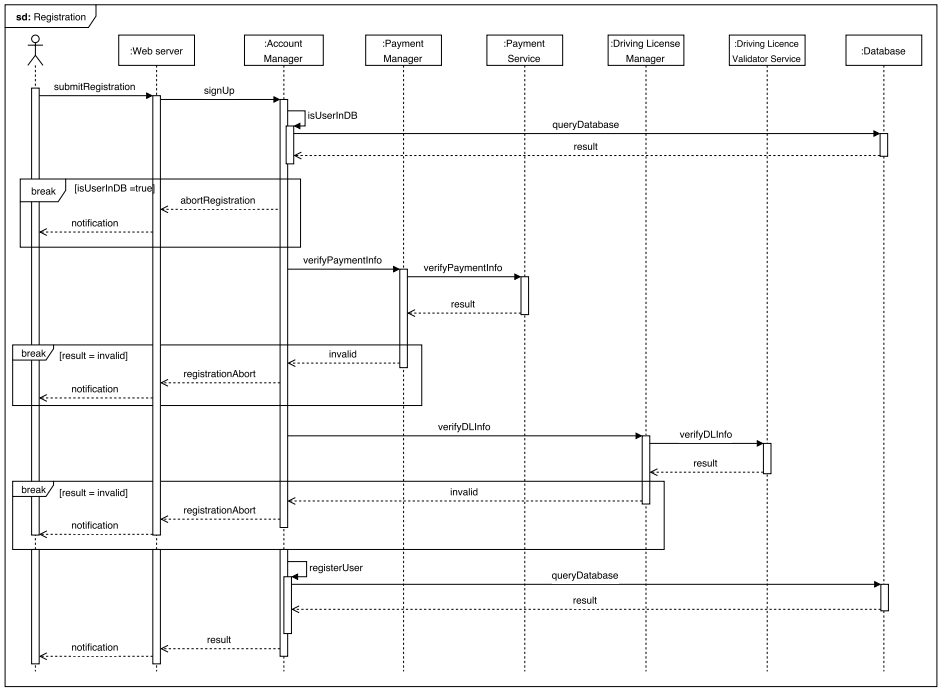

Figure 4: User registration sequence diagram

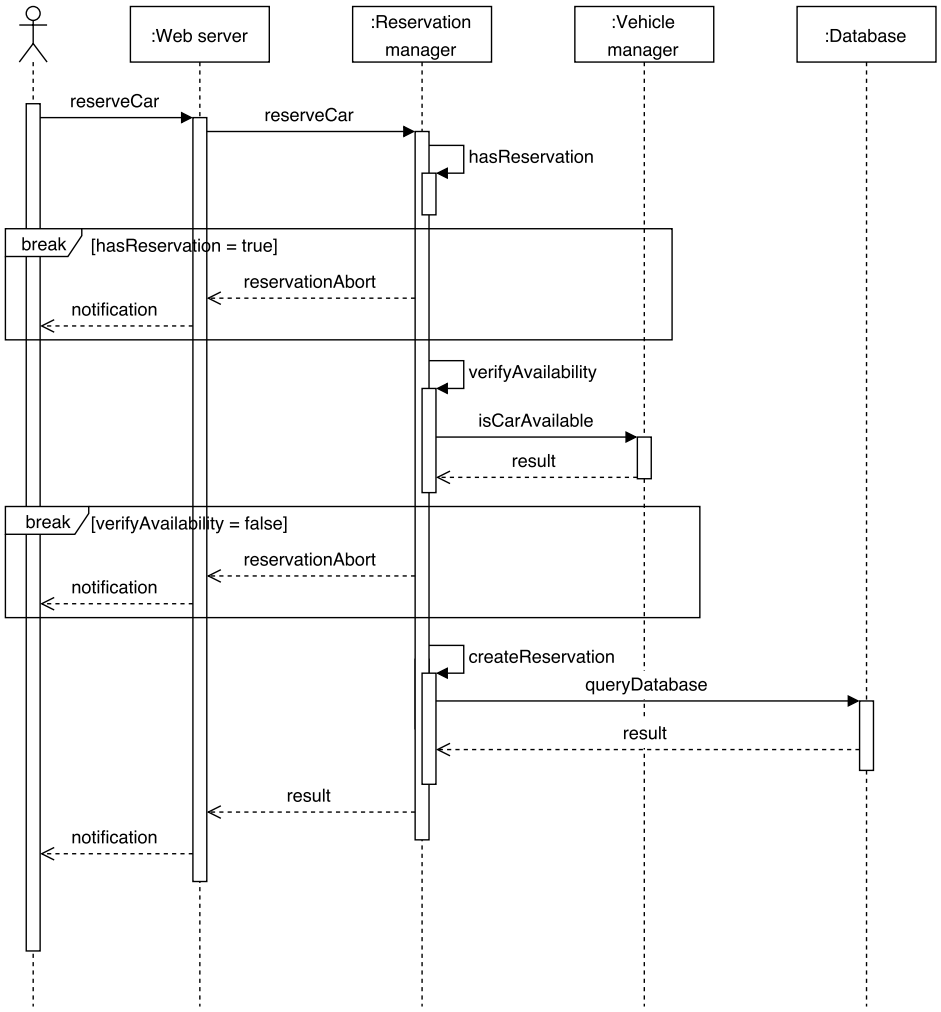

Figure 5: Reservation sequence diagram

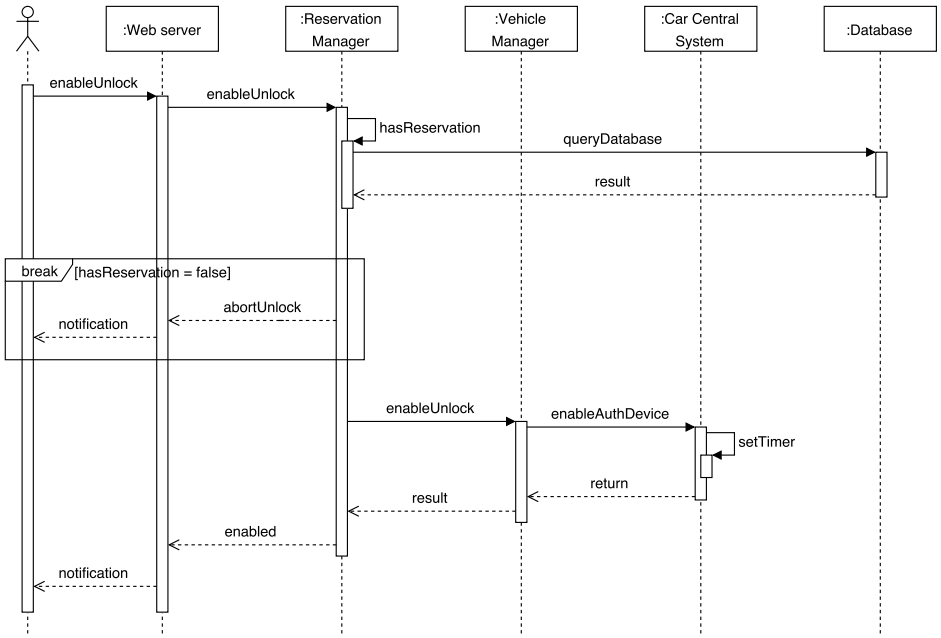

Figure 6: Unlocking pt.1 sequence diagram

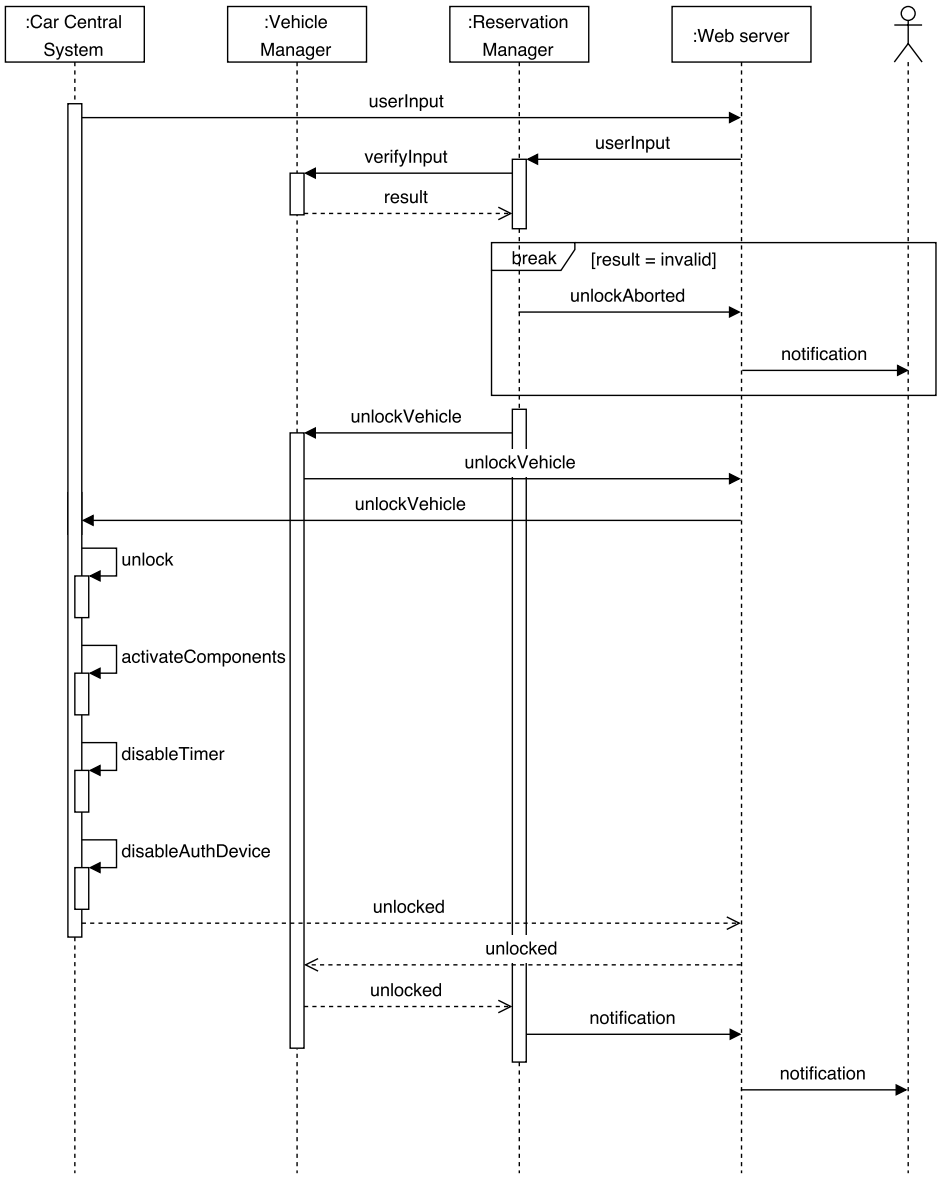

Figure 7: Unlocking pt.2 sequence diagram

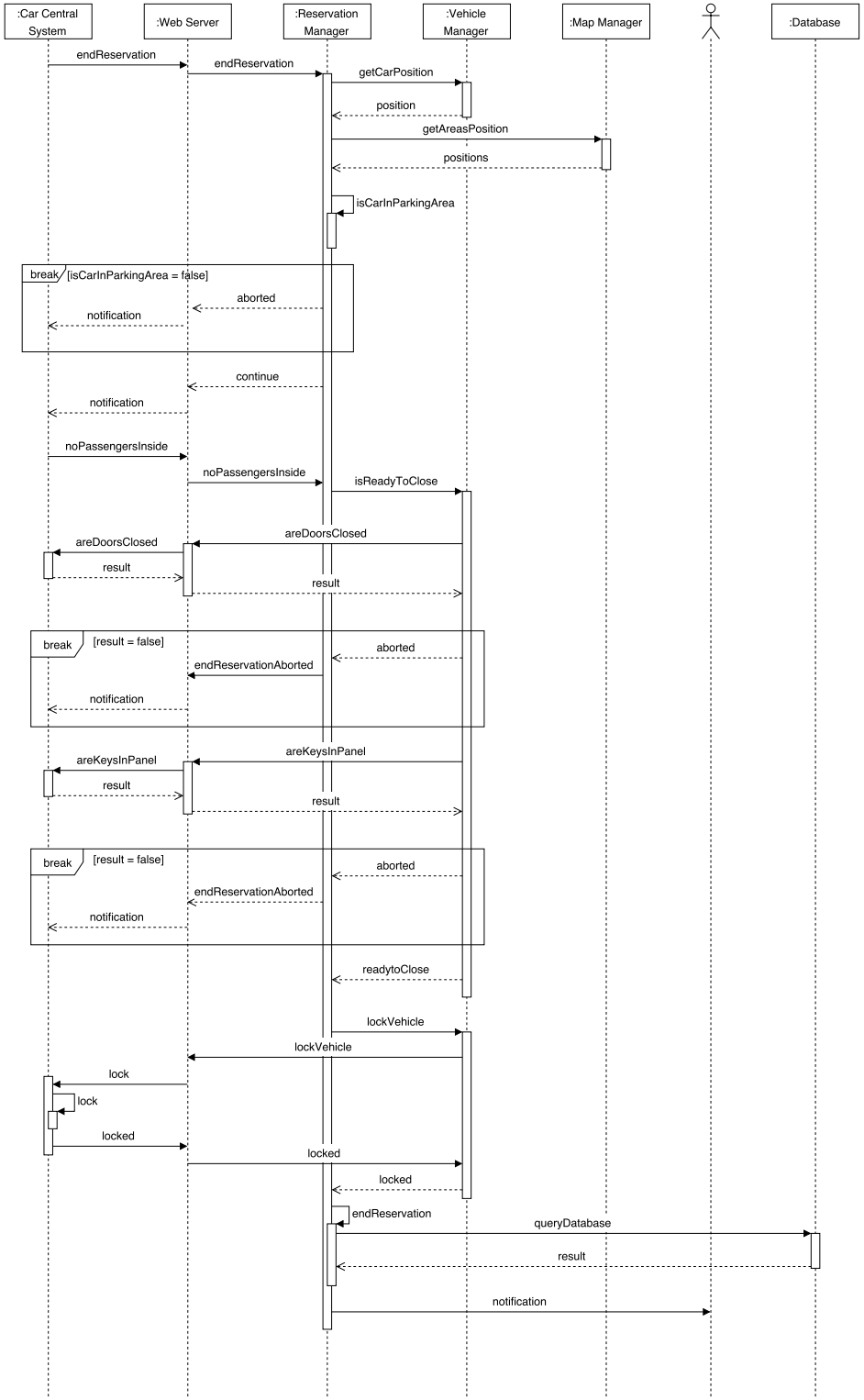

Figure 8: endReservation sequence diagram

User registration

2.6 Component Interfaces

2.7 Selected architectural styles and patterns

In this section the main architectural styles and patterns are listed and described, motivating the reason why they have been chosen and how they integrate into the system architecture.

- **Client-Server architecture**: the main interaction paradigm between customers of the service and the software system is based on a *request-response* model. Furthermore, interaction between Application tier and the Data tier work by means of the same paradigm. To support this kind of interaction the client-server architecture is chosen and integrated with the system trhough the introduction of servers, as shown in figure.
- **Distributed Representation**: in order to keep the client side as light as possible, a client - server paradigm based on a distributed representation is chosen: all the application logic layer and the data layer are place on system side, while the presentation layer, instead, is splitted in two part:
  - **System side**: the presentation layer consist in software for the generation of responses, like web pages.
  - **Client side**: the presentation layer consist in software for the interpretation of system's responses, such as a web browser.
- **4 Tiers**: the client/server architecture is split into 4 distinct tiers, decoupling data, application logic, request/response logic and client logic. As a consequence, the maintenance of the system is dramatically eased through this separation of concerns, introducing an modularity factor in the architecture.
- **Elastic Infrastructure**: the environment where the software system will be deployed is characterized by an unpredictable workload. To face optimaly the continuos and variable change in requests load, an elastic infrastructure is introduced to support the whole system. Through a set of API provided by the elastic infrastructure is possible to get informations about resources utilizations and to perform different tasks, even automatized, reagarding provisioning and decommissioning of IT resources.
- **Elastic Platform**: to get the maximum from each IT resource available, the software environment hosted by each resource, such as an operating system, can be shared among multiple components instances, for example multiple web servers or application components used by different users. As for the elastic infrastructure, the elastic platform exposes a set of API usable for provisioning and decommissioning of component instances.

- **Elastic Load Balancer**: to balance correctly the workload among the available resources, a load balancer is placed between the source of requests and the target. By the number of incoming requests and informations about the resources utilization, the load balancer distribute the work to the resources and can dynamically allocate more or less resources or application instances through the interface provided by the Elastic Infrastructure and the Elastic Platform.

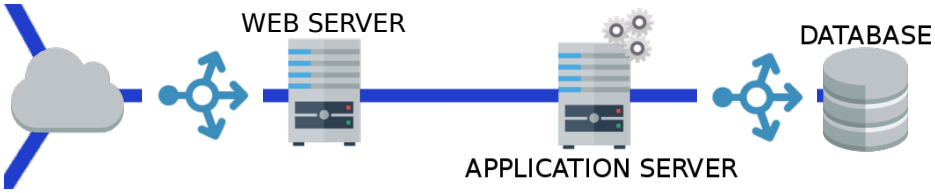

Figure 9: The load balancers' placement

2.8 Other design decisions

Other design decision not taken in account in previous sections are here exposed and discussed.

- **Firewalls**: to protect the system from malicious connections coming from the outside, firewalls are placed at the access points of each connections with the system, analyzing and monitoring the incoming requests and rejecting the invalid or suspicious ones.

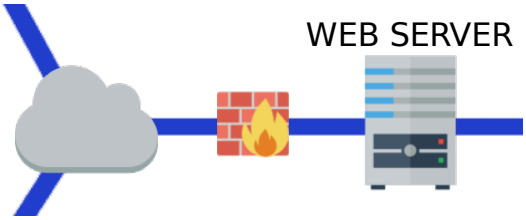

Figure 10: A firewall

- **Demilitarized zone**: since our system is connected to the internet, all the services exposed by the system and used by the system are publicy available and accessible. This is true not only for the web server services but, for instance, even for the database server, which services are used to access and manage data.

  In general, exposing services that directly manage sensitive informations or performe high-risk operations is not a secure choice. One of the most common and reliable architectural choice to increase the security of the

system is the creation of a *Demilitarized Zone*, also abbreviated as *DMZ*. In our architecture, a DMZ is created by means of two firewalls, one placed in front of the web server tier, one in front of data tier. This configuration place implicity into the DMZ the web server tier and the application tier.

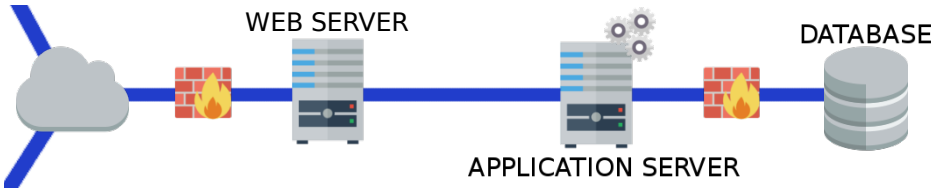

Figure 11: The Demilitarized Zone

- **Stored password protection**: system's breaching and data leakage is a possiblity that can occure even in the most protected environment. To further improve the security of customers' profiles, use of cryptographic hashing algorithms and salting method, are applied to passwords stored into the database.
- **Password security**: to avoid the main typical attack against password, like dictionary attacks and brute-force attacks, the password's characteristics accepted are designed to have minimal lenght of 8 characters, formed alphanumerical and special characters and contains at least one capital letter, one non-capital letter, one digit and one special character.
- **DoS/DDoS protection**:

3 ALGORITHM DESIGN

This sections present some of the main algorithms used in the application. Those algorithms are written in a pseudocode resembling the java syntax.

3.1 Available car search

The following algorithm searches all the available cars near the area specified by the user, if a car is within the visible area of the map, this function will place it on the map in the correct position so that the user can possibly reserve it. Referring to the Google Maps API this function should be added as a listener either to the "bounds_changed" event and to the map generation event, so that the user can drag the map and still see the available cars.

Algorithm 1 Car Search Handling Algorithm

```
1:  function nearbyAvailableCarSearch
2:      bounds = map.getBounds()
3:      availableCars = getAvailableCars()
4:      i = 0
5:      for i < availableCars.size() do
6:          currentLatLng = availableCars[i].getLatLng()
7:          currentId = availableCars[i].getId()
8:          if bounds.contains(currentLatLng) then
9:              map.addCar(
10:             position: currentLatLng,
11:             title: currentId,
12:             details: availableCars[i].getDetails()
13:             )
14:         else if map.contains(currentId) then
15:             map.removeCar(currentId)
16:         end if
17:         i++
18:     end for
19:     return
20: end function
```

3.1.1 Functions and variables description

- **map**: Variable referencing the map object created using the Google Maps API.
- **Map.getBounds()**: Method of the *map* object that returns an object describing the boundaries of the currently visible map.

- **getAvailableCars()**: Is a function that returns a list of car objects containing all the currently available cars.
- **List.size()**: Method that returns the number of elements contained in the list on which it is called.
- **Car.getLatLng()**: Method that returns a variable containing the current latitude and longitude of the car on which it is called.
- **Car.getId()**: Method that returns the identification code of the car on which it is called.
- **Bounds.contains(LatLng)**: This method returns true in the case in which the geographical position described by its argument *LatLng*, is inside the boundaries defined by the *bounds* object on which the method is called.
- **Map.addCar(CarMarker)**: This method adds the *CarMarker* object that receives as a parameter to the map on which it's called, making the car and all its informations visible on the map, and allowing the user to reserve it.
- **CarMarker**: Object containing the all the necessary informations concerning the car.
- **Map.removeCar(CarId)**: Method that removes the car identified by the *CarId* parameter from the map on which it is called.

3.2 Car reservation

The following algorithm takes care of the reservation of a car by a user. In order to avoid consistency issues we need to ensure the atomicity of the reservation so that a car cannot be reserved by more than one user. To do so we decided to use a lock-like pattern.

Algorithm 2 Car Reservation Handling Algorithm

```
1:  function carReservation(carId, user)
2:      reservationLock = null
3:      car = null
4:      try
5:          reservationLock = getReservationLock(carId, user)
6:          car = getCar(carId)
7:          if !car.isAvailable() then
8:              Throw carNotAvailableException
9:          end if
10:         if user.hasActiveReservation() then
11:             Throw multipleReservationException
12:         end if
13:         reservationTimer = Timer(reservation_timer)
14:         createReservation(carId, user, reservationTimer)
15:         ReservationLock.releaseLock()
16:     catch (LockTimeoutException, carNotAvailableException)
17:         return ReservationError
18:     catch (CarNotFoundException, multipleReservationException)
19:         reservationLock.releaseLock()
20:         return ReservationError
21:     end try
22:     return
23: end function
```

3.2.1 Functions and variables description

- **getReservationLock(CarId, User)**: Function that tries to acquire the lock on the reservation part of the Car object and the user object identified by the parameters *CarId* and *User*. If it can't get the lock after a certain amount of time the function throws a *LockTimeoutException*.
- **getCar(CarId)**: Function that returns the Car object identified by the *CarId* parameter. If no car corresponding to the specified ID is found, the function throws a *CarNotFoundException*.
- **Car.isAvailable()**: Method that returns *True* if the car on which it is called is currently available, returns *False* instead.

- **Car.reserveCar()**: Method that set the state of the car on which it is called to *Reserved*. This method is subject to the *ReservationLock* obtainable through the GETCARRESERVATIONLOCK(CARID) function.
- **User.hasActiveReservation()**: Method that returns *True* if the user on which it is called on already has an active reservation, returns *False* instead.
- **ReservationTimer(minutes)**: This timer expires after the amount of minutes defined by the parameter *minutes*, once expired it deletes the reservation it is associated to.
- RESERVATION_TIMER: Variable containing the duration of the *ReservationTimer* expressed in minutes.
- CREATERESERVATION**(CarId, User, ReservationTimer)**: Function that creates the reservation, associating the car, the user and the timer it gets as parameters. Among other things, it is in charge of setting the car as *Reserved*, and starting the *Timer*. The methods called in this function are subject to the *ReservationLock* obtainable through the GETRESERVATIONLOCK*(CarId, User)* function.
- **Lock.releaseLock()**: Function that releases the lock on which it is called.

4 USER INTERFACE DESIGN

4.1 UX diagrams

To provide a complete understanding of how the user interfaces are designed, both for the web application, the mobile application and the on-board software, user experience diagrams are provided in the following.

The user interface for the customers [figure 12] and the system's administrators [figure 13] are different since the functionalities provided them are design according to their relation with the system. This difference is here shown by the corresponding UX diagrams.

It's worth noting that the user interface for the web application and the mobile application share the same design, since the mobile application is only a mobile optimization of the former.

In order to keep the diagrams as clean and simple as possible, the Logout and Homepage navigation links are not displayed. It's assumed that every screen containing the logout() and homepage() functionalities refers, respectively, to the Logout screen and Logged Homepage screen.

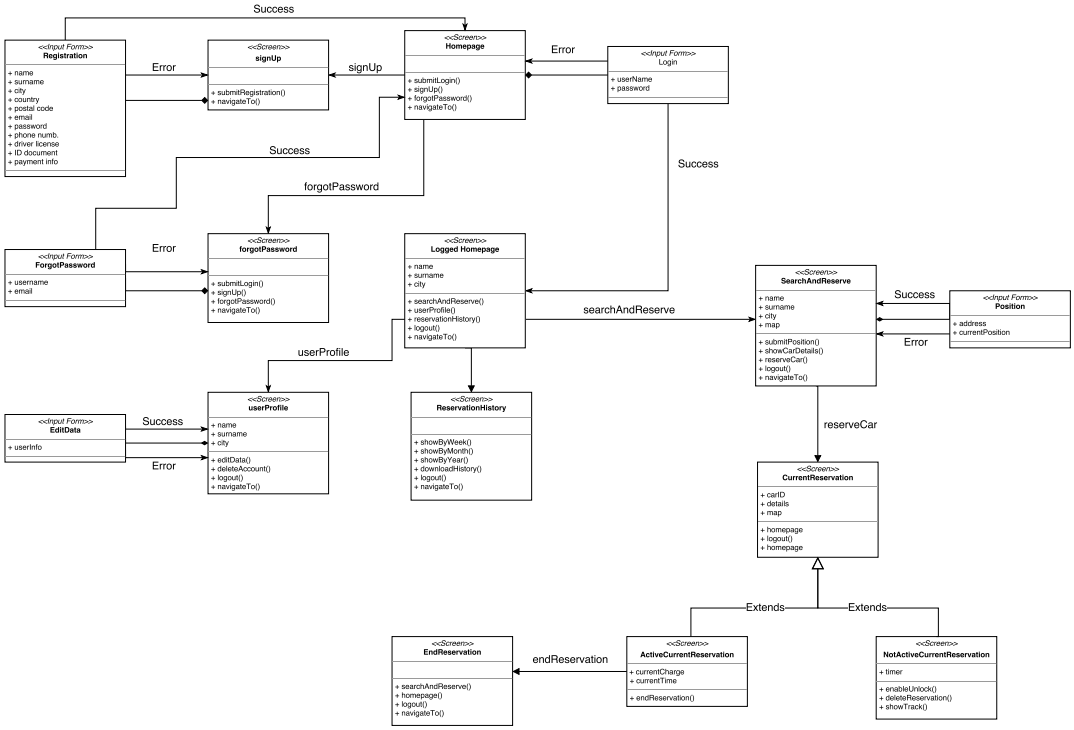

Figure 12: Web application & Mobile application UX

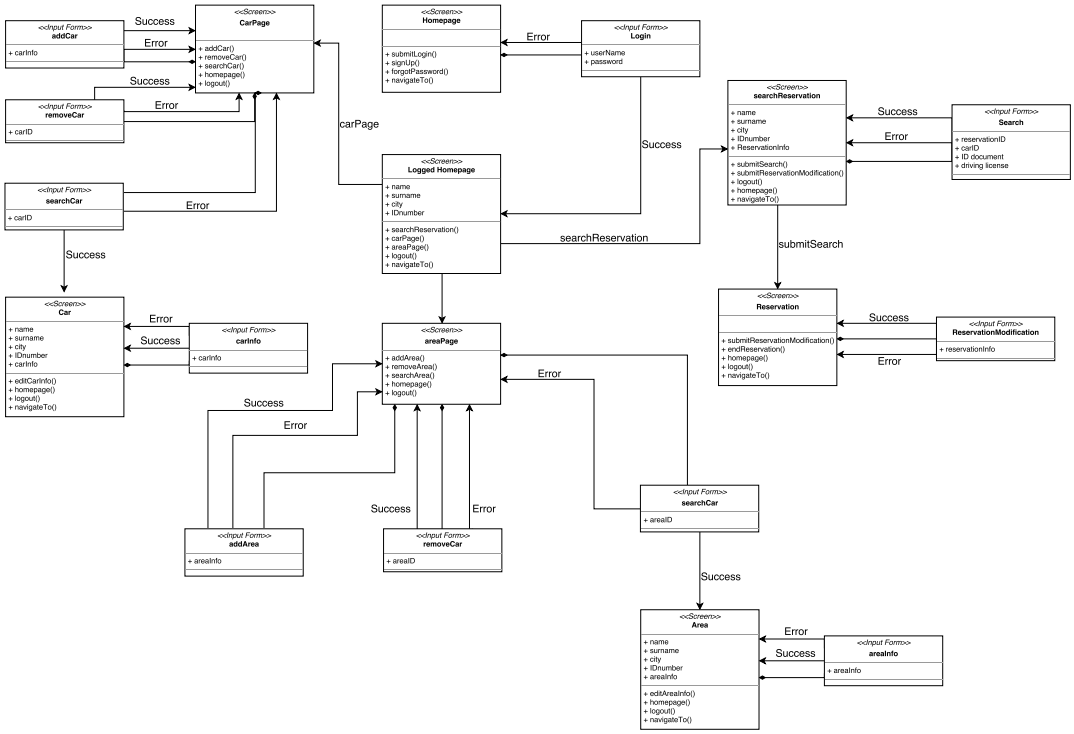

Figure 13: Administrator UX

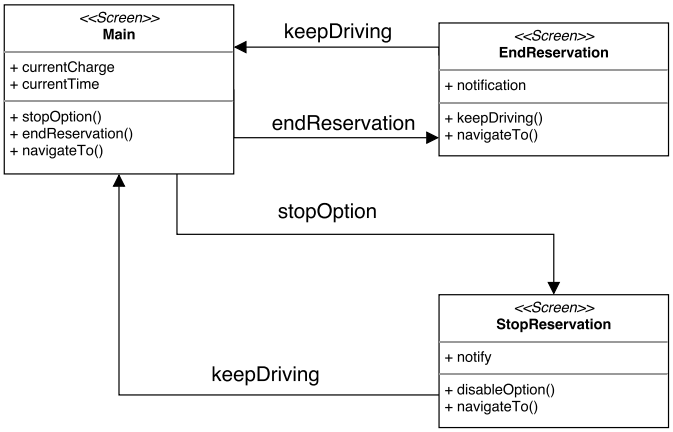

Figure 14: On-board software UX

4.2 Interface mockups

For some examples of graphic user interface of web application and of the on-board software, please refer to *RASD, subsection "User interface"*.

5 REQUIREMENTS TRACEABILITY

6 EFFORT SPENT

Lorenzo Casalino

- 26 November 2016 - 1 hour
- 27 November 2016 - 3 hours and 20m
- 28 November 2016 - 1 hour and 10m
- 29 November 2016 - 1 hour
- 1 December 2016 - 2 hours
- 2 December 2016 - 1 hour
- 3 December 2016 - 2 hours
- 5 December 2016 - 2 hour and 30 minutes
- 7 December 2016 - 4 hours
- 8 December 2016 - 2 hours
- 10 December 2016 - 2 hours
- 11 December 2016 - 2 hours
- 12 December 2016 - 2 hours
- 13 December 2016 - 3 hours
- 15 December 2016 - 4 hours
- 16 December 2016 - 1h 30min
- 17 December 2016 - 1 hour
- 19 December 2016 - 4 hours
- 20 December 2016 - 2 hours

7 REFERENCES