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.

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.

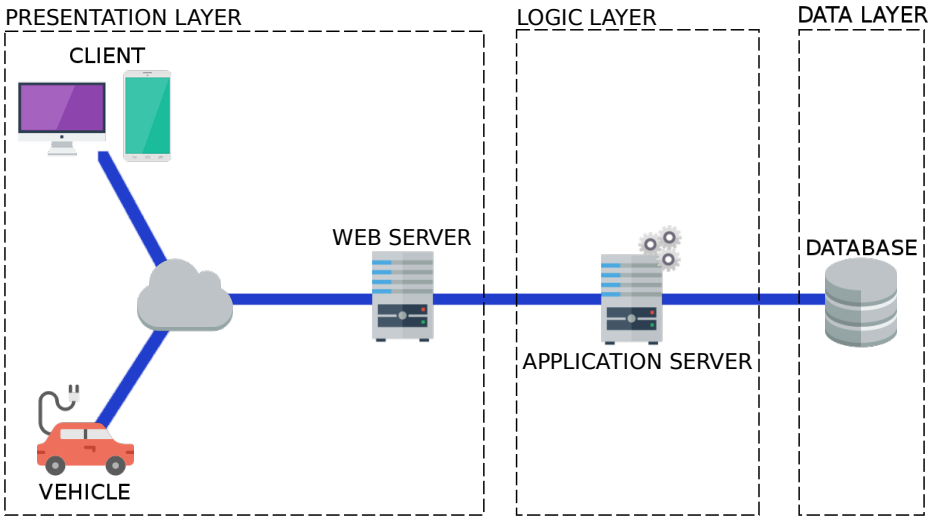

Figure 1: Architecture's division in layers

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].

For a more detailed description of each component and of each interfaces, required and provided, please refer to the corresponding sections.

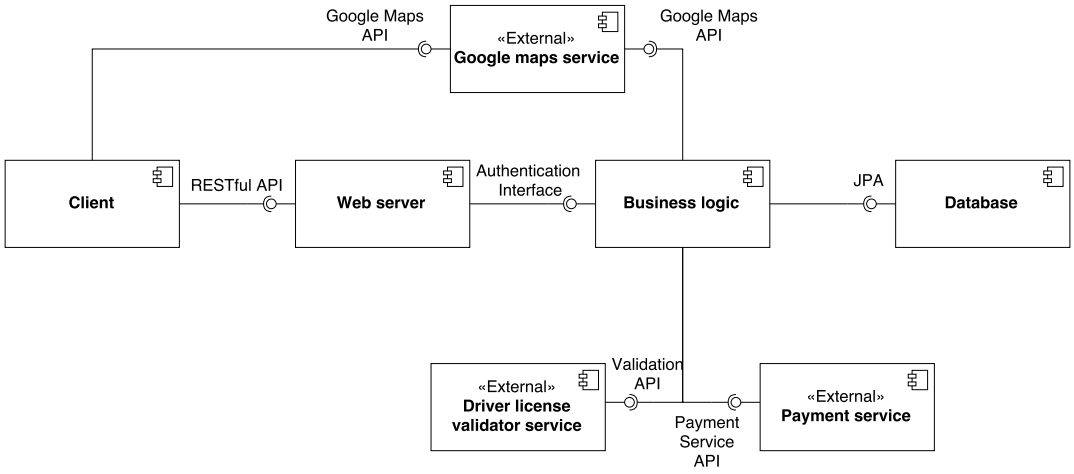

Figure 2: High-level component diagram

- **Client**: The software that a user can use to access to the services provided by the software system.
- **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, storage and data access.
- **Google maps service**: external service used to support geographical operations.
- **Payment service**: external service used to support the payment operations.
- **Driver license validator service**: external service used to verify the validity of driver licenses.

2.3 Component view

This section represents the whole system and shows the interactions between each part of it. Following this diagram are provided different subsections to see more in details how the system is composed.

2.3.1 Overall system view

The following image shows a lower level view of the system displaying the interfaces exposed by each component of the system.

Even though all the communications between clients and the apllication server pass through the the web server, we added some direct connection between different components for readability.

Our system relies on the Java Persistence API in order to maintain our data in a database, this allows an higher level of abstraction and is the reason why there is no direct connections between components and the data tier.

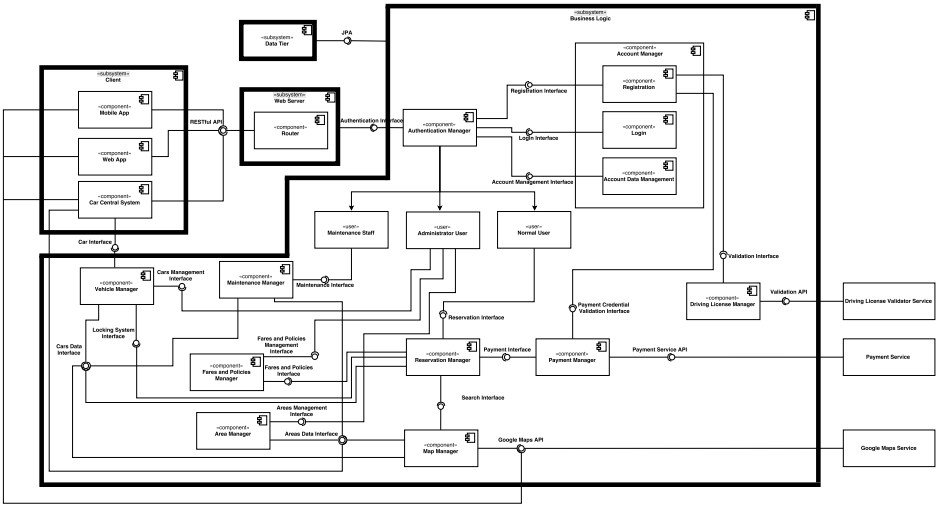

Figure 3: System component view

2.3.2 Web Server and Authentication

This image shows in detail the web server and the authentication component of our system.

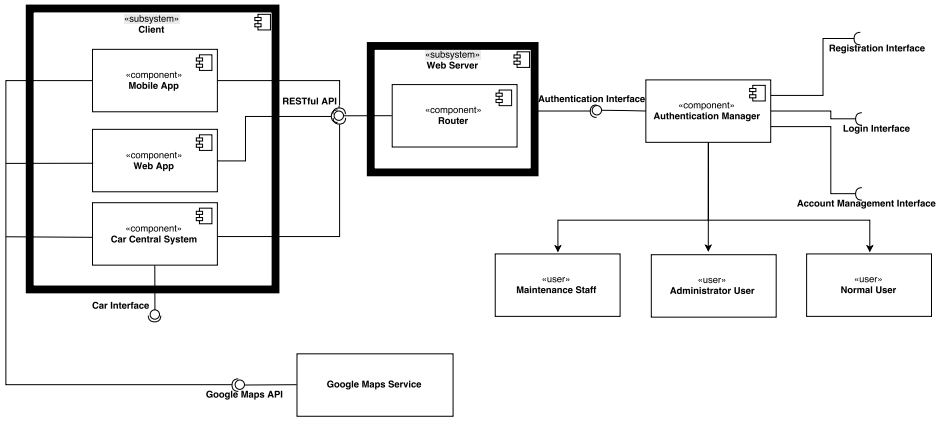

Figure 4: Web Server and Authentication Component View

- **Mobile App**: This is the main client component, it's realized starting from the web app and then adapted to the mobile market using the PhoneGap framework.
- **Web App**: This is client application our users can use in a desktop eonvironment.
- **Router**: This represents the main component of the web server used in our system that allows the clients to use our application.
- **Authentication Manager**: This component is responsible for the authentication of our customers. Since we implemented our service following the REST guidelines we had to implement an authentication mechanism.
- **Normal User, Administrator User, Maintenance Staff**: These are the three categories of user recognized by our system.

2.3.3 Account Manager

This image presents a closer look at the account manager component, the functions it provides and it's interactions with other components of the system.

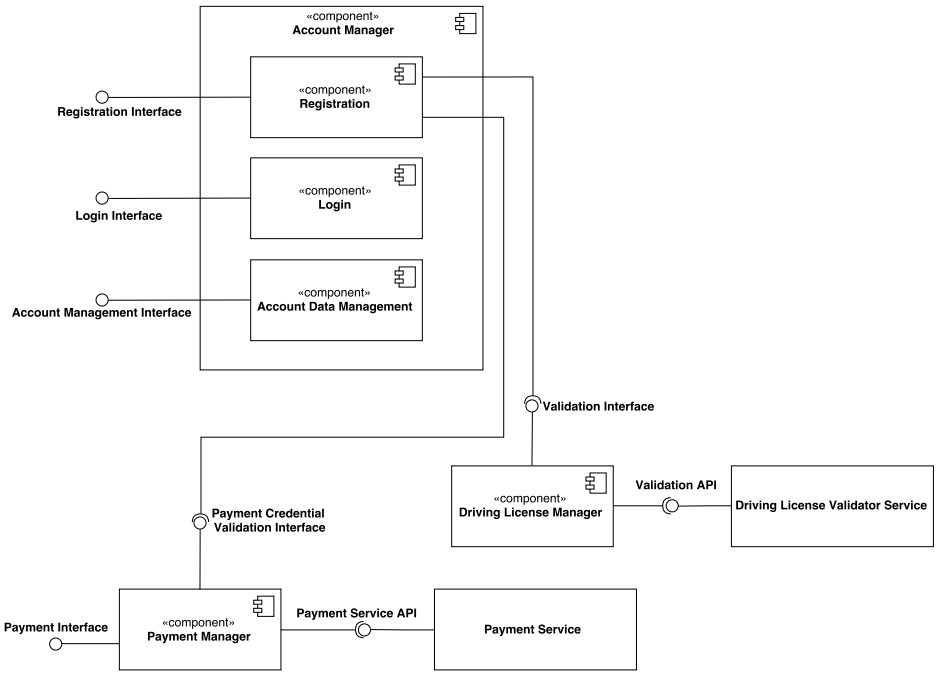

Figure 5: Authentication manager component view

- **Account Manager**: This is the component responsible for the login, the registration and the management of the account of each user. In fact this component exposes the *Registration Interface*, the *Login Interface* and the *Account Management Interface*.
  - **Registration**: This component takes care of the registration of a user. In order to do so it uses the *Validation Interface* exposed by the *Driving License Manager* to verify the authenticity of the user's driving license, and the *Payment Credential Validation Interface* exposed by the *Payment Manager* to check the payment informations submitted by the user.
  - **Login**: This is the component used when a user is logging into the system.
  - **Account Data Management**: This component allows the user to change the informations associated with his account.
- **Driving License Manager**: This component verifies whether or not the driving license submitted by the user is valid, to do so it uses the *Validation API* interface exposed by the *Driving License Validator Service*.
- **Payment Manager**: This is the component that handles the payments

and checks if the data submitted by the user during the registration process are related to a valid payment method.

- **Driving License Validator Service**: *IDCHECK.IO* is the external service we decided to use in order to check the driving license of the users that want to register in our system.
- **Payment Service**: *LevelUp* is the external service on which our system relies on in order to handle the payments.

2.3.4 Car System

This image presents the car system, it's interaction with the business logic and it's subcomponents.

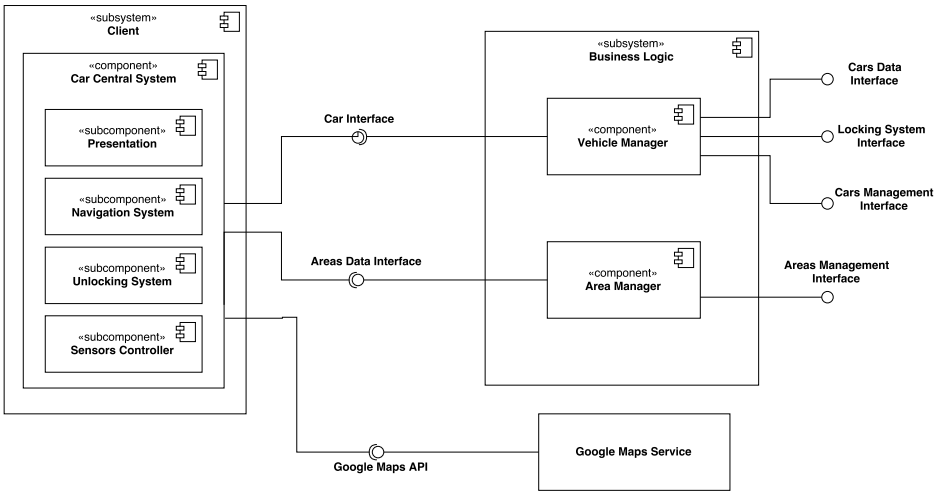

Figure 6: Car system component view

- **Car Central System**: This is the main component of each car, it is composed of different subcomponents each one performing a crucial task for the system.
  - **Presentation**: This is the subcomponent responsible for the interface shown to the user on the on-board touch display.
  - **Navigation System**: This subcomponent provides the user with a step-by-step navigation he can use to reach his destination. It recovers the informations about the parking areas position though the datas given by the *Areas Data Interface*.

- **Unlocking System**: This subcomponent communicates with the digital keyboard installed on the car's door allowing the user to lock and unlock the car.
- **Sensors Controller**: This is the subcomponent that collects all the datas coming from the different sensors placed on the car. (e.g. Battery Charge, GPS position,...)

- **Vehicle Manager**: This component interacts with the car through the *Car Interface* exposed by the *Car Central System*. This is the component inside the application server that keeps all the informations about the cars and is used for all the interactions with them.
- **Area Manager**: This is the component that controls all the parking areas of the company, it exposes the *Areas Data Interface* that the *Car Central System* uses to access the informations about the position of the parking areas, and the *Areas Management Interface* that allows an administrator to modify every aspect of the parking areas.

2.3.5 Normal User

This image provides an overview of the functionalities that are specifically available to a normal user of our system.

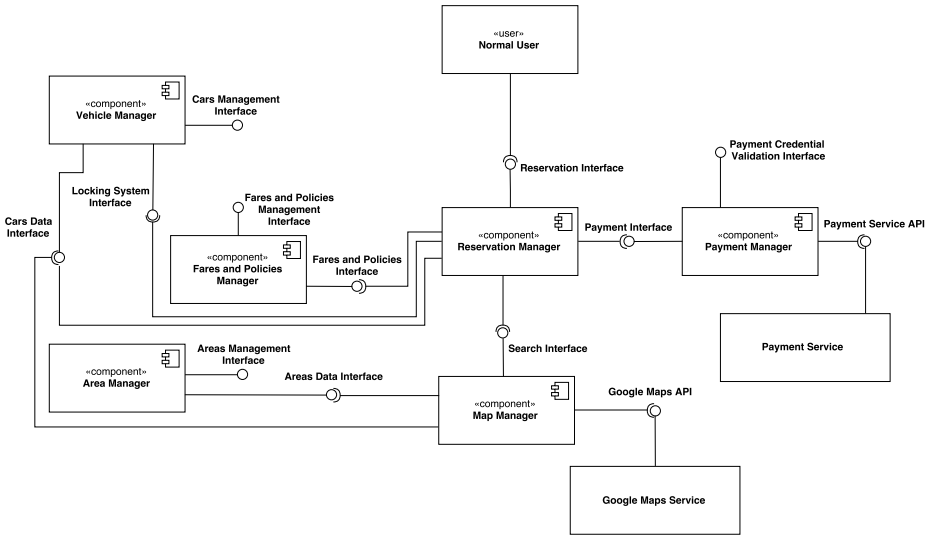

Figure 7: User Component View

- **Reservation Manager**: This is the component that manages all the aspects of a reservation. In order to do so it has to communicate with

different components of our system.

- **Fares and Policies Manager**: This is the component that handles all the fares and policies of the company. It can be accessed
- **Map Manager**: This is the component responsible for all the maps related functionalities present in the system.
- **Google Maps Service**: Google Maps is the service we decided to use to address our need of a reliable and functional service to handle a map.

2.3.6 Maintenance User

This image focuses on the maintenace user, showing the components it can access and it's funtionalities in our system.

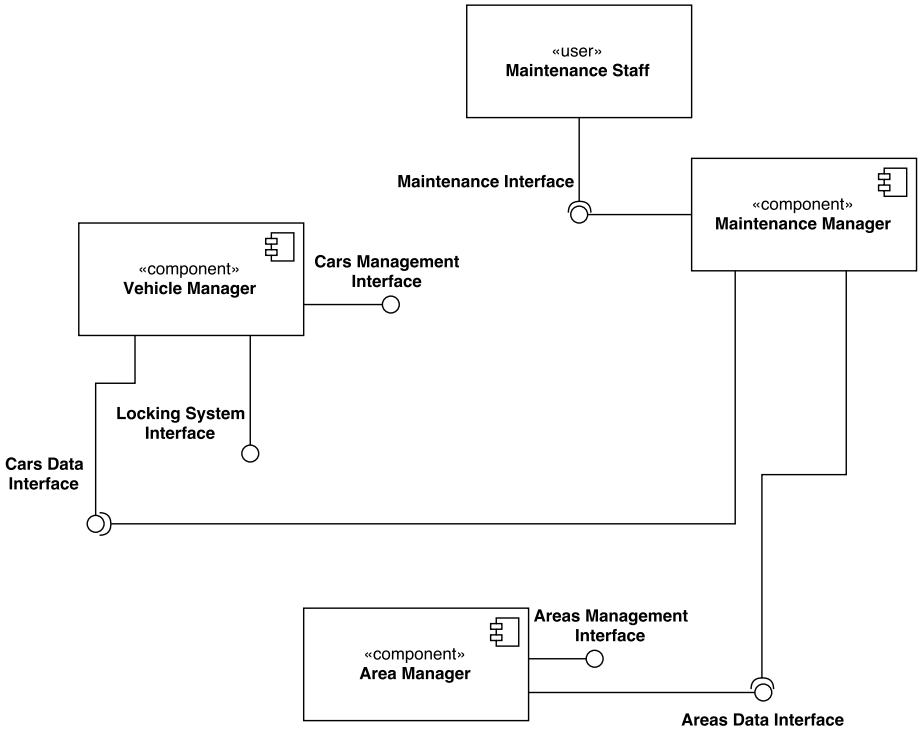

Figure 8: Maintenance User Component View

- **Maintenance Manager**: This component looks for any kind of problem regarding cars and parking areas so that the maintanace staff can promptly take action in order to resolve that problem.

2.3.7 Administrator User

This image shows the functionalities exclusively available to an administrator user of our system.

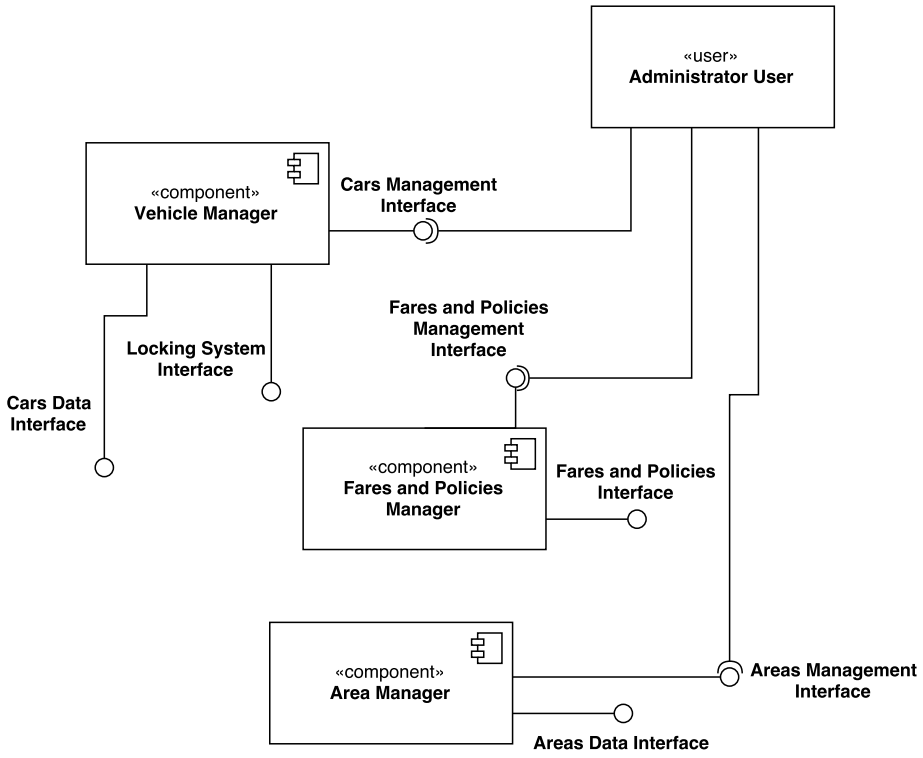

Figure 9: Administrator Component View

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.

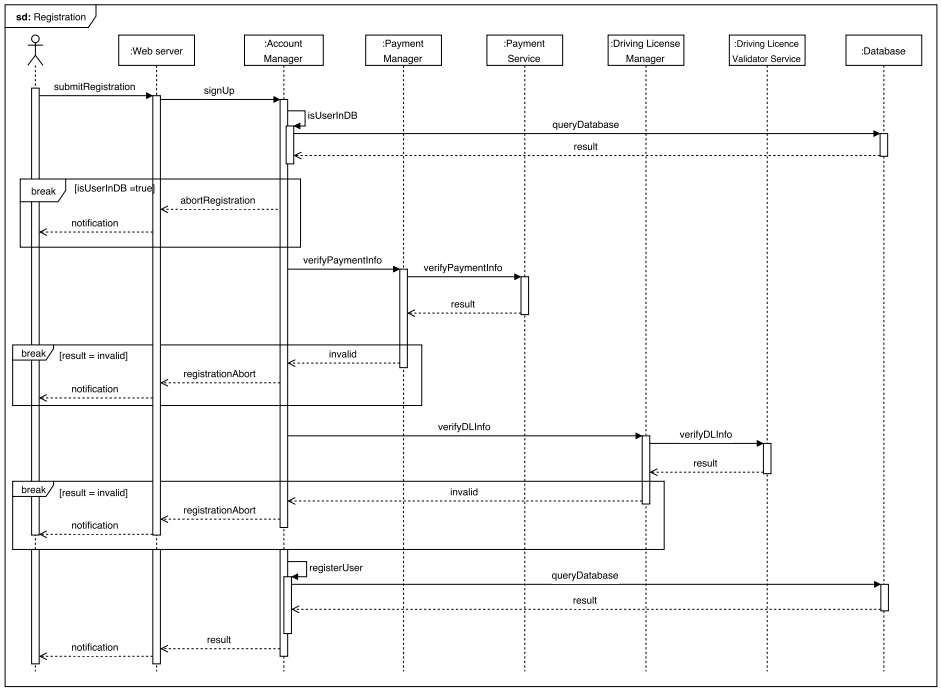

Figure 10: User registration sequence diagram

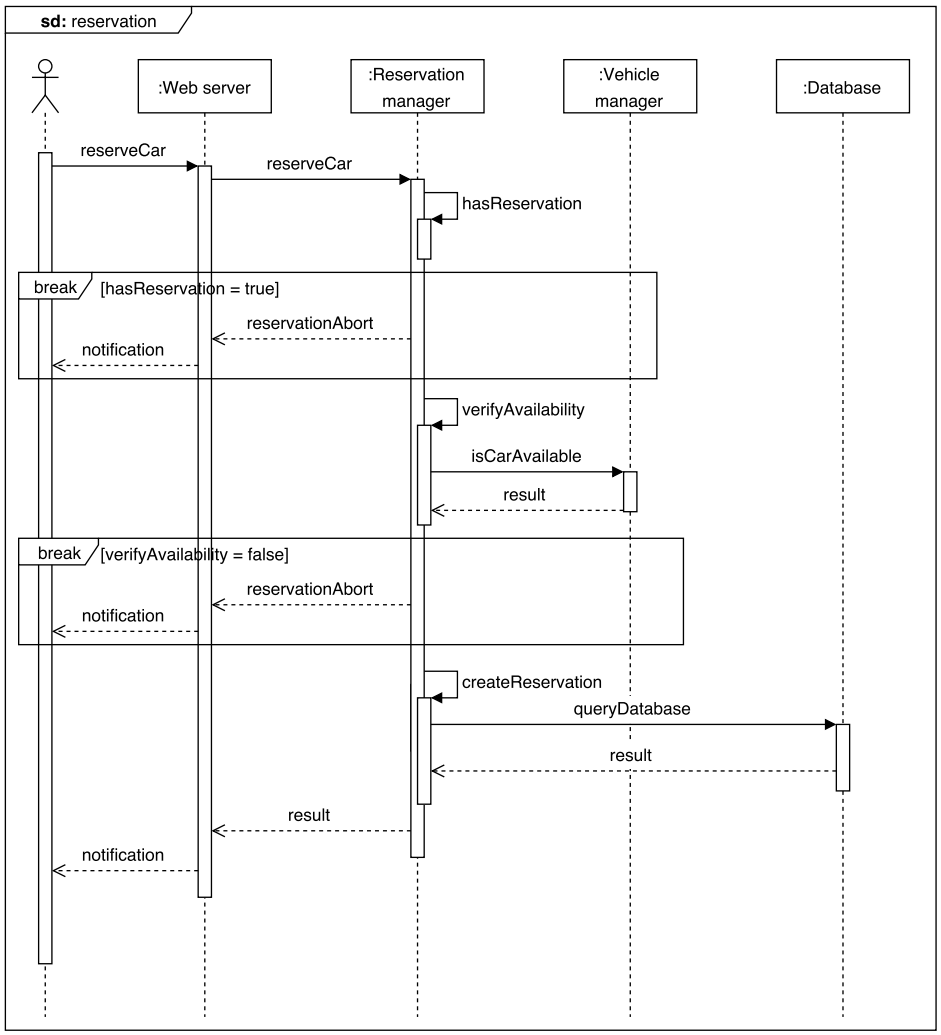

Figure 11: Create reservation sequence diagram

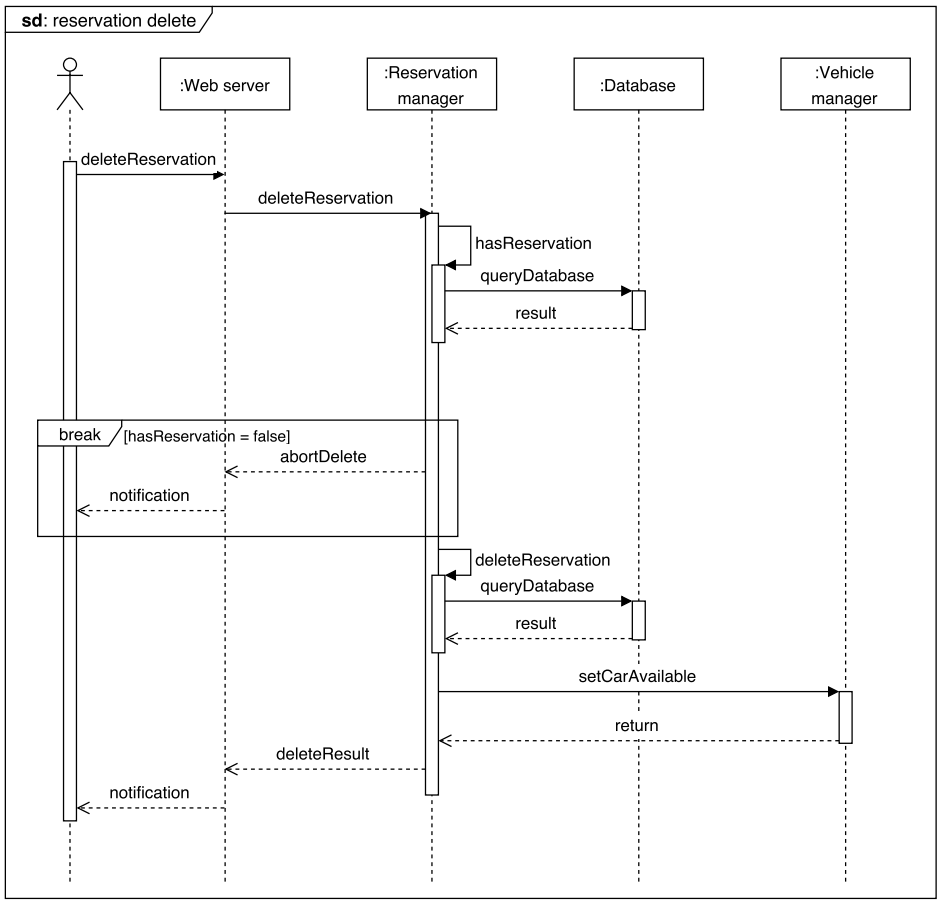

Figure 12: Delete reservation sequence diagram

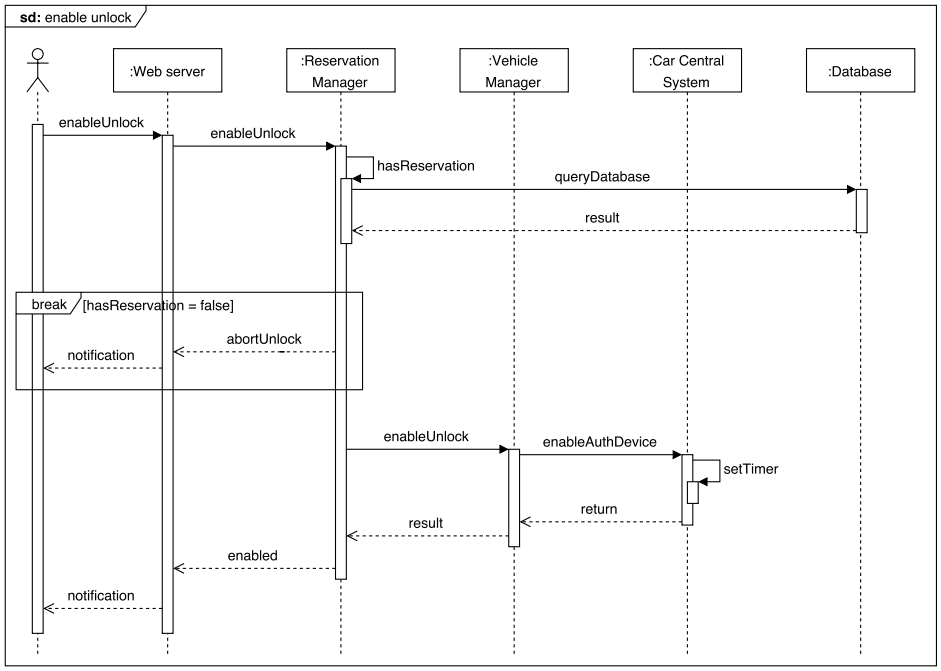

Figure 13: Enable unlock sequence diagram

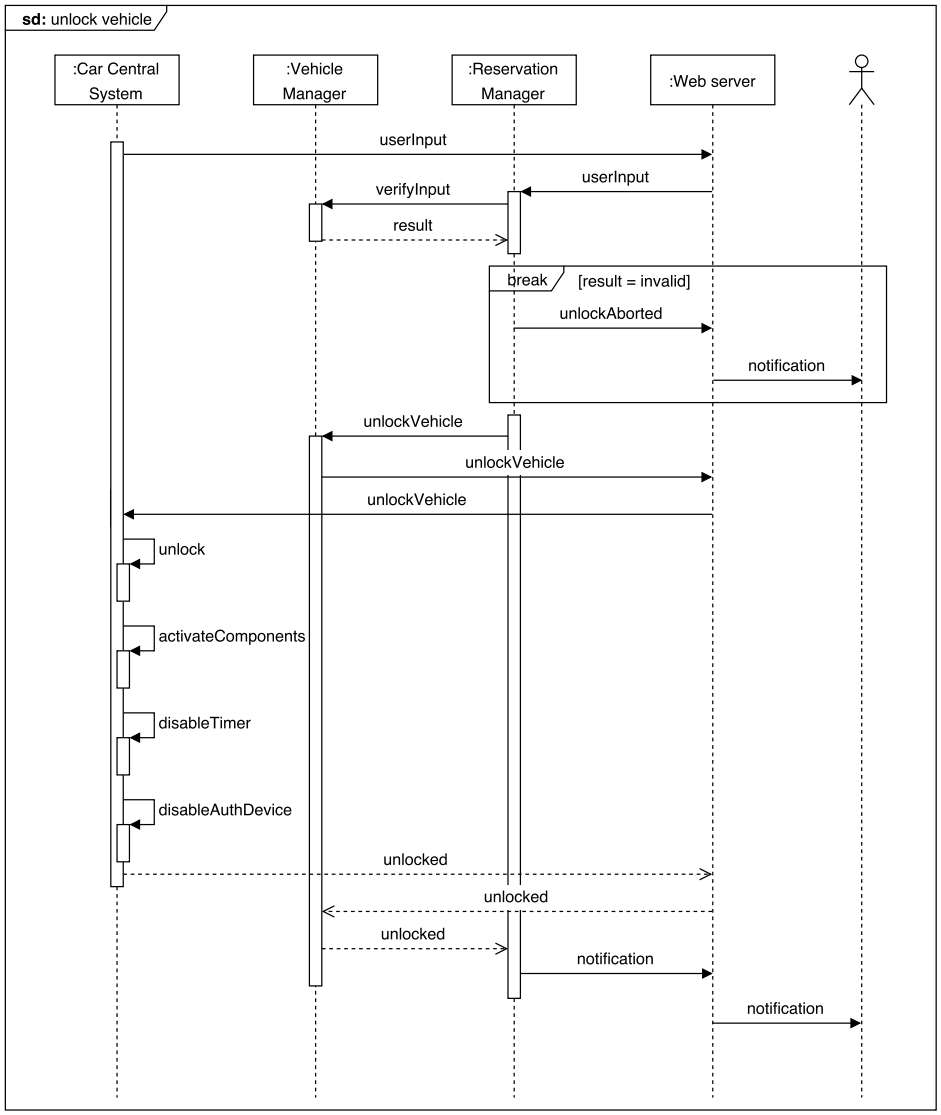

Figure 14: Unlock vehicle sequence diagram

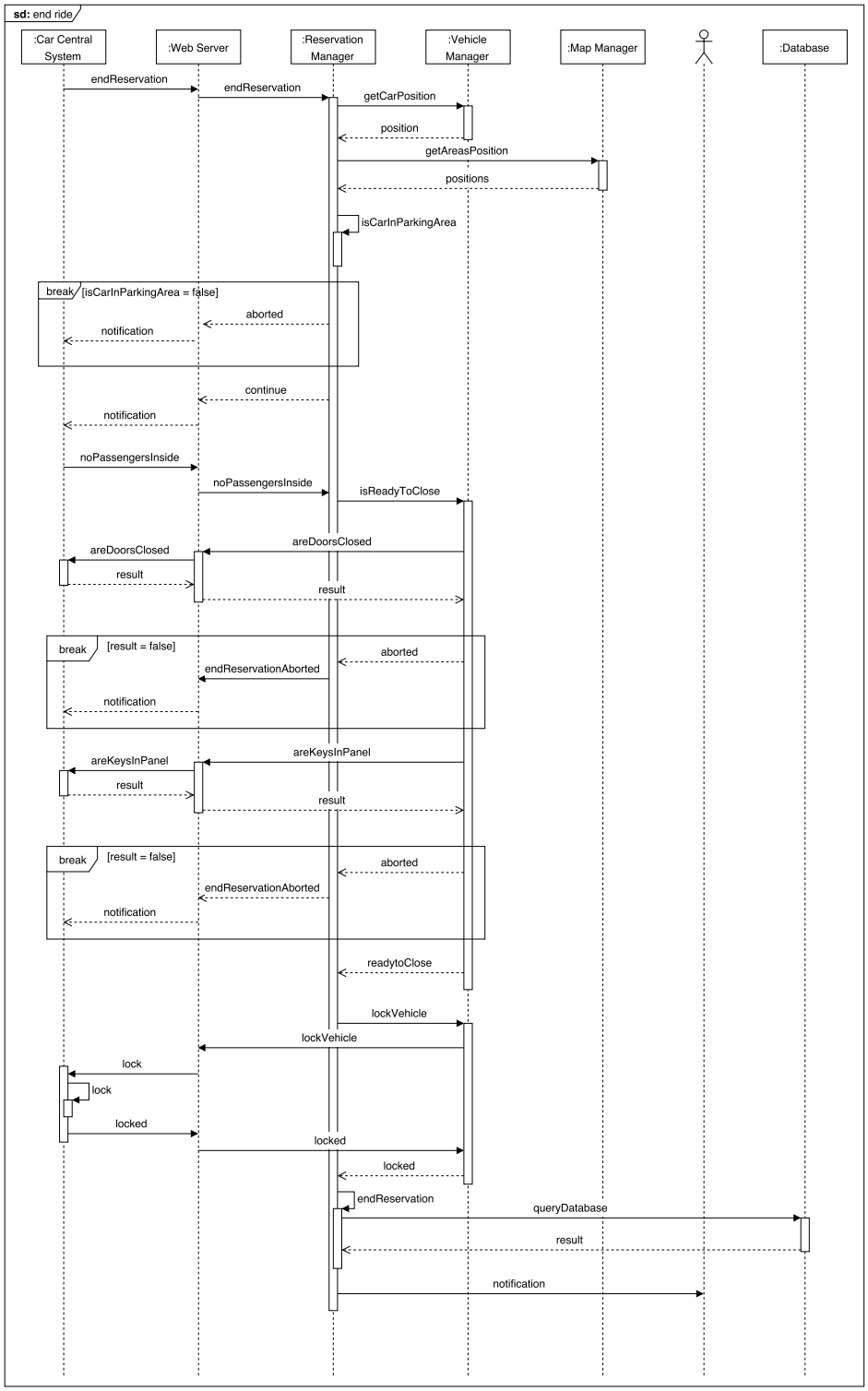

Figure 15: End reservation sequence diagram

User registration

2.6 Component interfaces

In this sections will be presented and described all the interfaces present in our system.

- **Web Server Interface**: Our web server exposes a RESTful interface over the HTTPS protocol.
- **Authentication Interface**: The access to the resources of our system is mediated by this interface that verifies the identity of each client that tries to connect to our system using HTTP basic auth over HTTPS.
- **Registration Interface**: This is the interface that provides the registration functionalities to the client connecting to our system, it relies on a document validation service and on a payment handling service.
- **Login Interface**: This inteface provides the login functionality to our system, it allows three different user type: Normal User, Administrator User and Maintenance Staff.
- **Account Management Interface**: This inteface allows any user to edit his account data or even to remove his account from the system.
- **Validation Interface**: This is the interface that connects our system to the external document validation service.
- **Validation API**: The *IDCHECK.IO* APIs are used in our system for the driving license validation. These are RESTful APIs that use JSON format. Given the picture of a driving license this service tells our system if the document is valid or not.
- **Payment Credential Validation Interface**: This interface connects our system to an exteral API that controls the authenticity of the payment informations submitted by the user.
- **Payment Service API**: The *LevelUp* APIs offer a lot of features related to the payments. They use JSON format response.
- **Reservation Interface**: This is the interface with which a normal user interacts in order to make a reservation. It interacts with different components in order to give the user all the informations he needs to make a reservation.
- **Payment Interface**: This is the interface responsible for the communication between the system and the external service that handles payments.
- **Search Interface**: This interface allows a user to search an available car.

- **Areas Data Interface**: This is the iterface used to fetch all the data about the parking areas, such as their position, and in case of special parking areas, the number of available plugs.
- **Areas Management Interface**: This is the iterface used by the administrator users to manage all the aspects of the parking areas. They can be added, removed or modified to reflect any changes in the real world.
- **Fares and Policies Interface**: This interface provides all the policies and is used to retrieve the fares to apply to the rides.
- **Fares and Policies Management Interface**: This interface provides to an administrator user all the functionalities to manage the fares and the policies of the company.
- **Maintenance Interface**: This interface provides the maintenance staff with all the information about malfunctioning or ordinary maintenance needed by cars and charging plugs.
- **Cars Data Interface**: This is the interface that serves all the datas that the sensors installed on the cars read and send to the system.
- **Locking System Interface**: This interface is used for the communications between the application and the car locking system. It is used to send the locking and unlocking signal, to alert the system if the doors are not closed, and to transmit the unlocking code inserted by the user on the keyboard placed on the car's door.
- **Cars Management Interface**: This is the interface used by an administrator user in order to manage the car fleet of the company. Cars can be added or removed, and the state of a car can be changed manually in case of malfunctioning of the automated system.

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.
- **REST architecture**: to improve the general performance and scalability of the system, adding a portability factor, a RESTful architecture is introduced as architectural base of the system.
- **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.

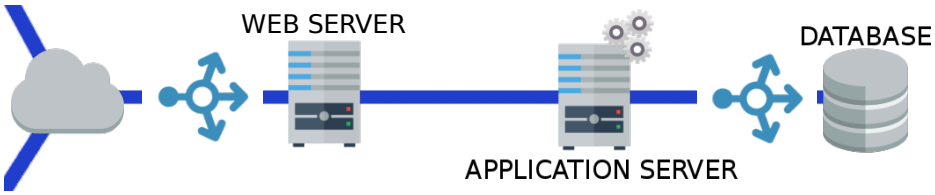

Figure 16: The load balancers' placement

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.

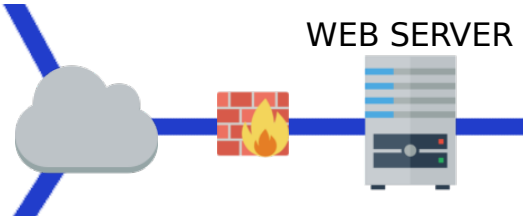

Figure 17: A firewall

- **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.

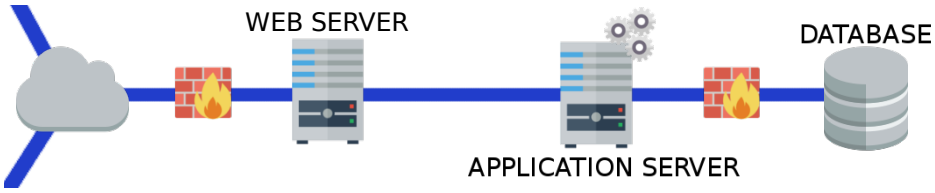

Figure 18: The Demilitarized Zone

- **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.
- **Secure connection**: nowaday everyone can, by means of a device and a packet sniffer, capture, monitor and tamper the internet traffic. Exchange of sensitive informations over the internet can be quite dangerous, either for software systems and users. Thus, the use of communication protocols strongly based on encryption between our software system and the devices (vehicles and user's devices) is required.
- **DoS/DDoS protection**: one of the most dangerous and serious attack for a society providing a service over the internet is the Denial of Service and its Distributed version. To leverage possible attack scenarios involving large volume of traffic from malicious users, firewalls and load balancer are placed at the entry of the software system's network.

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 in the area shown in the map, 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.

Algorithm 1 Car Search Handling Algorithm

```
1:  function nearbyAvailableCarSearch
2:      bounds = map.getBounds()
3:      availableCars = getAvailableCars(bounds)
4:      i = 0
5:      for i < availableCars.size() do
6:          map.addCar(
7:          position: availableCars[i].getLatLng(),
8:          title: availableCars[i].getId(),
9:          details: availableCars[i].getDetails()
10:         )
11:         i++
12:     end for
13:     return
14: 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(Bounds)**: Is a function that returns a list of car objects containing all the currently available cars inside the boundaries defined by the *Bounds* parameter.
- **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 19] and the system's administrators [figure 20] 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.

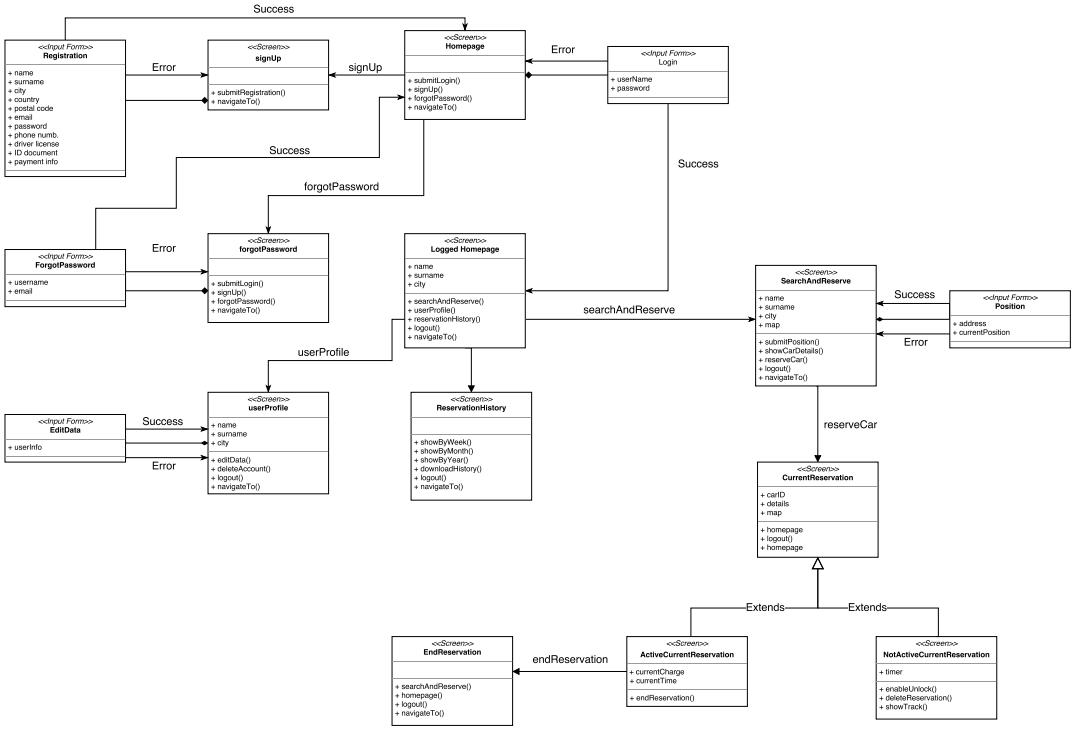

Figure 19: Web application & Mobile application UX

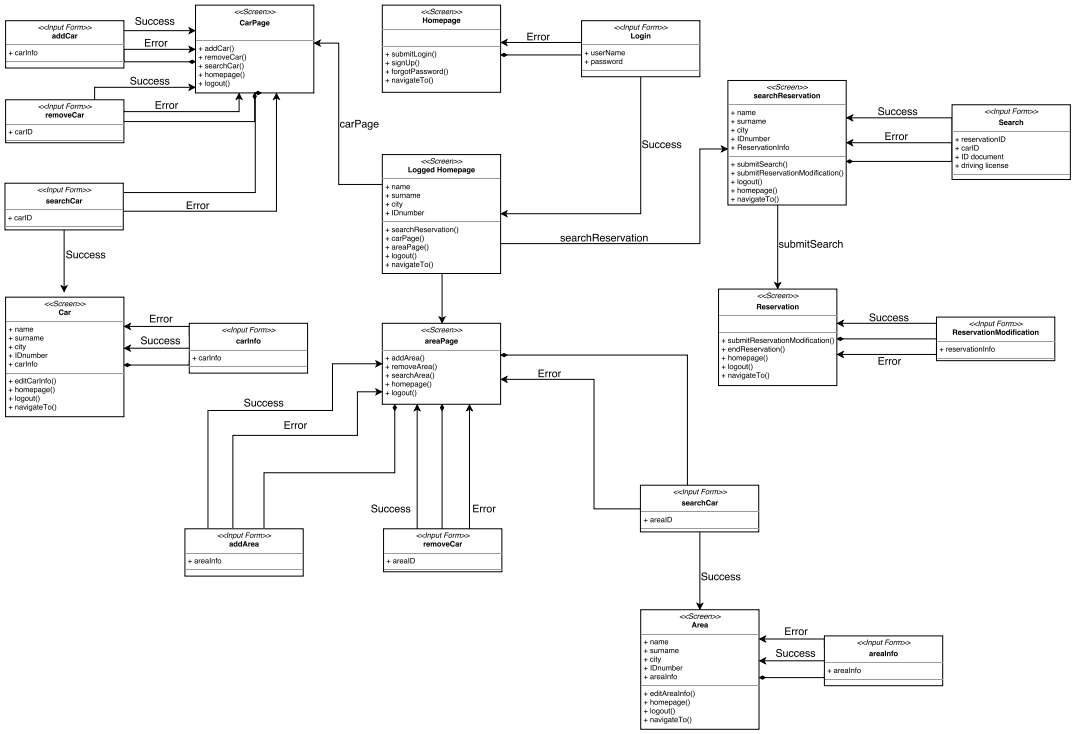

Figure 20: Administrator UX

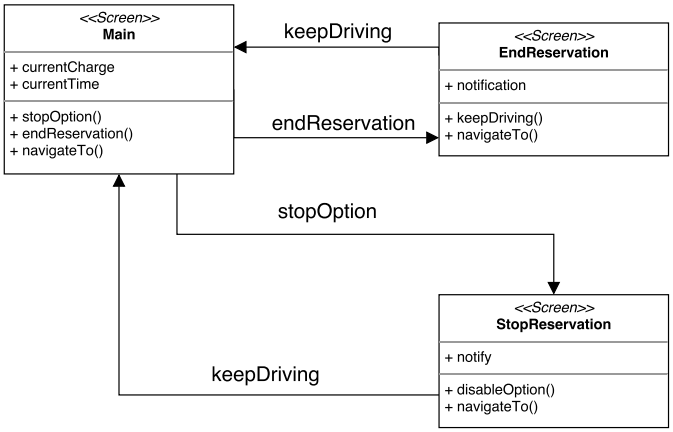

Figure 21: On-board software UX

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

A short and complete mapping between the requirements and architectural decisions taken is shown in the following, separating between the functional requirements mapping from the non-functional ones.

5.1 Functional Requirements

Component	**Requirement(s)**
Authentication Manager	(1.1) Separation of operation priviledges. (1.3) Multifactor mechanism.
Login	(1.2) Account existence verification.
Registration	(7.1) Users' informations registration. (7.2) Payment informations validation. (7.3) Driver license validation. (7.4) Unique user ID. (7.5) One user to one and only one driver license. One driver license to one and only one user.
Account Data Management	(7.7) User data elimination.
Vehicle Manager	(2.1) Comunication of nformations of cars nearby user's position. (2.3) Tracking available vehicle. (3.1) Enable vehicle unlock. (3.2) Control vehicles unlock. (4.1) Mark reserved vehicle as unavailable. (4.2) Mark as unavailable faulty vehicles. (4.4) Mark available car not reserved and not faulty. (5.3) End active reservation for cars left without passengers and lock them. (10.1.1) Notify user overcoming city boundaries. (10.2.1) Not allow reservation end if the vehicle's doors are not properly closed.

Reservation Manager	(2.1) Comunication of informations of cars nearby user's position. (2.6) Chosen car comunication. (2.7) Verify chosen vehicles availability. (2.8) Communication of car unavailability. (2.9) Record reservation details. (2.10) Retrieve and show position of parking areas. (3.1) Enable vehicle unlock. (3.2) Control vehicles unlock. (4.1) Mark reserved vehicle as unavailable. (4.2) Mark as unavailable faulty vehicles. (4.3) Automatically end out-of-time reservations. (4.4) Mark available car not reserved and not faulty. (5.1) Let user end reservation. (5.2) Let user leave car without end reservation applying special fare. (5.3) End active reservation for cars left without passengers and lock them. (5.4) Update informations of just terminated reservations. (5.5) Computation of ride's cost according to reservation time and special condition. (5.6) Verify enough money on payment system and deduct. Otherwise, forbid further reservation. (8.2) Car's conditions reports are stored. (9.1) Record cars' status changes and trggering avent. (9.2) Link cars' status changes to driver. (10.1.1) Notify user overcoming city boundaries. (10.1.2) Special fare applied to user that overcame the city boundaries. (10.2.1) Not allow reservation end if the vehicle's doors are not properly closed. (10.3.1) Notify user trying to end reservation outside parking areas.
Map Manager	(2.1) Comunication of nformations of cars nearby user's position. (2.10) Retrieve and show position of parking areas.
Car Central System	(2.3) Tracking available vehicle. (3.1) Enable vehicle unlock. (4.2) Mark as unavailable faulty vehicles. (5.3) End active reservation for cars left without passengers and lock them. (10.1.1) Notify user overcoming city boundaries. (10.2.1) Not allow reservation end if the vehicle's doors are not properly closed.
Area Manager	(2.10) Retrieve position of special parking areas. (6.1) Store parking areas informations.

Fares and Policies Manager	(5.2) Let user leave car without end reservation applying special fare (5.5) Computation of ride's cost according to reservation time and special condition. (10.1.2) Special fare applied to user that overcame the city boundaries.
Payment Manager	(5.6) Verify enough money on payment system and deduct. Otherwise, forbid further reservation.
Database	(6.1) Store parking areas informations. (7.1) Users' informations registration. (8.2) Car's conditions reports are stored. (9.1) Record cars' status changes.

5.2 Non-functionals Requirements

Unlike the previous subsection, here is given more emphasis to the non-functional requirements, mapping them to the design choices.

To have a better understanding about how the design choices satisfy the related requirements, see the *Selected architectural styles and patterns* and *namerefsec:other-design-decisions* sections.

Requirement	**Design Choice(s)**
Performance	Distributed representation Elastic infrastructure. Elastic platform. Load balancer. REST architecture.
Reliability	Elastic infrastructure. Elastic platform. REST architecture.
Availability	Elastic load balancer. Elastic infrastructure. Elastic platform. REST architecture.
Portability	REST architecture.
Security	Firewalls Demilitarized zone. Elastic load balancer. Password protection. Password security. Secure connection. DoS/DDoS protection.

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
- 21 December 2016 - 5 hours

Tommaso Castagna

- untill 9/12/16 about 10h
- 09/12/16 - 2h30m
- 12/12/16 - 3h30
- 13/12/16 - 30m
- 14/12/16 - 2h
- 15/12/16 - 1h30m
- 16/12/16 - 2h30m

- 18/12/16 - 1h
- 19/12/16 - 2h
- 20/12/16 - 3h
- 21/12/16 - 4h
- 22/12/16 - 8h