POLITECNICO MILANO 1863

POWER ENJOY

Design Document

Lorenzo Casalino

Tommaso Castagna

Document version 1.0

Contents

1 INTRODUCTION

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms and Abbreviations

- **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.

1.4 Reference Documents

1.5 Document Structure

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.

Since the context 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 1].

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.

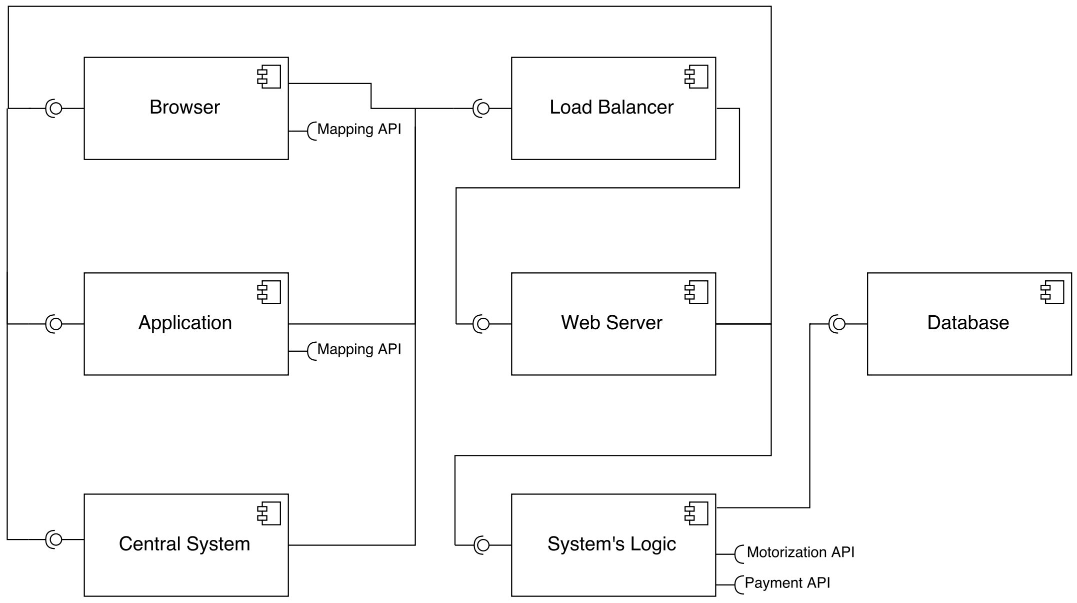

Figure 1: High-level component diagram

- **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

2.4 Deployment view

2.5 Runtime view

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 mainly 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. Therefore, the client/server paradigm is inserted dividing the request logic from the response logic and placing them into distinct tiers.
- **Distributed Representation**: in order to keep the client side as light as possible, a client/server paradigm based on distributed representation is chosen. All the application logic layer and the data layer are place on system side. The presentation layer, instead, is splittedin two part:
  - **System side**: the presentation layer consist in software for the generation of responses, like web pages or meta documents.
  - **Client side**: the presentatino layer consist in software for the interpretation of system's responses, such as web browser.
- **4 Tiers**: the client/server architecture is split into 4 distinct tiers, decoupling data, application logic, request/response logic and client logic. In this way, the general maintenance of the system is dramatically improved, introducing an high 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 unpredictable change in requests load, an elastic infrastructure is introduced to support the whole system. It exposes a set of API 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, for instance multiple web servers or application components used by different users. To achieve this result, a software called *component manager* is used to manage the software components. It exposes an interface 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.

2.8 Other design decisions

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

- **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.

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

```
function nearbyAvailableCarSearch
    bounds = map.getBounds()
    availableCars = getAvailableCars()
    i = 0
    for i < availableCars.size() do
        currentLatLng = availableCars[i].getLatLng
        currentId = availableCars[i].getId
        if bounds.contains(currentLatLng) then
            map.addCar(
            position: currentLatLng,
            title: currentId,
            details: availableCars[i].getDetails
            )
        else if map.contains(currentId) then
            map.removeCar(currentId)
        end if
        i++
    end for
    return
end function
```

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 served by more than one user. To do so we decided to use a lock-like pattern.

Algorithm 2 Car Reservation Handling Algorithm

```
function carReservation(CarId id, User user)
    $lock = null$
    $car = null$
    try
        $lock = getCarLock(id)$
        $availableCars = $ getAvailableCars()
        $car = availableCars.getCar(id)$
        $car.reserveCar()$
        $currentReservations = $ getCurrentReservations()
        if $currentReservations.contains(user)$ then
            Throw multipleReservationException
        end if
        $reservationTimer = Timer(60)$
        $reservationTimer.start()$
        createReservation(id, user, reservationTimer)
    catch (LockedResourceException)
        return ReservationError
    catch (CarNotFoundException)
        $lock.releaseLock()$
        return ReservationError
    catch (multipleReservationException)
        $lock.releaseLock()$
        $car.setAvailable()$
        return ReservationError
    end try
    return
end function
```

4 USER INTERFACE DESIGN

4.1 User eXperience (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.

It's worth noting that he 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.

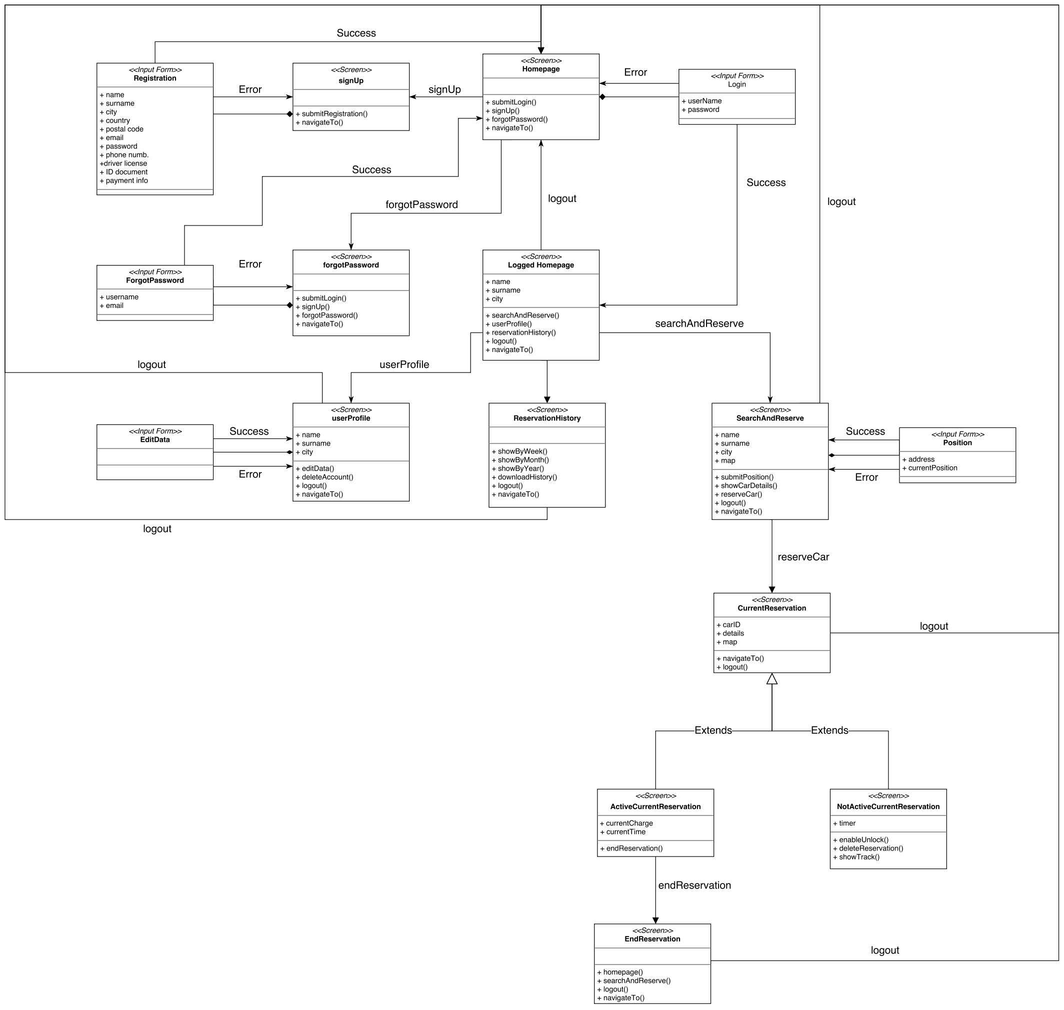

Figure 2: Web application & Mobile application UX

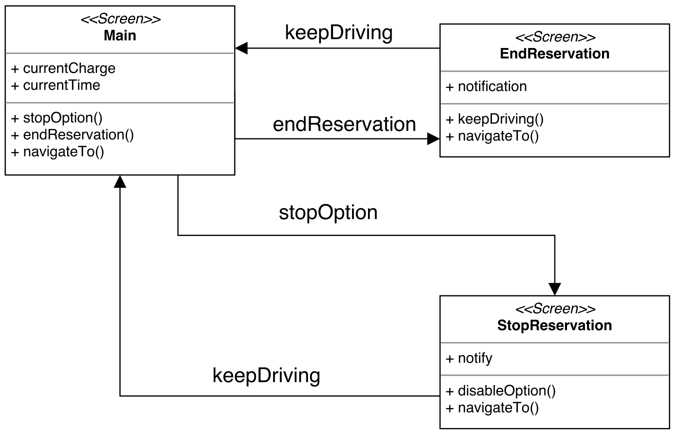

Figure 3: On-board software UX

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

7 REFERENCES