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 server tier**: layer containing the components created to implement and support the business logic.
- **Database 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.

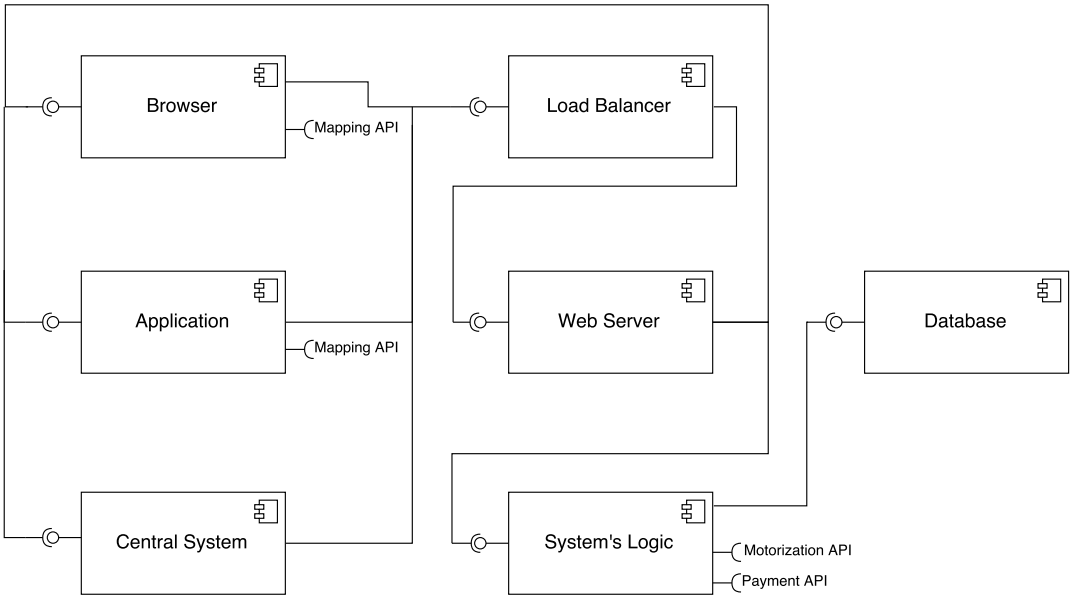

Figure 1: High-level component diagram

- **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

Here the main architectural styles and/or patterns are listed and described, motivating the reason why they have been chosen and how they integrate into the system.

- **Client/Server paradigm**: since most of the interaction between the system and the "outside" happens through requests and responses, the Client/Server paradigm result to be the most suitable design choice for the system. The paradigm is implemented through the introduction of web servers components that have the task of receive the incoming requests, process them and answer back with responses messages.
- **Distributed Representation**: in order to keep the client side as thin as possible, a client/server paradigm based on distributed representation is found to be the best solution possible for the project. All the business logic, the data and part of the presentation layer are placed on the server side, while the client side is loaded with part of the presentation layer.
- **4 Tiers**: the client/server architecture is split into 4 distinct tiers, decoupling data, business 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 Load Balancer/Elastic Component**: this cloud pattern is introduced in order to improve the general availability of the system. It is placed in front of the Web Server Tier and through the incoming requests and through the runtime information regarding the workload of the servers the Load Balancer allocate the workload in a balanced fashion.

2.7.1 Utility tree

2.8 Other design decisions

3 ALGORITHM DESIGN

This sections present some of the main algorithms used in the application.

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

3.2 Car reservation

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 generation event, so that the user can drag the map and still see the available cars.

Algorithm 2 Car Reservation Handling Algorithm

```
    function CARRESERVATION(CarId id, User u)
2:      bounds = map.getBounds()
        i = 0
4:      for i < availableCars.size() do
            currentLatLng = availableCars[i].getLatLng
6:          currentId = availableCars[i].getId
            if bounds.contains(currentLatLng) then
8:              map.addCar(
                position: currentLatLng,
10:             title: currentId,
                details: availableCars[i].getDetails
12:             )
            else if map.contains(currentId) then
14:             map.removeCar(currentId)
            end if
16:     end for
    end function
```

4 USER INTERFACE DESIGN

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

7 REFERENCES