POLITECNICO DI MILANO
Computer Science and Engineering
Software Engineering 2 Project

Tesla Car Sharing

Design Document

Danchenko Elena, 874840
Cilloni Stefano, 880924

December 12th, 2016

Document version: 1.0

Contents

1. Introduction

1.1. Purpose

This Design Document for the Tesla Car Sharing application aims to provide a functional description of the main architectural components, their interfaces and their interactions. It will give a more specific description of the architecture of the system project introduced by the RASD. With UML standards this document will show an overall structure of the designed system and how its modules interact together.
This document is written for developers, project managers and testers and it can be used to identify:

- The high level architecture of the system
- Used design patterns
- Main components operating in the system
- The cloud based environment runtime behaviour

1.2. Scope

The designed systems aims to offer a simple and reliable software that let users to benefit of the car sharing service thanks to web pages and mobile applications.
This Design Documents wants to describe our choices about architectural and algorithmic designs, designed user experiences that our application should provide to users and the requirement traceability in order to demonstrate that our decisions are compatible with requirements described in the RASD.
Furthermore, this document will explain the software structure that we decide to implement and software patterns adopted in order to provide an high level of maintainability and extensibility to easily implement functions in the future.

1.3. Definitions, acronyms, abbreviation

- **RASD:** Requirements Analysis and Specification Document.
- **DD:** Design Document (this document)
- **PaaS:** Platform as a Service is a category of cloud computing services that provides a platform allowing developers to run and manage applications without the complexity of building and maintaining the infrastructure typically associated with launching an application.
- **GCP:** The Google Cloud Platform is the collection of cloud computing services by Google that offers to developers cloud based environments powerful as the ones used internally for end-user products developed by Google (Google search engine, YouTube, Maps, etc).
- **GAE:** Google App Engine or simply App Engine is a Paas cloud computing platform for hosting web applications in Google-managed data centers that is part of the cloud

services provided by the GCP. In this kind of software environments applications are sandboxed and run across multiple servers to scale on workload demand.

- **Datastore:** It is the No-SQL (non relational) distributed database management system by the GCP. It is a DBMS (Database Management System) for cloud infractures). It is well integrated with the App Engine cloud environment through a large set of APIs.
- **UI:** User interface. It is the graphical software interface that present to users functionalities.
- **Application server:** it is the core layer of the whole application which implements through software logics and procedures that allow software to run.
- **Administration console**: Section of the web application accessed by the administrator and managers that offers system management functionalities
- **Back-end:** term used to identify the application server and sometimes both the application server and the database. It is the server-side part of the system.
- **Front-end:** term used to identify software components which use application server services and that are used by system users. They are the web application and the mobile applications.
- **MVC:** Model View Controller is a software development pattern that organize software code by its function.
- **API:** Application Programming Interface is a set of subroutine definitions, protocols, and tools for building application software.
- **REST:** Representational state transfer are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations.
- **HTTPS:** is is a protocol for secure communication over a computer network which is widely used on the Internet.
- **DBMS:**is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data.
- **Functional module:** Like a Java package it groups classes, functions and programming code. In the Atuin Framework it has a semantical meaning that help developer to well structure code.

1.4. Reference documents

This document refers to the following documents:

- Specification document: Assignments AA 2016-2017.pdf
- IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications
- Google App Engine Documentation
- Example document: Sample Design Deliverable Discussed on Nov. 2.pdf
- Google maps distance matrix documentation

1.5. Document structure

This document is structured in five parts:

- *Chapter 1: Introduction.* This first section introduces the DD document providing an introduction to the problem, the language used in the document, the referenced documents and the document structure.
- *Chapter 2: Architectural design.* This section describes the main components of the system with their sub-components and the interactions between components. This section also highlights design choices, styles, patterns, paradigms and framework adopted.
- *Chapter 3: Algorithm design.* This section presents and discusses in detail some hypothesized algorithms designed to implement system functionalities.
- *Chapter 4: User Interface Design.* This section shows samples of user interfaces and how they will look like and will behave.
- *Chapter 5: Requirements traceability.* This section reports requirements form the RASD document and how they will be satisfied by software functionalities described in this DD document.

2. Architectural design

2.1. Overview

This section of the design document provides a comprehensive view of the system components, both at a physical and at logical level.

The following schema represents the main components of the system and how they interact through API calls. The web application running in the browser will interact with the server with classical HTTP requests when the page visited by the user changes and also with API calls made with the AJAX technique to avoid page refresh for small interactions with the server.

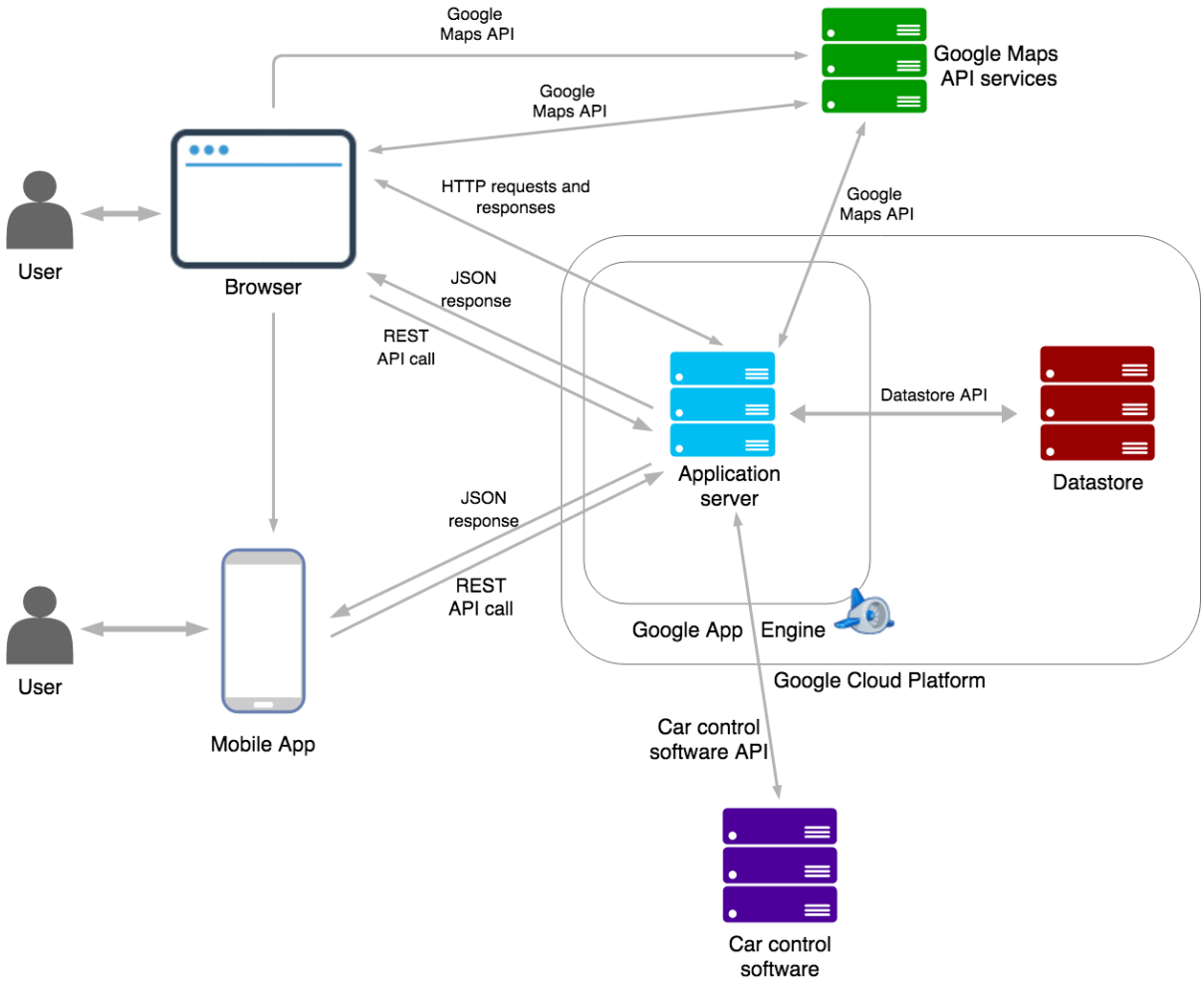

Figure 1. General system architecture

The following system description starts with the high-level components definition (section 2.2) and then, the next section 2.3, will detail the presented architecture. The 2.4 section will explain how the deployment procedure is supposed to be done to push the application

server on the cloud environment. The section 2.5 will describe the dynamic behaviour of the software and then, section 2.6, will focus on interactions between system components describing which protocol will be used to allow communications.
Design choices, patterns and frameworks used will be presented and explained in section 2.7

2.2. High level components and their interaction

The main high level components of the system are the following:

- **Datastore:** this data layer provides data management functions like transactions, atomicity, data reliability and scalability.
- **Application server:** this layer implements all the application logic for the system. Search algorithms, optimization, data management and formatting, policies, user permissions are performed here.
- **Mobile application:** it is a presentation layer with few computational capabilities mostly used to represent raw data in UIs. It connects directly to API provided by the application server to obtain information.
- **Web browser:** this presentation layer let the user asks for web pages provided by the application server. When web pages runs on the user browser they may ask directly to the application server information through API calls in an asynchronous way.

These high-level components can be structured into three layers shown in the following schema.
The system will be developed as a classical 3-tiered cross-platform application, separating the client side application part from the application server and the database.

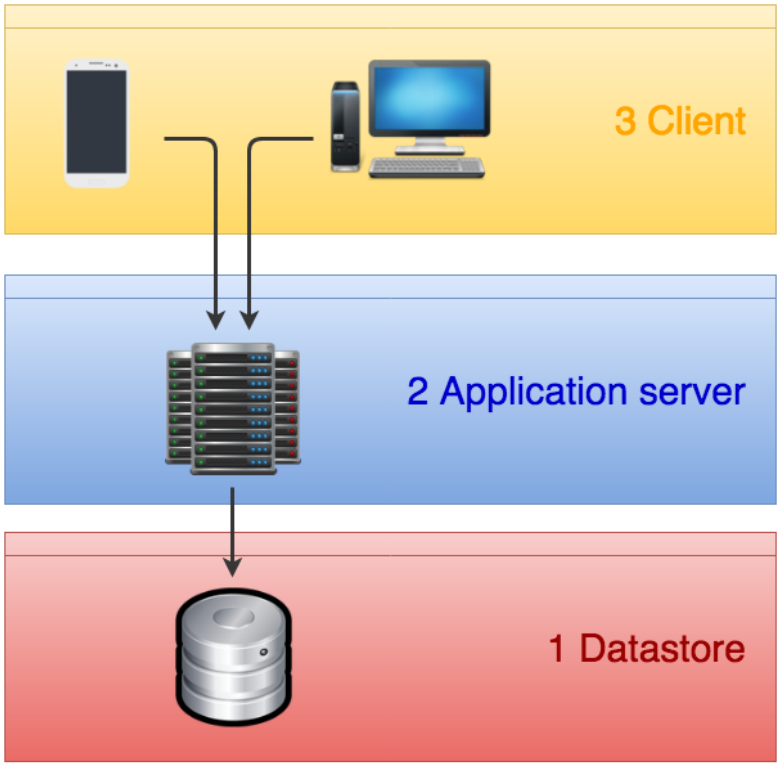

Figure 2. Layered system representation

The connections between the main components are shown in the following figure.

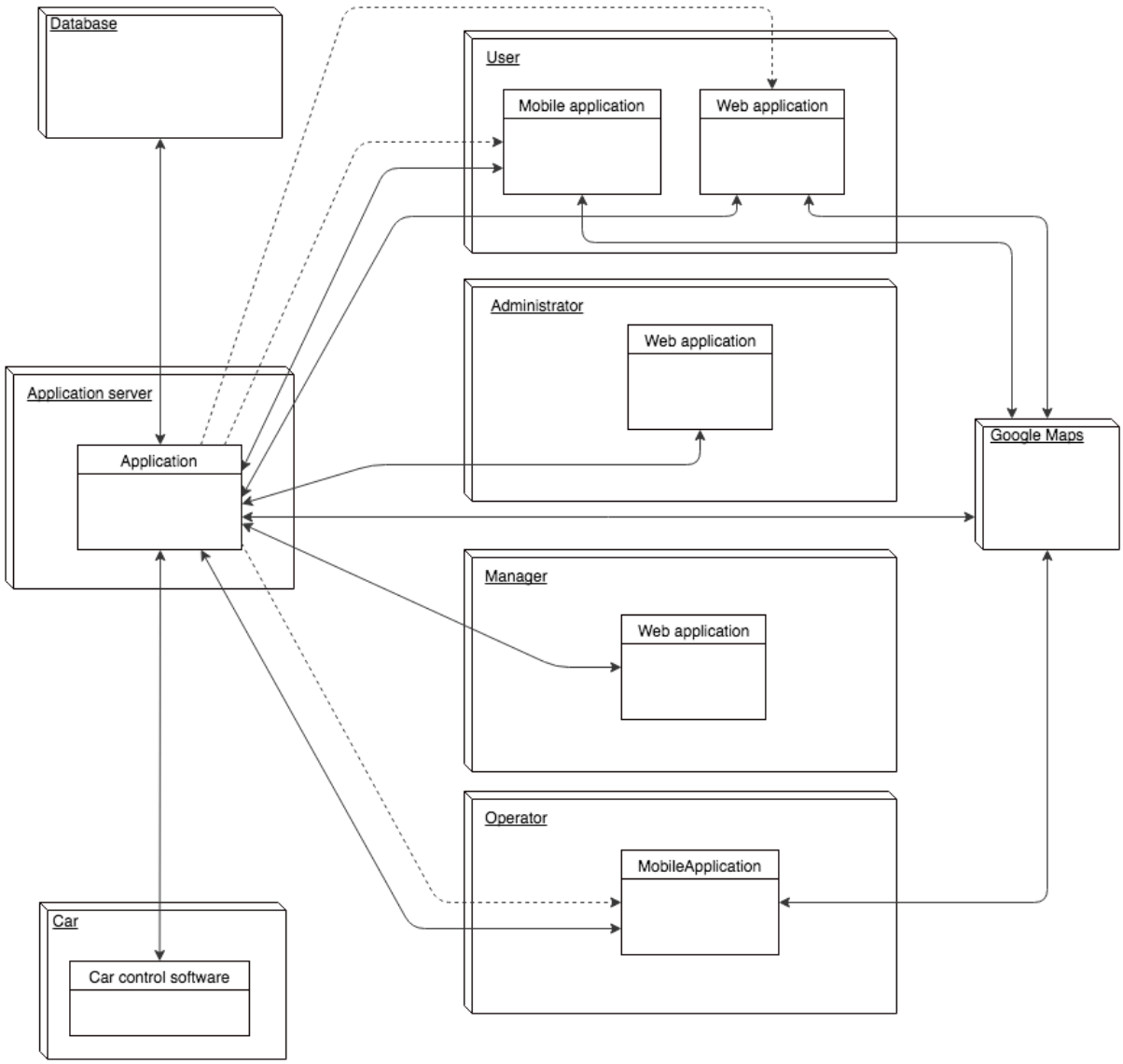

Figure 3. High-level components view

2.3. Component view

In the figure 4 component view of the system is represented.
The system is organized in functional modules based on the structure defined by atuin framework. Each functional module consists of three parts: administration, client, models.
The application layer is divided in following modules:

- BluePrint. The module functions as a router of the system sending requests to modules responsible for responses to them
- Authentication module. Consists of:
    1. PermissionController. Checks that the user type allowed to do an operation is the same that is logged in.
    2. AccountManagementController. Allows administrator to manage accounts.
    3. LoginManager. Gives software users the possibility to log in
- Car module. Consists of:
    1. CarMangementController. Allows managers to manage information about cars registered in the system
    2. CarInteractionManager. Communicates with cars software
    3. SearchController. Allows to search for cars
    4. ReservationController. Manages reservation and unlocking of the car
- Tasks module. Consists of:
    1. TaskController. Is responsible for task creation and changing of their status to completed
    2. TaskManager. Presents the tasks list to operators, gives the operators opportunity to choose, waive tasks and start the task completion confirmation procedure.
- Notifications module. Consists of:
    1. NotificationController. Is responsible for sending notifications to software users.
- Payment module.Consists of:
    1. PriceController. Allows managers to change price per minute
    2. ChargeCalculatorController. Is responsible for calculating final price for the ride

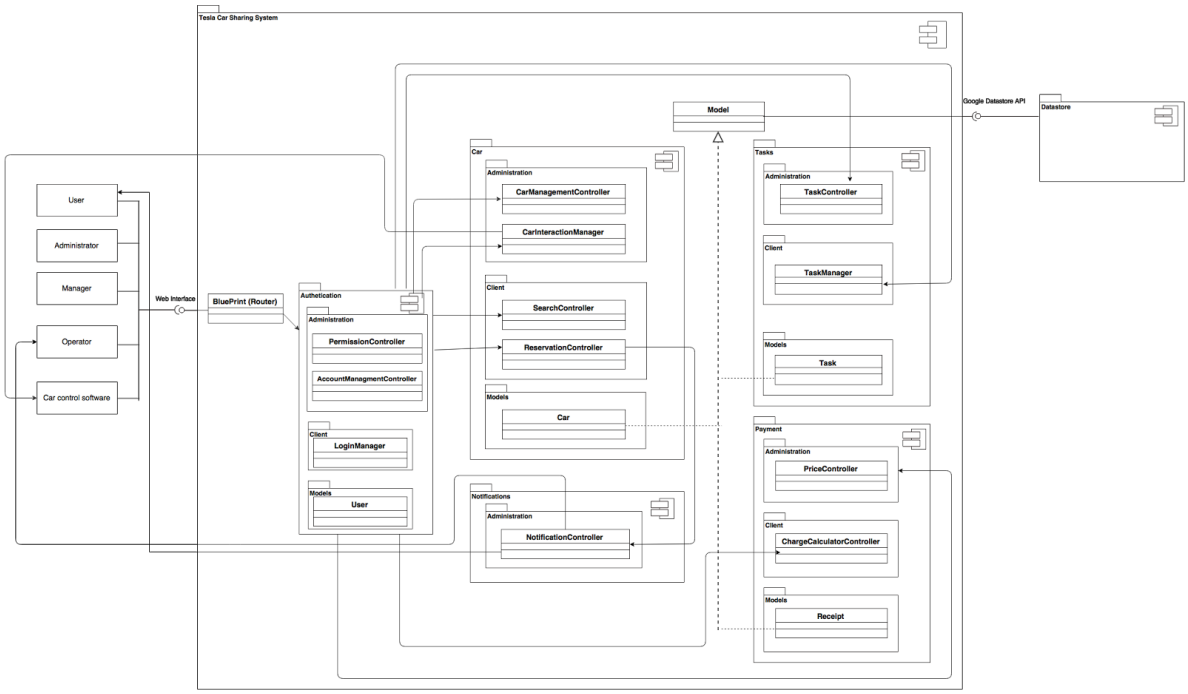

Figure 4. Component View diagram

2.3.1. Datastore

The technology chosen for the database tier is the Datastore engine provided by the GCP. Through a large set of API this distributed, No-SQL DBMS allows the application server to run queries as they're defined by the No-SQL logic. This DBMS will not be internally designed because it is an external service used as a "black box". To use it, it is only needed to enable Google Datastore API from the cloud console and then make API calls from the application server.

Communications between the application server and the datastore are secured and managed by the platform. The access permissions to data are checked by the application server that implement all permission logics to allow or deny users to retrieve data. Every software component that needs to access information stored in the database implements some logics to check user's permission before access data.

All persistent application information are stored in the database. Since the database itself does not support relations between entities these are implemented by the application server logic. We just want to give a conceptual representation of the structure design of the database with the following E-R schema.

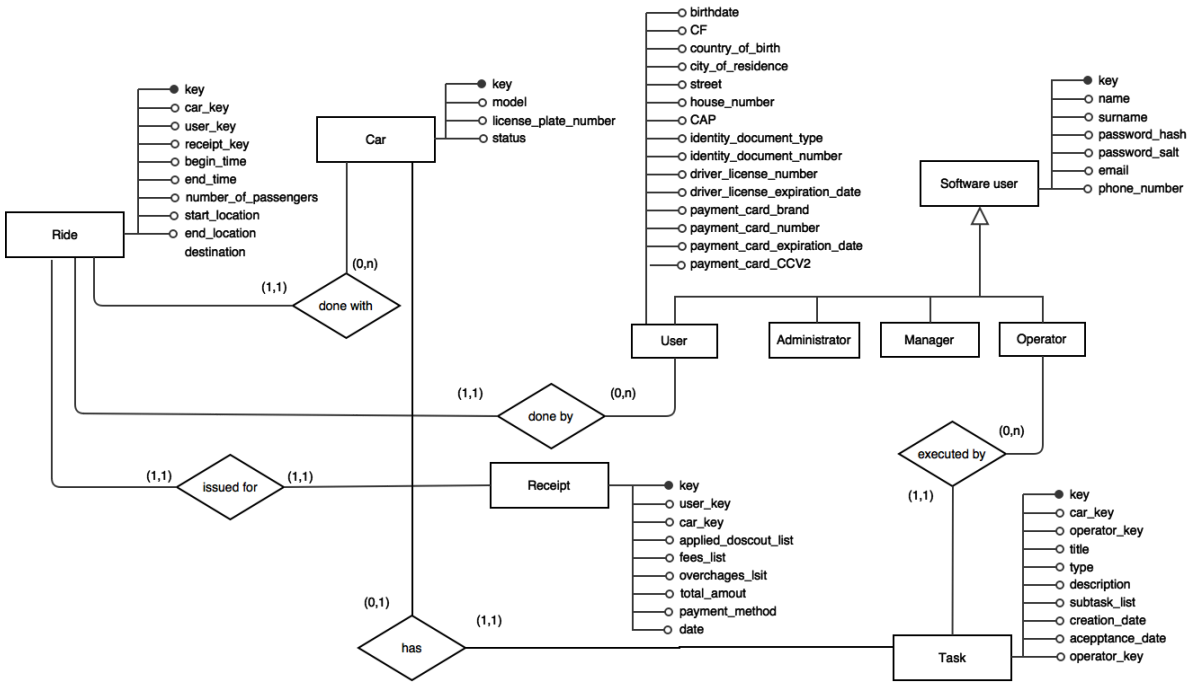

Figure 5. Entity-Relationship diagram

2.3.2. Application server

The application server is implemented using Python, Flask, Jinja2, Atuin framework and it runs on Google App Engine software cloud environment.
The application server uses Model-View-Controller pattern and takes advantage to the structure provided by the Atuin framework to well organize the code.

2.3.3. Mobile application client

Mobile application client is implemented for Android in Java and for iOS in Swift. Both mobile applications will use the MVC pattern to respect the mobile operative system behaviour. Both of them will call same APIs provided by the application server. We will implement both application with the native programming language of the system and not with a cross-platform framework due to the performance issues that come with these approaches.

2.3.4. Web application client

The users, administrator and managers can connect to the system through the web application. It will use methods like AJAX to avoid page refresh when little operations are done. In this way the response time will be shorter and the user interaction with the system will be faster.

2.4. Deployment view

The deployment view of the system is represented in Figure .
As was specified before the application layer and datastore are deployed on Google Cloud Platform.

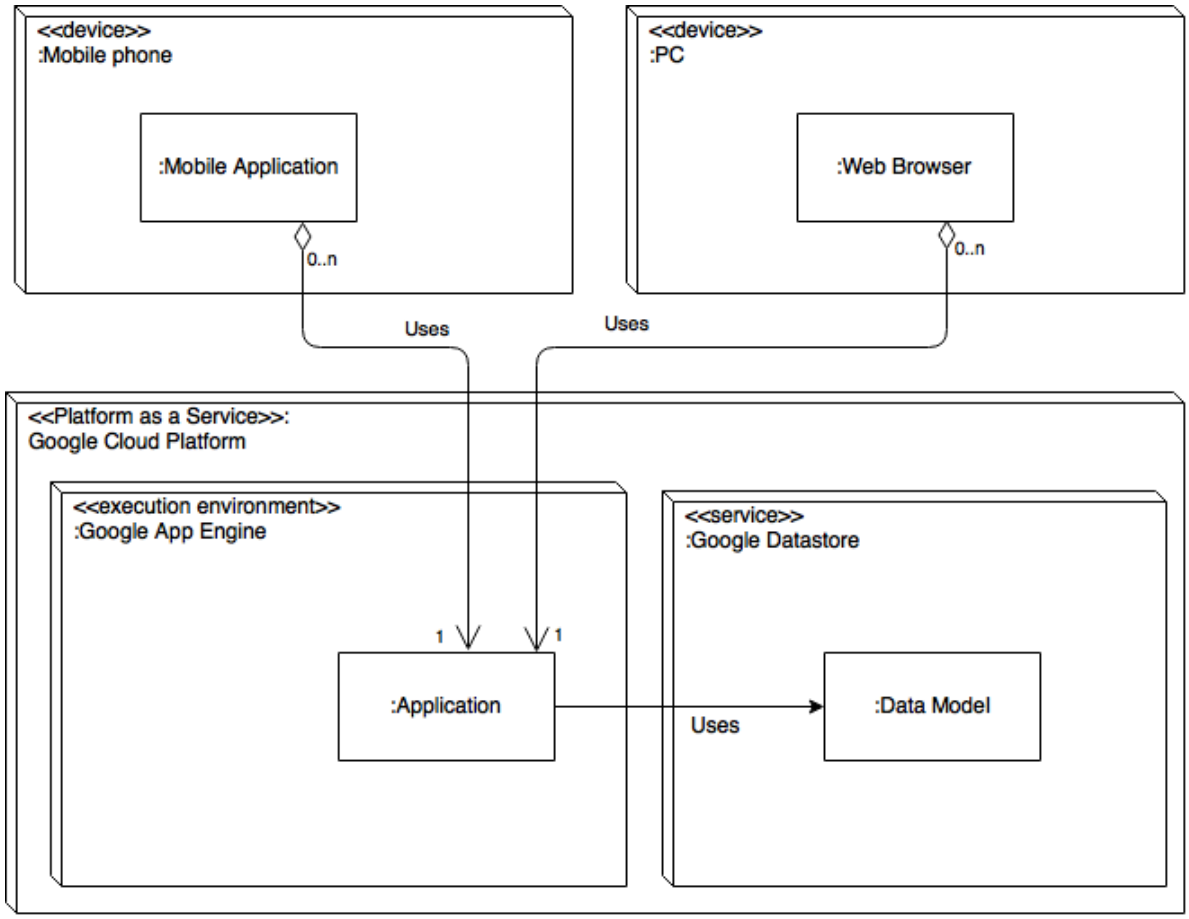

Figure 6. Deployment view

2.5. Runtime view

In this section the dynamical behavior of the system is described.
In particular by means of sequence diagrams is shown how the components described in section 2.3 are interacting with each other to process particular activity. There are represented diagrams for 2 activities chosen by criteria to clarify the most complicated interactions.

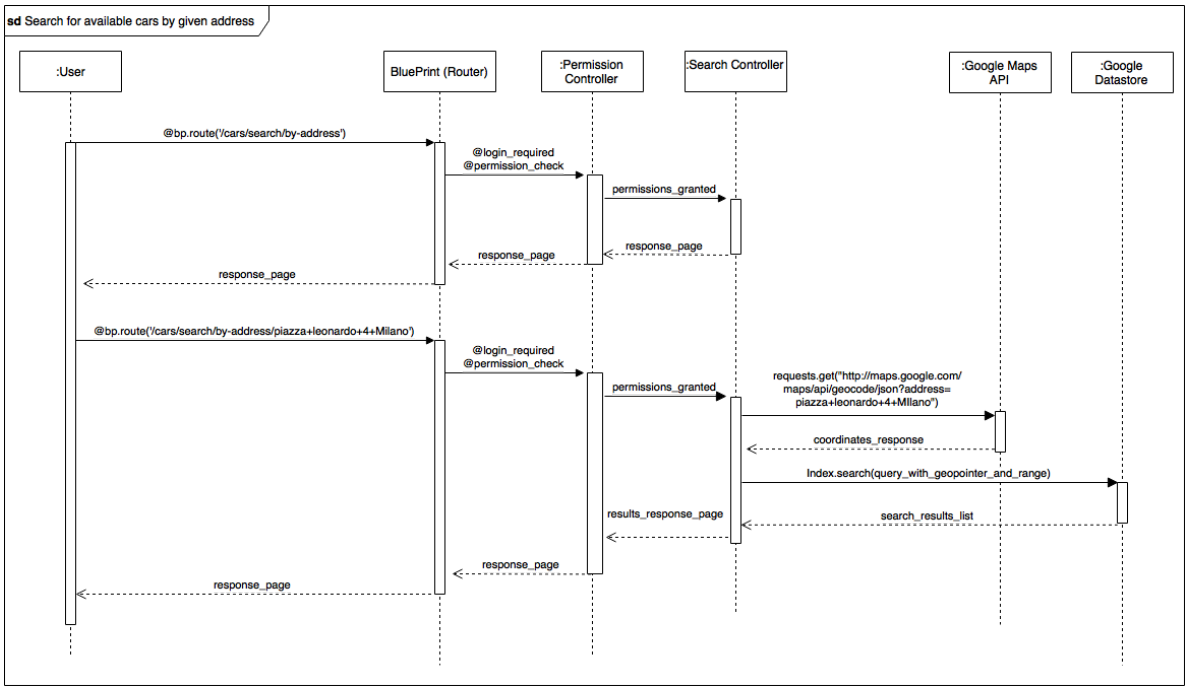

Figure 7. Sequence diagram. Search for car by given address

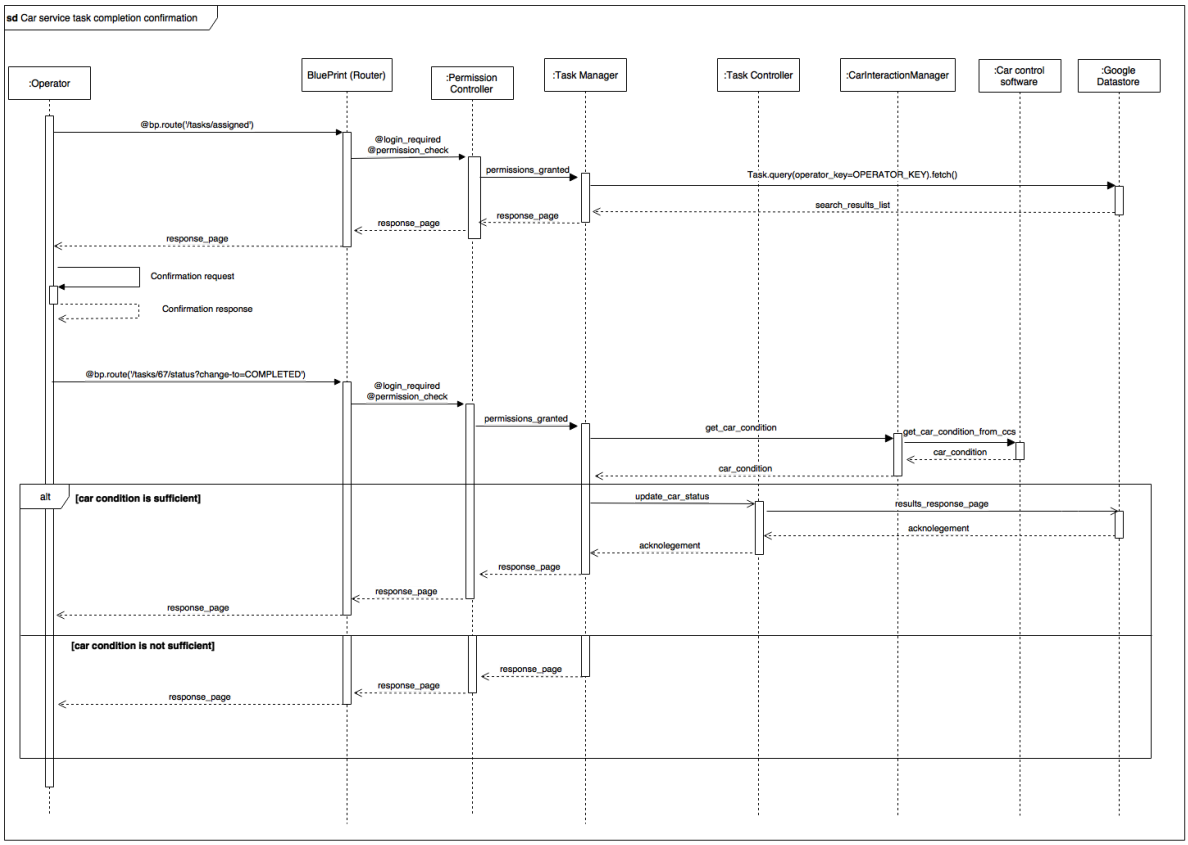

Figure 8. Sequence diagram. Car service task completion confirmation

2.6. Selected architectural styles and patterns

To provide an high level of software maintainability and in order to simplify future developments of other functionalities the software is organized in functional modules. These units, similar to Java packages, organize semantically and by meaning classes, functions and programming code.

Furthermore, since the proposed system architecture relies on a cloud based environment it is suitable to scale as the software work load increase.

The proposed modular structure is convenient to keep an high decoupling level and to encapsulate similar functionalities in order to decrease the dependency between functional modules increasing their reusability.

2.6.1. Overall architecture

The system is a three-tiered application:

1. Datastore (DBMS as a service)
2. Application Server (Application layer)

3. Clients (web application and mobile applications)

2.6.2. Protocols

The system relies on this protocolos to standardize communications between different parts of the software:

- JSON (Language independent data format used to represent information and transmit it between different parts of the system)
- HTTP (Network protocol used from browsers and servers to exchange data such as pages and requests)
- SSL (Secure Socket Layer is used over HTTP to secure connections)
- RESTful approach (Representational state transfer is a set of recommendations used to well organize and structure API semantically in order to improve understandability and maintainability)

2.6.3. Design patterns

MVC. Model-View-Controller pattern is used both for the server side and also for mobile applications. In the application server the Flask framework allows to structure the code with views, controllers and also models. Each functional module contains models, controllers and views.

Client-Server. The Tesla Car Sharing system adopts this pattern to provide the service to users. Users use clients like mobile applications and web application to interact directly with the application server.
This model is also useful to keep separated different part of the software. It helps to maintain in a better way differents part of the whole system like: mobile applications and the application server.

2.7. Other design decisions

2.7.1 Google Maps

The system uses the external services Google Maps to calculate distances between two geolocation point, to obtain precise geolocation coordinates by addresses and to show pins representing cars over a map. The reasons that bring us to choose an external service to operate over a map are the following:

- To develop from scratch a complete and reliable map system for cities is not a viable solution due to the huge effort required. It would be require a dedicated software architecture and many months or years to make maps work well. It is more cheap and with better result to rely on an existing system.

- Google Maps is a service well structured that provides a wide set of APIs that allow most of the needed operations over a map.
- Google Maps is a well-established, tested and reliable software that is the most used around the world.
- Google Maps APIs can be used both on the server side (calculation, accident reporting, geolocation coordinates retrieval, shortest paths) and also on the client side (map visualization, queries for path calculation that not overload the client component).
- The users feel confident with a software component that they already know and use daily.

2.7.2 Password storage

As in every system that take into account security problems, the application server saves in the datastore password in an encrypted version. Passwords are hashed with a dedicated salt hash in order to get, after the cryptographic function execution, a secured version of the password that cannot be reversed. This procedure of hashing and salt is a secure way to store passwords considering also the case of data theft from the database.

2.7.3 Cloud environment

The choice to develop the software over a cloud environment is done to guarantee an high level of availability and reliability of the software. Since the Google Cloud Platform provides service that are distributed and have many mechanisms to be fault tolerant it is supposed that the uptime guaranteed is about around at 99,99%.
Furthermore, the choice of the cloud environment is done also to take advantage of high computational power availability at cheap prices. Indeed, since the environment scale as the system work increase, at the starting of the project maintaining cost will be low.

3. Algorithm design

The following code samples aim to give an idea of how algorithms in the project will be implemented.

3.1. Search cars by given address

```
# - coding: utf-8 -
import json
from google.appengine.api import urlfetch
from google.appengine.api import searchAPI

from flask.blueprints import Blueprint
from flask import render_template, abort

from auth import login_required, permission_check

bp = Blueprint('cars.client', __name__)


@bp.route('/cars/search/by-address/<str:address>')
@login_required
@permission_check
def car_search_by_address(address=None):
    response_dict = None

    # return search page
    if not address:
        return render_template("cars/search/by-address.html", menuid='cars')

    # search by address
    else:

        # get coordinates for the address
        url = 'http://maps.google.com/maps/api/geocode/json?address=' + address

        try:
            result = urlfetch.fetch(url)
            if result.status_code == 200:
                # request goes well
                response_dict = json.loads(string_received)
            else:
                return abort('Wrong request.\nError code from APIs: ' + result.status_code, 400)
        except urlfetch.Error:
            logging.exception('Caught exception fetching url: ' + url + '\n' +
                              'Inserted address: ' + address )

        if not response_dict:
            return abort('Wrong response.', 500)

        # if request to get coordinates went well
        if 'results' in response_dict and \
            'geometry' in response_dict['results'] and \
            'location' in response_dict['results']['geometry']:

            location = response_dict['results']['geometry']['location']
            loc = (location['lat'], location['lng'])

            query = "distance(car_location, geopoint(" + loc[0] + ", " + loc[1] + ")) < 6000"
            try:
                index = searchAPI.Index(config.STORE_INDEX_NAME)
                found_cars = index.search(query)
            except search.Error:
                logging.exception('Caught exception while searching for cars')
                return abort('Something goes wrong during the search.', 500)

            return render_template("cars/search/results.html", menuid='cars', found_cars=found_cars)

        else:
            return abort('Wrong response.', 500)
```

3.2. Operator task completion

```python
# - coding: utf-8 -
from flask.blueprints import Blueprint
from flask import abort, jsonify

from auth import login_required, permission_check
from models import check_car_condition, car_status, get_car_parts_with_errors
from admin import get_car_condition_from_ccs, update_car_status_to_datastore

bp = Blueprint('tasks.client', __name__)


@bp.route('/tasks/<int:task_key>/completed')
@login_required
@permission_check
def task_completed(task_key=None):
    response_d = {'task_completion': '',
                  'car_parts_with_errors': []}

    if not task_key:
        return abort('Bad request', 400)

    car_condition = get_car_condition_from_ccs()

    '''
    example:
    car_condition = {
        'car_parts_condition' = {
            'engine' = 'OK',
            'wheel_pressure' = '2.5atm',
            'oil_level' = '80%',
            'water_level' = '85%',
            'locking_mechanisms' = 'OK'
        }

    }

    '''

    if check_car_condition(car_condition['car_parts_condition']):
        # car condition is OK
        update_car_status_to_datastore(car_status['AVAILABLE'])
        response_d['task_completion'] = 'OK'
        return jsonify(response_d)
    else:
        response_d['task_completion'] = 'KO'
        response_d['car_parts_with_errors'] = get_car_parts_with_errors(car_condition['car_parts_condition'])
        return jsonify(response_d)
```

4. User interface design

4.1. Mockups

4.1.1. Web application interfaces

Login page
The first page of the system is the registration/login page. Unregistered users can start the signup procedure by the button “Sign Up”. If a user chooses for the signup procedure he/she is redirect redirected to another page showing the appropriate form with all the necessary information to insert.
Instead, if an already registered user wants to access to the system he/she has only to fill the login form with email and password and press "Log In".
We create this mock graphical user interface to give an example.

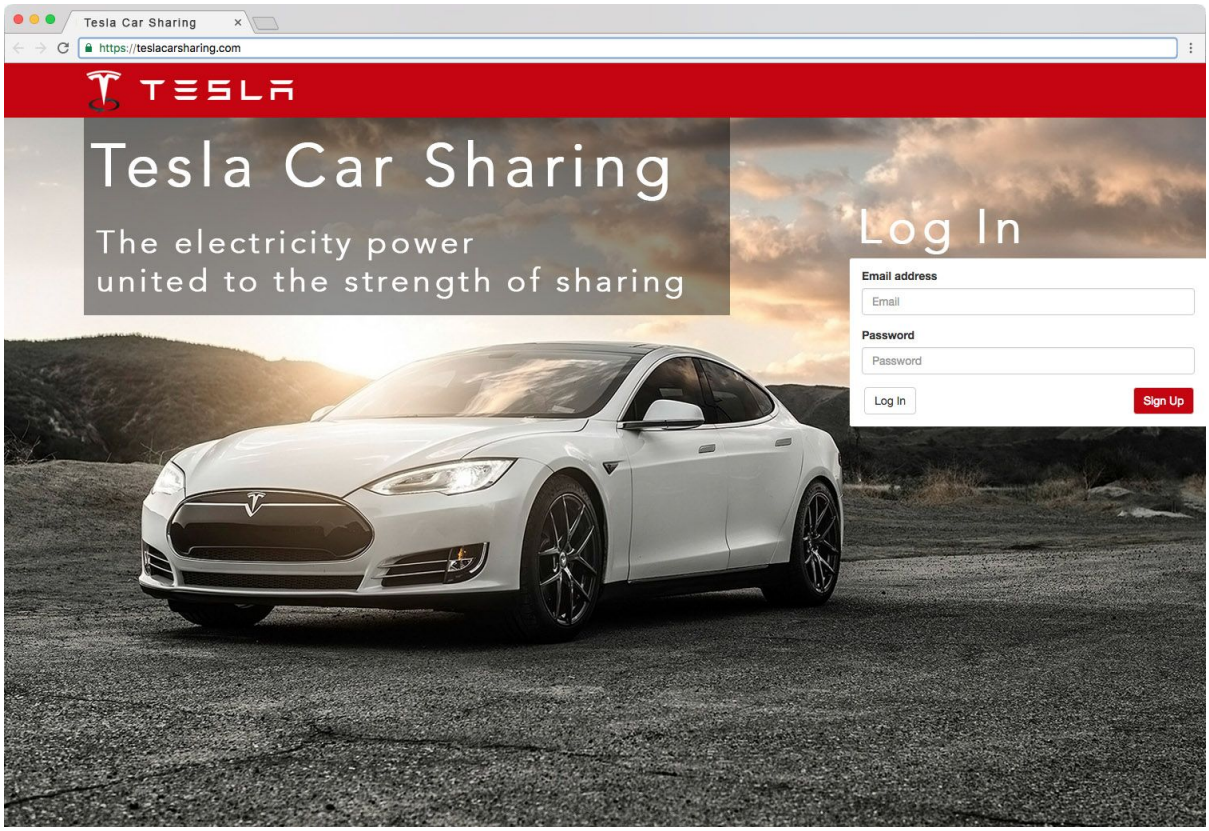

Image 1: Example of the graphical user interface of the login page.

Cars search page

The search page lets the user to search for available cars. He/she can search by an address putting it in the search bar or even by his/her current location by pressing the location button at the bottom right corner of the map.

Search results will appear on the left as a list of available cars. For each car some useful information are reported. For example, the car model, the battery percentage and the car status.

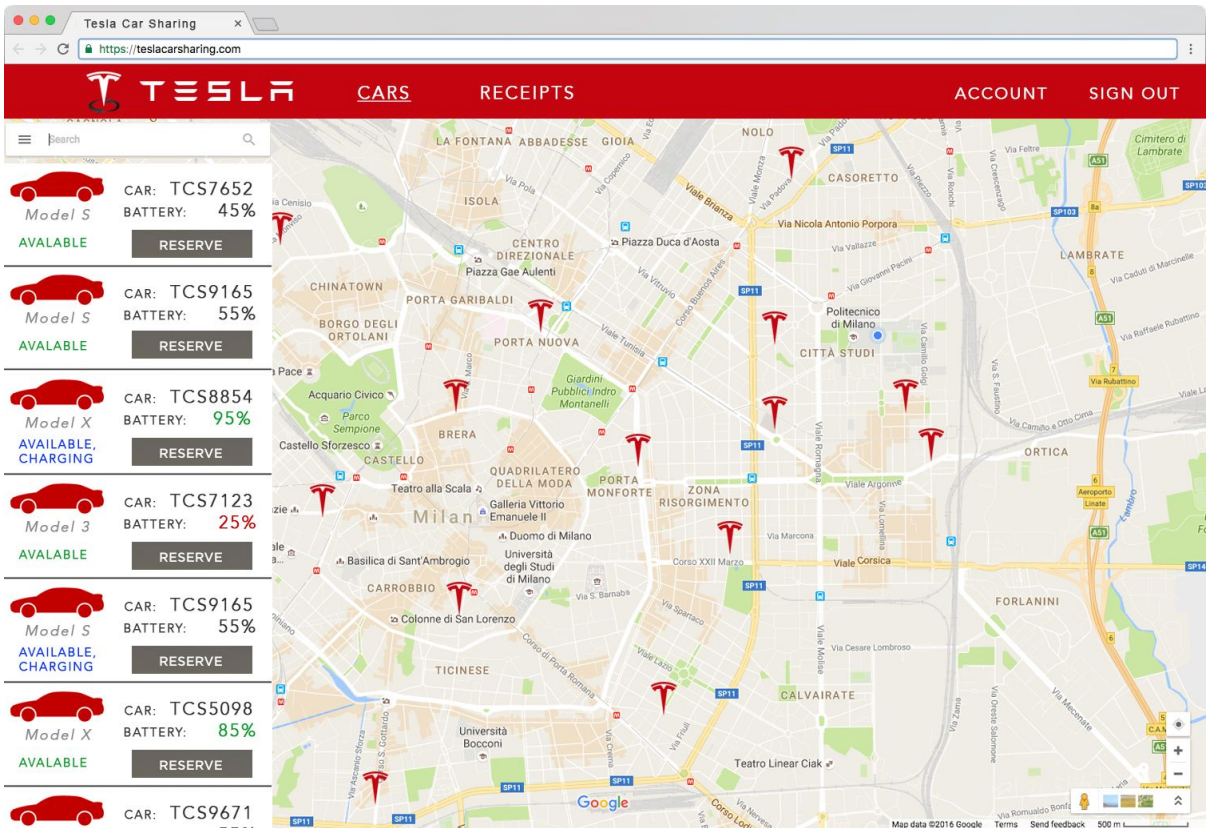

Image 2: Example of the graphical user interface of the car search page.

4.1.2. Mobile application interfaces

Login section

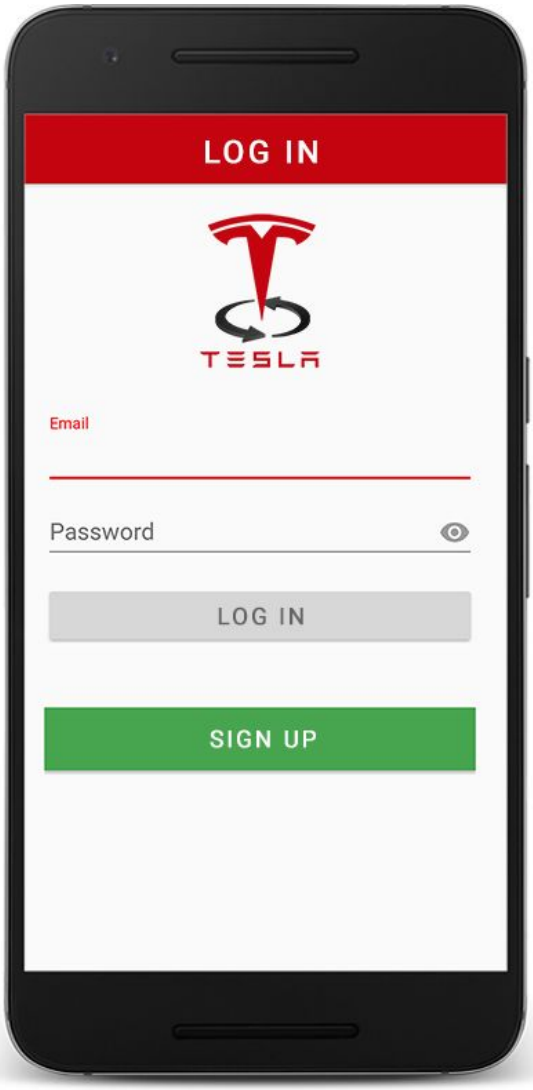

Image 3: Example of the mobile graphical user interface of the login section.

Cars search section

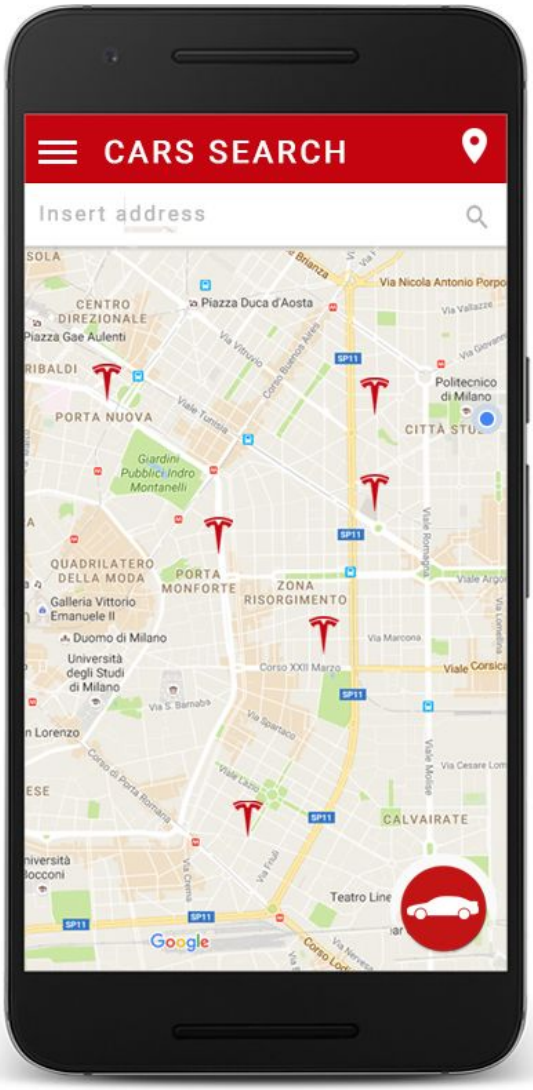

Image 4: Example of the mobile graphical user interface of the car search section.

Car reservation section

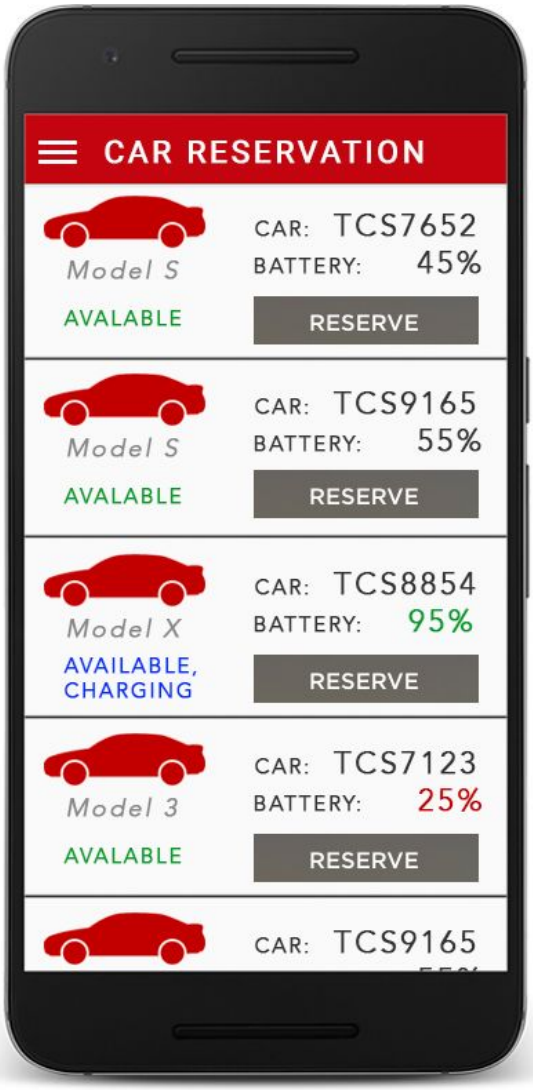

Image 5: Example of the mobile graphical user interface of the car reservation section.

5. Requirements traceability

The design of this project was made with respect to requirements and goals identified in RASD document.
In this section the connection between goals and components of the system is represented.

5.1. Users

[G.1.] Provide users the possibility to be identified by the system.
- LoginManager
- PermissionController

[G.2.] Provide users the possibility to search for available cars.
- SearchController

[G.3.] Provide users the possibility to reserve a car.
- ReservationController

[G.4.] Provide users the possibility to unlock the car when the user is near it.
- ReservationController

[G.5.] Provide the mechanism to charge users for the ride.
- PriceController
- ChargeCalculatorController

[G.6.] Incentivize virtuous behaviours of users.
- ChargeCalculatorController

5.2. Employees

[G.7] Provide employees the possibility to be identified by the system.
- LoginManager
- PermissionController

[G.8.] Provide the possibility for the administrator to manage employees accounts.
- AccountManagementController

[G.9.] Provide the possibility for managers to manage charges.
- PriceController

[G.10.] Provide the possibility for managers to manage car records.
- CarManagementController

[G.11.] Provide the possibility for operators to get information about cars requiring service.
- TaskManager

[G.12.] Provide the possibility for operators to deal with car service tasks.
- TaskManager
- TaskController

6. References

6.1 Used tools

The tools used to realize this DD document are:

- Google Drive
- Google Docs
- Draw.io
- GitHub
- Photoshop

9. Hours of work

9.1. Elena Denchenko

- 18/11/2016: 30 min
- 20/11/2016: 1h 30min
- 21/11/2016: 2h
- 24/11/2016: 3h
- 28/11/2016: 2h
- 29/11/2016: 1h
- 04/12/2016: 4h
- 10/12/2016: 2h
- 11/12/2016: 2h

9.2. Stefano Cilloni

- 18/11/2016: 30 min
- 20/11/2016: 30 min
- 21/11/2016: 2h
- 24/11/2016: 3h
- 28/11/2016: 2h
- 02/11/2016: 1h
- 08/12/2016: 5h
- 10/12/2016: 2h
- 11/12/2016: 2h

10. Changelog