

# Design Document for *myTaxiService*

Version 1.1

Alberto Mario Pirovano

Alessandro Vetere

21/01/2016



**POLITECNICO**  
MILANO 1863

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Purpose . . . . .	3
1.2	Scope . . . . .	3
1.3	Definitions, Acronyms, Abbreviations . . . . .	3
1.4	Reference Documents . . . . .	4
1.5	Document Structure . . . . .	4
<b>2</b>	<b>Architectural Design</b>	<b>5</b>
2.1	Overview . . . . .	5
2.2	High level components and their interaction . . . . .	5
2.2.1	Layers . . . . .	6
2.2.1.1	View . . . . .	6
2.2.1.2	Controller . . . . .	6
2.2.1.3	Model . . . . .	7
2.2.2	Tiers . . . . .	7
2.3	Component view . . . . .	8
2.4	Deployment view . . . . .	16
2.5	Runtime view . . . . .	17
2.6	Component interfaces . . . . .	21
2.7	Selected architectural styles and patterns . . . . .	23
2.8	Other design decisions . . . . .	25
<b>3</b>	<b>Algorithm Design</b>	<b>27</b>
3.1	Taxi Rides Serving Management . . . . .	27
3.1.1	Queue Manager . . . . .	27
3.2	Geolocation . . . . .	30
3.2.1	Point . . . . .	30
3.2.2	Triangle . . . . .	30
3.2.3	Zone . . . . .	32
3.2.4	Zones . . . . .	32
<b>4</b>	<b>User Interface Design</b>	<b>34</b>
4.1	Passenger mockups . . . . .	35
4.2	Taxi Driver mockups . . . . .	39
4.3	Administrator mockups . . . . .	43
<b>5</b>	<b>Requirements Traceability</b>	<b>46</b>
5.1	Registration . . . . .	46
5.2	Login . . . . .	46
5.3	Logout . . . . .	47
5.4	View Request and Reservations . . . . .	47
5.5	Handle Personal Profile . . . . .	47
5.6	Taxi Reservation . . . . .	48
5.7	Taxi Request . . . . .	48

5.8	Notify Problem . . . . .	49
5.9	End of Ride . . . . .	49
<b>6</b>	<b>Appendix</b>	<b>50</b>
6.1	Tools . . . . .	50
6.2	Hours of Work . . . . .	50
6.3	Version History . . . . .	50

# 1 Introduction

This section contains a brief but complete introduction of *myTaxiService* Design Document.

## 1.1 Purpose

The purpose of this document is to explain the Design and Architectural elements that characterize *myTaxiService* as a software system. The level of the description is high enough for all the stakeholders to capture the information they need in order to decide whether the system meets their requirements or in order to begin the development work.

## 1.2 Scope

This document provides a detailed description of *myTaxiService* software design and architectural choices. It consists in natural language descriptions, UML Diagrams and portions of Java code, proposed in a way that should be clear and direct. Every portion of the document is designed itself to be comprehensible, but a big picture of the system must be present to the reader in order to obtain the best knowledge on the matter when consulting this document.

## 1.3 Definitions, Acronyms, Abbreviations

In the document are often used some technical terms whose definitions are here reported:

- **Layer:** A software level in a software system.
- **Tier:** An hardware level in a software system.
- **Relational Database:** A digital database whose organization is based on the relational model of data, as proposed by E.F. Codd in 1970.
- **Cocoa MVC:** A strict application of MVC principles.
- See the correspondent section in the **RASD** for more definitions.

For sake of brevity, some acronyms and abbreviations are used:

- **DD:** Design Document.
- **GPS:** Global Positioning System.
- **GUI:** Graphic User Interface.
- **API:** Application Programming Interface.
- **MVC:** Model View Controller.
- **ETA:** Estimated Time of Arrival.

- See the correspondent section in the **RASD** for more acronyms and abbreviations.

## 1.4 Reference Documents

This document contains references to:

- ***myTaxiService* RASD v1.3:** Requirements Analysis and Specification Document for *myTaxiService*
- **IEEE Std 1016-2009:** IEEE Standard for Information Technology - Systems Design - Software Design Descriptions
- **ISO/IEC/IEEE 42010:** International Standard for Systems and software engineering - Architecture description

## 1.5 Document Structure

This document is divided in 4 main sections:

- **Architectural Design:** This section exposes in details the design chosen for the architecture of the system to be, using user friendly diagrams, UML Diagrams and natural language.
- **Algorithm Design:** Here are exposed some of the algorithms that have been designed in order for the application to work in the desired way. Some Java code is used to show in a more coherent, correct and meaningful way such algorithms.
- **User Interface Design:** In this section are presented some mockups of the system that is going to be developed. In particular, we are presenting mockups for the four GUIs:
  - Passenger Web
  - Passenger Mobile
  - Taxi Driver Mobile
  - Administrator Workstation
- **Requirements Traceability:** In this part of the document are mapped the requirements that were discovered in the RASD to design and architectural decisions.

## 2 Architectural Design

This section exposes *myTaxiService* Architectural Design in a complete e comprehensible way.

### 2.1 Overview

A quick but meaningful overview on the system architectural design is presented with the help of the following diagram, which is self explanatory:

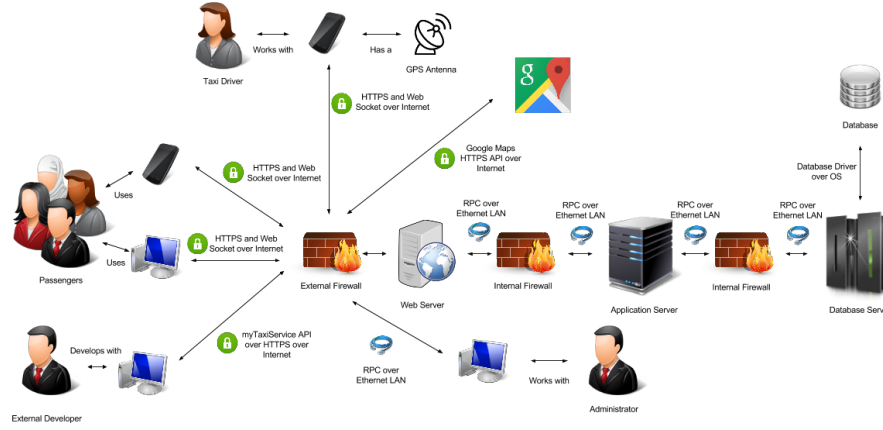


Figure 1: Overview Diagram

In addition are given many **UML Diagrams** integrated with complete descriptions of the software components in the following subsections:

- **Component View:** Describes the division of the system in **Layers** and how the work flow is organized through them. It contains a **UML Component Diagram** that best delivers a comprehensive overlook.
- **Deployment View:** Explains how the division of system in **Tiers** is done with the help of an **UML Deployment Diagram**.
- **Runtime View:** Describes how each software components interacts with each other. To better explain the method call order and which component is used to obtain a given functionality, a set of **UML Sequence Diagram** is presented.

### 2.2 High level components and their interaction

In the following section are deeply covered all the architectural aspects that characterize the software to be. In particular, here it is explained at a high level of abstraction how is designed the division of the system into **Layers** and **Tiers**.

### 2.2.1 Layers

The software architecture that has been chosen follows the principles of the **Model View Controller** architectural pattern. Therefore three main software components have been identified and those are without any coup de theatre: the **Model**, the **View** and the **Controller**. As already mentioned and explained through the **Overview Diagram**, the system has many distributed components: those will communicate with a **Client-Server** style and through **Point to Point** messaging system. The **Client-Server** style is used in order to receive different requests (e.g. a **Taxi Reservation** request) from the many **Clients** connected to the **Server** to use the service. The **Point to Point** bidirectional communication channel is made necessary to enable the **Server** the delivery of various messages and requests to the **Clients**. These messages could be generic notifications, service messages or, in the particular case in which the connected client is a **Taxi Driver**, the request of serving a **Taxi Ride** and the request of an updated **GPS Data**. **Model, View and Controller** are then mapped to three different relevant **Layers**.

**2.2.1.1 View** The role of this **Layer** is the one of processing remote **Clients** commands, and convert them into requests addressed to the **Controller** layer, that is connected to the **View** through a communication facility (e.g. The Internet). There exists four different types of **View**, each one designed specifically for the particular **Client**:

- **Passenger Web View:** Interface developed for the **Passengers**, accessible via a **Web Browser**.
- **Passenger Application View:** Interface developed for the **Passengers**, accessible via a dedicated **Mobile Application**.
- **Taxi Driver Application View:** Interface developed for the **Taxi Drivers**, accessible via a dedicated **Mobile Application**.
- **Administrator View:** Interface developed for *myTaxiService* **Administrator**, accessible via the dedicated **Administrator Workstation**.

**2.2.1.2 Controller** This second **Layer** is composed itself of two families of components with specialized functionalities:

- **Networking Components Family:** This family groups the software components that are involved in sending messages to the various **Views**, following the logic implements in the **Business Components Family**. This is also the entry point for requests generated by the **Views**, that are then dispatched to the various business components.
- **Business Components Family:** In this family are included all the software components that implement the system logic, processing requests and generating either responses or asynchronous events.

**2.2.1.3 Model** The third and last **Layer** is the **Model**, that should guarantee a high level interface to store and manage all the *myTaxiService* relevant data. It abstracts a **Relational Database** in a software component that is in direct connection with the **Controller**.

## 2.2.2 Tiers

Then it comes the division of the system in different **Tiers**:

- **Clients:** The distributed clients of the application. The **View** layer is mapped only to this **Tier**.
- **Web Server:** A server that is in charge of dynamically generate web pages and receive requests through the web. It is also the server where are hosted the **Assets** of the web portal. The **Networking Components Family** of the **Controller** is mapped to this **Tier**.
- **Application Server:** This is the most important **Tier** of the system. Here are done all the logics and calculations that constitute the core part of *myTaxiService*. The **Business Components Family** is hosted on this server and therefore it is a part of the **Controller** that is separated from the **Web Server**. It is necessary to have the **Controller** divided into two different **Tiers** in order to separate the most important software parts, that are the business logic components, from the network components, for security concerns.
- **Database Server:** In this **Tier** it is hosted the **Database** that allows the service data persistence. The **Model** layer is entirely mapped to this **Tier**.

This architecture, with three different **Layers** and four different **Tiers**, makes possible to have a **Client-Server** style with a **Fat-Server** and a **Thin-Client**: this is due to the mapping of the **View** to the **Clients** and the mapping of the **Controller** and the **Model** to the **Server**.



## 2.3 Component view

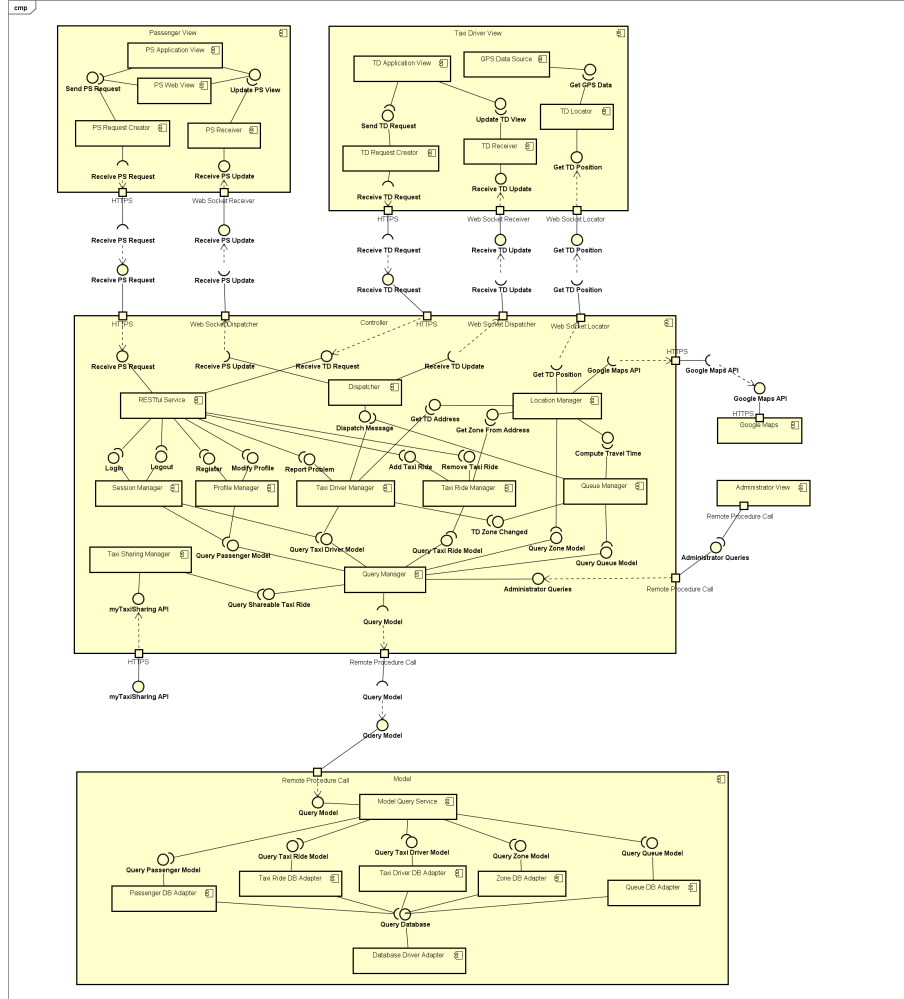


Figure 2: UML Component Diagram

Below are described all the software component that interact inside every subsystem; for the most important ones is also provided a description of the internal interfaces that they use to communicate one with each other. The identified subsystems are:

- **Passenger View:** In this subsystem are contained the following components:
  - **PS Application View and PS Web View:** These are the software components that catch **Passenger** generated events, like the click

on a button. In addition they are capable of showing responses and notifications received via network. It is an **Observable** component, which **Observer** is **PS Request Creator**.

- **PS Request Creator:** This software component translates each **Passenger** generated event into a request that is intended to be sent to the server. It is also capable of handling the server response. Furthermore it is an **Observer** component that is notified when specific events occur.
- **PS Receiver:** This component is in charge of handling asynchronous server notifications or requests. It is an **Observable** component which **Observer** is the **PS Application View** and **PS Web View**.
- **Taxi Driver View:** In this subsystem are contained the following components:
  - **TD Application View:** These are the software components that catch **Taxi Driver** generated events, like the click on a button. In addition they are capable of showing responses and notifications received via network.
  - **TD Request Creator:** See **PS Request Creator**.
  - **TD Receiver:** See **PS Receiver**
  - **TD Locator:** This is the software component that accesses the interface to the **Taxi Driver GPS Data Source**. Once invoked by the **Location Manager**, it retrieves the **GPS Data** that identify the **Taxi Driver Position**. It is a fundamental part of the **Taxi Driver View** because it makes possible for the server to have a constantly updated **Model**.
  - **GPS Data Source:** It is a software component that connects to the **GPS Driver** of the **Taxi Driver Smartphone**. It provides an object that represents the most recent position retrieved by the GPS Antenna; this object contains many data, among which the most important are **Latitude** and **Longitude**. In fact through these data it is possible to determine in a precise way the **Taxi Driver** position.
- **Controller:** In this subsystem are contained the following components:
  - **RESTful Service:** This **Networking Family** component receives requests from the **Clients**, elaborates them and then forwards related requests to components of the **Business Family**. It can handle a given number of requests in parallel: this number is configurable. In order to allow interaction with other components, it implements and exposes the following *interfaces*:
    - \* **Receive TD Request:** Allows **Clients** to send requests.
    - \* **Receive PS Request:** See **Receive TD Request**.

In addition it uses the following *interfaces*:

- \* **Login:** Used to manage a login request made by a **Passenger** or a **Taxi Driver**.
  - \* **Logout:** Used to handle a logout request made by a **Passenger** or a **Taxi Driver**.
  - \* **Register:** Used to manage a registration request made by a **Passenger**.
  - \* **Modify Profile:** Used to manage a profile modification request made by a **Passenger**.
  - \* **Report Problem:** Used to manage a report problem request coming from a **Taxi Driver**.
  - \* **Add Taxi Ride:** Used to handle a **Taxi Request** or **Taxi Reservation** request made by a **Passenger**.
  - \* **Remove Taxi Ride:** Used to manage a **Taxi Ride** cancellation request coming from a **Passenger**.
- **Dispatcher:** This software component is in charge of dispatching notification messages, system messages, and asynchronous requests (e.g. the request of accepting a **Taxi Ride** to a **Taxi Driver**) from the **Server** to the designed **Client**. In order to allow interaction with other components, it implements and exposes the following *interfaces*:
- \* **Dispatch Message:** Allows other **Controller** software components to send many type of messages to the designated **Client**.
- In addition it uses the following *interfaces*:
- \* **Receive PS Update:** Used to make possible for **Clients** to receive messages coming from **Controller** internal software components.
  - \* **Receive TD Update:** See **Receive PS Update**.
- **Location Manager:** This software component implements and exposes the following *interfaces*:
- \* **Get Zone From Address:** Associate a **Zone** of the **City** to a specific **Address**. It is used to categorize a just arrived **Taxi Ride**.
  - \* **Get TD Address:** Allows the **Taxi Driver Manager** to request the actual **Address** of a given **Taxi Driver**, in order to store it in the **Model**.
  - \* **Compute Travel Time:** Allows the calculation of the travel time between two **Addresses**.
- In addition it uses the following *interfaces*:
- \* **Get TD Position:** Used to request **GPS Data** to a given **Taxi Driver**.
  - \* **Google Maps API:** Used to request travel time between two **Addresses** and to associate **GPS Data** to the nearest **Address**.

- \* **Query Zone Model:** Used to query the **Model** about which **Zone** contains a given **Point** defined by the **Longitude** and **Latitude** of a **GPS Data** object. This interface is called to associate the **GPS Data** received from a **Taxi Driver** to a given **Zone**.
- **Queue Manager:** This software component is in charge of:
  - \* Manage **Taxi Driver Queues**, that are one for each **Zone** in the **City**.
  - \* Each **Queue** contains only the **Available Taxi Drivers** in the corresponding **Zone**.
  - \* When a **Taxi Driver** accepts the submitted **Taxi Ride**, this object generates a notification message. This message is clearly addressed to the **Taxi Ride's Passenger** and it contains a reference to the **Taxi Driver** that has just accepted the request and the **Travel Time** remaining for the that **Taxi Driver** to reach the **Passenger** that is waiting. To compute the **Travel Time**, is called the **Location Manager's** dedicated interface.
  - \* When a **Taxi Driver** moves from one **Zone** to another one, this component is in charge of moving him from the old **Queue** to the new one.
  - \* Serves a given number of **Taxi Rides** in parallel. That number is configurable. The exceeding request are put into a queue waiting to be served.
  - \* Redistribute the **Taxi Rides**. When the **Queue** associated to the starting **Zone** of a served **Taxi Ride** is empty, the system waits a given timeout. After that if no **Taxi Driver** can be found in the current **Queue**, the **Queue Manager** searches for a **Taxi Driver** in **Queues** of the neighbour **Zones**.
  - \* Redistribute **Taxi Drivers**. When a **Taxi Driver** refuses a **Taxi Ride** request, it is dequeued from the current **Queue** and then enqueued in the same **Queue**: this moves him to the last position.

This software component implements and exposes the following *interfaces*:

- \* **TD Zone Changed:** Allows the **Taxi Driver Manager** to notify that a given **Taxi Driver** has moved to a different **Zone**.

In addition it uses the following *interfaces*:

- \* **Query Queue Model:** Used for moving a **Taxi Driver** from a **Queue** to another one, when he/she has changed **Zone**. In addition this interface allows to pick a **Taxi Driver** from the related **Queue** and associate he/she to a given **Taxi Ride**.
- \* **Compute Travel Time:** Used to calculate the **Travel Time** needed by the **Taxi Driver** for moving from his/her actual position to the **Taxi Ride Start Address**.

- \* **Dispatch Message:** Used to send a notification to a **Taxi Driver** that has been associated to a given **Taxi Ride**.
- **Session Manager:** It is in charge of handling the sessions of *myTaxiService* users. This software component implements and exposes the following *interfaces*:
  - \* **Login:** Allows a **Client** to login to *myTaxiService* and therefore start a valid session.
  - \* **Logout:** Allows a **Client** to logout from *myTaxiService* and therefore terminate the current session.

In addition it uses the following *interfaces*:

  - \* **Query Passenger Model:** Used to manage **Passenger** active sessions (e.g. retrieve valid session tokens) in the **Model**.
  - \* **Query Taxi Driver Model:** Used to manage **Taxi Driver** active sessions (e.g. retrieve valid session tokens) in the **Model**.
- **Profile Manager:** It handles the registration of a new **Passenger** to the system and the modification of a given **Registered Passenger**'s profile. This software component implements and exposes the following *interfaces*:
  - \* **Register:** Allows registration to *myTaxiService* for **Non Registered Passengers**.
  - \* **Modify Profile:** Allows profile modification to **Registered Passengers**.

In addition it uses the following *interfaces*:

  - \* **Query Passenger Model:** Used to manage the **Model** part related to **Registered Passenger**, inserting new entries or modifying existing ones.
- **Taxi Driver Manager:** It is in charge of manage all the business logic behind **Taxi Drivers**. This is a core component that, among the other things, periodically asks **GPS Data** to **Taxi Drivers**. It implements and exposes the following *interfaces*:
  - \* **Report Problem:** Allows **Taxi Driver** either to report a **Problem** or to communicate its subsequent **Solution**.

In addition it uses the following *interfaces*:

  - \* **TD Zone Changed:** Used as a shortcut to notify the **Queue Manager** that a given **Taxi Driver** has moved from one **Zone** to another one.
  - \* **Query Taxi Driver Model:** A useful interface to the **Model**. It is used in the following cases:
    - When the update of a given **Taxi Driver** geolocation data is retrieved, in order to maintain coherent the **Model**.
    - When a **Problem** is reported, in order to store it in the **Model**.

- When a **Problem** is notified as **Solved** by the relevant **Taxi Driver**, in order to update the **Model**.
  - \* **Dispatch Message**: Used to send personalized messages to a given **Taxi Driver**.
  - \* **Get TD Address**: Used to ask a given **Taxi Driver** his/her current location.
- **Taxi Ride Manager**: It is in charge of manage all the business logic behind **Taxi Rides**. This software component implements and exposes the following *interfaces*:
  - \* **Add Taxi Ride**: Allows to add **Taxi Ride** request to the **Model**. A **Taxi Ride** can be of two different types: **Taxi Request** and **Taxi Reservation**. The software behaves differently, basing of the request nature:
    - **Taxi Request**: The request is added to a dedicated data structure in the **Model**, and it will be then served as soon as possible by the **Queue Manager**.
    - **Taxi Reservation**: The request is added to a different dedicated data structure in the **Model**, and a timer is set to wake up a routine that, 10 minutes before the **Start Time** of the given **Taxi Reservation**, builds a **Taxi Request** out of the same **Taxi Reservation**, preserving all the relevant data. The timer uses the interface itself to perform its task.
  - \* **Remove Taxi Ride**: Allows **RESTful Service** to remove a **Taxi Ride** request already present in the **Model**.

In addition it uses the following *interfaces*:

- \* **Get Zone From TD Address**: Used to compute the **Zone** to which a given **Address** belongs, in order to associate a **Starting Zone** to a given **Taxi Ride**. This is a core procedure of *myTaxiService* application because querying the **Zone** field of a given **Taxi Ride** it is possible to know in which **Queue** should look for a **Taxi Driver**.
  - \* **Query Taxi Ride Model**: Used to add and remove **Taxi Rides** from the **Model**.
- **Taxi Sharing Manager**: It is in charge on handling all the necessary operation related to **Taxi Sharing**. This software component implements and exposes the following *interfaces*:
  - \* **myTaxiSharing API**: Allows **External Developers** to access *myTaxiSharing* functionality through the use of standard set of programmatic APIs.

In addition it uses the following *interfaces*:

- \* **Query Shareable Taxi Ride**: Used to manage the **Model** part that is related to **Shareable Taxi Rides**.

- **Query Manager:** It is a **Controller** component that acts as a uniform layer built above the **Model**, that is in fact located in to a different machine. This software component implements and exposes the following *interfaces*:
  - \* **Query Queue Model:** Allows the **Queue Manager** to manage **Queues**.
  - \* **Query Zone Model:** Allows the **Location Manager** to retrieve the **Zone** associated to a given **GPS Data**.
  - \* **Query Passenger Model:** Allows the **Session Manager** and the **Profile Manager** to manage the data related to **Registered Passengers**, and insert new ones.
  - \* **Query Taxi Driver Model:** Allows the **Taxi Driver Manager** to retrieve, store and modify data related to **Taxi Drivers**.
  - \* **Query Taxi Ride Model:** Allows the **Taxi Ride Manager** to read, update and insert data related to **Taxi Rides**.
  - \* **Query Shareable Taxi Ride:** Allows the **Taxi Sharing Manager** to manage data related to **Shareable Taxi Rides**.
  - \* **Administrator Queries:** Allows the **Administrator** to perform queries related to **Taxi Driver** profile management and **Taxi Driver** registration.

In addition it uses the following *interfaces*:

- \* **Query Model:** Used to forward the method calls received by this component to the **Model** itself.
- **Model:** In this component are included the following subcomponents:
    - **Model Query Service:** This component provides an aggregated interface rich of methods to access the relational model underlying with a high level of abstraction, protection and security. The interface that it implements is called Query Model and it is used by the Query Manager to query the model via RPC.
    - **Passenger DB Adapter:** It is a utility component that implements and exposes Passenger related query methods, built above the Query Database interface to the Database Driver Adapter.
    - **Taxi Ride DB Adapter:** It is a utility component that implements and exposes Taxi Ride related query methods, built above the Query Database interface to the Database Driver Adapter.
    - **Taxi Driver DB Adapter:** It is a utility component that implements and exposes Taxi Driver related query methods, built above the Query Database interface to the Database Driver Adapter.
    - **Zone DB Adapter:** It is a utility component that implements and exposes Zone related query methods, built above the Query Database interface to the Database Driver Adapter.

- **Queue DB Adapter:** It is a utility component that implements and exposes Queue related query methods, built above the Query Database interface to the Database Driver Adapter.
- **Database Driver Adapter:** This component is a wrapper built on the Database Driver (that can be JDBC) that provides a set of methods to perform the common operations that could be done on a relational database. In fact it connects to the Database Management System via a Database Driver. Those methods are exposed in a interface that is called Query Database, and the various DB Adapters calls them in a way that depends on the caller component itself.



## 2.4 Deployment view

In this subsection is proposed a **UML Deployment Diagram** that shows how the software components are mapped into different **Tiers** that compose the system.

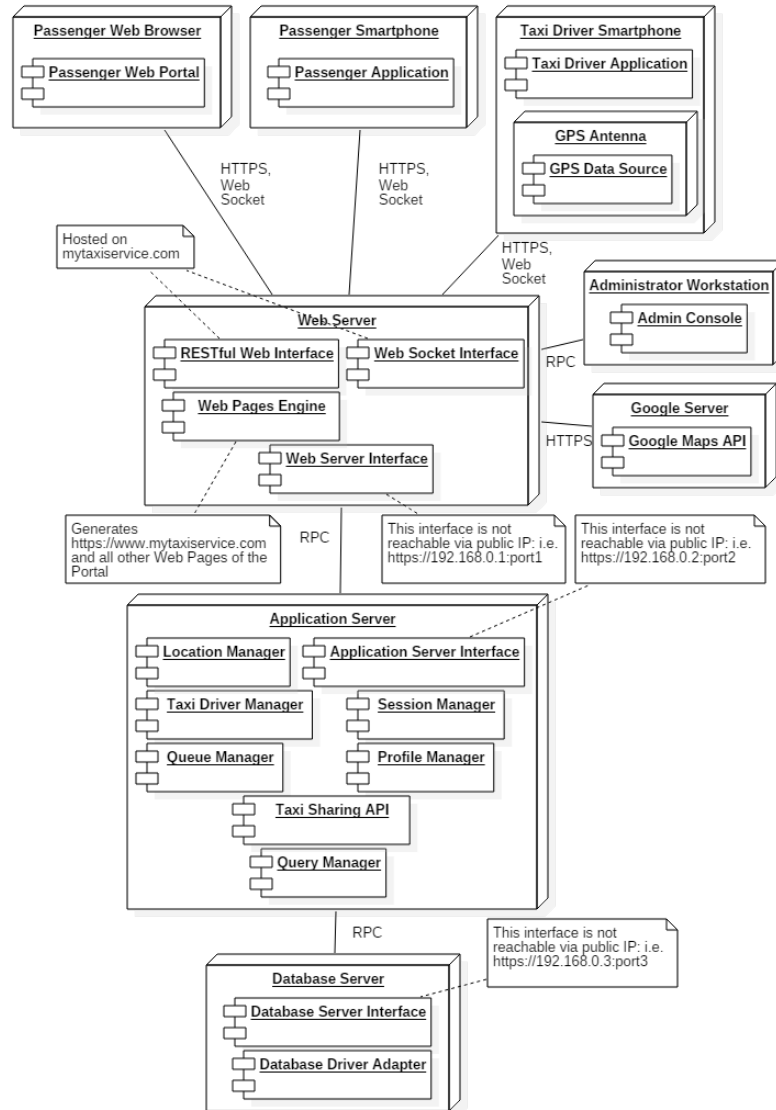


Figure 3: UML Deployment Diagram

## 2.5 Runtime view

In this subsection is proposed a **UML Sequence Diagrams** that shows how the software components that compose the system interacts in order to deliver the requested functionality.

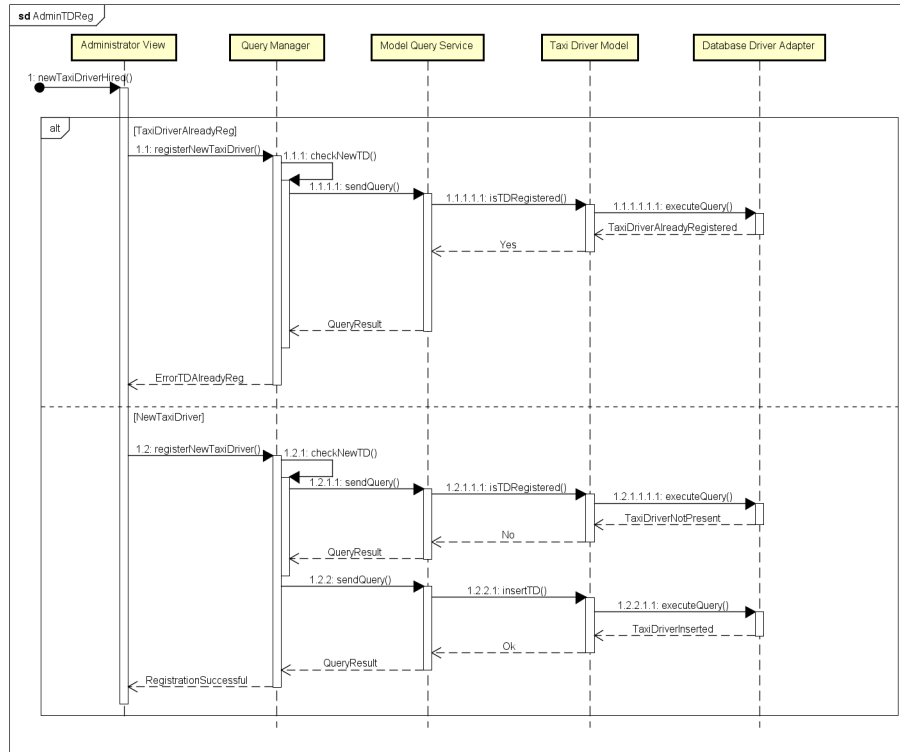


Figure 4: UML Sequence Diagram - Taxi Driver Registration

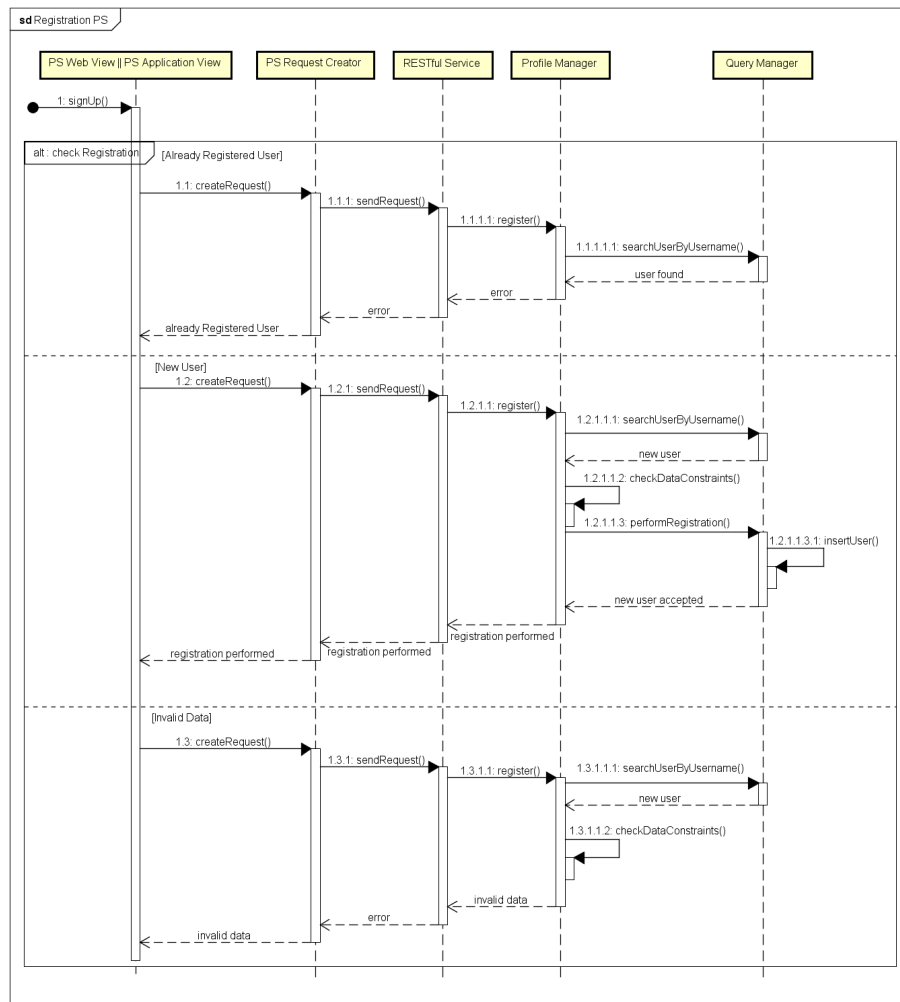


Figure 5: UML Sequence Diagram - Passenger Registration

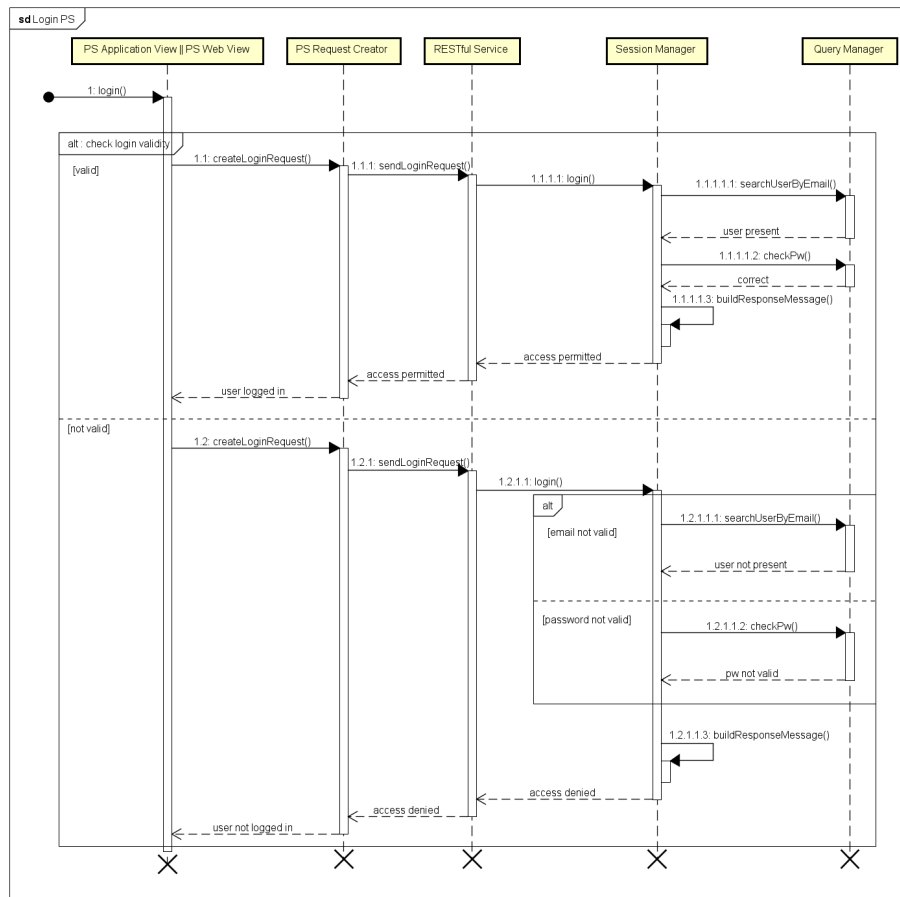


Figure 6: UML Sequence Diagram - Passenger Login

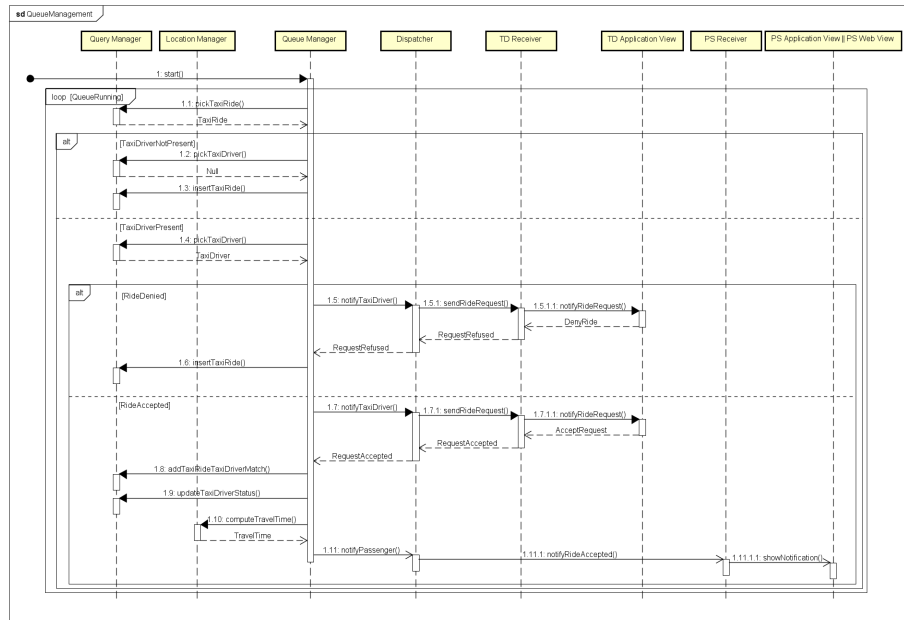


Figure 7: UML Sequence Diagram - Queue Management

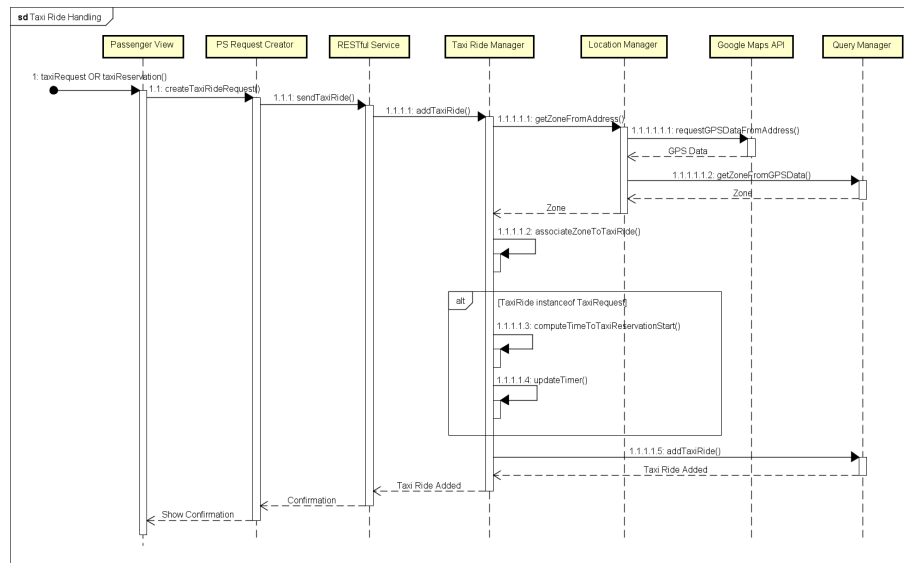


Figure 8: UML Sequence Diagram - Taxi Ride Handling

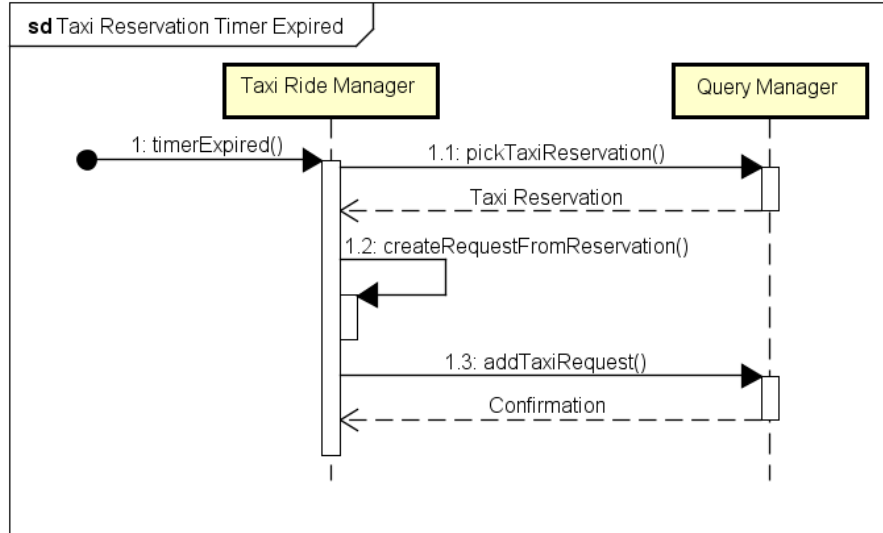


Figure 9: UML Sequence Diagram - Taxi Reservation Timer Expired

## 2.6 Component interfaces

Being our system composed of different software components that need to interact together, several interfaces have been put among them. Subsequently are listed all the main software components and their interfaces, each one described in detail:

- **Passenger View:** This component implements one interface:
  - **Receive PS Update:** An interface over the **Web Socket** technology that is implemented by the **Passenger View** and called by the **Controller** in order to deliver updates to a particular Passenger. It is required that the **Passenger View** opens a **Web Socket** connection with the **Controller** in order to permit him of sending personal messages. Message delivery is intended to be asynchronous but in some cases the **Controller** waits, in the same communication channel, the response of the **Passenger View**. In this case the communication is synchronous.

This component utilizes one interface:

- **Receive PS Request:** An interface implemented by the **Controller** that exposes the necessary methods to build up a **RESTful Service** over the **HTTPS** protocol. Through this interface the **Passenger View** can send specific requests to the **Controller**, that synchronously responds with the necessary information.

- **Taxi Driver View:** This component implements two interfaces:
  - **Receive TD Updated:** As for the **Receive PS Update** interface, this interface over **Web Socket** technology is called by the **Controller** to deliver messages and requests to a particular Taxi Driver. It is required that a **Web Socket** connection is opened by the **Taxi Driver View** with the **Controller** in order to permit it to send messages. The delivery of message is asynchronous when a direct answer by the **Taxi Driver View** is not needed, in other cases the **Controller** makes a synchronous call to the **Taxi Driver View** and waits for it to send back the desired data.
  - **Get TD Position:** This interface over **Web Socket** is called synchronously by the **Controller** to get the last position registered in the **Taxi Driver View**. This could be a crucial part of the system and therefore it is intended that no Taxi Driver hacks his own smart-phone GPS location system in order to deliver wrong position (maybe introducing an important fee to be paid in case of this misbehaviour). The **Taxi Driver Web** must open a **Web Socket** connection with the **Controller** in order to permit him to ask the position.

This component utilizes one interface:

- **Receive TD Request:** An interface over **HTTPS** that is implemented by the **Controller** and like the **Receive PS Request** does for the Passenger, allows the Taxi Driver to send request to the **Controller**, while waiting synchronously for a response.
- **Administration View:** This component utilizes one interface:
  - **Administrator Queries:** This interface is provided by the **Controller** via **RPC** technology (whose implementation could be **RMI**) and allows the **Administrator View** to perform some crucial activities and queries on the *myTaxiService* model in a more direct and powerful way. Through this interface it is allowed the registration of a new Taxi Driver. This interface is not intended to be publicly accessible, and therefore it is better not to expose it through a public IP. In addition, because of the very direct and uncontrolled nature of this communication channel, it is intended that the Administrators are trained personnel and are very discouraged to damage or break the system they are interacting with.
- **Model:** This component implements one interface:
  - **Query Model:** It is an interface over **RPC** technology (could as well be **RMI**) that allows the **Controller** to do all kind of interesting relational queries on the *myTaxiService* model, through a set of exposed and controlled methods. It abstracts the concept of relational query to a higher level of usability and security. The model

to which the interface refers is at its core a relational database, that is accessed through a **Database Driver Adapter**, that could wrap up a database driver like **JDBC**.

- **Controller:** This is the central part of *myTaxiService* system and has the greatest number of implemented and utilized interfaces. It implements 4 different interfaces:
  - **myTaxiSharing API:** This interface provided through **HTTPS** is intended to be utilized by the developers that wants to further develop *myTaxiService*, by adding the Taxi Sharing capability. It is not intended to be accessible publicly and therefore it must be protected using firewalls and access control management, with credential authentication. A developer that wants to utilize the API, must request the permission to the Administrator, that creates valid credentials, configures the system accordingly, and communicate him/her those credentials.
  - **Receive PS Request:** Already presented above in **Passenger View** interfaces analysis.
  - **Receive TD Request:** Already presented above in **Taxi Driver View** interfaces analysis.
  - **Administrator Queries:** Already presented above in **Administrator View** interfaces analysis.

## 2.7 Selected architectural styles and patterns

Several architectural styles and patterns were chosen in order to build *myTaxiService* as a modern software. The main pattern that was recursively adopted is the **Model View Controller** architectural pattern:

- **System Level:** All the clients that use *myTaxiService* (i.e. the **Passengers**, the **Taxi Drivers**, and the **Administrator**) are seen as **Views**, that following the **Cocoa MVC** pattern, are connected to a **Controller**, the **Web Server** or the **Application Server**, that is itself connected to the **Model** that is hosted on the **Database Server**.
- **Client Level:** The **Views** in the various **Clients** are designed internally to conform to the **MVC** pattern. For instance the **Passenger View** is composed of a **Model** that is an internal representation of the data received by the **Web Server**, a **Controller** that performs network activities and connects the **Model** with the **View**, and the **View** that is either a **Mobile Application GUI** or a **Web Page**.
- **Server Level:** The server part of *myTaxiService* is divided also physically into the three different entities of the **MVC** pattern:



- **Web Server:** It is the **View** of the server part of the application, and gives to clients a representation of the internal data. The **Web Server** is connected only to the **Application Server**. It contains all the assets included in the web portal. On its external interface, it can handle properly **HTTPS** requests and manage communication over **Web Sockets**. On its internal interface it sends and receives objects via **RPC** from the **Application Server**, that is in charge of the application logic.
- **Application Server:** That is the **Controller** of the server and it is connected with the **Web Server** and the **Database Server** via **RPC**. It contains most of the business logic behind *myTaxiService* and therefore it is a core part of the system. It should be reachable only within the enterprise network, and not via public IP. The connection to the **Web Server** is used to receive requests, to deliver responses to clients and to send asynchronous requests to them, when it is needed. On the other hand is necessary a connection with the **Database Server** in order to connect the **Controller** with the **Model**. It is also used to store all the important data and react to some data modifications that occurs during the execution of the application.
- **Database Server:** It is the **Model** of the server, and in fact is the **Model** of the whole *myTaxiService* system. It relies on an **RPC** connection with the **Application Server** and it is not directly reachable via a public IP. The **Database Server** is connected to a relational database via a **Database Driver Adapter**, that is a wrapper built over the **DBMS** installed on the **Database Server Operating System**. Following the **Cocoa MVC** pattern, it is not directly connected to the **View** of the server, that is the **Web Server**, but only to the **Controller**, that is the **Application Server**.

Alongside the **MVC** architectural pattern, also the **Client-Server** style is used for all the requests done by the various clients connected to the **Web Server** of *myTaxiService*. The **Taxi Driver Application** and the **Passenger Application** can use a standardized **Client-Server** protocol via **HTTPS** that follows the principle of a **RESTful Service**. The **Administrator** application is connected via **RPC** to the **Web Server** and can perform more critical requests, like the registration of a new **Taxi Driver** into the system. It is required that the **Administrator** application opens a **RPC** connection to the **Web Server** to start the communication. A **Point to Point** bidirectional messaging system is established between the **Clients** and the **Web Server** at the *boot* of the client application. The client should explicitly request a connection to the server that is listening for clients' connections. It is the connection over **Web Socket** protocol that allows the **Web Server** to send asynchronous messages and requests to which the client can respond using the same channel. The main reasons why this protocol is used are sending a **Taxi Ride** proposal to a given **Taxi Driver**, that can either accept or deny the proposal, and allowing

the server to ask the **Taxi Driver** an updated geolocation data. The **Client-Server** style and **Point to Point** bidirectional messaging system are used to implement properly the **MVC** pattern in this three **Layers**, four **Tiers** system. The three **Layers** have already been discussed and are:

- **Data Layer:** That is the System Level **Model**.
- **Business Logic Layer:** That is the System Level **Controller**.
- **Presentation Layer:** That is the System Level **View**.

These **Layers**, as already said, are mapped into four different **Tiers**:

- **Clients:** They map the System Level **View**.
- **Web Server:** It maps a portion of the System Level **Controller**, that is the one of the **RESTful Service**, **Dispatcher**, and part of the **Location Manager**, onto software objects that provide the required network capabilities.
- **Application Server:** This **Tier** maps the purely logical part of the System Level **Controller**.
- **Database Server:** It maps the System Level **Model**.

## 2.8 Other design decisions

Several technologies have been chosen in order to best fit the needs of the system to be. Not all the required functionalities of *myTaxiService* are already mapped onto specific products because in those cases the choice done would matter less. But for the cases in which a technology has already been proposed, it is because a clear design decision was mandatory. As for the communication protocols between clients (excluded the Administrator client) and the server have been chosen:

- **HTTPS:** The secure version of **HTTP** was a mandatory choice as security and privacy concerns are of major importance nowadays.
- **Web Socket:** This innovative socket technology has been chosen although is relatively new because it implements a full duplex socket communication channel using web technology and therefore using the port 80, which is in almost every case not blocked by any firewall.

For what concerns the network reachability has been chosen to make discoverable only the Web Server through the assignment to it of a public IP. All the other servers in *myTaxiService* system should be reachable only within the enterprise network. Between the **Web Server** and the external network is installed a firewall that controls all the incoming connections. In particular it must accept only incoming **HTTPS** connections, **Web Socket** connections and **RPC** connections. A firewall is also used to protect the Database Server from

the **Application Server** in the unlikely case that the **Application Server** is attacked through the **Web Server** or the **Application Server** for some reasons stops working correctly and start behaving in a way that will damage the application **Model**. Also, another interesting design choice that has been made concerns the way in which the **GPS coordinates** obtained from a given **Taxi Driver** are mapped into a specific **Zone**. It could have been possible, but not feasible, to do such a thing:

1. Obtain **GPS Data** via **Web Socket** from the selected **Taxi Driver**.
2. Calculate the nearest **Address** of the given **GPS Data** using **Google Maps HTTPS API**.
3. Query the **Model** to obtain the **Zone** to which belongs the given **Address**.

But this solution requires to have a precomputed data structure that associates every **Address** in the **City** to the corresponding **Zone** (that could have been a relational table with as many rows inside as **Addresses** in the **City**, each address associated with the corresponding **Zone**), that is heavy to manage and maintain, although if correctly installed and filled, it gives for certain good performances. A less heavy weight solution has been found: this solution expects every **Zone** of the **City** to be divided in several convex **Polygons**, for instance **Triangles**, that have interesting properties for our application. In *myTaxiService*, **Zones** of regular shape are intended to be designed, and therefore the number of **Triangles** in which a **Zone** should be decomposed is very limited. So, such a flow is followed:

1. Obtain **GPS Data** via **Web Socket** from the selected **Taxi Driver**
2. For each **Zone**, check if the **Point** that the **Longitude** and **Latitude** from **GPS Data** identify is contained inside any **Triangle** in which the **Zone** is divided. If that is the case, then the **Zone** is found. If that is not the case, then another **Zone** could contain the given Point. If no **Zone** contains the Point, then we can assume that the **Point** refers to **GPS Data** that identify a geographical point outside of the **City**.

The computation of the **Point in Triangle** test is simple and efficient (e.g. using barycentric coordinates) and an algorithm designed in **Java** is provided in the following section.

## 3 Algorithm Design

This section presents two of the most important algorithms used in *myTaxiService* software system. The algorithms are reported with a level of detail that is enough high to give software developer a predefined route to follow when developing the system.

### 3.1 Taxi Rides Serving Management

In this subsection is presented some Java code that implements in a naive but meaningful way the algorithm that is used to handle the association of a Taxi Ride to a Taxi Driver.

#### 3.1.1 Queue Manager

---

```
package mts.queue;

import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * This class manage Queues related tasks.
 */
public class QueueManager {
    private class ManageRide implements Runnable {
        private final TaxiRide taxiRide;

        protected ManageRide(TaxiRide taxiRide) {
            this.taxiRide = taxiRide;
        }

        @Override
        public void run() {
            manageTaxiRide(taxiRide);
        }
    }

    private static final int TD_SEARCH_TIMEOUT_SEC = 10;
    private static final int RESPONSE_TIMEOUT_SEC = 10;
    private static final int PARALLEL_TASKS = 15;

    private final QueryManager queryManager;
    private final Dispatcher dispatcher;
    private final LocationManager locationManager;

    public QueueManager(QueryManager queryManager, Dispatcher dispatcher,
        LocationManager locationManager) {
```

```

        this.queryManager = queryManager;
        this.dispatcher = dispatcher;
        this.locationManager = locationManager;
    }

    private TaxiDriver getTaxiDriver(Zone taxiRideStartingZone) {
        // Search for an available Taxi Driver in the Taxi Ride Starting
        // Zone,
        // If found removes it from the queue and returns it;
        // BLOCKING METHOD, NULL returned after TD_SEARCH_TIMEOUT
        TaxiDriver taxiDriver =
            queryManager.dequeueTaxiDriver(taxiRideStartingZone,
            TD_SEARCH_TIMEOUT_SEC);

        // Taxi Driver found in Taxi Ride Starting Zone
        if (taxiDriver != null) {
            return taxiDriver;
        }

        // Taxi Driver not found in Taxi Ride Starting Zone
        // Search in near zones
        Set<Zone> nearZones = taxiRideStartingZone.getAdjacentZones();
        for (Zone nearZone : nearZones) {
            // Search for an available Taxi Driver in the selected Zone -
            // If found removes it from the queue and returns it;
            // BLOCKING METHOD, NULL returned after TD_SEARCH_TIMEOUT
            taxiDriver = queryManager.dequeueTaxiDriver(nearZone,
            TD_SEARCH_TIMEOUT_SEC);
            // Taxi driver found
            if (taxiDriver != null) {
                return taxiDriver;
            }
        }

        // No suitable Taxi Driver can be found
        return null;
    }

    private void manageTaxiRide(TaxiRide taxiRide) {
        // This method call searches for a suitable Taxi Driver
        TaxiDriver taxiDriver = getTaxiDriver(taxiRide.getStartingZone());

        // This means that no suitable Taxi Driver has been found
        if (taxiDriver == null) {
            // Need to reinsert the Taxi Ride in the pending list
            queryManager.insertTaxiRide(taxiRide);
            return;
        }

        // Ask Taxi Driver to accept the Taxi Ride Request
    }

```

```

boolean accepted = dispatcher.notifyTaxiDriver(taxiDriver,
        taxiRide, RESPONSE_TIMEOUT_SEC);

if (!accepted) {
    // Need To reinsert the Taxi Ride in the pending list
    queryManager.insertTaxiRide(taxiRide);
    // Reinsert the Taxi Driver in the Queue, so that he/she will
        be in
    // the last position of that Queue
    queryManager.enqueueTaxiDriver(taxiDriver);
} else // Request accepted
{
    // Add the taxiRide - taxiDriver match to the model
    queryManager.addRideDriverMatch(taxiRide, taxiDriver);
    // Set his/her status to WORKING
    queryManager.updateTaxiDriverStatus(taxiDriver,
        TaxiDriverStatus.WORKING);
    // Compute the Taxi Driver ETA
    int travelTime =
        locationManager.computeTravelTime(taxiDriver.getCurrentAddress(),
            taxiRide.getStartAddress());
    // Notify the TD ETA to the interested Passenger
    dispatcher.notifyPassenger(taxiRide.getPassenger(), taxiRide,
        travelTime);
}
}

//This method starts the activity of serving Taxi Rides. It is non
    blocking.
public void start() {
    // Start an Executor Service with a fixed capacity
    final ExecutorService executorService =
        Executors.newFixedThreadPool(PARALLEL_TASKS);

    // Creates a worker thread that runs the Queue Management logic
    Thread worker = new Thread(new Runnable() {
        @Override
        public void run() {
            // During the execution of the service
            while (true) {
                // Picks a Taxi Ride from the pending list
                // to be served and removes it
                // from the list - BLOCKING METHOD CALL
                TaxiRide taxiRide = queryManager.getPendingTaxiRide();
                // Execute the task to manage the picked ride
                executorService.submit(new ManageRide(taxiRide));
                // Loop
            }
        }
    });
}

```

```

    });

    // Starts the worker thread
    worker.start();
}
}

```

---

## 3.2 Geolocation

This subsection presents a naive implementation of the algorithm used to associate a given GPS Data to a Zone, or to show that the GPS Data is not contained in the City.

### 3.2.1 Point

---

```

package mts.zones;

public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}

```

---

### 3.2.2 Triangle

---

```

package mts.zones;

public class Triangle {
    private final Point p3;

    private final double a;
    private final double b;
    private final double c;
}

```

```

private final double d;

public Triangle(Point p1, Point p2, Point p3) {
    this.p3 = p3;

    this.a = p1.getX() - p3.getX();
    this.b = p2.getX() - p3.getX();
    this.c = p1.getY() - p3.getY();
    this.d = p2.getY() - p3.getY();
}

// This methods check if the given point is inside this Triangle,
// or on its borders, vertices included, with a given
// approximation error.
// It spoils a theorem about convex polygons that says that a point P
// is inside a given convex polygon (like a Triangle is) if the given
// vector associated to the point P is a convex combination
// (a linear combination with all the coefficients
// non negative that sum to 1) of the polygon vertices.
// So, here we calculate if such coefficients exists solving
// a vector equation:
// 1)  $P = \alpha_1 * P_1 + \alpha_2 * P_2 + \alpha_3 * P_3$ ;
// with P, P1, P2, P3 in  $R^2$ ;
// with  $\alpha_1, \alpha_2, \alpha_3$  in  $R$ ;
// and  $\alpha_1 \geq 0, \alpha_2 \geq 0, \alpha_3 \geq 0$ ;
// and  $\alpha_1 + \alpha_2 + \alpha_3 = 1$ .
// 2)  $P = \alpha_1 * P_1 + \alpha_2 * P_2 + (1 - \alpha_1 - \alpha_2) * P_3$ .
// 3)  $P - P_3 = \alpha_1 * (P_1 - P_3) + \alpha_2 * (P_2 - P_3)$ .
// This equation can then be split into two scalar linear equations
// in the x and y components.
// The system is solved using Cramer's rule and then it is
// checked that  $\alpha_1$  and  $\alpha_2$  (and  $\alpha_3$ ) found by
// solving the system satisfy the constraints.
public boolean contains(Point p) {
    double e = p.getX() - p3.getX();
    double f = p.getY() - p3.getY();

    double delta = a * d - b * c;

    if (delta == 0)
        return false;

    double delta_alpha_1 = e * d - f * b;

    double alpha_1 = delta_alpha_1 / delta;

    if (alpha_1 < 0)
        return false;

    double delta_alpha_2 = a * f - c * e;

```



```

        double alpha_2 = delta_alpha_2 / delta;

        if (alpha_2 < 0)
            return false;

        double alpha_3 = 1 - alpha_1 - alpha_2;

        if (alpha_3 < 0)
            return false;

        return true;
    }
}

```

---

### 3.2.3 Zone

```

package mts.zones;

import java.util.Set;

public class Zone {
    private final Set<Triangle> triangles;

    public Zone(Set<Triangle> triangles) {
        this.triangles = triangles;
    }

    public boolean contains(Point p) {
        for (Triangle t : triangles)
            if (t.contains(p))
                return true;

        return false;
    }
}

```

---

### 3.2.4 Zones

```

package mts.zones;

import java.util.Set;

public class Zones {
    private final Set<Zone> zones;

```

```
public Zones(Set<Zone> zones) {
    this.zones = zones;
}

public Zone getContainingZone(GPSData gpsData) {
    Point p = new Point(gpsData.getLongitude(), gpsData.getLatitude());

    for (Zone z : zones)
        if (z.contains(p))
            return z;

    return null;
}
}
```

---

## 4 User Interface Design

Four different graphical user interfaces are provided: two of them for both **Registered and Non Registered Passengers**, one for the **Taxi Drivers**, and the last one is dedicated to the **Administrator**. The interfaces provided to **Passengers** are both Web based and Mobile Application based. For **Taxi Drivers** it is only provided a Mobile Application interface, and for the **Administrator** a Desktop Application interface is built. No additional user interfaces are provided, not graphical nor textual. The system architecture does not admit the usage of a textual interface (e.g. a CLI). Below are provided three lists of mockups that have the aim of graphically explaining navigation through the service. For this reason we have scratched only the most important and relevant features of *myTaxiService*. First, are provided the **Passenger** mockups, then **Taxi Driver** screens are shown and at last, a set of mockups about the **Administrator** application are provided. The screens about the **Passenger** and about the **Taxi Driver** are the same reported in the RASD Document, and are here provided to make easier for the readers to access them. On the other hand the last screens are about a desktop-based **Administrator** interface that allows the administrators to perform some critical queries.

## 4.1 Passenger mockups

- Homepage

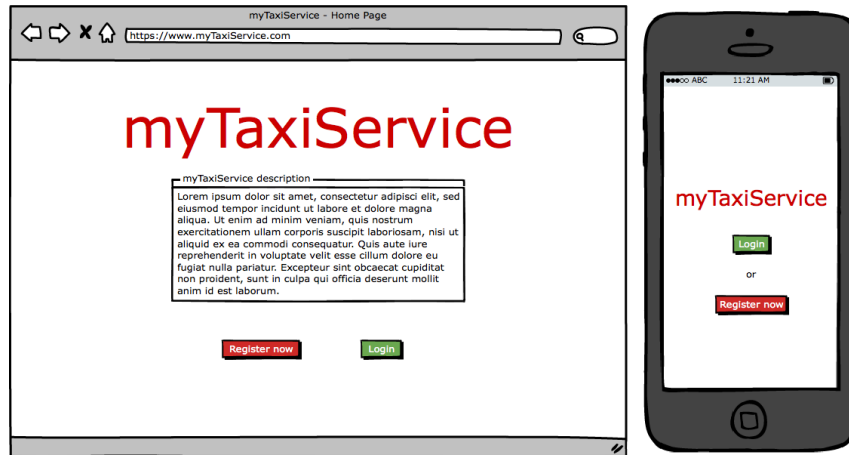


Figure 10: *myTaxiService* homepage.

- Registration

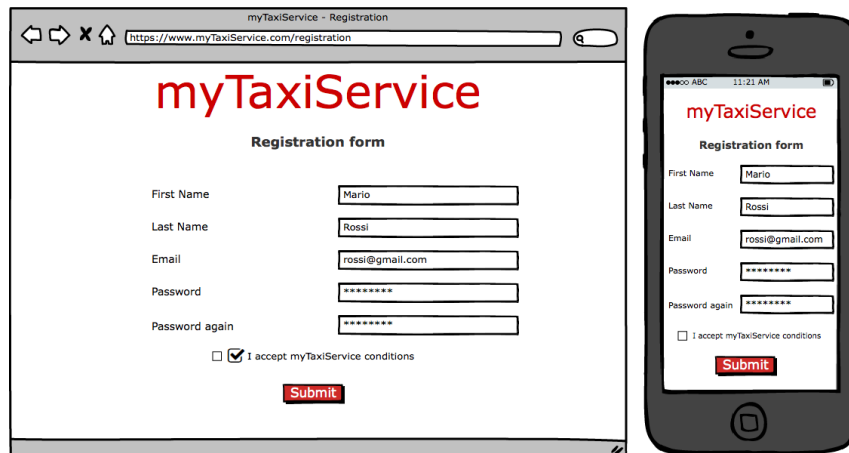


Figure 11: Registration screens.

- Login



Figure 12: Login screens.

- Empty registered passenger home page

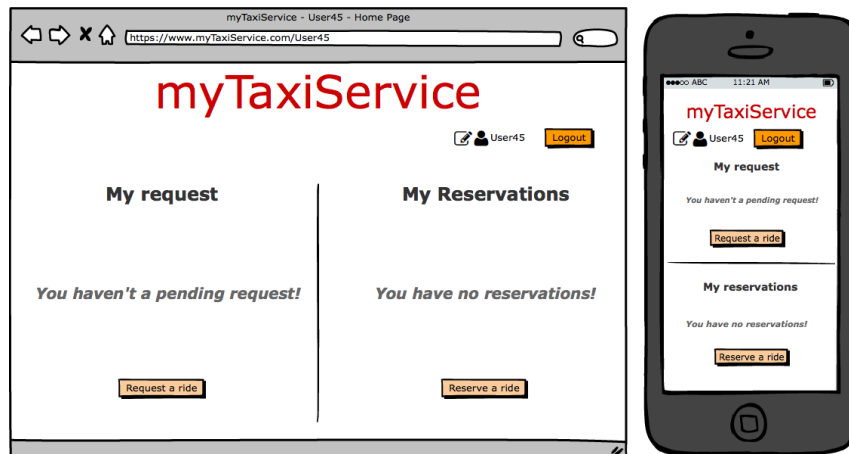


Figure 13: State 1 of RP home page.

- Request a taxi home page

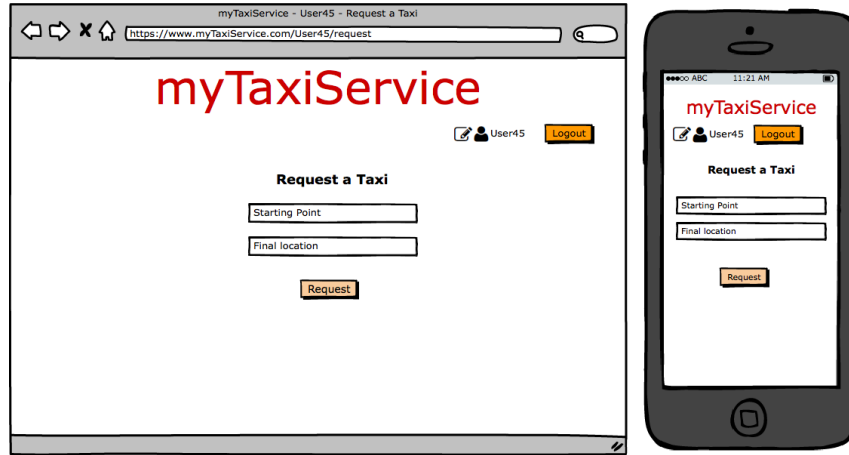


Figure 14: Taxi request.

- Reserve a taxi home page



Figure 15: Taxi Reservation.

- Registered passenger home page

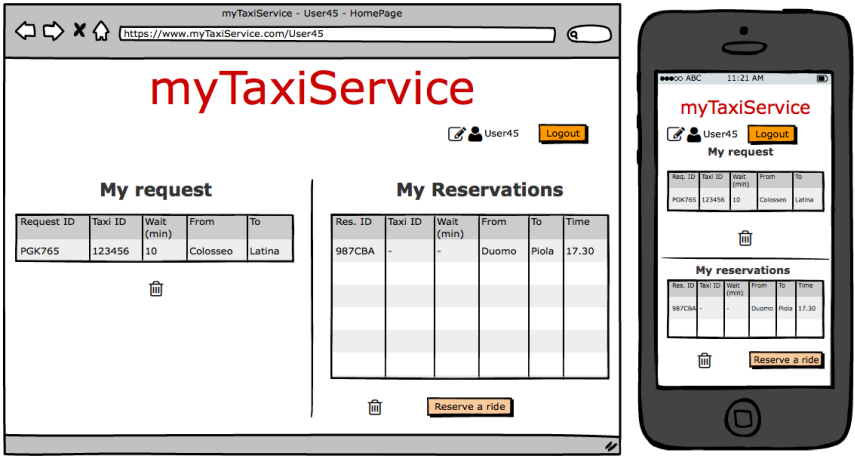


Figure 16: State 2 of RP home page.

- Cancellation of a ride in registered passenger home page

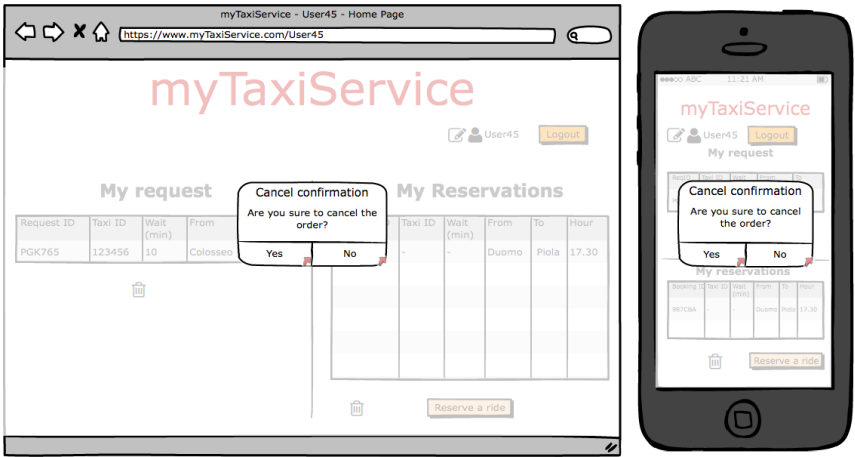


Figure 17: State 3 of RP home page.

## 4.2 Taxi Driver mockups

- Homepage



Figure 18: TD homepage.

- Login



Figure 19: TD login page.



- Empty driver home page

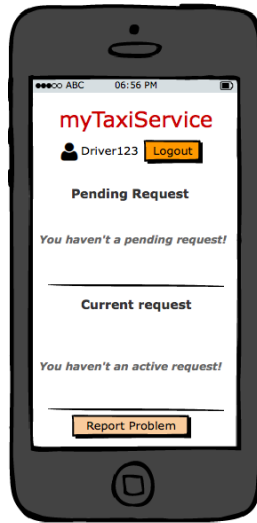


Figure 20: State 1 of TD personal page.

- Taxi driver home page with a pending request

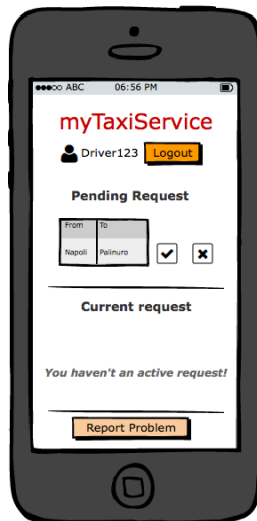


Figure 21: State 2 of TD personal page.

- Taxi driver home page while serving a request

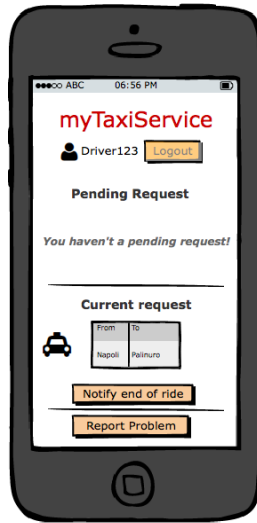


Figure 22: State 3 of TD personal page.

- Report problem



Figure 23: TD report problem screen.

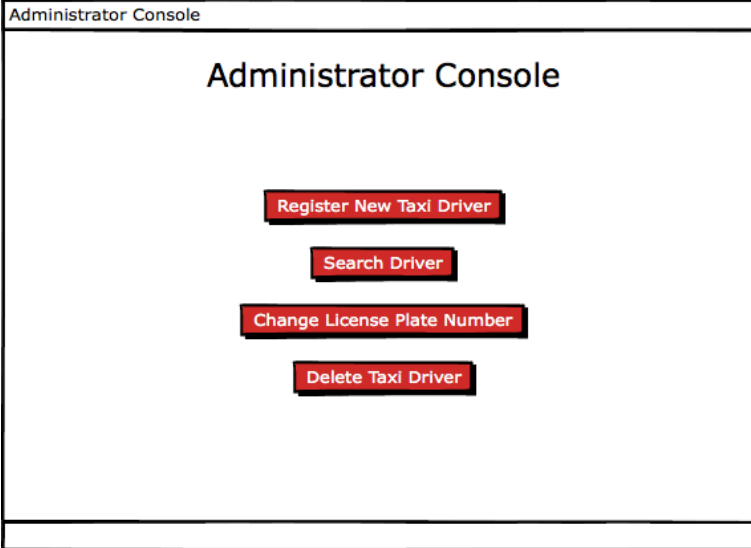
- Report solution



Figure 24: TD report solution screen.

### 4.3 Administrator mockups

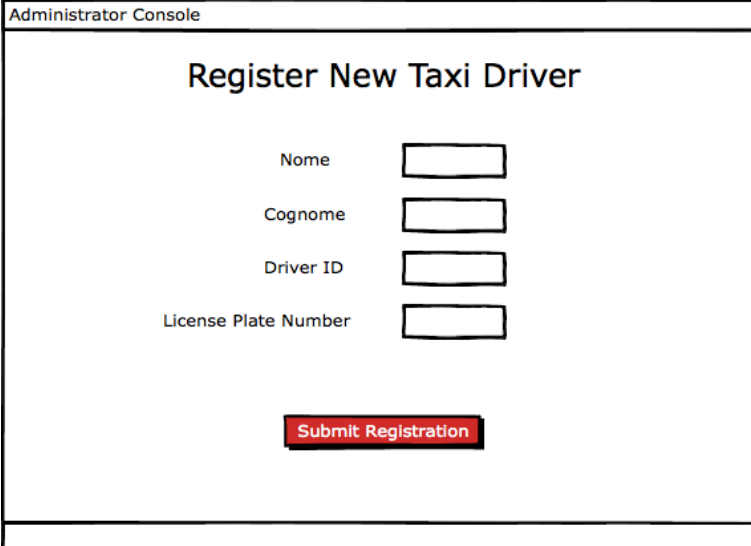
- Administrator Console



The mockup shows a web browser window titled "Administrator Console". The main heading is "Administrator Console". Below the heading, there are four red buttons with white text, arranged vertically and centered: "Register New Taxi Driver", "Search Driver", "Change License Plate Number", and "Delete Taxi Driver".

Figure 25: Administrator splash screen.

- Register New Taxi Driver



The mockup shows a web browser window titled "Administrator Console". The main heading is "Register New Taxi Driver". Below the heading, there are four text input fields, each preceded by a label: "Nome", "Cognome", "Driver ID", and "License Plate Number". Below the input fields, there is a red button with white text labeled "Submit Registration".

Figure 26: State 1 of Administrator personal page.

- Search Taxi Driver

Administrator Console

## Search Taxi Driver

XML Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<taxi_driver>
  <name>Mark</name>
  <surname>Reds</surname>
  <driverid>345565</driverid>
  <license_plate_number>EG 004 AD</license_plate_number>
</taxi_driver>
```

Figure 27: State 2 of Administrator personal page.

- Change License Plate Number

Administrator Console

## Change License Plate Number

Driver ID

New License Plate Number

Figure 28: State 3 of Administrator personal page.

- Delete Taxi Driver

Administrator Console

Delete Taxi Driver

Driver ID

345565

Delete Driver

Figure 29: State 4 of Administrator personal page.

## 5 Requirements Traceability

In this section are mapped all the functional requirements identified in the RASD, grouped by the **Use Case** they refer to.

### 5.1 Registration

<i>Functional Requirements</i>	<i>Design Elements</i>
FR1, FR2	The <b>Profile Manager</b> perform data validation and checks if the passenger is already registered to <i>myTaxiService</i> .
FR3	The registration process is handled as an atomic procedure by the <b>Profile Manager</b> : if the registration process is stopped, there are no choices of resuming.
FR4	Through the response to the registration request sent to the <b>RESTful Service</b> , the passenger is informed about the result of the process.
FR5	Through an internal method of the <b>Profile Manager</b> is possible to send an email to the just registered <b>Passenger</b> .
FR6	If the registration process is completed successfully, then the <b>Profile Manager</b> has to call a method of the <b>Query Passenger Model</b> in order to make the registration effective and persistent. From this moment, if that Passenger tries to login, the system must recognize him as a user of the service and allow him to use all the features of <i>myTaxiService</i> .

### 5.2 Login

<i>Functional Requirements</i>	<i>Design Elements</i>
FR7	The validation of Passenger inserted data is done by the <b>Session Manager</b> after checking for a possible duplication of the user session.
FR8	The <b>Session Manager</b> generates the Login Token once it is sure of the correctness of the login data, and associates it to the user via the <b>Query Manager</b> .
FR9	The notification of the result of the login procedure is produced in the <b>Session Manager</b> and sent as response to the user that requested the login.
FR10	The <b>PS Request Creator</b> , <b>TD Request Creator</b> and the <b>RESTful Service</b> are capable of handling connection errors without making the relevant systems crash.

### 5.3 Logout

<i>Functional Requirements</i>	<i>Design Elements</i>
FR11	The Passenger can perform the Logout procedure by interacting with the <b>PS Web View</b> or <b>PS Application View</b> .
FR12	All the logic about the navigation inside the application and in general about the building of the View is contained into the <b>PS Web View</b> or <b>PS Application View</b> .
FR13	The <b>PS Request Creator</b> , <b>TD Request Creator</b> and the <b>RESTful Service</b> are capable of handling connection errors without making the relevant systems crash.

### 5.4 View Request and Reservations

<i>Functional Requirements</i>	<i>Design Elements</i>
FR14	The data is accessible through the <b>Query Manager</b> , that calls the <b>Model Query Service</b> to reach the <b>Passenger DB Adapter</b> .
FR15	<b>PS Application View</b> and <b>PS Web View</b> handle this interface logic.
FR16	<b>PS Application View</b> and <b>PS Web View</b> handle this interface logic.
FR17	The <b>PS Request Creator</b> and the <b>RESTful Service</b> are capable of handling connection errors without making the relevant systems crash.

### 5.5 Handle Personal Profile

<i>Functional Requirements</i>	<i>Design Elements</i>
FR18	This validation is done by the <b>Profile Manager</b> .
FR19	The <b>Profile Manager</b> uses the interfaces exposed by the <b>Query Manager</b> to guarantee so.
FR20	This information is sent as a response to a profile modification request by the <b>RESTful Service</b>
FR21	The <b>PS Request Creator</b> and the <b>RESTful Service</b> are capable of handling connection errors without making the relevant systems crash.



## 5.6 Taxi Reservation

<i>Functional Requirements</i>	<i>Design Elements</i>
FR22	This validation is done by the <b>Taxi Ride Manager</b> .
FR23	The <b>Taxi Ride Manager</b> uses the interfaces exposed by the <b>Query Manager</b> to guarantee so.
FR24	This information is sent as a response to a Taxi Reservation request by the <b>RESTful Service</b>
FR25	The <b>PS Request Creator</b> and the <b>RESTful Service</b> are capable of handling connection errors without making the relevant systems crash.

## 5.7 Taxi Request

<i>Functional Requirements</i>	<i>Design Elements</i>
FR26	Data validation is done by the <b>Taxi Ride Manager</b> . If the data don't respect the constraints, then the <b>Taxi Ride Manger</b> informs the related Passenger of the issue.
FR27	When a Passenger sends a Taxi Ride request to <i>myTaxiService</i> , the <b>RESTful Service</b> handles it. Then the <b>Taxi Ride Manager</b> completes the <b>Taxi Ride</b> request by adding the Zone of the Starting Address. In addition, this component keeps the model updated by adding the ride to the persistent data layer. Later on the <b>Queue Manager</b> picks the ride from the Model and associates it to a Taxi Driver. In conclusion this component uses the <b>Dispatcher</b> to notify the Passenger with notification message that contains the Taxi Driver ID and Travel Time.
FR28	Through the <b>Dispatcher</b> , users can be reached in every moment by notification messages.
FR29	The <b>PS Request Creator</b> , the <b>PS Receiver</b> , the <b>RESTful Service</b> and the <b>Dispatcher</b> are capable of handling connection errors without making the relevant systems crash.

## 5.8 Notify Problem

<i>Functional Requirements</i>	<i>Design Elements</i>
FR30	The <b>TD Application View</b> is provided with a specific screen that allows the Taxi Driver to report a problem. In addition, Taxi Driver has a chance to specify whether or not he can solve the given problem These data are then sent to the <b>RESTful Service</b> in order to be processed by the <b>Taxi Driver Manager</b> .
FR31	This task is done by the <b>Taxi Driver Manager</b> .
FR32	This is sent a response to a Notify Problem request by the <b>RESTful Service</b> .
FR33	This is done by the <b>Taxi Driver Manager</b> via the <b>Dispatcher</b> .
FR34	The <b>TD Request Creator</b> , the <b>TD Receiver</b> , the <b>RESTful Service</b> and the <b>Dispatcher</b> are capable of handling connection errors without making the relevant systems crash.

## 5.9 End of Ride

<i>Functional Requirements</i>	<i>Design Elements</i>
FR35	The Taxi Driver pushes a button on the <b>TD Application View</b> that calls the <b>TD Request Creator</b> , which sends a request to the <b>RESTful Service</b> . The request is then forwarded to the <b>Taxi Ride Manager</b> by the <b>RESTful Service</b> . The <b>Taxi Ride Manager</b> handles the operation.
FR36	When a Taxi Driver has notified the end of his current Taxi Ride, he is associated to a Available status. Then the <b>Queue Manager</b> is waiting for Taxi Drivers to become Available to enqueue them into their current Zone.
FR37	The response to the request made to achieve FR35 contains the result of the <b>Taxi Driver</b> action.
FR38	The <b>TD Request Creator</b> and the <b>RESTful Service</b> are capable of handling connection error problem without making the relevant systems crash.

## 6 Appendix

### 6.1 Tools

- **TeXstudio:** L<sup>A</sup>T<sub>E</sub>Xeditor used to write this document.
- **StarUML:** Used to design the *UML Deployment Diagram*.
- **SourceTree:** Used to allow team work and synchronize GitHub repository.
- **Blasamiq Mockups 3.0:** Used to build *dekstop-based application*, *web application* and *mobile application* mockups.
- **Astah Professional:** Used to design the *UML Component Diagram* and *UML Sequence Diagrams*.
- **Eclipse Mars:** Used to write and test the *Java* code.
- **Google Draw:** Used to draw the *Overview Diagram*.

### 6.2 Hours of Work

- **Alessandro:** 37
- **Alberto Mario:** 37

### 6.3 Version History

<i>Version Number</i>	<i>Release Date</i>	<i>Changelog</i>
1.0	04/12/2015	Initial Release.
1.1	21/01/2016	Diagrams updated. Fixed grammar.