Politecnico di Milano

A.A. 2015-2016

Software Engineering 2 Project – “myTaxiService”

Prof. Raffaela Mirandola

Design **D**ocument

Version 3.0

Christian Zichichi (mat. 840565), Luigi Marrocco (mat. 854884)

January 12, 2016

Table of Contents

1. INTRODUCTION

1.1 Purpose

This is the Design Document (DD) for myTaxiService application, which aim is to provide a medium/base level description of the system design in order to allow software developers to proceed with an understanding of what is to be built and how it is expected to be built. It contains a functional description of the main architectural components and their interactions. Using UML standards, it will be possible to specify the structure of the system and the relationships between the modules.

1.2 Scope

The system aims to offer a taxi calling and reservation service for Big_City citizens through a dedicated website and a mobile application. It has to simplify the users' access and provide a reliable taxi queue management, as well as a straightforward on-board application for taxis. Every component must be conveniently thin and must incapsulate a single functionality. The dependency between them has to be unidirectional and coupling must be avoided in order to increase the re-usability of the software. The system functionalities will be provided by interfaces or abstract classes, which will be implemented in concrete ones. This will simplify future maintenance and the implementation of additional services. For other details regarding system requirements and specifications, please see chapter 1.4 regarding the reference documents.

1.3 Definitions, acronyms, abbreviations

- **Tier:** It is a hardware level in a generic architecture
- **Layer:** It is a software level in a generic software system
- **GUI:** Graphical User Interface. It is the set of graphical elements such text, buttons and images which the user can interact with, usually abbreviated to **UI**
- **FIFO:** First In First Out

- **GPS:** Global Positioning System
- **Thin Client:** It is a design style for the client in which the application running on the client contains the least business logic possible. In our case the client applications will only serve as a presentation interface
- **RDBMS and DB:** Relational Data Base Management System and Data Base Layer
- **Application Server:** the component which provides the application logic and interacts with the DB and with the front-ends
- **Back-end:** other term to identify the Application server
- **Front-end:** the components which use the application server services (e.g., the web front-end and the mobile applications)
- **Web Server:** the component that implements the web-based front-end. It interacts with the application server and with the users' browsers
- **API:** Application Programming Interface
- **Java EE:** Java Enterprise Edition
- **JDBC:** Java Data Base Connectivity
- **JPA:** Java Persistence API that describes the management of relational data in applications using Java EE
- **JSON:** JavaScript Object Notation, it is a lightweight data-interchange format (attribute–value pairs) that usually replaces XML for asynchronous browser/server communication
- **JSF-JSP:** Java Server Faces – Java Server Pages
- **Cronjob:** a scheduled task that is executed at a specified time/date. In our case it is periodic

1.4 Reference documents

This document refers to the following documents:

- myTaxiService – **R**equirement **A**nalysis and **S**pecification **D**ocument
- myTaxiService – **I**ntegration **T**esting **P**lan **D**ocument

1.5 Document structure

This document is structured in four main parts:

- **Architectural Design:** this section shows the main components of the systems with their sub-components and their relationships, along with their static and dynamic design. This section will also focus on design choices, styles, patterns and paradigms
- **Algorithm Design:** this section presents and discusses in detail the algorithms designed for the system functionalities
- **User Interface Design:** this section provides an overview on how the UI(s) of the system will look like
- **Requirements Traceability:** this section shows how the requirements defined in the reference RASD map into the design elements defined in this document

2. ARCHITECTURAL DESIGN

2.1 Overview

To develop this project we decided to use the Java Enterprise Edition architecture, that is four-tiered, with GlassFish as Application Server and MySQL as RDBMS. Moreover, we decided to decouple the business logic from the view, using a RESTful architecture, since we have completely different interfaces to deal with (mobile application and web application for users, mobile application for taxis embedded in the car):

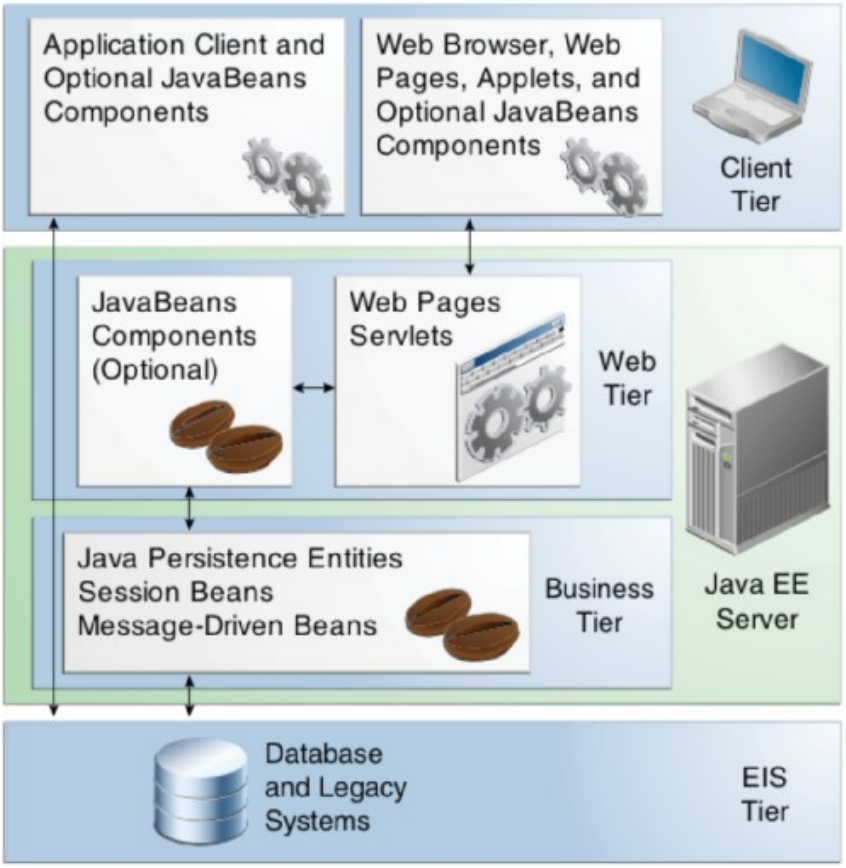
• **Client Tier:** it contains Application Clients and Web Clients and it is the layer that interacts directly with the actors. As far as the web application part is concerned, the client will use a web client to access pages, that consists of dynamic pages with markup language (like HTML and XML) and a web browser (Google Chrome, Mozilla Firefox) that renders the pages received from the web server. The mobile application for users, instead, will be developed using a native language such as Objective C or

Swift for iOS and Java for Android or a framework like Apache Cordova (HTML/CSS/JavaScript) for a cross-platform single-page application. The application for the Android Auto device embedded in the car will be developed using Java. All of them will require resources, obtained by calling specific APIs

• **Web Tier:** it contains web components that can be Servlets or Dynamic Web pages created with JSF and/or JSP technology. It is possible to use some Java Beans components to send input data (in JSON format) made by users to other beans running in the business tier and forward the response to the client. Moreover, It is the layer that is exposed to the outside: client requests are processed by a dispatching component (WebController) which decides which controller to execute. Every controller uses the components it needs to perform an action

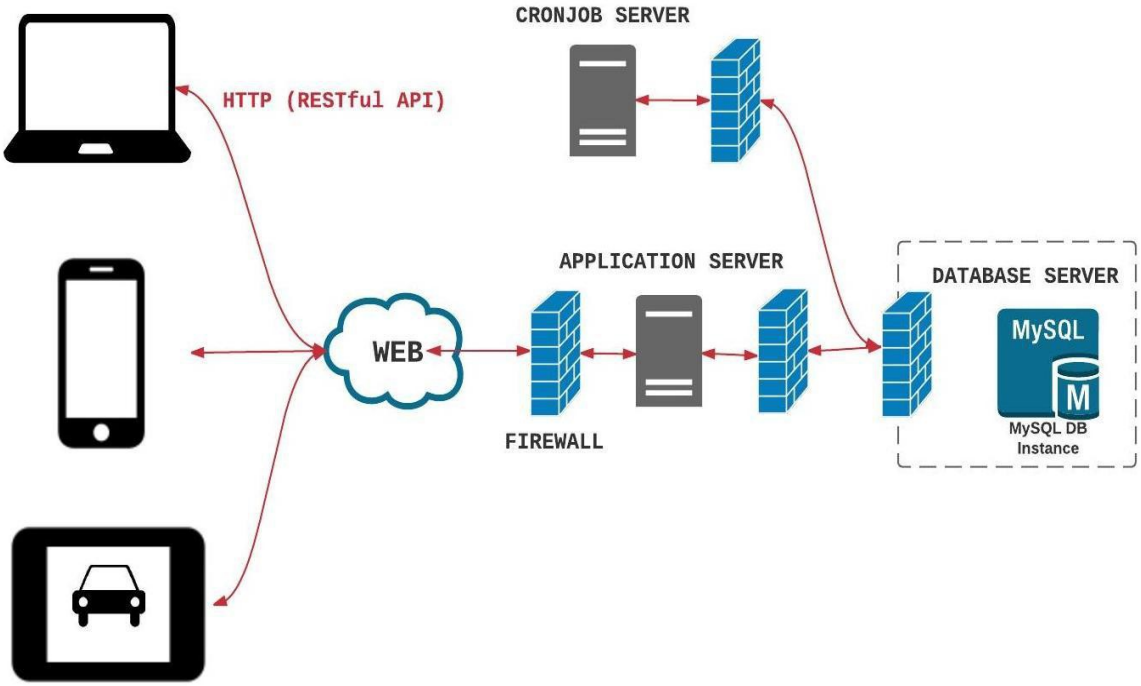
• **Business Tier:** essentially it is the logic part that solves the needs of a particular business domain. It consists of Enterprise Java Beans (EJB), that process data coming from the client and send it to the Enterprise Information System tier for storage or the other way around (using JPA), and Java Persistence Entities (Session Beans and Message Driven Beans). In our case, Stateless Session Beans are used since they do not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the client-specific state is not retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. Because they can support multiple clients, stateless session beans can offer better scalability for message-driven applications that require large numbers of clients (like ours) than the stateful ones. Nevertheless, our architecture is RESTful, so it must be stateless by definition

• **Enterprise Information System (EIS) Tier:** it includes enterprise infrastructure systems with the DBs in which data concerning taxis, passengers and rides are stored and retrieved. Java EE application components need access to this tier for database connectivity (JDBC)



This design choice also makes it possible to deploy the application server and the web server on different tiers, improving scalability where many web servers talking to a single application server are desired. The interactions between the main architectural components are shown in the hardware representation, in which the web server is omitted for simplicity since it can be included in the application server (detailed deployment view in chapter 2.4). They are all synchronous and multi-threaded (the more the users of the system, the more the instances). Security measures are present to encrypt communications and protect the data from unauthorized access and malicious users. The functional role of the Cronjob Server instance will be explained in detail in chapter 2.8.

HARDWARE REPRESENTATION



Our RESTful architecture permits to have a single back-end which can be used from all client applications, that are the mobile applications (users and taxi) and the web application for users. Every client makes an API call to the web-server and handles the answer in different ways according to where the request came from. It is important to state that mobile applications (passenger and taxi) and web application for users have a different view logic. A client-side real time polling technique is also used to notify taxi drivers and passengers that an event has occurred (e.g., a taxi driver has been picked up from a queue for a ride), which periodically makes a request to the server-side asking for updates.

As far as security is concerned, every HTTP request must have an associated authentication-authorization token (created at login time to identify the client). In fact, the request is sent by the "Dispatcher" component to the controller if and only if the associated token is correctly validated by the "Security" component. The controller, then, will satisfy the desired function using other components. Authentication is a process in which the credentials provided are compared to those on the database. If the credentials match, the process is completed with the recognition of the client (with their type: passenger or taxi driver), that is then granted authorization for access the functionalities of the system specifically designed for them. The produced response is a JSON string that all client applications can handle.

One of the advantages of Java EE is represented by the fact that many services have been produced by other developers and can be used in our application: for instance, JAX-RS (Java API for RESTful Web

Services) allows to create APIs following the RESTful paradigm, in order to extend the functionalities provided by our system. In addition, our application uses external APIs such as the ones provided by Google Maps which allow us, for instance, to compute distances and the waiting times.

2.2 High level components and their interactions

Dispatcher

This component dispatches all the requests that come from a passenger or taxi driver. When a request is received the component checks if the request is valid and, after that, if it is, the request is forwarded to the correct component. Instead, if the request is not valid, the dispatcher returns an error message to the user. When it receives a response from a component, it sends it to the correct user.

Security

The unique function of this component is to control if a client's request is valid. After receiving a request, this component has the responsibility to check whether who (user or taxi driver) made it is allowed to do that or not.

AccountManager

This component manages the requests of the user that deals with account management. When a user logs in this component creates a token and stores it in the database. Instead, when a user logs out this component removes the relative token from the database.

PassengerManager

This component is in charge of managing all the requests made by a passenger.

The functionalities are:

- Create reservations and requests

- Cancel reservations
- Manage the list of taxis booked by a passenger

TaxiManager

This component manages all the functionalities that allow taxi drivers to set their availability status.

These functionalities are:

- Set their availability on
- Set their availability off

RideManager

The function of this component is to find a taxi for the ride and to manage the response of the taxi driver.

The functionalities are:

- Find a taxi for a ride
- Manage the confirmation of a ride by the taxi driver
- Manage the refusal of a ride by the taxi driver
- Check if the taxi driver is involved in a ride

QueueManager

This component manages all the requests concerning the queues.

The functionalities are:

- Add a taxi to a queue
- Remove a taxi to a queue

- Return the first taxi of the queue
- Move a taxi in the last position of the queue

ZoneManager

This component returns the relative queue from a location (point in a zone) received as input.

DataLayer

This component manages all the operations regarding the data base (Insert, Update, Delete).

PollingManager

The functional role of this component is fully explained in chapter 2.8.

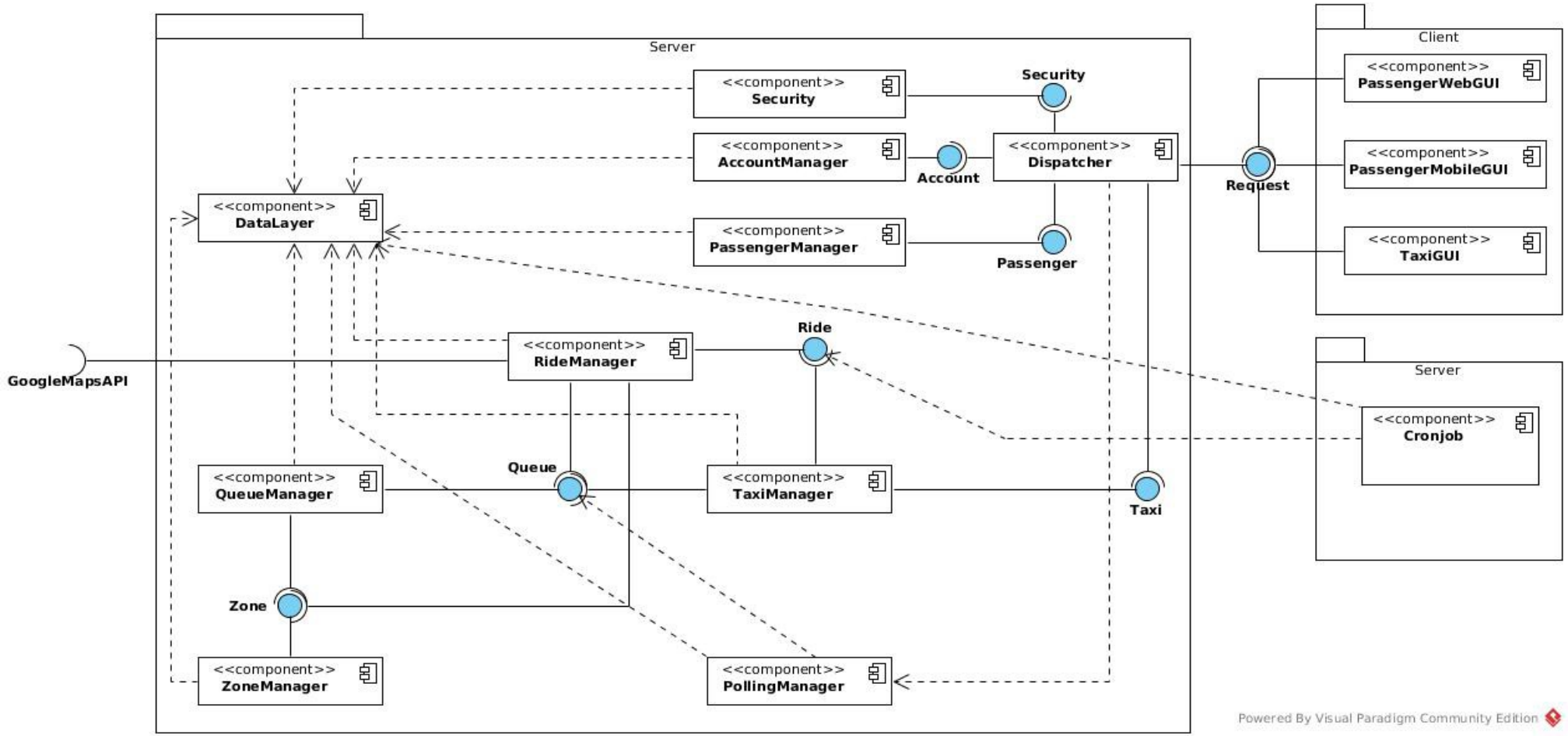
Cronjob

The functional role of this external component is fully explained in chapter 2.8.

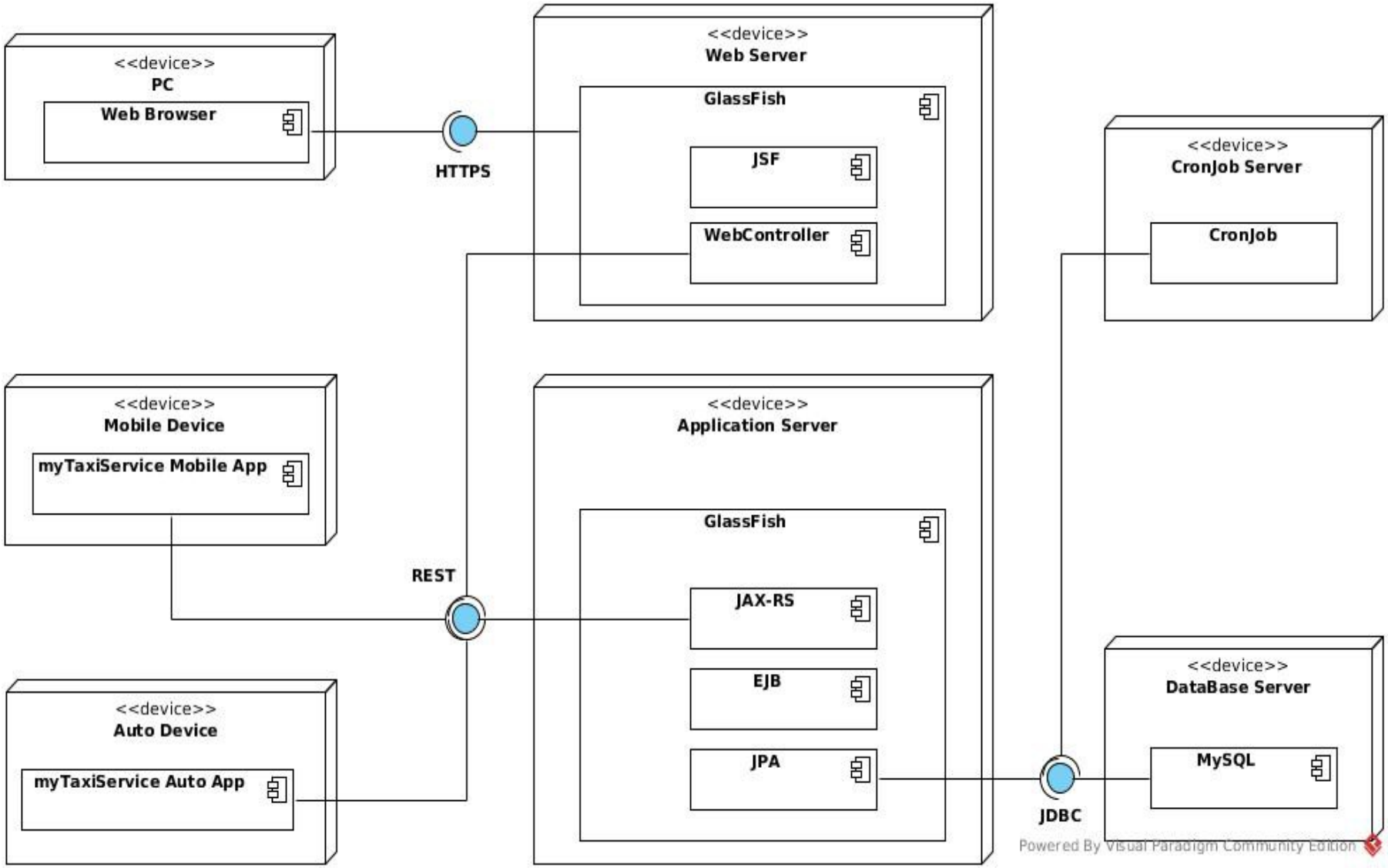
PassengerWebGUI / PassengerMobileGUI / TaxiGUI

These client-side components deals with the view of the user (passenger or taxi) by showing the requested page or the message that should be displayed, like a ride proposed to a taxi driver or a confirmation page to the passenger.

2.3 Component view

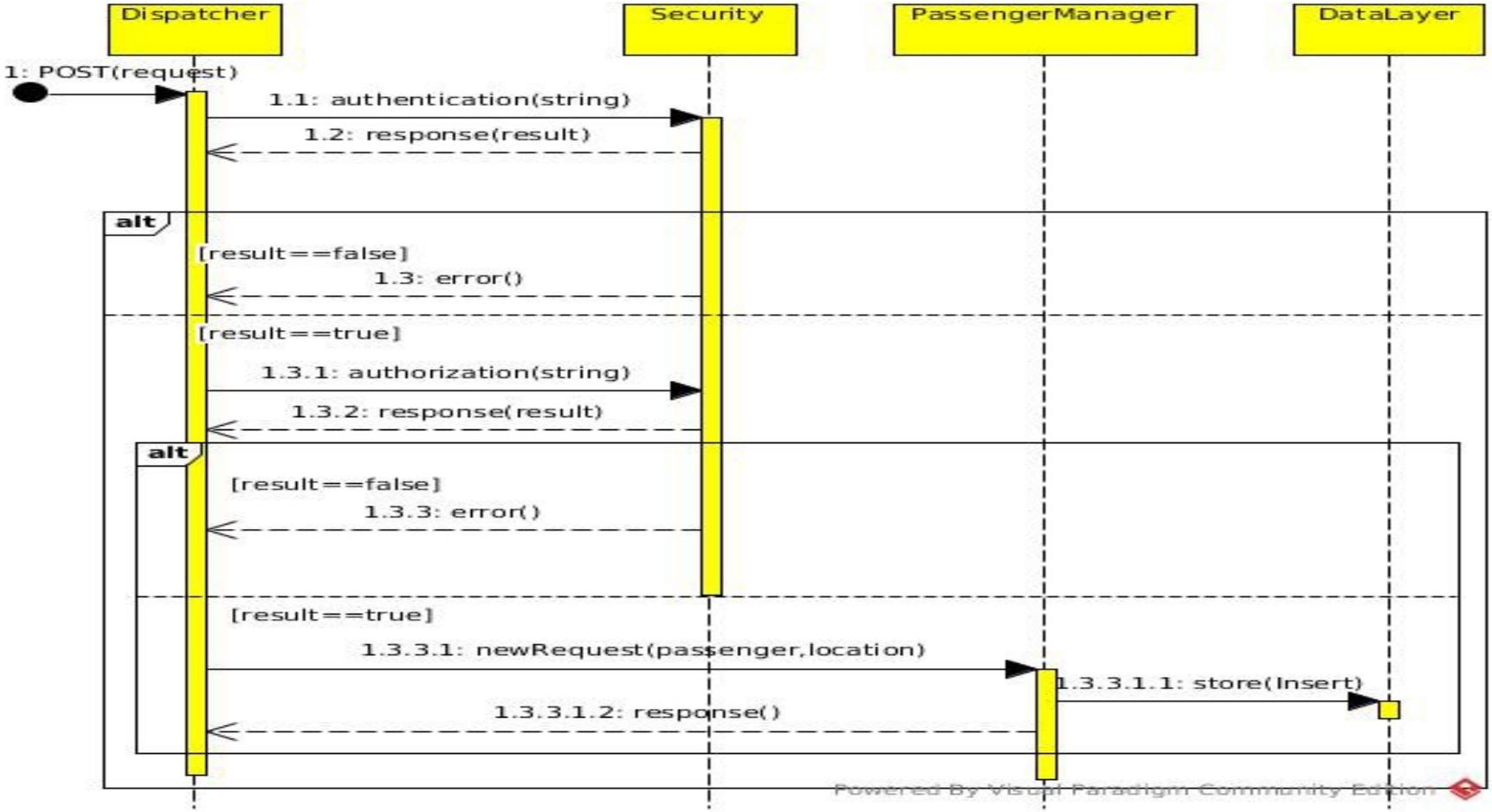


2.4 Deployment view

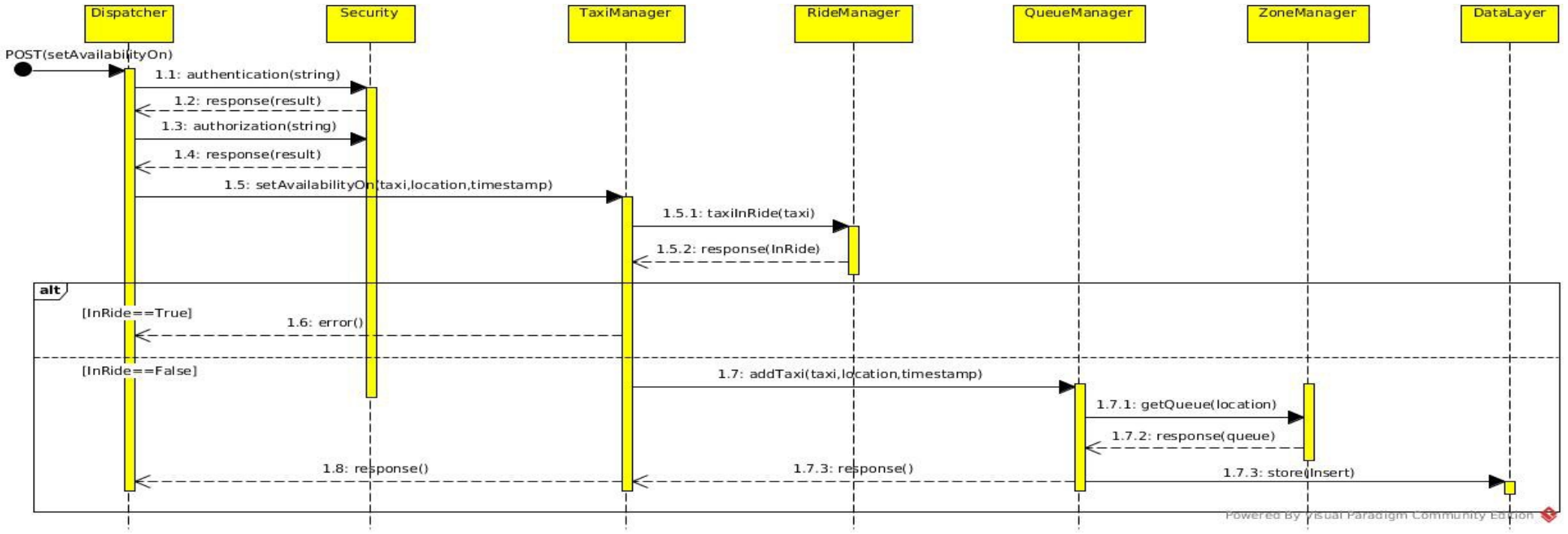


2.5 Runtime view

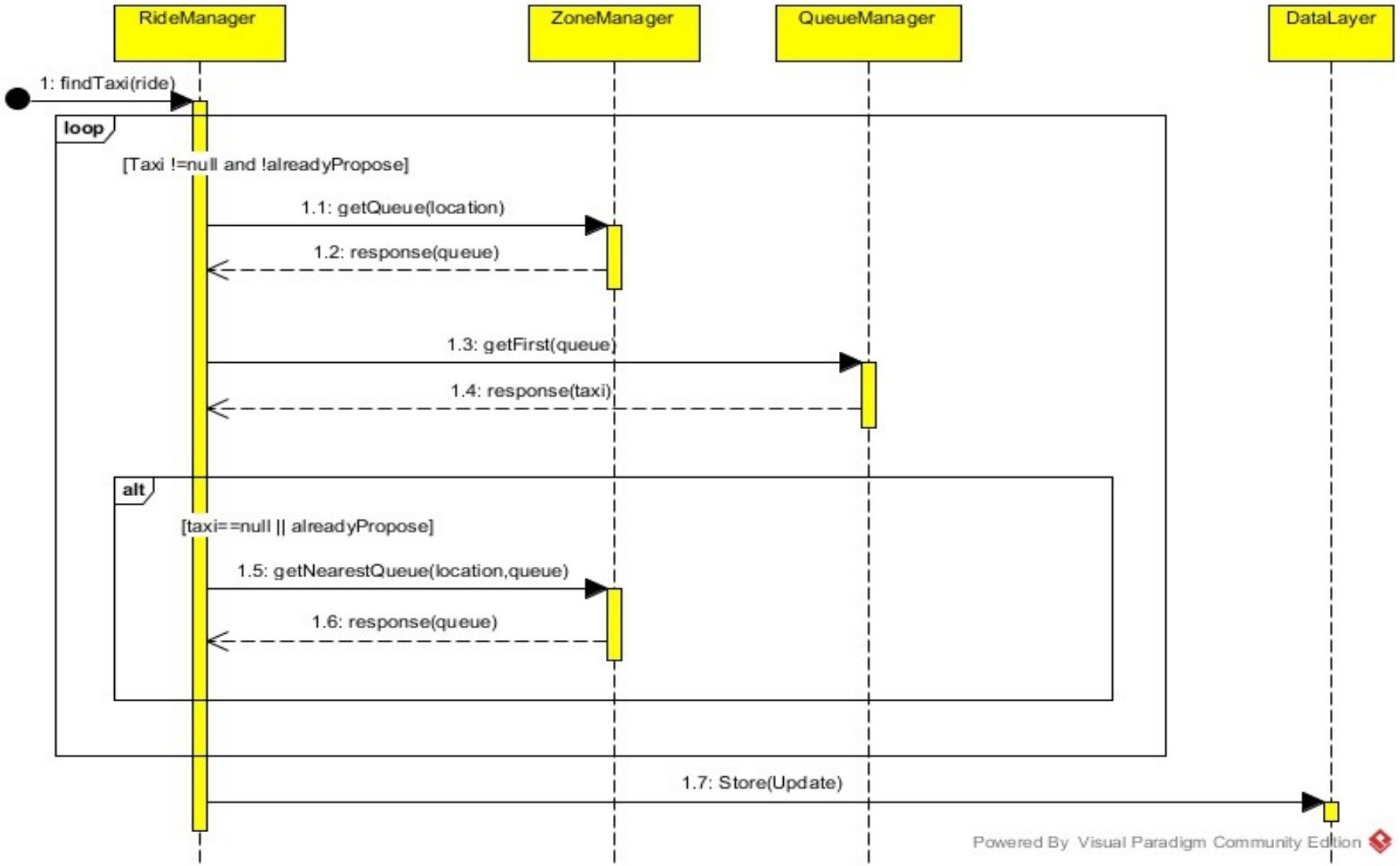
A USER MAKES A TAXI RIDE REQUEST FROM THE MOBILE APPLICATION



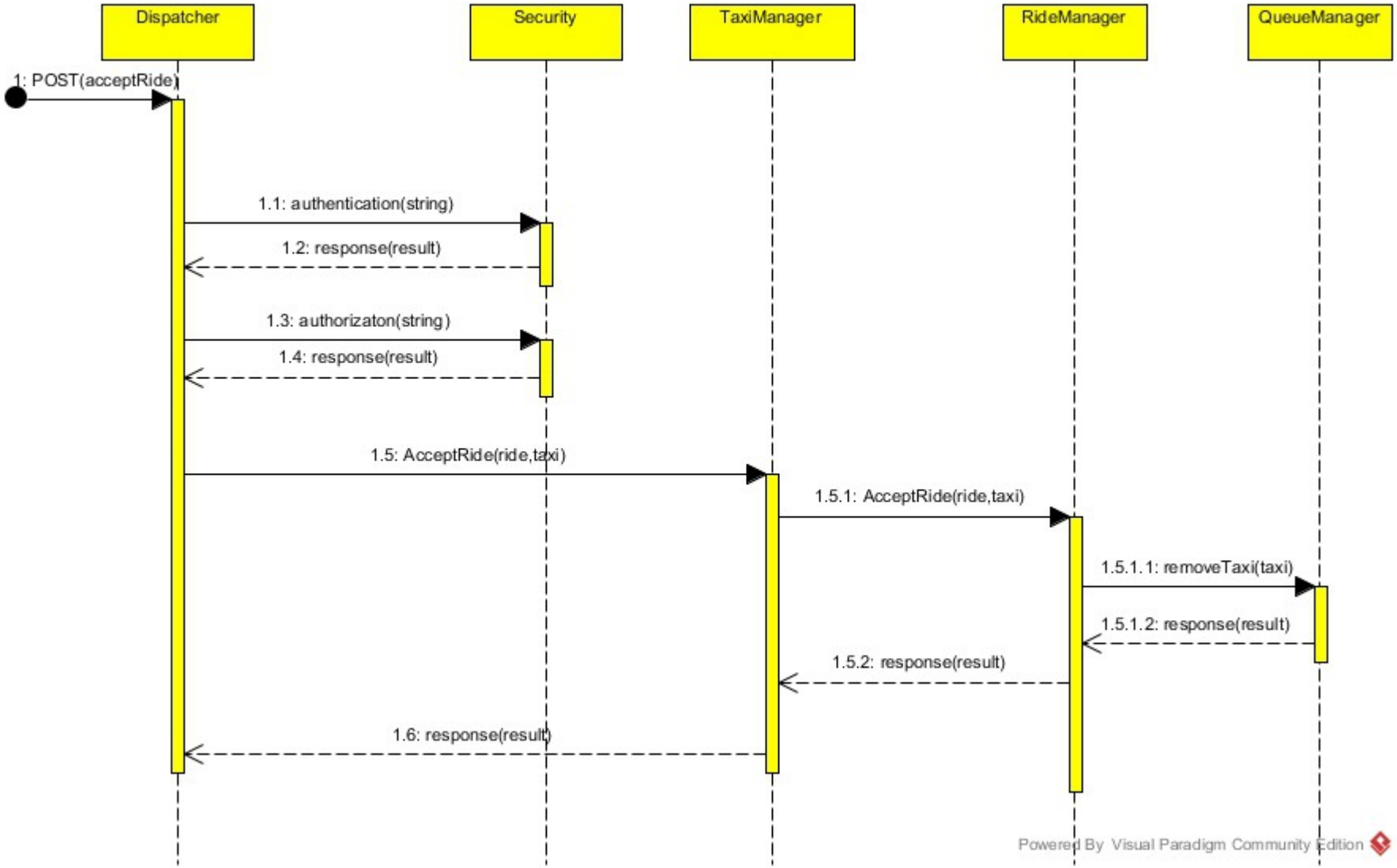
A TAXI DRIVER SETS THEIR AVAILABILITY ON



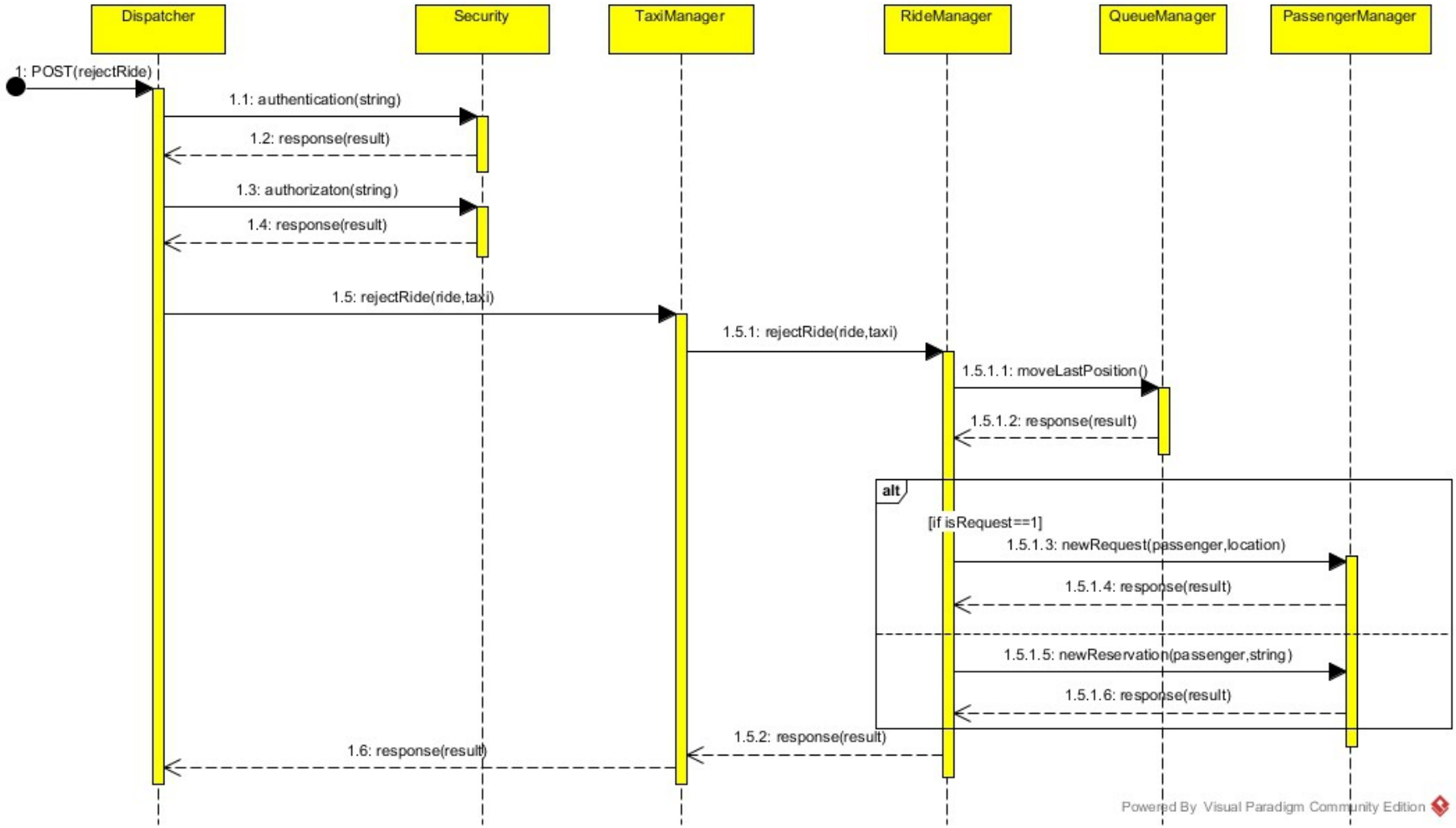
A RIDE IS PROPOSED TO A TAXI DRIVER



A TAXI DRIVER ACCEPTS A RIDE



A TAXI DRIVER REJECTS A RIDE



2.6 Component interfaces

Security interface

```
Interface Security{
        Public Boolean authentication(String token);
        Public Boolean authorization(String token);
}
```

Account interface

```
Interface Account{
        Public Response signIn(String s);
        Public Response signUp(String s);
        Public Response signOut(String s)
}
```

Queue interface

```
Interface Queue{
        Public Boolean addTaxi(Taxi t, Location l, Timestamp ts);
        Public Boolean removeTaxi(Taxi t);
        Public Taxi getFirst(Queue q);
        Public Boolean moveLastPosition(Taxi t)
}
```

Passenger interface

```
Interface Passenger{
        Public Response newRequest(Passenger p, Location l);
        Public Response newReservation(Passenger p, String s);
        Public Response cancelReservation(Reservation r);
        Public Response ViewBookedList(Passenger p);
}
```

Taxi interface

```
Interface Taxi{
        Public Response setAvailabilityOn(Taxi t, Location l, Timestamp ts);
        Public Response setAvailabilityOff(Taxi t);
        Public Response acceptRide(Ride r, Taxi t);
        Public Response rejectRide(Ride r, Taxi t);
        Public Response endRide(Ride r);
}
```

Zone interface

```
Interface Zone{
        Public Queue getQueue(Location l);
        Public Queue getNearestQueue(Location l, Queue q); /* returns the queue of the zone nearest to
                                                                the location after the submitted queue q*/
}
```

Ride interface

```
Interface Ride{
        Public Boolean acceptRide(Ride r, Taxi t);
        Public Boolean rejectRide(Ride r, Taxi t);
        Public Boolean endRide(Ride r);
        Public Boolean taxiInRide(Taxi t);
        Public void findTaxi(Ride r);
}
```

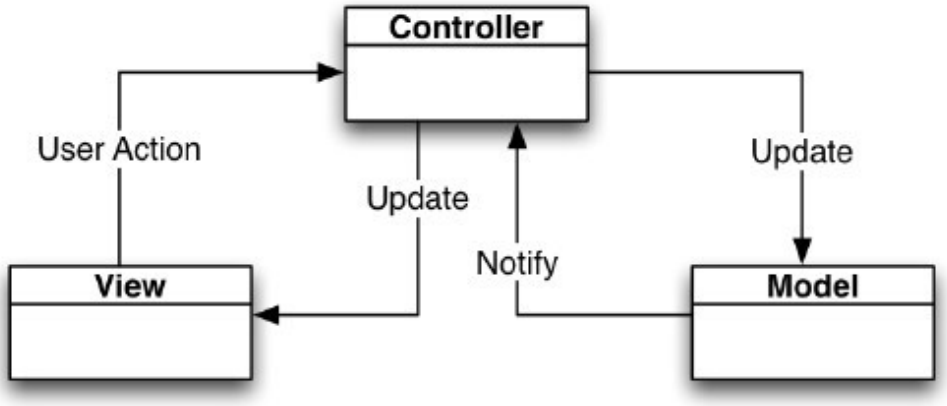
Dispatcher interface

```
Interface Request{
        Public Response request();
}
```

2.7 Architectural styles and patterns

The following design patterns have driven the design process of this project:

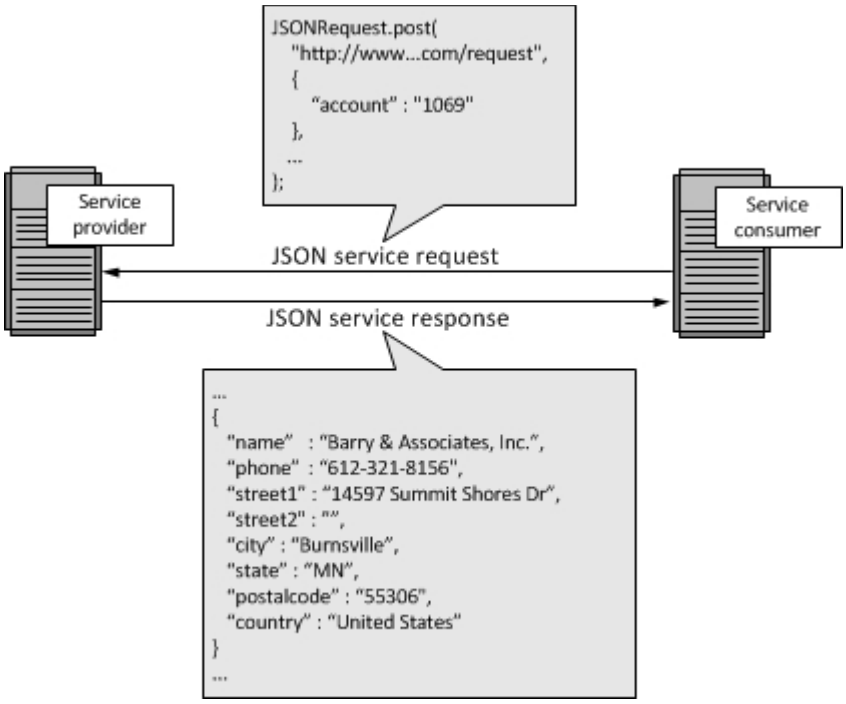
• **MVC design pattern:** Model-View-Controller. This software architectural pattern divides the business data, the user interface (or the interface between systems) and the core modules that run the business logic, so as to separate internal representations of information from the way they are presented to or accepted from the user. In myTaxiService it will be applied to the client side (User and Taxi applications).



• **Thin Client-Fat Server architecture:** The client–server model is a distributed application structure that divides tasks between the providers of a resource or service (servers) and service requesters (clients). The clients in our architecture are thin because they contain only an interaction/visualization layer and all the business logic of the application is concentrated in the application server

• **RESTful architecture:** Representational State Transfer (REST) is a style of architecture based on a set of principles that describe how networked resources are defined and addressed. An application or architecture considered RESTful is characterized by these properties: state and functionality are divided into distributed resources; every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet); the protocol is client/server, stateless, layered, and supports caching.

The following figure illustrates the usage of REST for Web Services with JSON as data-interchange format:



2.8 Other design decisions

As far as taxi ride requests and reservations are concerned, a periodic cronjob which run autonomously in a dedicate server will assign to a request/reservation the first taxi driver from the queue corresponding to the zone where it is made from calling the RideManager Component. The cronjob periodically checks in the data base if no ride is assigned to a taxi. If it is so, it calls the RideManager component to assign it a ride. Checking the data base, the cronjob also verifies whether a ride has been rejected by a taxi driver and in that event it creates a new entry of that ride (in order to assign it to another taxi).

Every client of the system periodically sends requests to the server in order to get updates through notifications with a polling request (forwarded by the Dispatcher component to the PollingManager component). For instance, taxi drivers get notifications when a ride has been assigned to them and they have to accept or reject it; passengers, instead, have to be informed about waiting time and taxi code of a reserved ride 10 minutes before it occurs (e.g., with a push notification on the smartphone). Moreover, in order to allow the system to update the queues if a taxi moves from one zone to another one while it is available, the taxi mobile application has to constantly send its current GPS location. In this way, the polling controller (PollingManager) is able to verify whether a taxi driver is still in the same zone (and in the same queue) or it has to be moved into a new queue (corresponding to the new zone).

3. ALGORITHM DESIGN

Here below are reported some mockups of the algorithms we developed for our system (not managing database errors).

FAIR MANAGEMENT OF TAXI QUEUES

*/*when a taxi driver set their availability on, the function SetAvailabilityOn on a taxiManager is invoked*/*

FUNCTION response SetAvailabilityOn(Taxi t, Location l, Timestamp ts){

*/*check if the taxi has a pending request. Taxis cannot set their availability on if they are already involved in a ride*/*

```
        if(rideManager.taxiInRide(Taxi t))
                return "error";
        else{
                /*add taxi to queue*/
                queueManager.add(Taxi t, Location l, Timestamp ts);
                // update taxi driver timestamp
                dataLayer.query("Update");
                return "add";
        }
}
```

```
FUNCTION boolean addTaxi(Taxi t, Location l, Timestamp ts){
        //take the queue relative to the position of a taxi driver
        queue q=zoneManager.getQueue(Location l);
        // add taxi to queue
        dataLayer.query("Insert");
        return true;
 }
```

*/*When a taxi driver set their availability off the function SetAvailabilityOff on the taxiManager is invoked*/*

```
FUNCTION response SetAvailabilityOff(Taxi t){
                /*remove taxi from queue*/
                queueManager.remove(Taxi t);
                return "delete";
        }
 }
```

```
FUNCTION boolean removeTaxi(Taxi t, Location l, Timestamp ts){
        // remove taxi from queue
        dataLayer.query("Delete ");
        return true;
 }
```

A RIDE IS PROPOSED TO A TAXI DRIVER

*/*The cronjob searches all the rides that are either requests or reservations and have meetingTime - nowTime <= 10 minutes. After that, for all the rides found, it calls the method findTaxi of a TaxiManager*/*

```
FUNCTION void findTaxi(Ride r){
    boolean taxiFound;
    do{
        //take the queue relative to the position of a passenger
        queue q=zoneManager.getQueue(Location l);
        //take the first taxi of the queue
        taxi t= queueManager.getFirst(Queue q);
//return the taxi of the queue with the lowest timestamp and that have no pending requests
//if the result is null or in the database there is another ride with the same idRide and TaxiAssigned (we have already proposed the ride to this taxi driver) we take another queue
        if(t is null || datalayer("Select")){
            q=zonemanager.getNearestQueue(Location l, Queue q);
            taxiFound=false;
        }
        else taxiFound=true;
    }while(taxiFound==false)
    //update the rideResponse of the taxiFound set to 1
```

```
        dataLayer.query("Update");
        //update the taxiAssigned of the ride
        dataLayer.query("Update");
    }
```

*/*Now with the polling technique the ride is proposed to the taxiDriver. If the taxi driver accepts the proposal, acceptRide is invoked on the taxiManager that invoked the acceptRide on the rideManager*/*

```
    FUNCTION boolean acceptRide(Ride r, Taxi t){
        //remove the taxi from the queue
        queue.removeTaxi(Taxi t);
        //update the rideResponse of the taxiFound set to 0
        dataLayer.query("Update");

        //update of the ride with resultRequest set to 1
        dataLayer.query("Update");
        return true;
    }
```

*/*if the taxi driver refuses the proposal, acceptRide is invoked on the taxiManager that invoked the rejectRide on the rideManager*/*

FUNCTION boolean rejectRide(Ride r, Taxi t){

//move the taxi to the last position of the queue

queue.moveLastPosition(Taxi t);

//update the rideResponse of the taxiFound to 0

dataLayer.query("Update");

//update of the ride with resultRequest set to 0

dataLayer.query("Update");

return true;

}

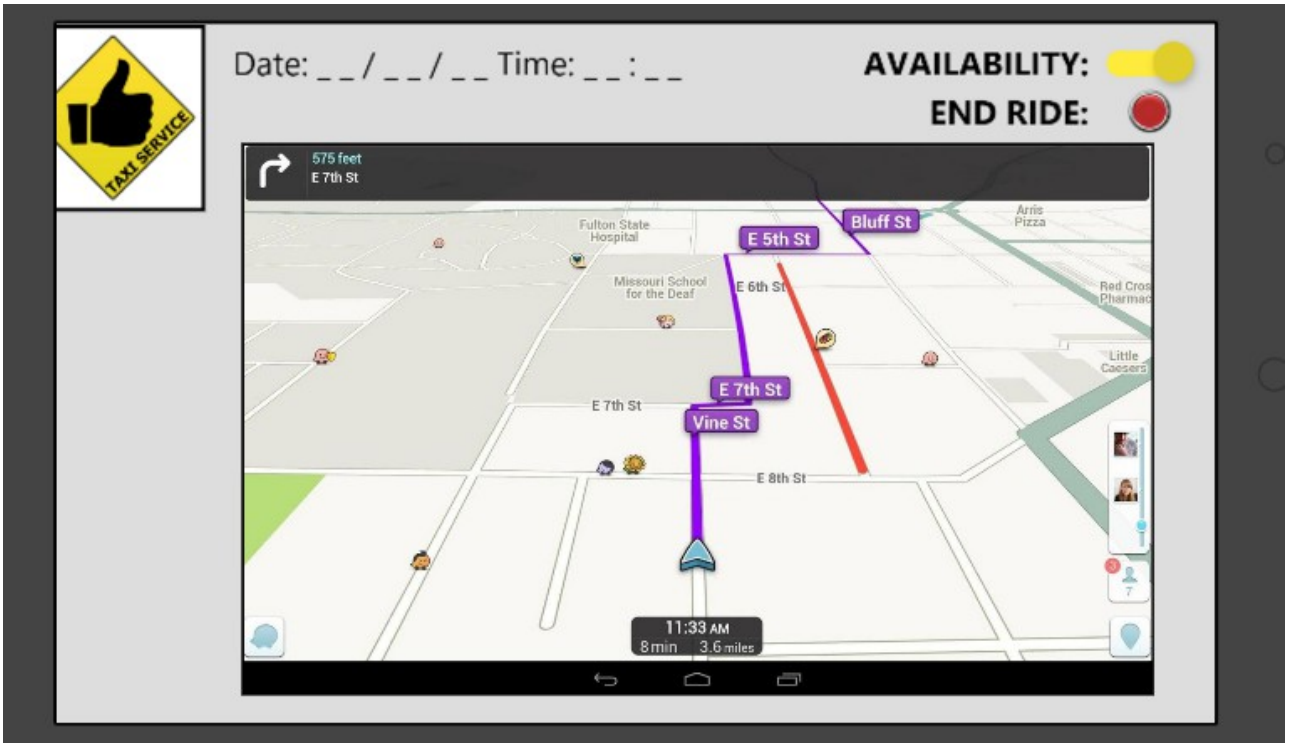
4. USER INTERFACE DESIGN

The GUI of the web application for Users and its relative usage is very similar to the mobile one (described in detail in the chapter 3.2.1 of the RASD document).

Here below, a couple of examples:

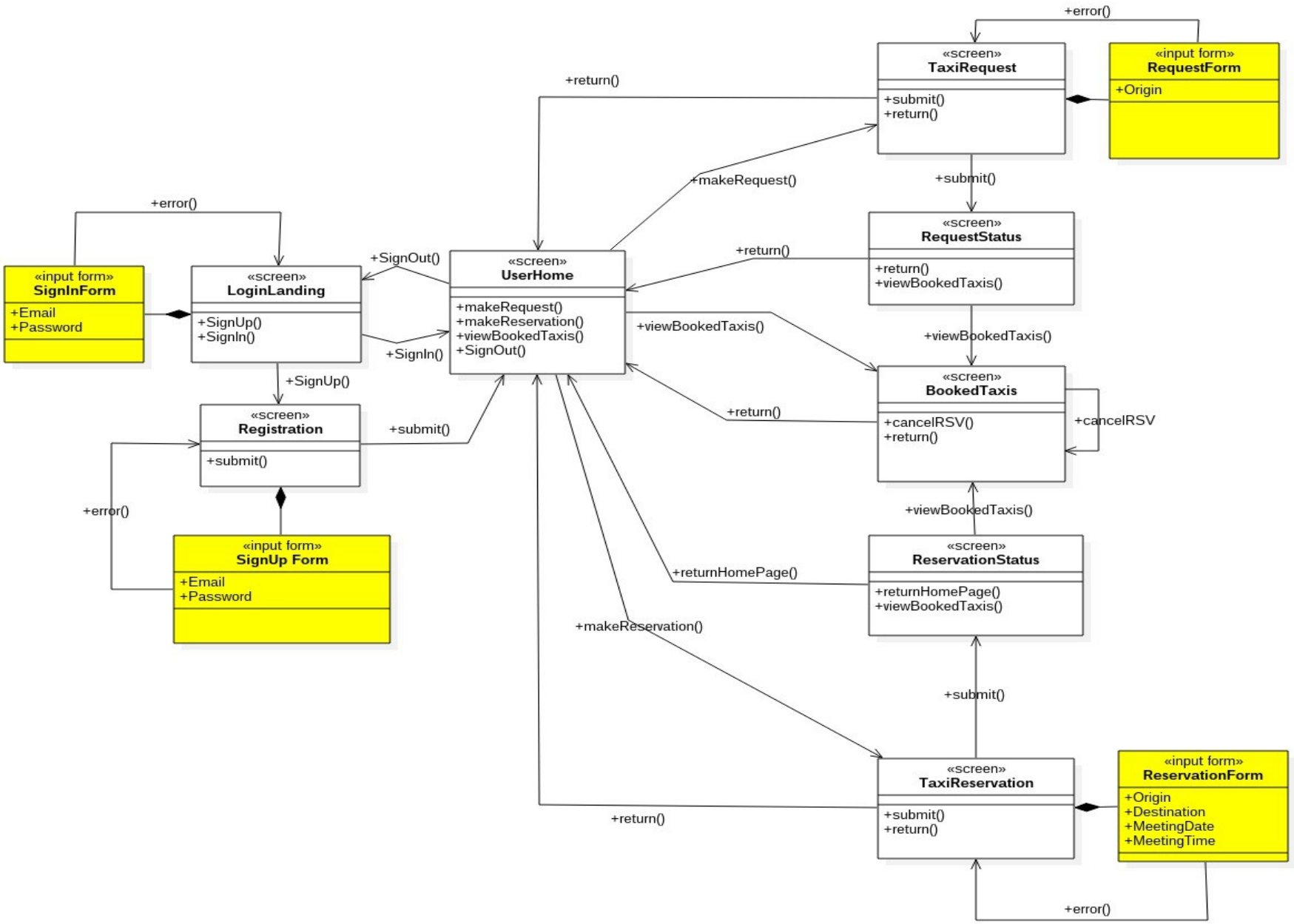
The only difference from the mobile application is the presence of a form concerning the Origin of a ride in the Request page (since the position cannot be acquired via GPS).

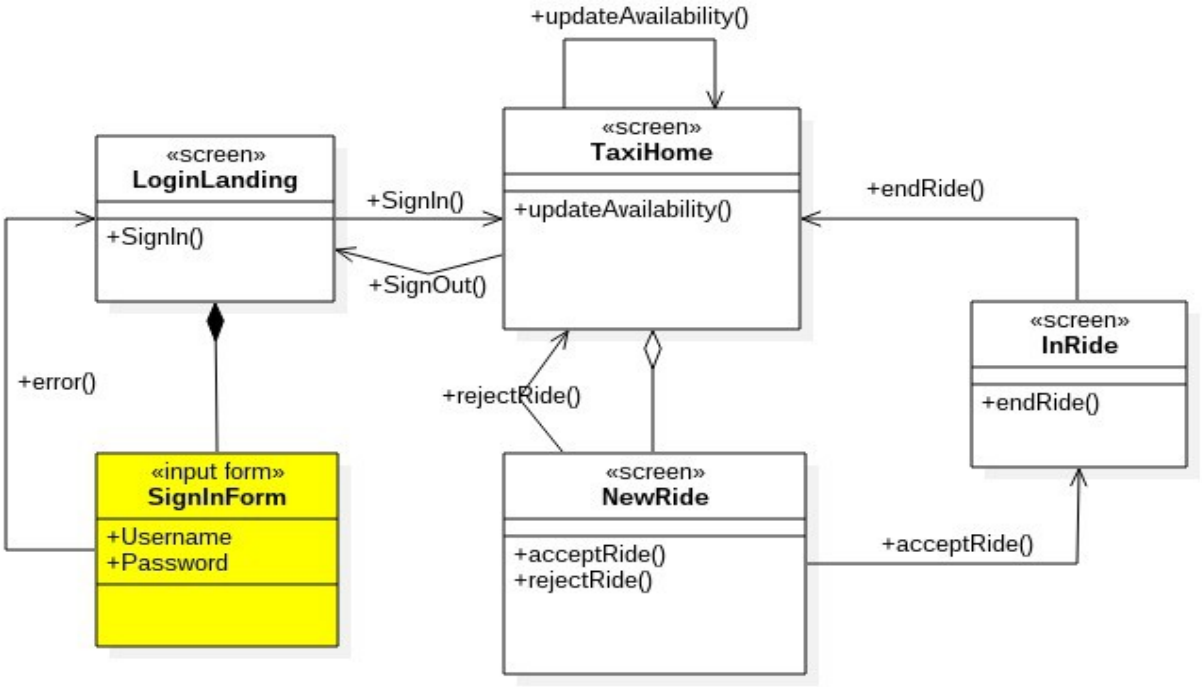
As far as the GUI for the taxi application is concerned, with respect to the one presented in the RASD, we pointed out how myTaxiService communicate with Google Maps through its API in order to provide the right directions to a driver that confirms a call. Here below an example of how the GUI should look like:



The user experience is represented with UX diagrams, that are composed by different objects. A <<screen>> is a web/mobile page that contains some elements, like for example an <<input form>> where the user can insert some information. Here below, to better represent the totality of the user experience, we chose to show the **UX diagrams for the web application for users and the Auto application for taxis**.

(For a detailed description of the flow of events, see chapter 3.2.1 of RASD document).


«input form»
SignInForm
+Email
+Password
«screen»
LoginLanding
+SignUp()
+SignIn()
«screen»
Registration
+submit()
«input form»
SignUp Form
+Email
+Password
«screen»
UserHome
+makeRequest()
+makeReservation()
+viewBookedTaxis()
+SignOut()
«screen»
TaxiRequest
+submit()
+return()
«input form»
RequestForm
+Origin
«screen»
RequestStatus
+return()
+viewBookedTaxis()
«screen»
BookedTaxis
+cancelRSV()
+return()
«screen»
ReservationStatus
+returnHomePage()
+viewBookedTaxis()
«screen»
TaxiReservation
+submit()
+return()
«input form»
ReservationForm
+Origin
+Destination
+MeetingDate
+MeetingTime
+error()
+SignOut()
+SignIn()
+SignUp()
+submit()
+error()
+return()
+makeRequest()
+submit()
+return()
+viewBookedTaxis()
+viewBookedTaxis()
+return()
+cancelRSV
+viewBookedTaxis()
+returnHomePage()
+makeReservation()
+submit()
+return()
+error()



5. REQUIREMENTS TRACEABILITY

ACTOR:

REQUIREMENT → COMPONENT

GUEST:

Sign up into the system → LoginManager

USER:

Log in into the system → LoginManager

Create a ride request → PassengerManager

Create a ride reservation → PassengerManager

Check the list of booked taxis → PassengerManager

Cancel a ride reservation → PassengerManager

TAXI DRIVER:

Confirm a ride (reservation or request) → RideManager

Reject a ride (reservation or request) → RideManager

update taxi availability → TaxiManager

SYSTEM:

Assign a ride to a taxi → Cronjob + RideManager + PollingManager + TaxiGUI

Confirmation of request/reservation to the user with related information → PollingManager + PassengerWebGUI /PassengerMobileGUI

Insert, remove or move a taxi from a queue → PollingManager + QueueManager

Update availability of taxis → TaxiManager

6. USED TOOLS

- **LibreOffice Writer 5.0:** to redact and format the document
- **Proto.io:** to create the user interface mockup of the mobile applications (User and Taxi)
- **Balsamiq Mockups**: to create the user interface mockup of the web application
- **StarUML 2:** to create UX diagrams
- **Visual Paradigm 12.2 Community Edition:** to create component, deployment and sequence diagrams
- **Lucid Chart:** to create the architectural model

7. HOURS OF WORK

Since we are neighbors, we have worked **together almost all the time** at each other's home and **equally shared all the tasks and efforts**. We have worked on this document **for a total of 30 hours.**

8. REFERENCES

Here is a short list of the references for this document:

- Slides of the Software Engineering 2 course (from the Beep Platform)
- Design Document Template (by Prof. Raffaela Mirandola)
- http://www.service-architecture.com
- https://docs.oracle.com/javaee/