Software Design Document

# MyTaxiService

*Authors:*
M. Albanese, M. Bianchi, A. Carlucci

POLITECNICO
MILANO 1863

# Contents

# 1 Introduction

## 1.1 Purpose

This document shows the architecture underlying myTaxiService, starting from the specifications and requirements described in the RASD document. It is addressed to developers and maintainers primarly. The main focus is to show the major choices about software and hardware, through several UML and UX diagrams in increasing detail: the goal is to explain how each component interact with each other using a standard language.

## 1.2 Scope

The system will be an optimization of a pre-existing, non-software solution for renting taxis already in use in the city. The new system will let users to rent or reserve a taxi through a mobile or a web application and will also let taxi drivers to take care of the users' requests in a more simple and effective way. In addition to a better user interface, the new system will focus on a smarter organization of the vehicles deployed in each city zone, resulting in a more efficient service for the citizens.

## 1.3 Definitions, acronyms and abbreviations

The following are used throughout the document:

**RASD** Requirements Analysis and Specification Document.

**DD** Design Document.

**RDBMS** Relational Data Base Management System.

**DB** DataBase, handled by a RDBMS.

**JDBC** Java DataBase Connectivity.

**UI** User Interface.

**Application server** the component which provides the application logic that interacts with the DB and with the front-ends.

**Back-end** term used to identify the Application server

**MVC** Model-View-Controller.

**SOAP** Simple Object Access Protocol.

**JAX-WS** Java API for XML Web Services.

**JPA** Java Persistence API.

# 2 Architectural Design

## 2.1 High level components and their interaction

According to the requirements described in [1] and the typical guidelines for an enterprise application, we designed the architecture depicted in Figure 1.

The main logic lies in the Application Server, which contains 5 **main macrocomponents**:

- **Data Layer**: *data* and *data access*;
- **UserManager**: this layer contains all operations regarding *guests* and registered *users* (login, logout, editing profile. . . ); it interacts with the data layer to accomplish them.
- **RideManager** which contains all operations regarding *calls* (be them *reservations* or *requests*) and associated *rides*. Common operations involve inserting a new call (even a shared one) and getting info about a ride (for example, the precalculated fee or the available payment methods). It interacts with the data layer to accomplish all this.
- **TaxiAllocationManager** which contains all operations regarding the *allocation* of a new ride. It interacts with the data layer to accomplish them.
- **TaxiDriverManager** which contains all operations for the interaction between the application server and the driver mobile app. It uses

Two **clients** interact with the application server: the usual *client interface* (via web or mobile application) and the *mobile driver application*, which has dedicated functions and a completely different UI.

Eventually two **external services** are used: one handling payments done via POS and another one for pushing notifications into the driver app (for more information on this, see [3]).

For a detailed description of each component please refer to section 2.2 instead.

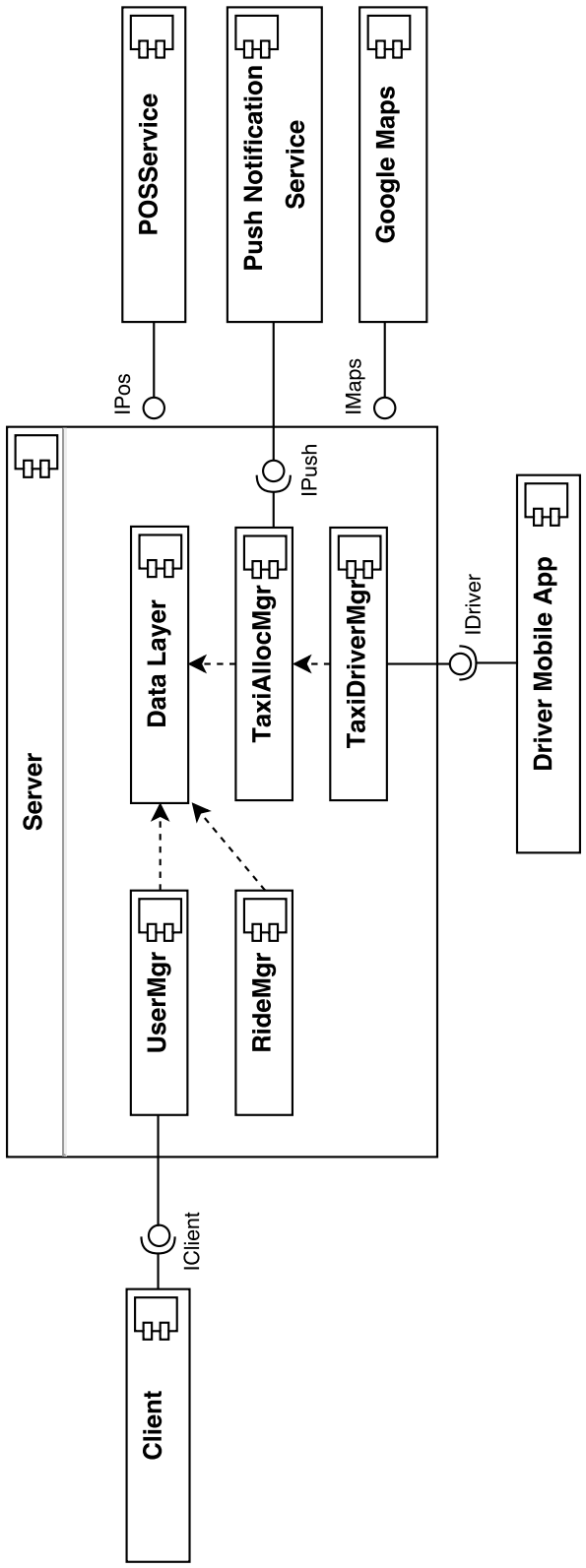



Figure 1: High Level Architecture

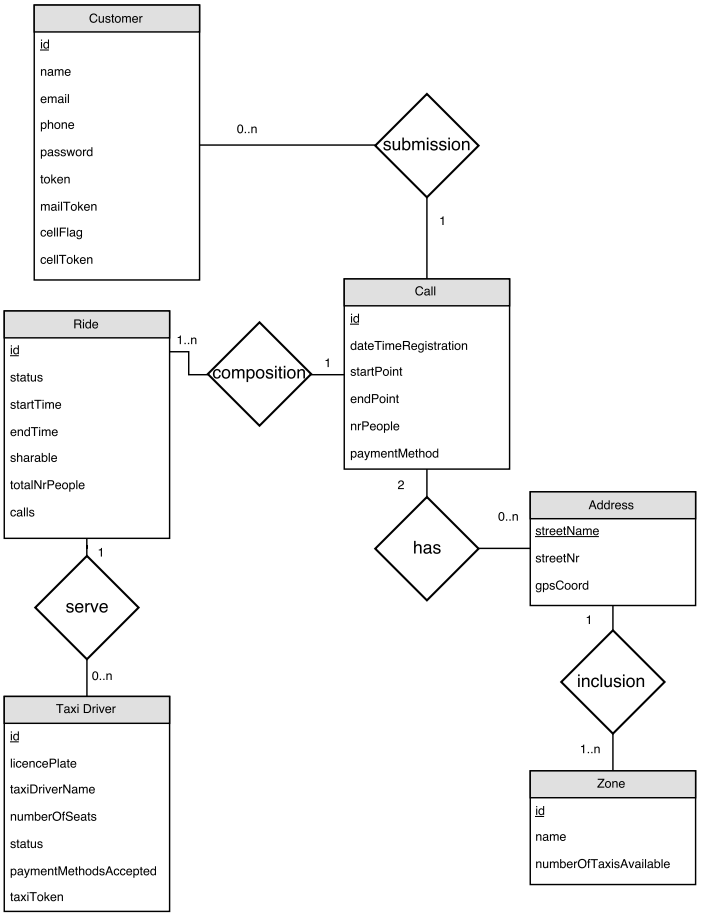



Figure 2: Entity Relationship Diagram

## 2.2 Component view

### 2.2.1 Data Layer

This layer contains all *Entity Beans* (object/relational mappings with the DB) and the classes that guarantee access to the DB (and actually retrieve data for these entities, the so-called *DAOs*). In order to better represent the structure of the database supporting this service, a ER diagram is provided.

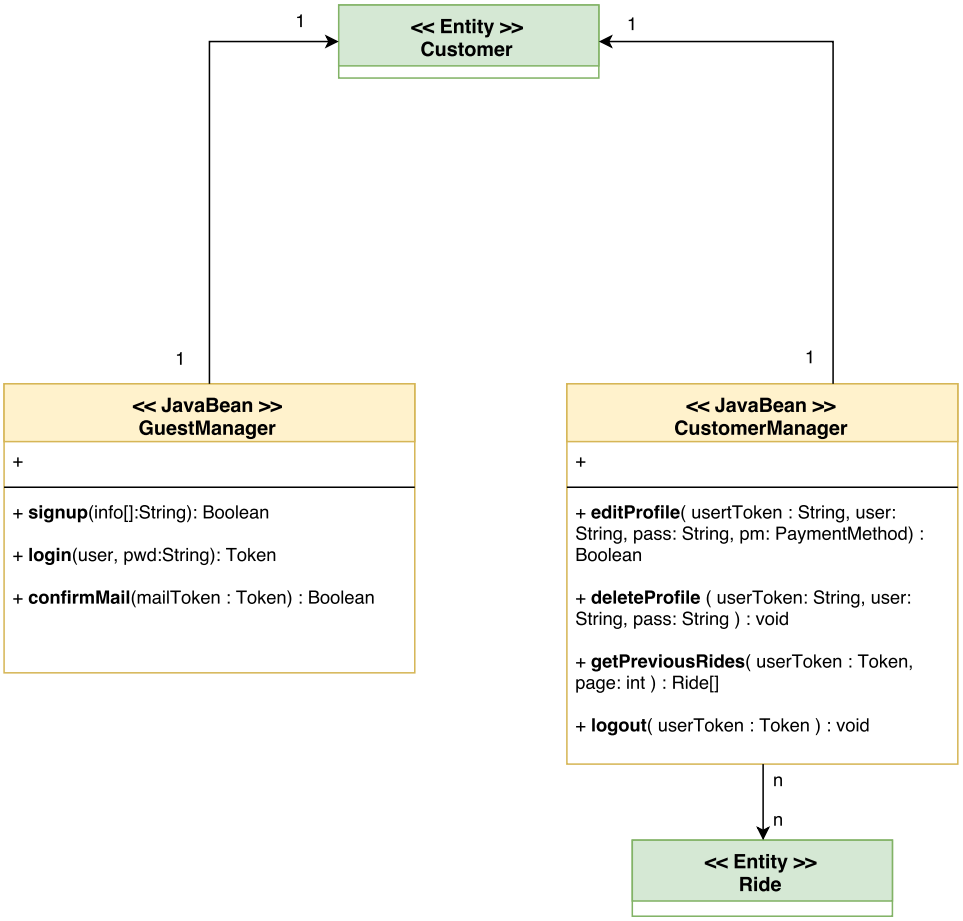



Figure 3: Component view: Client

### 2.2.2 Client

Here are contained all classes related to a *customer*. In particular, two managers (one for guests and one for registered users) are given.

Whenever a *guest/customer* wants to do something strictly related to themselves (e.g. signing up, logging in...) he has to refer to this component.

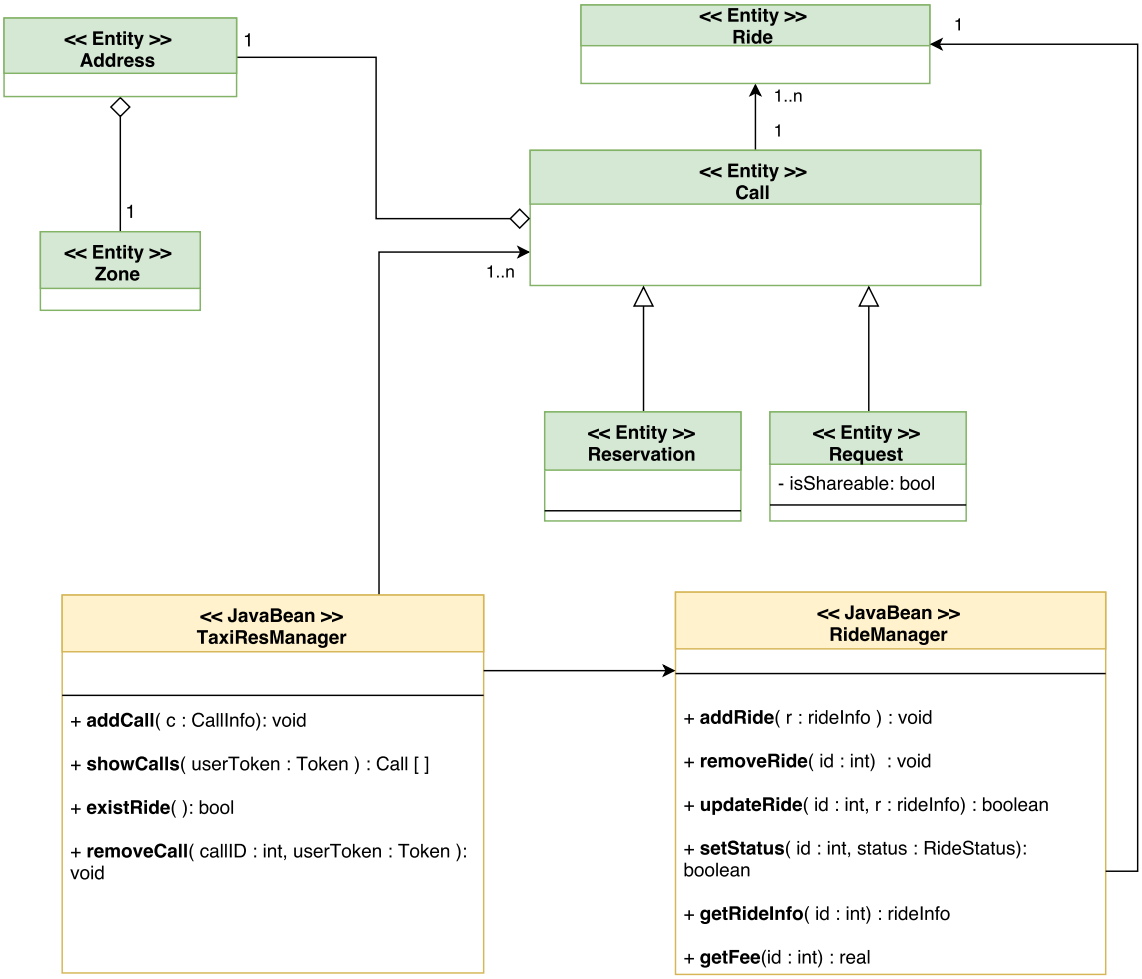



Figure 4: Component view: Taxi Reservation

### 2.2.3 Taxi Reservation

This component is responsible for two main objects in the entire system, Calls and Rides. A *Call* is either a request or a reservation made by a customer, whereas a *Ride* describes the effective ride by a Taxi and can be associated with multiple Calls.

Whenever an action involves one of these classes, this is the component that must be used.

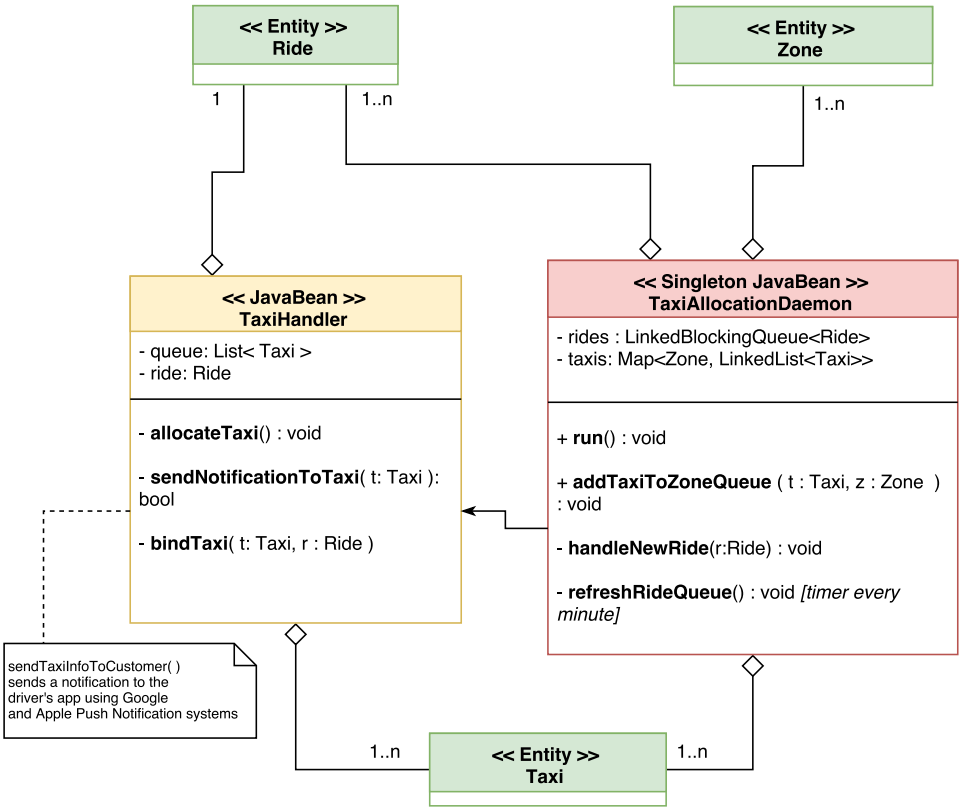



Figure 5: Component view: Taxi Allocation

### 2.2.4 Taxi Allocation

This is a central component for the service, since it is responsible for the actual *allocation* of a taxi for a certain ride. It holds a daemon always refreshing the ride queue (`refreshRideQueue()`) to be processed from the DB by any TaxiHandler created by it.

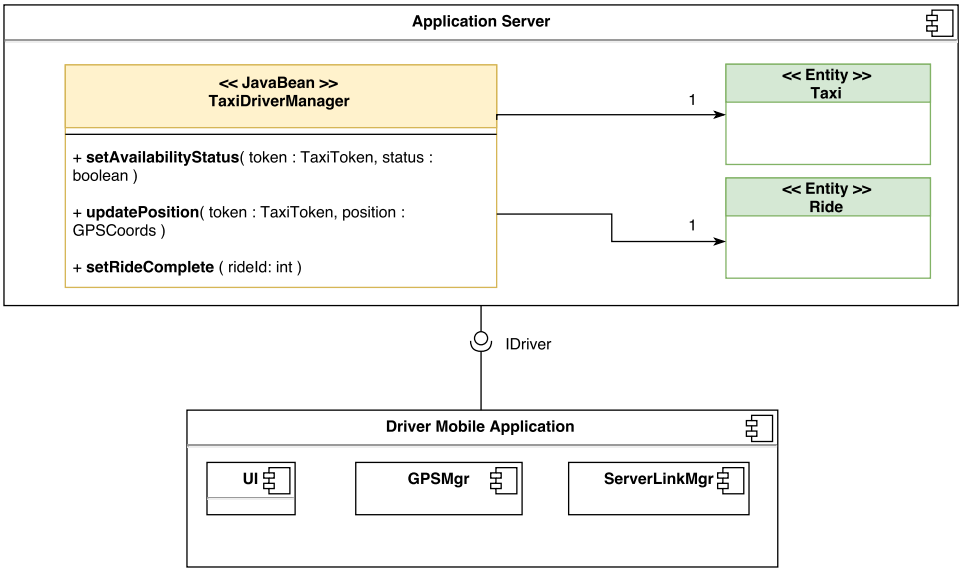



Figure 6: Component view: Taxi Driver

### 2.2.5 Taxi Driver

This component is responsible for the interaction between the *mobile application* and the application server (for example, updating the position of a cab according to the GPS data). Whenever a taxi driver has to do something related to the system, he has to use classes from this component. Please note that only classes on the server side are specified: those regarding the mobile application are no further detailed.

## 2.3 Deployment view

In this section we will talk describe the deployment of the application. In Figure 7 you can easily see the physical structure of the system. In the following paragraphs we will explain in more detail relevant items.

**Servers** We choose to use three different physical servers as *Web server*, *Application server* and *Database server*.

Each of them is a *Dell PowerEdge R220*[7], a very common and powerful rack server. Each server is running a Linux distribution called Debian [1] which is a very secure and customizable operating system, the right choice for our system.

All of these three servers must be physically protected and the communication between every tier must be encrypted using *Transport Layer Security* (TLS) protocol.

**Database server** This server will run the *Database Management System* (DBMS). MySQL Community Edition is the designated software. It is highly reliable, free and widely used. When more processing power is needed, the DBMS will be migrated to *MySQL Cluster Edition*. Application server will be configured to use this server as database backend. This server should communicate only with the application server using the standard MySQL protocol, wrapped into a TLS layer.

**Application server** GlassFish, one of the most used J2EE implementations will be installed in this server and configured to serve our application. Database communication is made possible by *Java Persistence API* (JPA), which will wrap all database functionalities. This server will handle all the logic behind the system. GlassFish is a highly scaleable software server, if more computational power is needed we will increase the number of the servers in this tier. Communication with the Web Tier and with the smartphone App relies on *Simple Object Access Protocol* (SOAP).

**Web server** This server is responsible for the Web Interface. The whole web interface will be implemented using *Java Server Faces* (JSF), while the software web server will still be Glassfish Server.

**Clients** We currently support 2 different kind of clients: *Smartphone users* and *Web Interface users*. The smartphone application directly connects to the Application Server via SOAP protocol while the Web interface will be accessible by connecting to our Web Server via a common Web browser.

[1] Debian is a free operating system based on Linux and the ISO image is downloadable from its website, `https://www.debian.org`

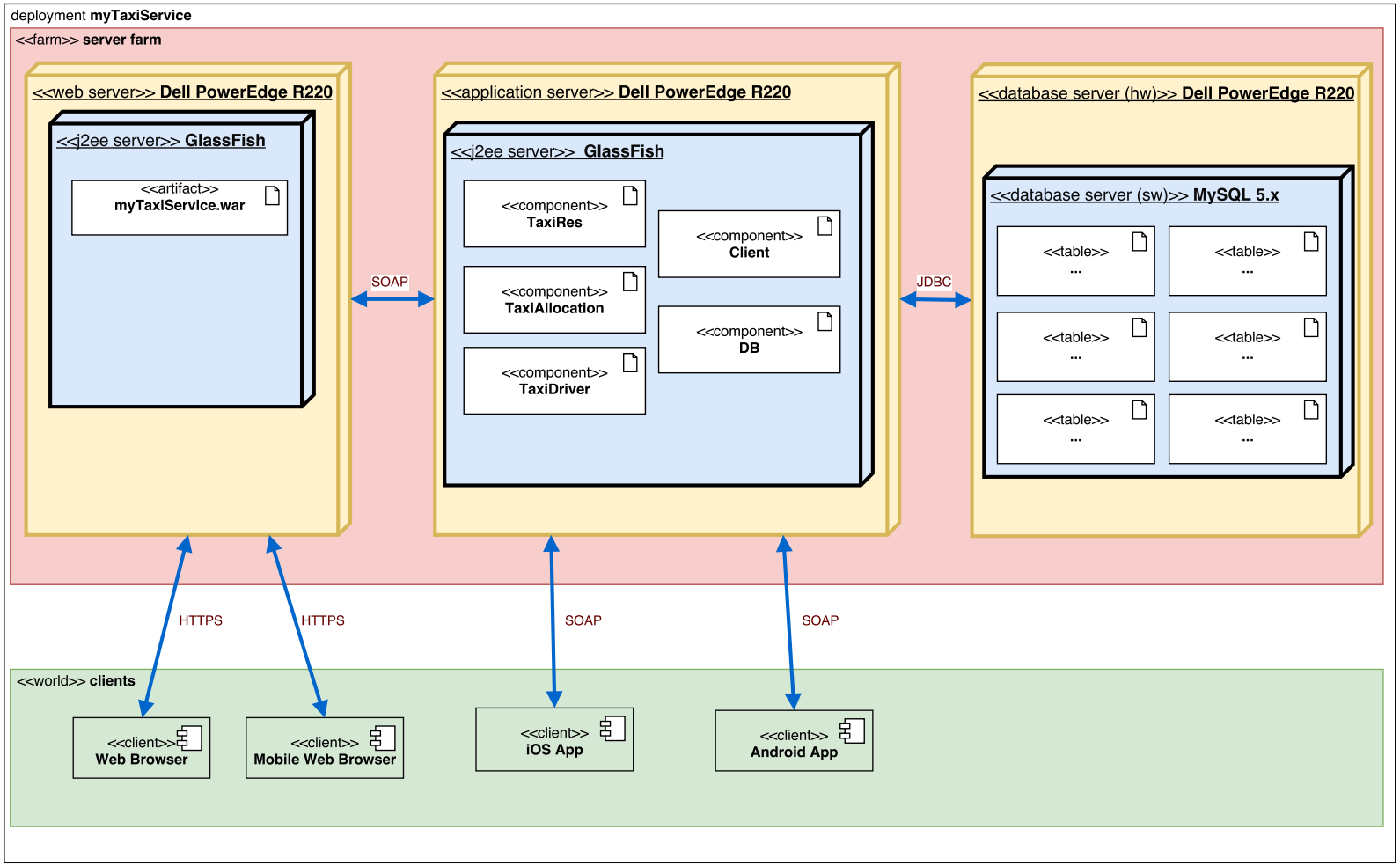



Figure 7: Deployment Diagram

## 2.4 Runtime view

Since the actual representation of processes and threads running within myTaxiService would have been quite complex and difficult to explain, no runtime unit diagrams have been used. As shown in the algorithm section (3), there is however something important on this point: TaxiAllocationDaemon and TaxiHandler work in a similar way to SocketManager and Socket. In fact, TaxiAllocationDaemon *listens* to new rides through a repeated fetch operation on the DB and allocates a new TaxiHandler to manage one ride at a time.

Nevertheless, in order to catch the behaviour of the overall system, here are presented the improved sequence diagrams, better explaining what happens *inside the box*.

### 2.4.1 Sign in

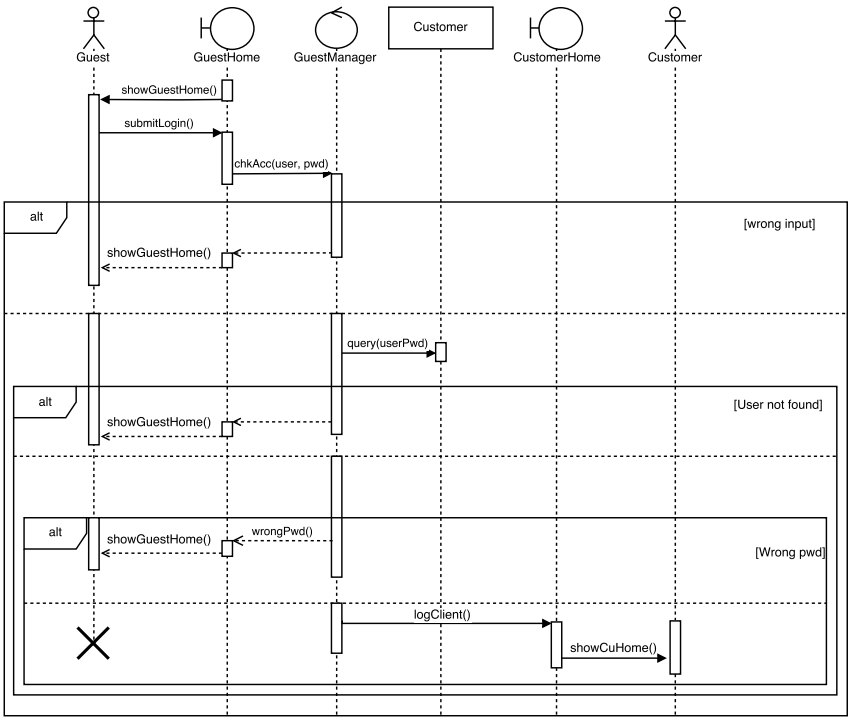



Figure 8: Sequence Diagram, Sign In

### 2.4.2 Sign up

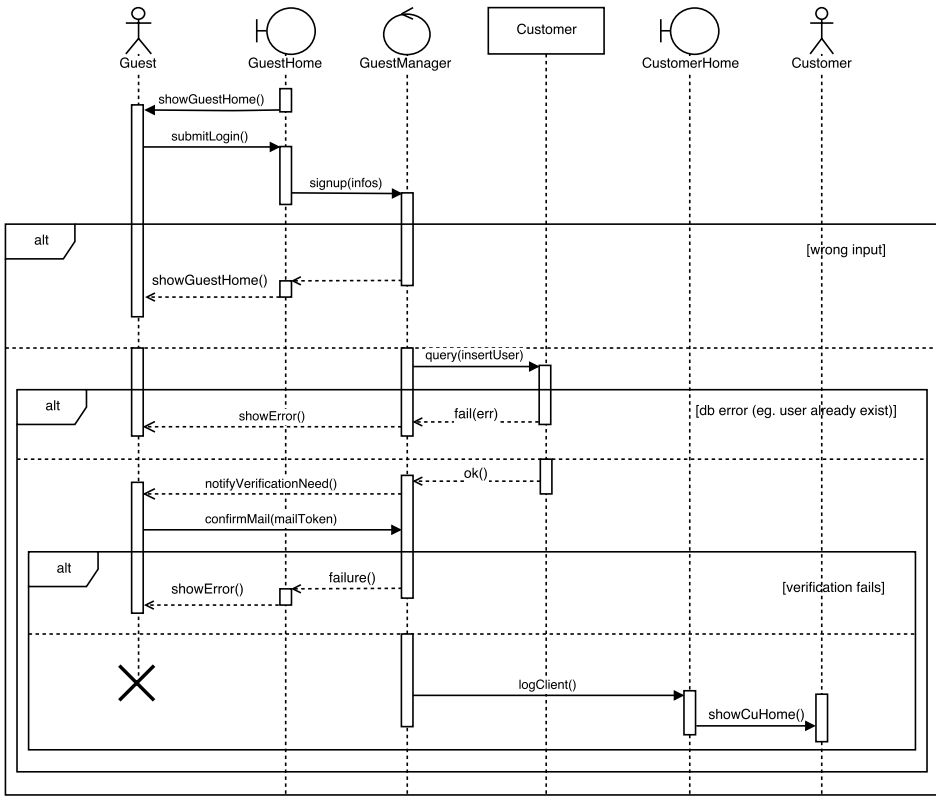



Figure 9: Sequence Diagram, Sign Up

### 2.4.3 Allocate a taxi

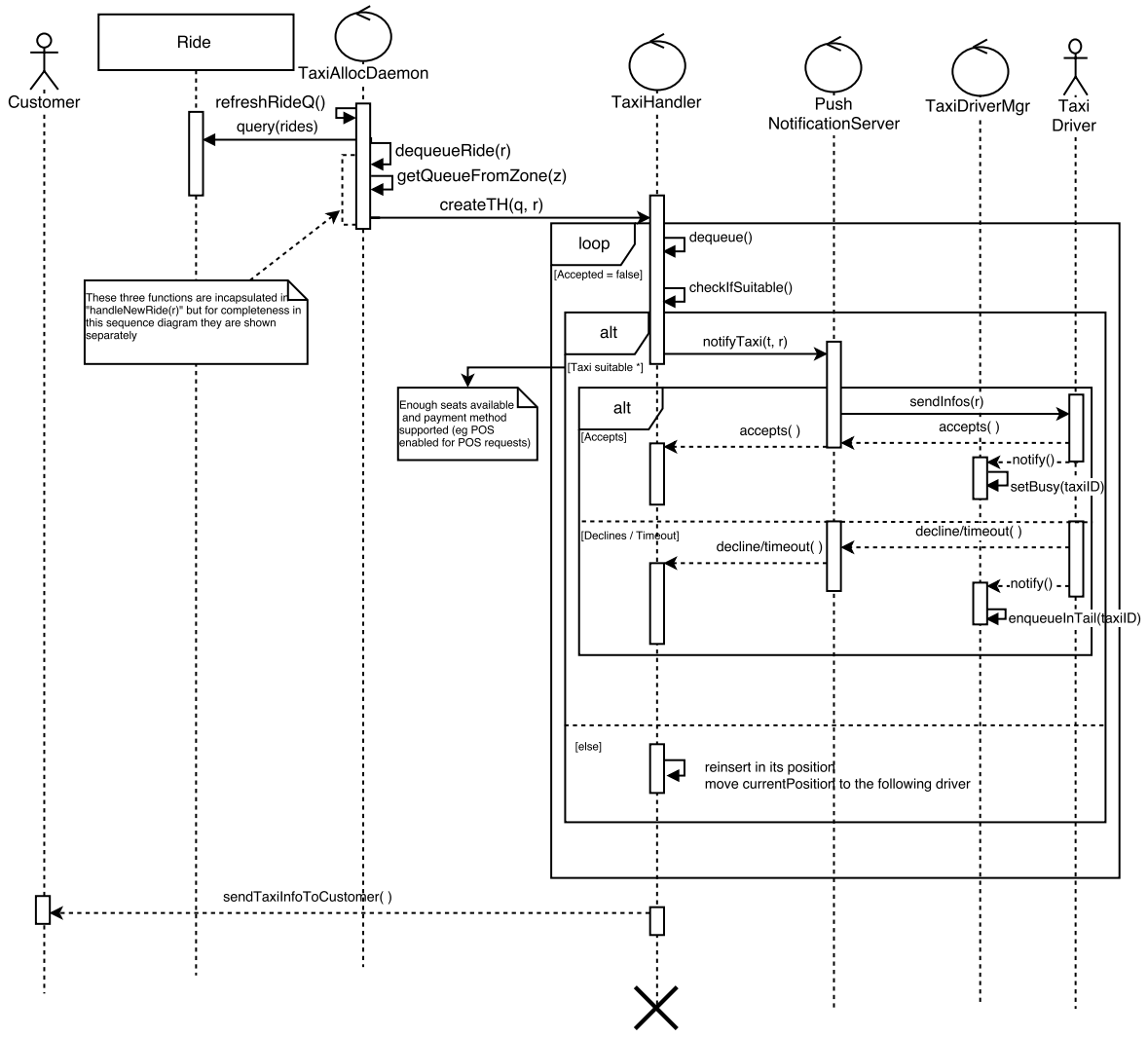



Figure 10: Sequence Diagram, Allocate a Taxi

### 2.4.4 Request a taxi

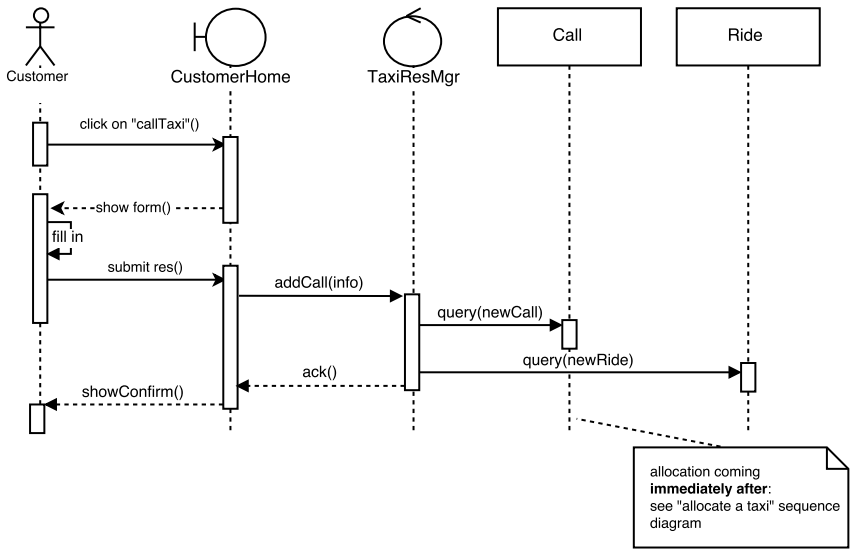



Figure 11: Sequence Diagram, Request a Taxi

### 2.4.5 Reserve a taxi

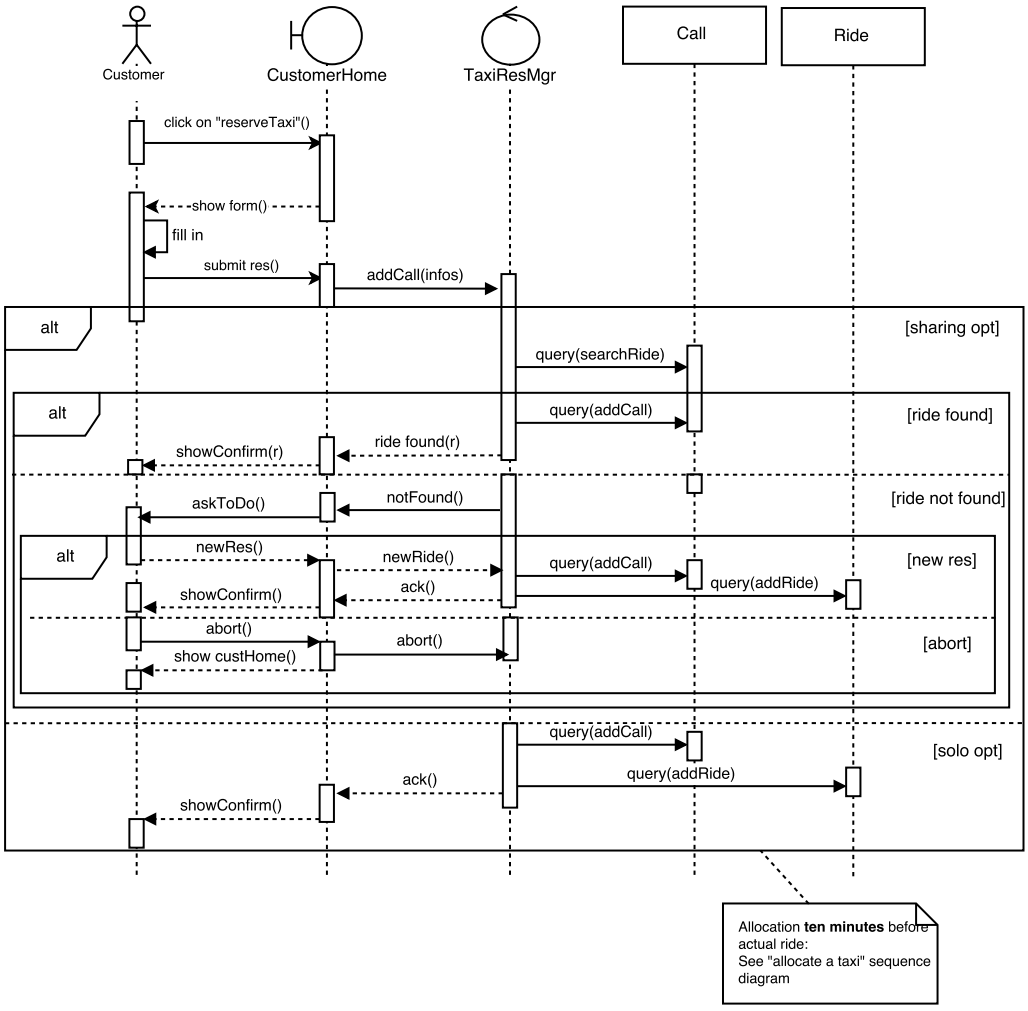



Figure 12: Sequence Diagram, Reserve a Taxi

### 2.4.6 Delete reservation

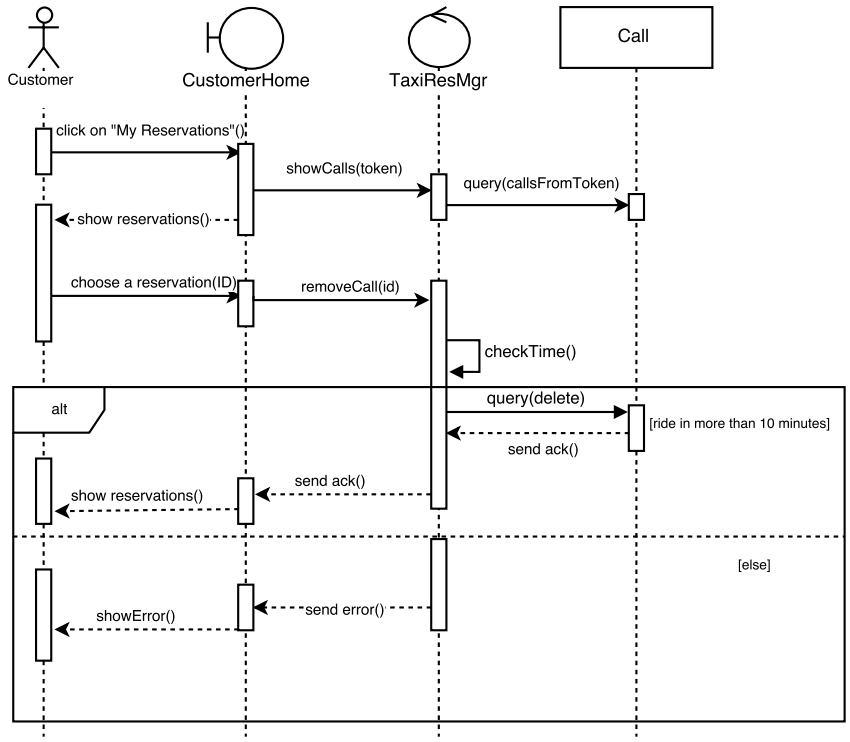



Figure 13: Sequence Diagram, Delete reservation

## 2.5 Selected architectural styles and patterns

This section describes high level patterns we decided to use in myTaxiService.

**Deployment** A four-tier architecture is used in this project. The whole infrastructure is divided in the following tiers:

- **Client tier** which is composed of web browsers and mobile app.
- **Presentation tier** whose aim is to generate user interface and send it to clients.
- **Logic tier** which coordinates the application: it performs calculation, makes logical decisions and moves data between Presentation tier and Data tier.
- **Data tier** mainly made of entity beans and database. Here information is stored and retrieved.

We choose to use this kind of architecture because it provides a model by which we can create a flexible and scalable system.

**Communication** We chose to use a *Service-Oriented Architecture*, an architecture which provides application functionality as a set of services, and applications that use those software services.

Services are *loosely coupled* units of functionality that are self-contained, Each service is an implementation of some interfaces, which provide a communication schema with each application. Also, interfaces can be published and invoked.

**Structure** Here we chose to use the Object-Oriented Architectural Style. This is a design paradigm based on the division of responsibilities for a complex system into small and reusable parts called "Objects". They communicate with each other through interfaces, by sending and receiving messages or by calling methods in other objects.

The main benefits of this approach are that it is:

- **Extensible** because there are a lot of structures which ensures that a change of implementation does not imply a change of interface.
- **Reusable** if each object should be developed as a reusable small piece of code.
- **Testable** because of the incapsulation, which improves testability.
- **Understandable** because it maps the application more closely to the real world.

## 2.6 Component interfaces

### 2.6.1 Connection client - web server

The connection through the website to the service is guaranteed through standard HTTPS requests and are not described in detail here.

### 2.6.2 Connection application server - DB

The application server uses EntityBeans to map relations into Java objects, which themselves refer to the underlying DB through JDBC calls. Since these connections are pretty standard (i.e. connection, query, handling results), they are not described in detail here.

### 2.6.3 Web Service: JAX-WS

The following public methods are available through a Web service, which is handled by JAX-WS on the server side. This means that each client knowing the address of the service can use it according to the functions described in detail below. Authorization for specific operations is controlled by tokens, as it is required in the parameter list.

**Guest Manager** The GuestManager bean is responsible for operations on *unregistered* users; it let customers register, sign in or confirm their email after signing up.

| Method name | Token | User | Parameter Name | Parameter Description | Returns |
|---|---|---|---|---|---|
| signup | NO | Guest | Username<br>Password<br>Mail | Username for this customer<br>Password for this customer<br>Mail for this customer | True if operation succedeed |
| login | NO | Guest | Username<br>Password | Username chosen during signup<br>Hashed password for this user | Access token |
| confirmEmail | NO | Customer | Token<br>Mail | Mail token sent during registration<br>Mail to be confirmed | Boolean response |

**Customer Manager** The CustomerManager bean is responsible for operations on *registered* users;

| Method name | Token | User | Parameter Name | Parameter Description | Returns |
|---|---|---|---|---|---|
| editProfile | YES | Customer | Username<br>Password<br>Payment Method | New username<br>New password<br>New payment method (POS or cash) | Boolean response |
| deleteProfile | YES | Customer | Username<br>Password | Username chosen during signup<br>Password for this username | - |
| getPreviousRides | YES | Customer | Page | Page number of rides (10 rides per page) | List of rides |
| logout | YES | - | - | - | - |

**TaxiResManager** TaxiResManager is responsible for the management of any call (aka request or reserve).

| Method name | Token | User | Parameter Name | Parameter Description | Returns |
|---|---|---|---|---|---|
| addCall | YES | Customer | dateTime<br>rideDateTime<br>startPoint<br>endPoint<br>nrPeople<br>paymentMethod | When the call was made<br>When the ride starts<br>Starting point in associated ride<br>Ending point in associated ride<br>How many seats are reserved<br>POS / Cash | - |
| showCalls | YES | Customer | - | - | Array of calls by this customer |
| removeCall | YES | Customer | IDCall | Call ID in DB table | - |

**TaxiDriverManager** TaxiDriverManager is responsible for the management of a single taxi driver through his own app.

| Method name | Token (Taxi) | User | Parameter Name | Parameter Description | Returns |
|---|---|---|---|---|---|
| setAvailabilityStatus | YES | Taxi Driver / System | Status | New status for this taxi | - |
| updatePosition | YES | Taxi Driver | GPSCoords | Current GPS Coordinates for taxi | - |
| setRideComplete | YES | Taxi Driver | rideID | ID for current ride | - |

### 2.6.4 Internal classes

Here are described the classes that are somewhat important for the service but that are not available publicly.

**RideManager** RideManager is responsible for any action involving a ride: adding one when a call is added to the DB, modifying one when something changes (for example, a new client in a shareable ride), deleting one or setting a new status for it.

| Method name | Token | User | Parameter Name | Parameter Description | Returns |
|---|---|---|---|---|---|
| addRide | YES | System | IDRide<br>startTime<br>nrPeople<br>expectedFee | Ride ID<br>When the ride starts<br>Total number of people for this taxi<br>Fee calculated in advance for each customer | - |
| removeRide | YES | System | IDRide | Ride ID in DB table | - |
| updateRide | YES | System | RideID<br>nrPeople<br>expectedFee<br>endTime | Ride ID<br>New total number of people<br>New expected fee<br>When the ride ends (optional) | Boolean response |
| setStatus | YES | Systen | IDRide<br>Status | Ride ID<br>New status for this ride | - |
| getRideInfo | YES | System | IDRide | Ride ID in DB table | Ride Info |
| getFee | YES | System | IDRide | Ride ID in DB table | - |

**TaxiAllocationDaemon** This is a Singleton JavaBean, responsible for allocating a taxi for each ride when necessary. It also holds all the Queues, divided by zone, for fast access.

| Method name | Token | User | Parameter Name | Parameter Description | Returns |
|---|---|---|---|---|---|
| addTaxiToZoneQueue | YES (Taxi Driver) | Taxi Driver | taxi<br>zone | The taxi to be added<br>The zone where to add the taxi | - |
| handleNewRide | NO | System | - | - | - |
| refreshRideQueue | YES | System | - | - | - |

## 2.7 Other design decisions

There are two aspects of myTaxiService we haven't discussed yet: communication with smartphones and geolocalization.

**Mobile communication** We need to send instant notifications to user or taxi's mobile devices. There is a standard way to do this, which is the one we have chosen, *Push Notification*.

We decided to support both iOS and Android smartphones. In order to send push notifications to iOS smartphone, we have to use *Apple Push Notification Service*. The same applies to Android world, Google has its own push service, called *Google Cloud Messaging*.

**Geolocalization** We decided to use Google Maps as geolocalization service because it is highly reliable, fast and extremely used in the world. Also, it offers powerful APIs that will be used to calculate the best route among two or more points in the map (for example, when shared rides are involved).

# 3 Algorithm Design

## 3.1 Taxi Allocation Daemon

In this chapter there are some of the useful algorithms used in this project. The first one is the taxi allocation daemon which receives new taxi requests and allocates a TaxiHandler for each request.

Listing 1: Taxi allocation daemon

```
1  @Startup
2  @Singleton
3  class TaxiAllocationDaemon {
4      /** Available taxis */
5      private Map<Zone, LinkedList<Taxi>> taxis;
6
7      /** A blocking queue containing new rides */
8      private LinkedBlockingQueue<Ride> rides;
9
10     /** ----- Class Methods ----- */
11
12     /** Class constructor */
13     public TaxiAllocationDaemon () {
14         /** Move arguments to class variables */
15         taxis = new HashMap<Zone, LinkedList<Taxi>>();
16         rides = new LinkedBlockingQueue<Ride>();
17     }
18
19     /** Main function */
20     @PostConstruct
21     public void run(){
22         /** Keep looping while system is running */
23         while( true ) {
24             /** Get the next ride.(Blocking call) */
25             Ride ride = rides.take();
26
27             /** Get the associated zone */
28             Zone zone = ride.getZone();
29
30             /** Create and launch a TaxiHandler */
31             new TaxiHandler(taxis.get(zone), ride)
32                 .start();
33         }
34     }
35
```

```
36      /** Add a taxi to the specified zone queue */
37      public void addTaxiToZoneQueue(
38          Taxi taxi, Zone zone) {
39          LinkedList<Taxi> list = taxis.get(zone);
40
41          list.add(taxi);
42      }
43
44      /** This method is called when a new ride
45       * is ready to be served */
46      private void handleNewRide(Ride ride) {
47          rides.add(ride);
48      }
49
50      /** This method is called every minute */
51      @Schedule(minute="*")
52      private void refreshRideQueue() {
53          // (...)
54          if( newRideAvailable ) {
55              handleNewRide(ride);
56          }
57      }
58  }
```

### 3.2 TaxiHandler

This code is needed to find a suitable taxi for the designated ride.

Listing 2: TaxiHandler

```
1  class TaxiHandler extends Thread {
2      private List<Taxi> taxis;
3      private Ride ride;
4
5      public TaxiHandler(LinkedList<Taxi> taxis,
6          Ride ride) {
7          this.ride = ride;
8          this.taxis = taxis;
9      }
10
11     @Override
12     public void run() {
13         selectTaxi();
14     }
15
16     private void selectTaxi() {
17         /** Get the number of passengers */
18         int nrPeople = ride.getNumberOfPassengers();
19         PaymentMethod method = ride.getPaymentMethod();
20         Taxi designatedTaxi = null;
21
22         while( designatedTaxi == null ) {
23
24             /** Find a valid taxi */
25             for(Taxi taxi : taxis) {
26                 if( taxi.availableSeats() >= nrPeople
27                     && taxi.getPaymentMethods().
28                         contains(method)
29                 ){
30                     designatedTaxi = taxi;
31                     break;
32                 }
33             }
34
35             /** No taxi available */
36             if(designatedTaxi == null) {
37                 throw
38                     new AgainstAssumptionsException();
39             }
```

```
40
41              if(!sendNotificationToTaxi(
42                  designatedTaxi)) {
43
44                  /** Re-enqueue taxi */
45                  taxis.remove(taxi);
46                  taxis.add(taxi);
47
48                  /** Choose another taxi */
49                  designatedTaxi = null;
50              }
51          }
52
53          /** Bind the taxi with the ride */
54          bindTaxi(designatedTaxi, ride);
55
56          /** This taxi is now busy. */
57          taxis.remove(designatedTaxi);
58      }
59
60      // (...)
61  }
```

### 3.3 Process call algorithm

This algorithm is needed to process a call and add that in a suitable ride. It takes as input parameter the call and returns the associated ride.

Listing 3: processCall()

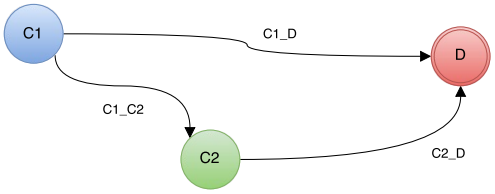
```
1  public Ride processCall(Call call) {
2      /** Is this call is a shareable reservation? */
3      if(call instanceof Reservation &&
4          ((Reservation)call).isShareable() {
5
6          /** Get call parameters,
7           *  used to lookup active rides */
8          Address startPoint = call.getStartPoint();
9          Address endPoint = call.getEndPoint();
10         int nrPeople = call.getNrPeople();
11
12         /** Find a suitable ride */
13         Ride ride = lookupSharedRide(
14             startPoint,
15             endPoint,
16             nrPeople
17         );
18
19         /** There is a suitable shared ride. */
20         if(ride != null) {
21             ride.addCall(call);
22             return ride;
23         }
24
25         /** Otherwise, create a new shared ride */
26         return createSharedRide(call);
27     } else {
28         /** This call is not shareable.
29          *  Let's create a new normal ride
30          */
31         return createNormalRide(call);
32     }
33 }
```

### 3.4 Divide fees

This is a quite simple algorithm used to divide the fee between people who are joining the ride. We decided to equally divide the fee because even if the first customer would have to pay some extra route in order to reach others, everyone starts from the same area; this means that $C1_{C2} \ll C1_D$ and $C1_D \simeq C2_D$, where $X_Y$ is the cost of reaching Y from X. This implies that

$$C1_D \gg \frac{C1_{C2} + C2_D}{2} \tag{1}$$

which makes the assumption reasonable.




The actual fee is calculated in advance, when a ride is created or updated (for example, for sharing o for a different destination), according to the following formula:

$$fee(h, km) = c(h) + km * d \tag{2}$$

where $c(h)$ represents a fixed, hour-dependent cost (night rides are more costly than day ones), whereas $d$ represents the *total distance*, evaluated using Google Maps API (see 2.7).

It finally follows that:

$$fee_i = \frac{fee}{n} \tag{3}$$

where $n$ is the number of customers in that taxi.

# 4 User Interface Design

This section briefly illustrates the design behind the UI for the main app and the taxi driver app. In order to accomplish this, two UX diagrams are provided:

- The first one represents the structure of the taxi driver's mobile app. A much simpler interface is provided (mainly for two operations: accept/refuse request and obtain directions towards destination).
- The second one represents the structure of the official mobile and web app. Each screen gives the possibility to use a main functionality (request a taxi, reserve one, sign up...). For a visual representation of each screen, please refer to the RASD document.

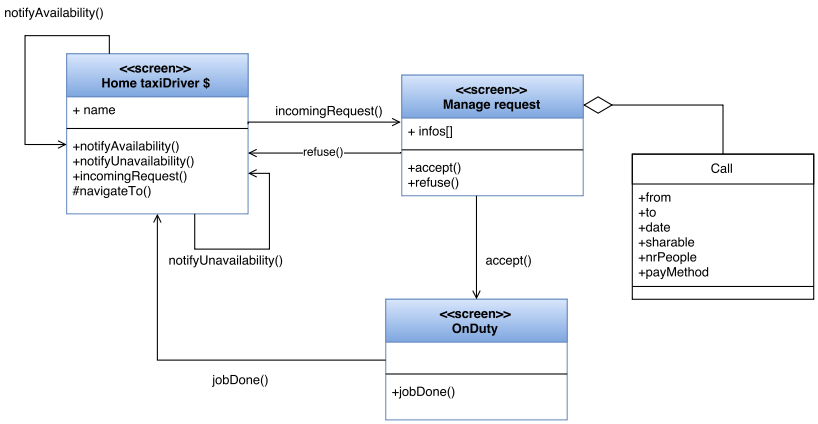



Figure 14: UX for driver app

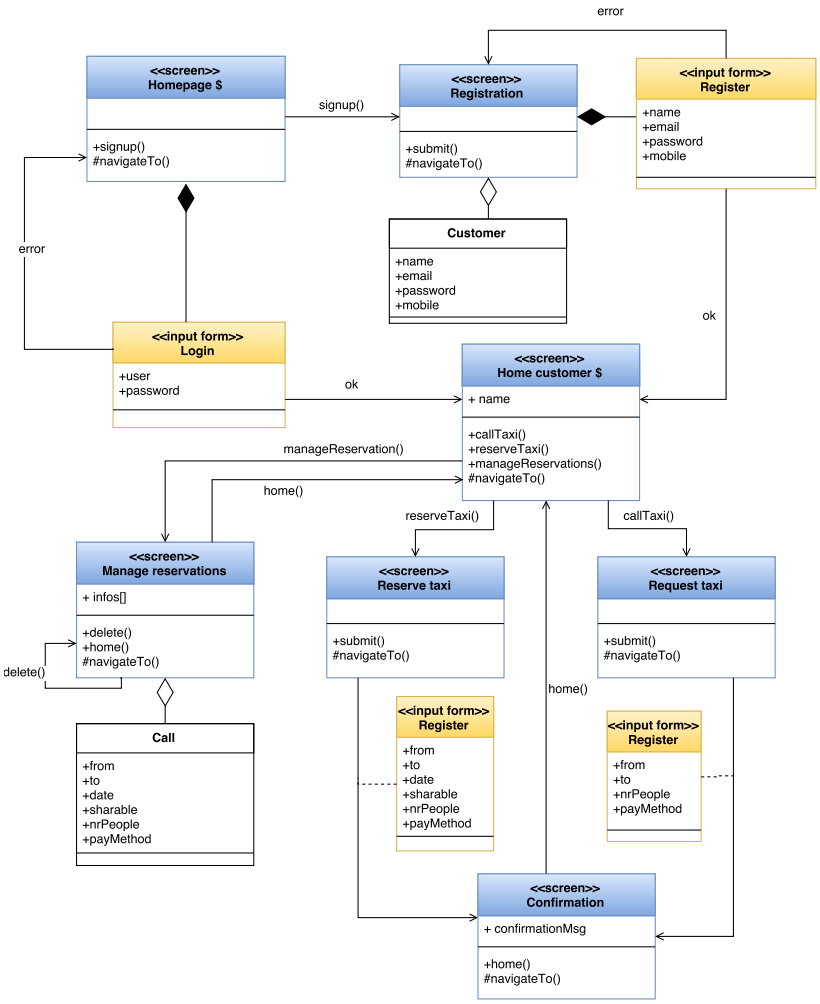



Figure 15: UX for main application

# 5 Requirements traceability

## Functional requirements

All functional requirements expressed in section 2 in the Requirements Document have been analyzed in order to comprehend which components were necessary and how they had to relate one to the other.

In the following table, each requirement is backed up by some model (Entity) and business logic classes already presented in section 2.2. Please refer to each subsection for more information on each class. In addition to this, a sequence diagram is provided where possibile.

| **Functional requirement** | **Business Logic** | **Entity** | **More info in section** |
|---|---|---|---|
| **G1:** Allow users to sign up | GuestManager | Customer | 2.4.2 |
| **G2:** Allow users to log in | GuestManager | Customer | 2.4.1 |
| **G3:** Guarantee a fair management of any queue | TADaemon, TaxiHandler, DriverManager | Ride, Taxi | 2.4.3 |
| **G4:** Allow taxi drivers to notify their availability | DriverManager | Taxi | - |
| ~~**G5:** Guarantee presence of taxi drivers in every zone~~ | - | - | - |
| **G6:** Public API | GuestManager | Customer | 2.6.3 |
| **G7:** Allow customers to request a taxi | TaxiResManager | Call, Ride | 2.4.4 |
| **G8:** Allow the system to allocate a taxi | See G3 | | |
| **G9:** Allow users to reserve a taxi | TaxiResManager | Call, Ride | 2.4.5 |
| **G10:** Allow users to share a ride | See G9 | | |
| **G11:** Allow reservation management | TaxiResManager | Ride, Call | 2.4.6 |

### User Interface

Several mockups were already created and inserted into the RASD document; since no new mockups were added, please refer to section 1.5.1 in [1]. However two UX diagrams have been created to describe in detail how a customer or a taxi driver should use his app. For more information, see section 4.

### Other interfaces

The optimal route between two or more points is delegated to Google Maps API, as stated in section in section 1.5.3 in [1].

# 6 References

## References


[1] *myTaxiService - Requirement Analysis and Specification Document* Albanese M., Bianchi M., Carlucci A. - Politecnico di Milano

[2] *Apple Push Notification Service* - https://developer.apple.com/notifications/

[3] *Google Cloud Messaging* - https://developers.google.com/cloud-messaging/

[4] *Service-Oriented Architecture Definition* - http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html

[5] *Architectural Styles, Design Patterns, and Objects* - Monroe R. T., Kompanek A., Melton R., Garlan D. B. - Carnige Mellon University

[6] *Java How to Program, 7th Edition* - Harvey M. Deitel, Paul J. Deitel

[7] *Dell PowerEdge R220* - http://www.dell.com/us/business/p/poweredge-r220/fs

# Appendix A

## Tools used

The following tools were used to produce this document:

- **Draw.io** to draw all the diagrams - `https://www.draw.io`
- **LaTeX** as typesetting system to write this document
- **VIM** as editor - `https://www.vim.org`
- **TeXMaker** as editor - `http://www.xm1math.net/texmaker`

## Revision notes

- **High Level Architecture** section rewritten in order to accomodate the new high level component diagram.
- **Diagrams** slightly revisited: better layout and some error fixes
- Minor fixes in overall **layout**

## Hours spent

The following hours were spent to produce this document:

**Elenco ore DD**

| | Michele | Alain | Mattia | Cosa si è fatto? |
|---|---|---|---|---|
| 22/11/2015 | 0 | 0.5 | 0 | Al: Start deployment diagram |
| 23/11/2015 | 2 | 2 | 2 | Al: alg start, Tia:UX, Mitch:high level arch |
| 24/11/2015 | 2 | 1.5 | 2 | Al: alg finished, tia:component view, mitch:component view |
| 25/11/2015 | 2 | 1.5 | 2 | Mitch: component view (comp diagram), tia: Sequences updated, Al: ended deployment view |
| 26/11/2015 | | 1 | 1 | Tia: sequences, Al: study Java EE + draft runtime view |
| 27/11/2015 | 2 | 2 | 2 | Tia: sequences (almost finished, need 1 more hour), Mitch: Switching from php to javaEE, Al: deploymen view updated to java ee |
| 30/11/2015 | | | 1 | Tia: finished sequences |
| 01/12/2015 | 3 | 5 | 1 | Mitch: java EE study + diagrams updated, Al: Architectural styles and patterns, Tia: first ER sketc |
| 02/12/2015 | 3 | 3 | 4 | Mitch: class diagrams + interface list; Tia: ER diagram+reshape sequences; Al: Other * (gmaps + push) + Biblio |
| 03/12/2015 | 5 | 4.5 | 6 | Al + Mitch: review component, Lavoro assieme (2h) |
| 04/12/2015 | 6 | 3 | 2 | Mitch: requirement traceability + fix Al: description of deployment + figures referenced and fixed |
| | | | | |
| | | | | |
| | 25 | 24 | 23 | |