Politecnico di Milano

Software Design Document

"myTaxiService"

Nicolas Tagliabue(matr. 853097), Matteo Pagliari(matr. 854353)

December 4, 2015

Versione 1.0

Contents

1 Introduction

1.1 Purpose

The Software Design Document, is aimed to help the developers during software development, in particular looking at the architectural and core design parts of it. This document contains both textual and visual descriptions of the software project like the component, deployment and runtime views as well as a more detailed information about the graphical user interface. The SDD also helps the developers by providing an overall view of the whole project to all the people involved in the development: for example either the software development team and the graphical department can retrieve useful information from this document.

1.2 Scope

This SDD is intended for the development of myTaxiService which is a web and mobile application intended to implement an easier way to coordinate the taxi system of a particular city. This document will first of all focus on the more general aspects of the system and then will deal about more specific parts.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Client/Customer: identifies a person who is going to take usage of the myTaxiService for requesting/reserving a taxi. These two words will be used as synonymous. Note that a client/customer must be a registered user.
- Reservation: when a customer reserves a taxi via the "Reserve a taxi" functionality in either the web or mobile application.
- Request: when a customer requests a taxi via the "Request a taxi" functionality in either the web or mobile application.

1.3.2 Acronyms

- SDD: Software Design Document.
- RASD: Requirements Analysis and Specifications Document.
- MVC: Model-View-Controller.
- JEE: Java Enterprise Edition.
- JSP: Java Server Page.

1.4 Reference Documents

See the Requirements Analysis and Specifications Document (RASD)

1.5 Document Structure

The main part of the document consists in the architectural design chapter, in which are described the choices we made about the structure of the system as well as some details about the algorithms and user interface that will be implemented. It is also shown how some of the core components of the system will work together and what these components are.

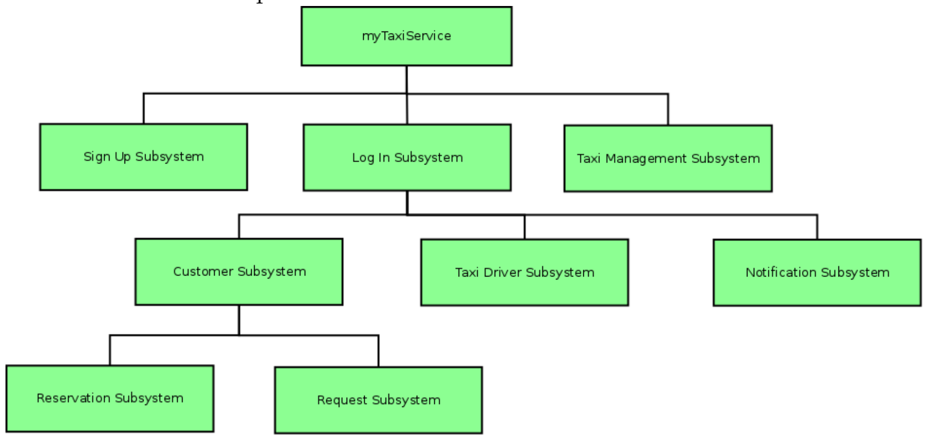
2 Architectural Design

2.1 Overview

In this part of the SDD we will focus on explaining the main components of the system, starting form a general and high level view of the whole system and then going deeper with the description. In this section we will also explain in which way software components are related to different hardware parts. In order to accomplish this, we will use a variety of different diagrams including sequence, component and deployment diagrams, and also Entity-Relations diagram.

2.2 High level components and their interactions

Here are presented all the subsystem in which our system will be divided, completed with a brief description.



Signup Subsystem

Signup subsystem allows an unregistered user to become either a taxi driver or a customer: it's job is to verify if valid data inputs are provided and store them in the database. It verifies that the username and email are univocal and also not already present in the database but also checks that the driver licence number is valid and not yet registered to this system.

Login Subsystem

Log in subsystem deals with the authentication part of the system: it grants the two different kinds of registered user their respective permits to access the different functions of the system.

Customer Subsystem

The Customer subsystem deals with all the request and reservation made by the customers as well as the management of his/her profile page. In fact it is strictly connected to the request and reservation subsystems. Both this two subsystems takes care of all the calls made by the customers passing all the required information to the other subsystems involved in order to satisfy the request like the taxi management subsystem. It checks also that all the user provided information are valid and respectful of the assumptions and constraints made in the RASD.

Taxi Management Subsystem

The Taxi management subsystem takes care of everything needed in order to manage all the taxi registered. It deals with the queue manager that adds or removes the selected taxi to the specific queue based on their location and availability status. It processes the information received from the Customer subsystem regarding the incoming taxi requests and reservations and selects the first taxi available based on customer position and queue situation. It also shares information with the other subsystems involved.

Taxi Driver Subsystem

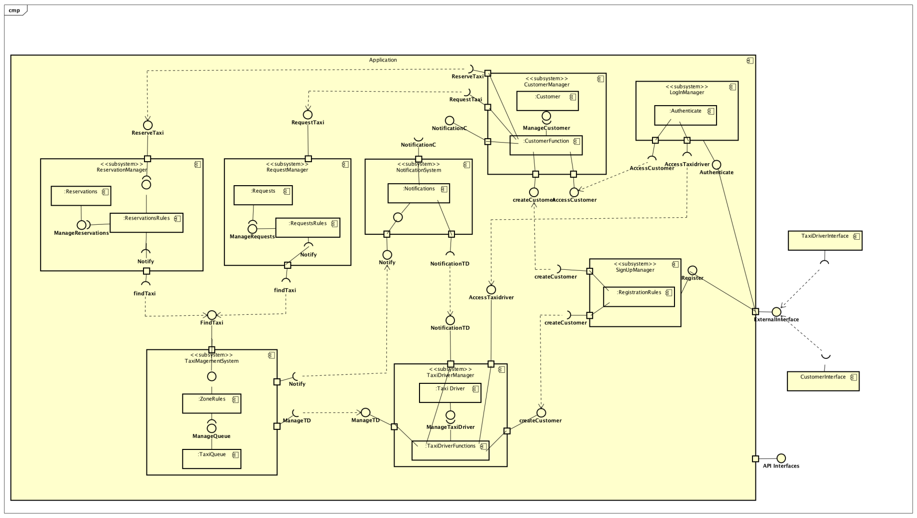
The Taxi driver subsystem takes care of allowing the taxi driver to set his/her availability status and to receive the notifications of the incoming request calls.

Notification Subsystem

The notification subsystem takes care of all the messages and notification that occur while the system is working. This subsystem provides notifications for a new customer call on the taxi driver side, but also provides notifications on the customer side, when a taxi has been assigned to him/her.

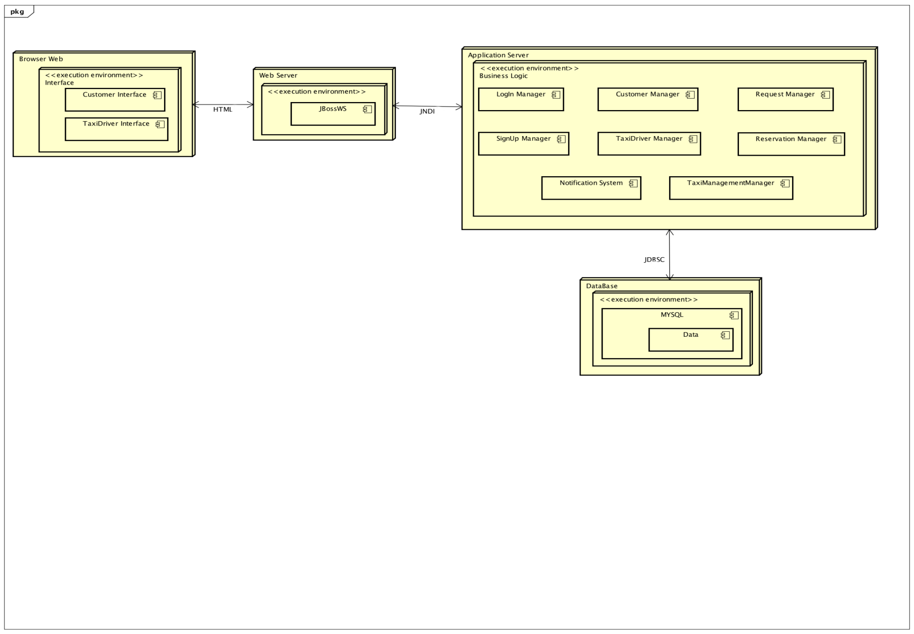
2.3 Component View

This is the component diagram of our application.



2.4 Deployment View

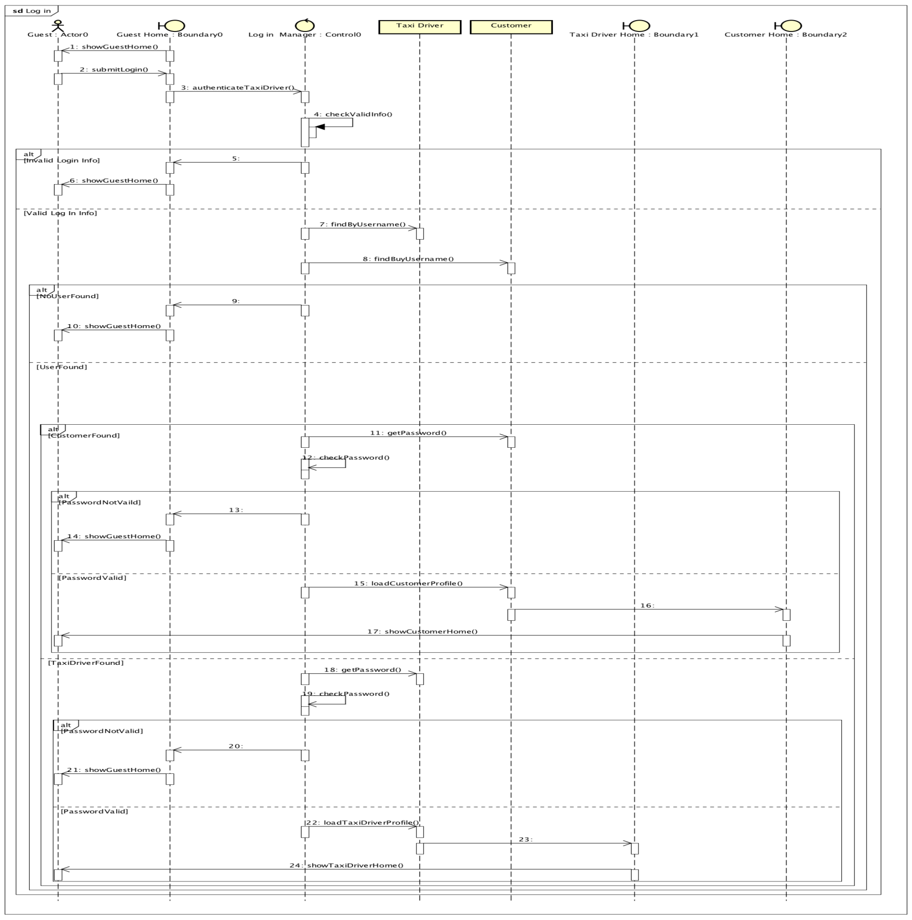
This is the deployment diagram of our system in which is represented how the software components are matched with the hardware.



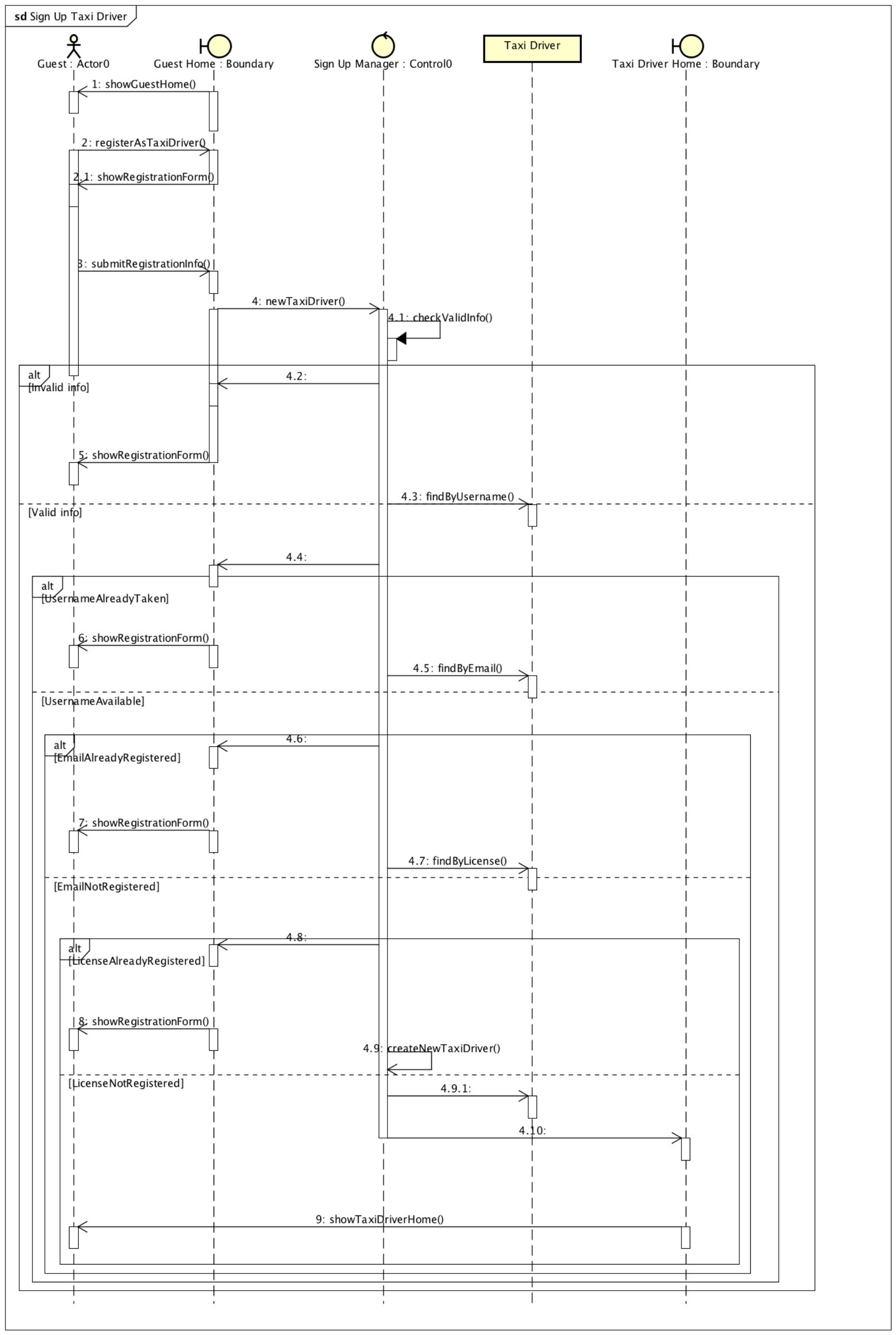
2.5 Runtime View

Here are presented some sequence diagram describing different functionality

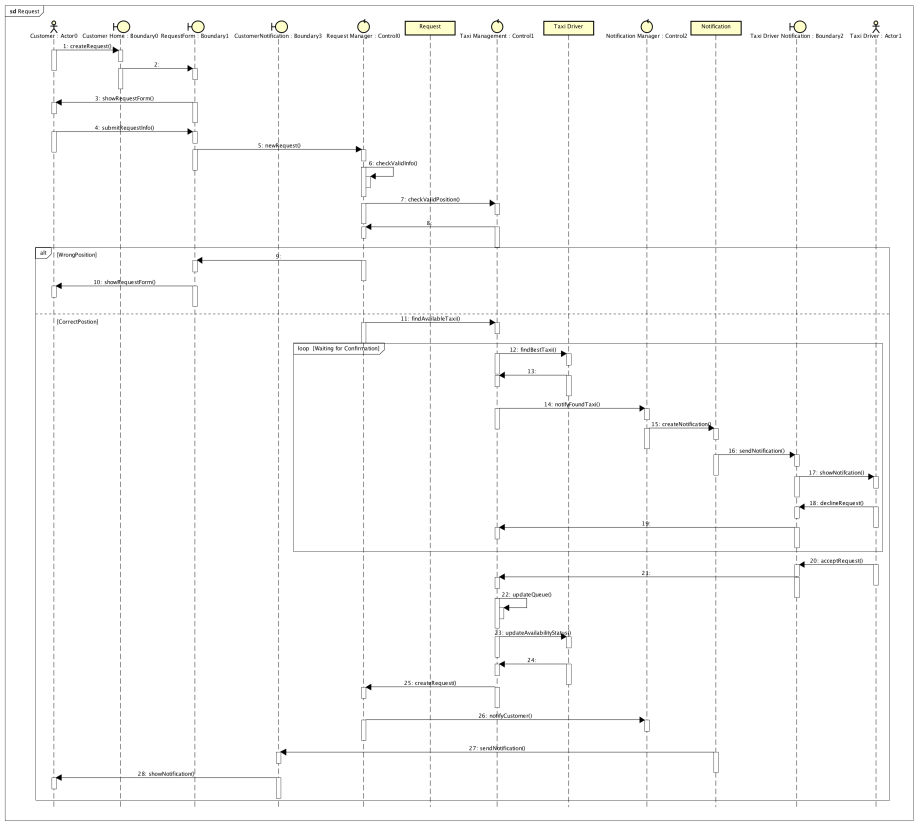
2.5.1 Log in of a Taxi Driver



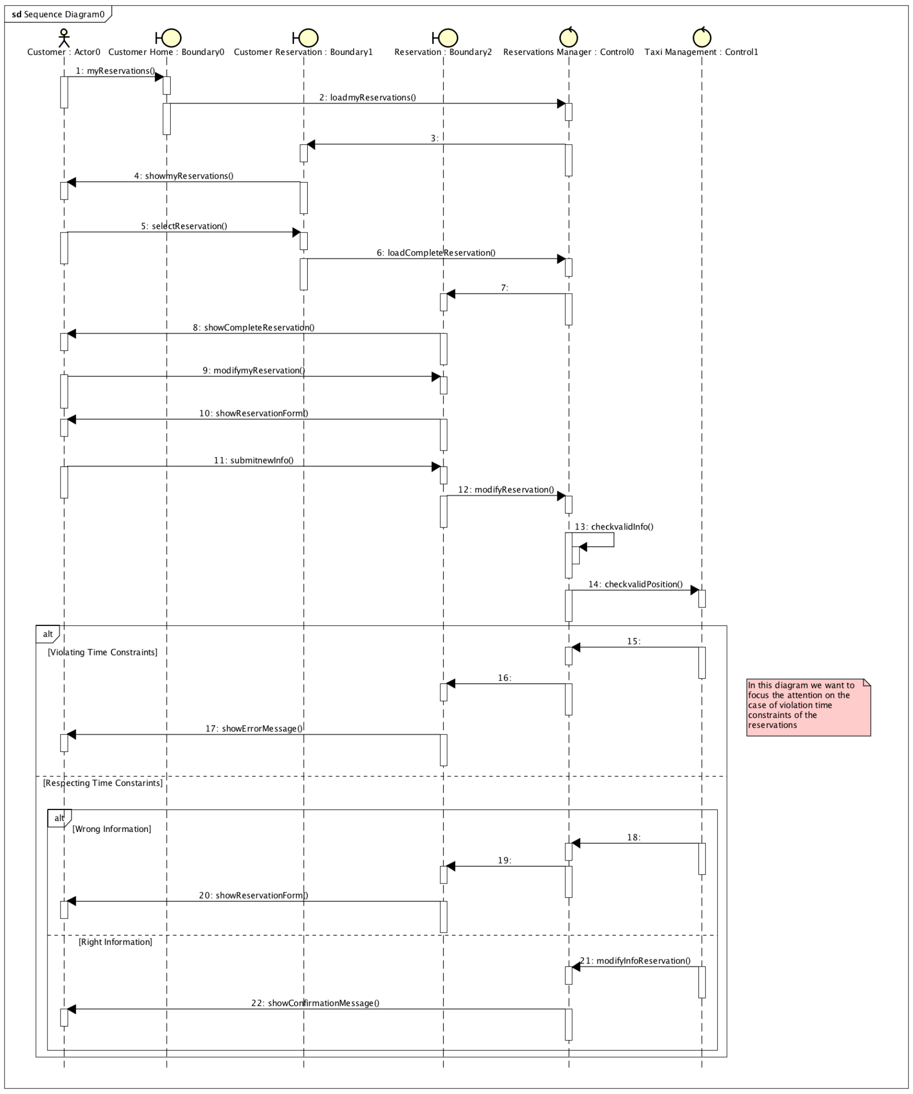
2.5.2 Sign Up of a Taxi Driver



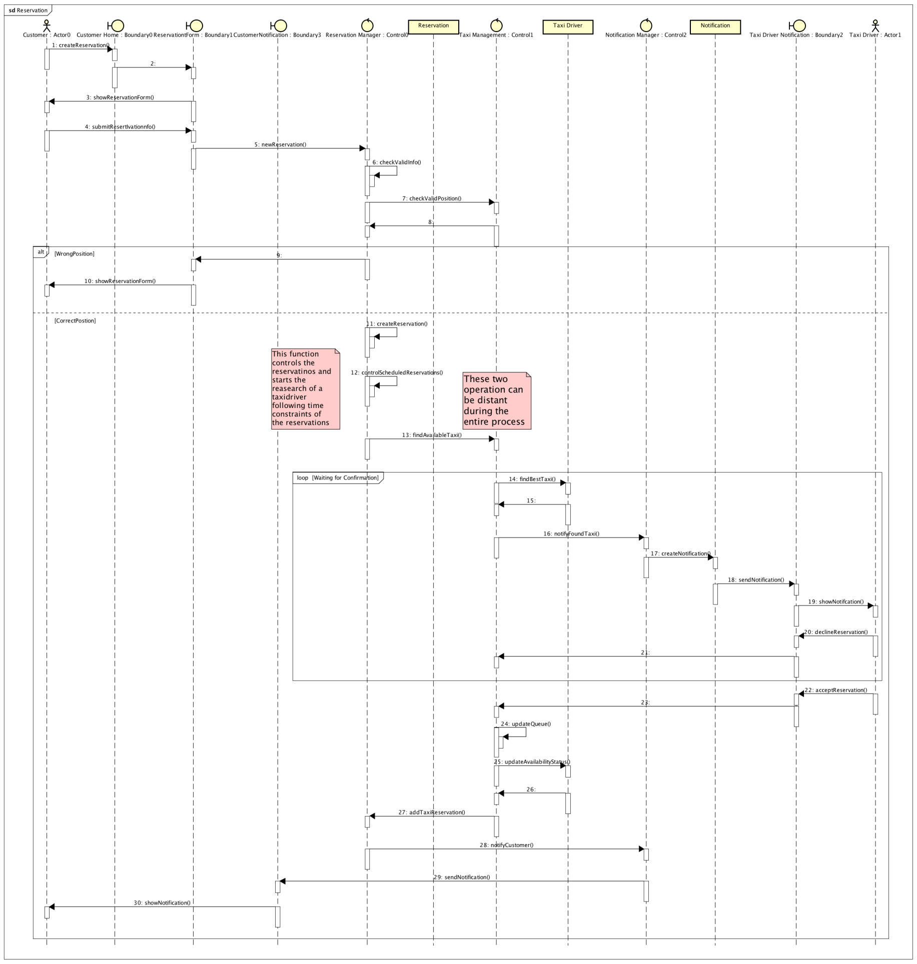
2.5.3 Request of taxi cab made by a Customer



2.5.4 The act of modifying a Reservation already made


sd Sequence Diagram0
Customer : Actor0
Customer Home : Boundary0
Customer Reservation : Boundary1
Reservation : Boundary2
Reservations Manager : Control0
Taxi Management : Control1
1: myReservations()
2: loadmyReservations()
3:
4: showmyReservations()
5: selectReservation()
6: loadCompleteReservation()
7:
8: showCompleteReservation()
9: modifymyReservation()
10: showReservationForm()
11: submitnewInfo()
12: modifyReservation()
13: checkvalidInfo()
14: checkvalidPosition()
alt
[Violating Time Constraints]
15:
16:
17: showErrorMessage()
[Respecting Time Constarints]
alt
[Wrong Information]
18:
19:
20: showReservationForm()
[Right Information]
21: modifyInfoReservation()
22: showConfirmationMessage()
In this diagram we want to focus the attention on the case of violation time constraints of the reservations

2.5.5 Making a Reservation



2.6 Component Interfaces

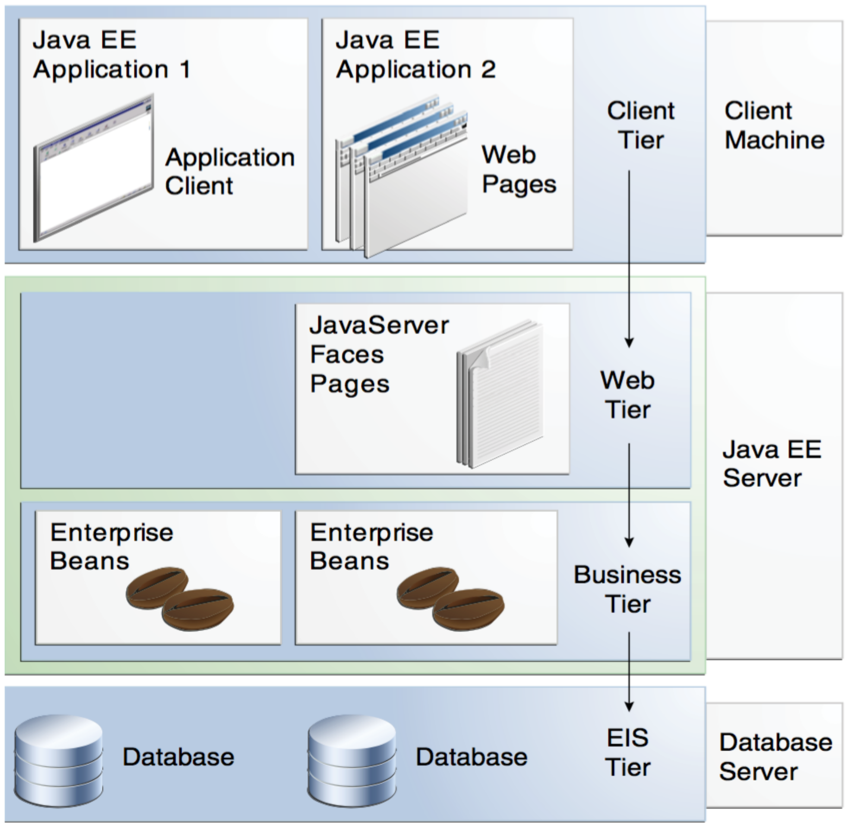
Component	**Interfaces**	**Description**
Sign up component	newCustomer newTaxiDriver	Allow not registered user to sign up into the system
Log in component	authenticateCustomer authenticateTaxiDriver	Allow registered user to authenticate into the system and access to their dedicated functionality
Customer component	getActiveReservation getActiveRequest findByUsername findByEmail	Allow customer to see their active request/reservations
Taxi Driver component	getAvailability setAvailability getCab findByUsername findByEmail findByLicense	Allow a taxi driver to see his/her availability Allow a taxi driver to modify his/her status For future implementations, this interfaces allows the customer to select different cabs size
Request component	newRequest	Allow a customer to make a new request
Reservation component	newReservation editReservation deleteReservation getActiveReservation getActiveRequest	Allow a customer to make/modify/delete reservation and also to see active reservations/requests
Taxi Management component	findAvailableTaxi	Allow request and reservations components to find the first available taxi in a certain zone
Notification component	getCustomerNotification getTaxiDriverNotification notifyCustomer notifyFoundTaxi	Allow registered users to see their notifications.

2.7 Selected Architectural Styles and Patterns

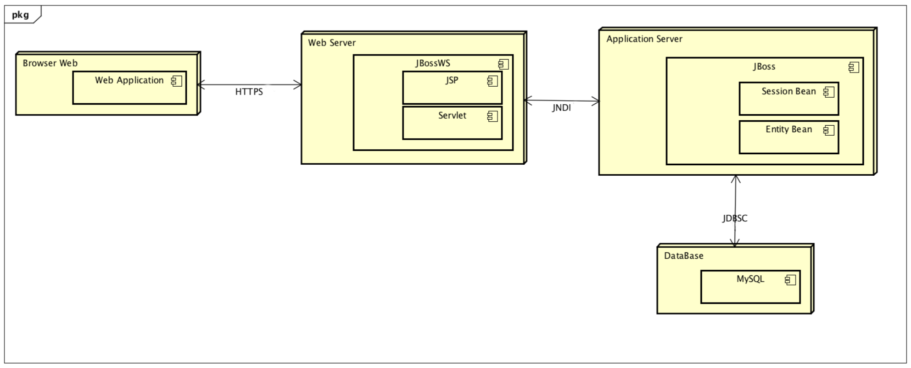
2.7.1 Architecture

We imagined the system to be divided in a 4 tier architecture based on the JEE architecture:

- Client
- Web
- Business logic
- EIS



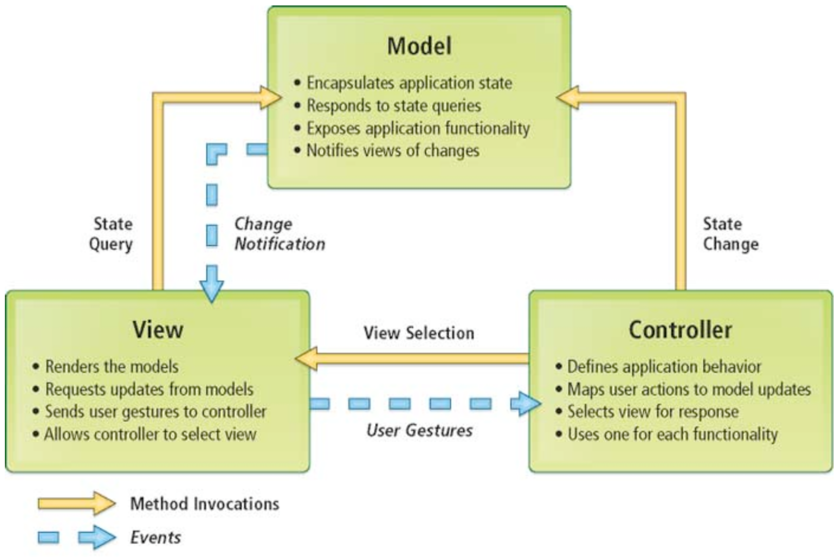
The Client tier is the highest level one and it represents the front-end of the application: the interaction between the User and the application occurs via web browser. The Web tier manages the connections between the Client tier and the Business logic tier. The Business Logic is the core of the system, it manages the business logic of the application and it is responsible for example to find a taxi for a customer and it also manages the queues of taxis into the city. The EIS contains the data source of the system and it manages the operations of data access, write/modify data and retrieve it. The data are not only in one copy, but we decided to insert into this tier another system for data storage in order to preserve all the data in case of any problems (Mirror 1). According to the Java Enterprise Edition architecture, every tier contains a particular software to manage its operations. The Client tier contains J2EE7 and JSP pages to support Web Browser to manage bean. The Web tier contains Servlet and Dynamic Web Pages that needs to elaborate. The Business Logic contains Enterprise Java Bean to handle the business logic. Four tier architecture permits to modify the system according to a new system requirements or features that the system will provide in the future without too much hassle. In particular, if the system will be used in the future by more people, the data tier could be enlarged and improved without modifying other parts.



2.7.2 Architectural and Design Patterns

Architectural Patterns

The architecture of the system is a client/server architecture. In this type of architecture, the server provides some services to the possible clients. A client that want to benefit of its service, requests the server, the server receives its in form of messages or remote invocation and then it answers the client. The client can access the server only with interfaces. Client and server are mapped into different hw/sw modules. The architectural pattern that we want to insert in the system is the Model-View-Controller. MVC is a high-level architecture and it is used in several applications, in particular in web application. This pattern is very general and it doesn't consider the physical tiers of the system in order to allow an easy development of the system. MVC separates data model, user interface, and control logic; in this way, these three components are divided, and modifications to one component can be made with a minimal impact to the others. The model contains the data source of the system and the logic part regarding data access and easy operations on the data like sum, average and simple similar operations. The view is the subsystem that creates an interface, in particular a user interface, to allow the users to interact with the application and benefit its services. The view is a representations of the model and for a single model you can have different views. The controller has to respond to user interactions and its function is to modify the model according to what the users want. MVC allows developers to decide where put the business logic of the system: in our case the client itself does not contain any part of the application logic, according to the definition of a thin client. A client contains only the view of this architecture, in particular his/her user interface. In our system, all business logic is inside the controller that manages all the operation regarding the services providing by the application, requests and reservations. The model is in practical the database of the system. Between the view and the controller there is a layer that handles the communication with the users.


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• Uses one for each functionality
Method Invocations
Events

Design Patterns

Inside our application we decided to insert these design pattern:

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern. In our particular case the Observer Pattern can be used to monitor the availability status of a taxi driver: when this happens the system is notified and the changes are saved.

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. We can use this pattern to create different types of user accounts with different factories.

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This pattern is often used in control class to coordinate the operations in the system. We can use this pattern in the classes that manage requests and reservations and also in the class that manages the queue of taxis.

2.7.3 Other Design Decisions

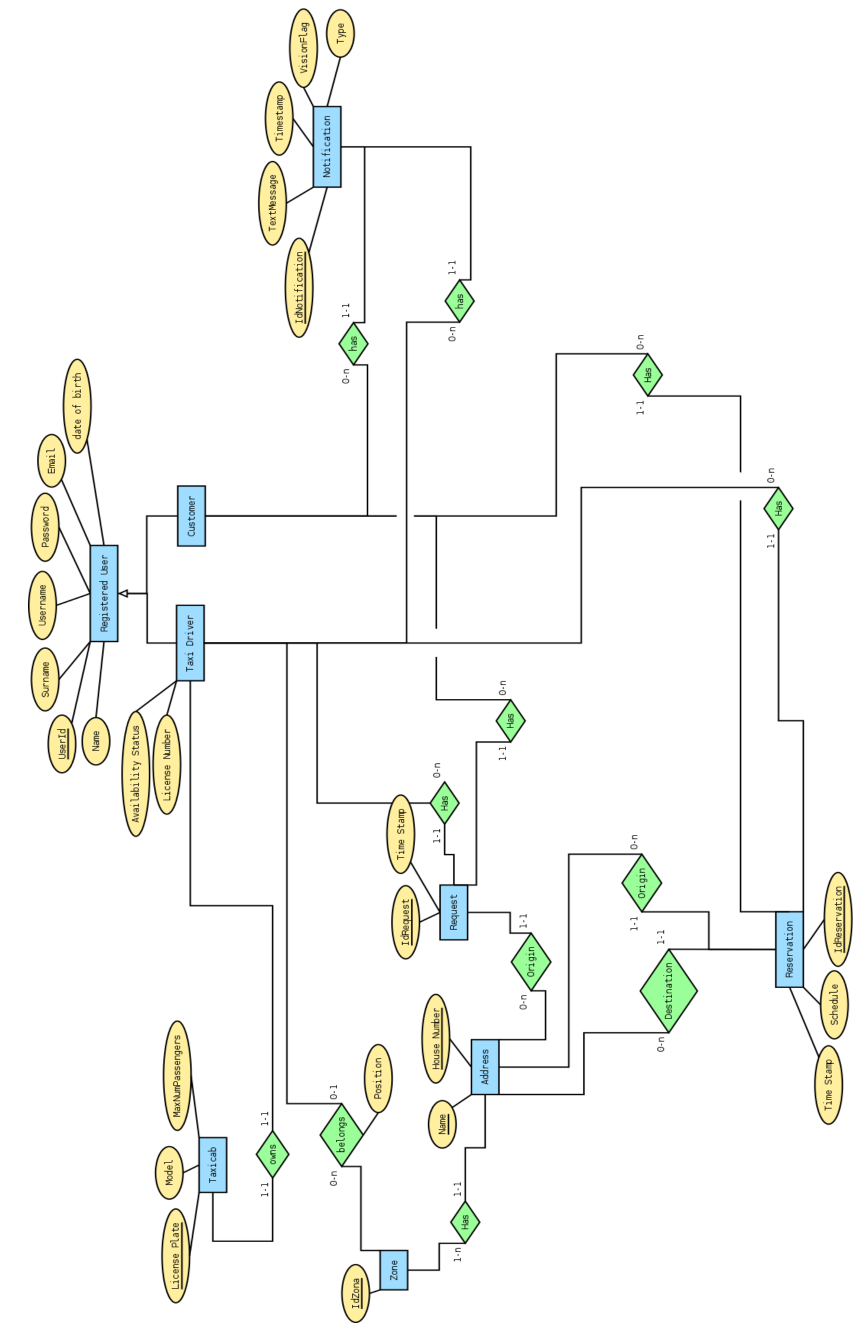
ER Diagram

We have decided in our assumption that each address is contained into only one zone of the city. In order to satisfy this constraint, we created an entity Zone and every address is associated to its zone. The zones have an auto increment id and in this way the operations of add/modify a zone are not so difficult. The address entity contains the name of the street and the house number. For future possible implementations of the system, we can add in this entity the name of the city or we can add new entity to use this application in another place.

To manage the two different types of users of the system, we use a generalization hierarchy to distinguish between Customer and TaxiDriver. If a user wants to log into the system, he/she has to insert his/her username and password, The mandatory fields of the registration form are: Name, Surname, Username, date of birth and password. For each user the system creates an auto increment id. If a person wants to register himself as a TaxiDriver has to insert also his/her license number. Taxi driver are associated to a cab and every cab has a model and a number of passengers. The system creates automatically a binary field Availability that represents the status of the taxi driver and this field is set during the registration process as false. Each taxi driver, during his/her job, is in a particular zone of the city and we create a relation between these two entities to indicate this fact; this information came from GPS position. The taxi driver has an attribute Position that identifies his/her position in the queue of the zone.

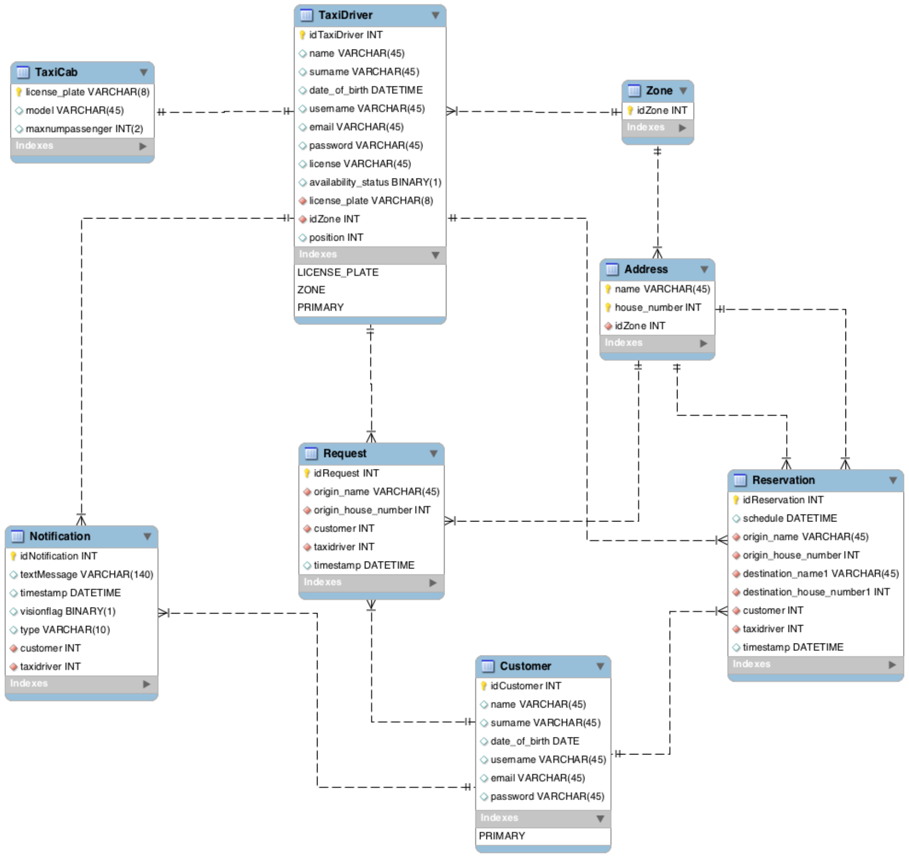
The core part of the system are requests and reservations. For these we created two separated entity, both have and id to identify trips in an unique way. Request contains a reference to the customer who wants to use the service, the taxi driver who answered the call and the origin address of the route. Reservation has also a reference to the destination of the route and its schedule. When a reservation is requested, the taxidriver field is empty; this field will be completed only when the system finds the taxi driver, following the time constraints specified in the RASD.

In the application there are two types of notifications: one for the taxi driver to inform him/her about a call and one for customer to inform him/her about the found taxi. Every notification has a timestamp that represents the time when the notification is send and a visionflag, a binary variable that represents if a user has seen the notification and a field textMessage where there is the principal part of the notification. A notification has also a reference to the taxi driver and the customer. A call notification is send by the system to the taxi driver and contains the information of the route (field textMessage) and the customer. A found taxi notification is send by the system to the customer to inform him/her about the taxi inside the field textMessage.


Registered User
Customer
Taxi Driver
Notification
Request
Reservation
Address
Zone
Taxicab
date of birth
Email
Password
Username
Surname
UserId
Name
Availability Status
License Number
IdNotification
TextMessage
Timestamp
VisionFlag
Type
has
has
Has
Has
Has
Has
Time Stamp
IdRequest
Origin
Origin
Destination
House Number
Name
Position
belongs
owns
Has
MaxNumPassengers
Model
License Plate
IdZona
IdReservation
Schedule
Time Stamp
1-1
0-n
0-1
1-n

In our schema there is only one hierarchy and following the rules of the logical model, we have to eliminate it to produce a correct model. We decided to eliminate the generalization of Registered user creating two different entities, one for the customer and one for the taxi driver, to underline the different functions and roles of these actors inside the system. We choose this structure also because taxi driver has too many field that customer doesn't have to avoid some optional attribute inside the model.

Here is presented the relational model diagram in which is represented how we assigned the foreign keys in the different tables of the database:



The final model is the following physical structure:

TaxiDriver(idTaxiDriver, name, surname, date_of_birth, email, password, license, availability, *license_plate, idZone*, position)
Customer(idCustomer, name, surname, date_of_birth, email, password)
Zone(idZone)
Address(name, house_number, *idZone*)
TaxiCab(license_plate, model, max_num_passengers)
Request(idRequest, *origin_name, origin_house_number, idCustomer, taxidriver*, timestamp)
Reservation(idReservation, schedule, *origin_name, origin_house_number, destination_name, destination_house_number, idCustomer*, *taxidriver*, timestamp)
Notification(idNotification, textMessage, timestamp, visionflag, type, *taxidriver*, *customer*)

3 Algorithms

3.1 The queue of Taxis

In this section we want to explain one of the core part of the system: the queue of taxis. We didn't use a particular programming language but we wrote our algorithms in pseudo code to provide only a high level view.

```
TaxiDriver findAvailableTaxi(){
        ad = request.origin.address;
        z = getZone(ad); // this function return the zone where the address is
        td = z.getFirst(); // this function return the first taxi in queue
        return td;
}
```

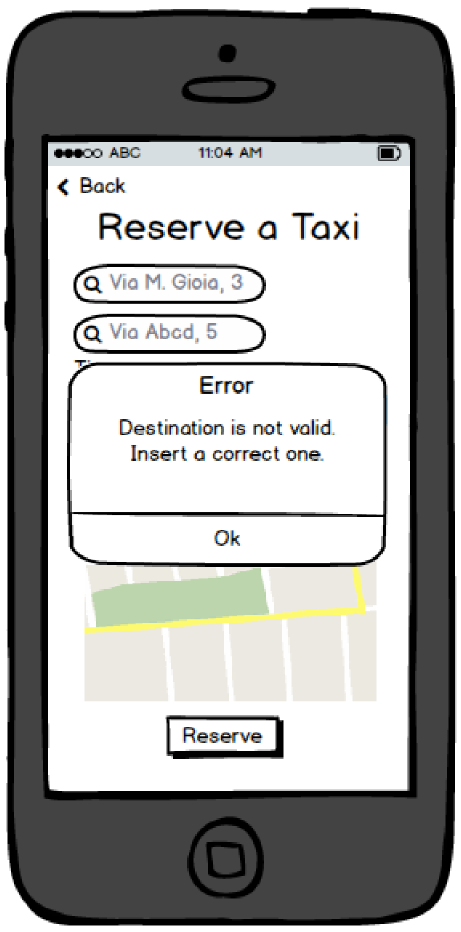
When the system finds the taxi driver, this function is used to update the queue: the first operation removes the taxi driver from the queue and sets his/her availability as false; then the position of the other taxi drivers is updated. If the found taxi is the first in queue all the others are updated, but if one or more taxi drivers refuse the call, this function modifies only the position of the remaining taxi drivers until the last taxi driver that refused the call. We assumed that if a taxi driver refuses a call, the request is send to the next taxi in queue but his/her availability doesn't change.

```
void updateQueue(){
        ftd //found taxi driver for a particular request/reservation
        ftd.availability = false;
        remove(ftd); //remove the found taxi from the queue
        for(TaxiDriver td in queue; td.position<ftd.position;)
                td.position = td.position-1;
}
```

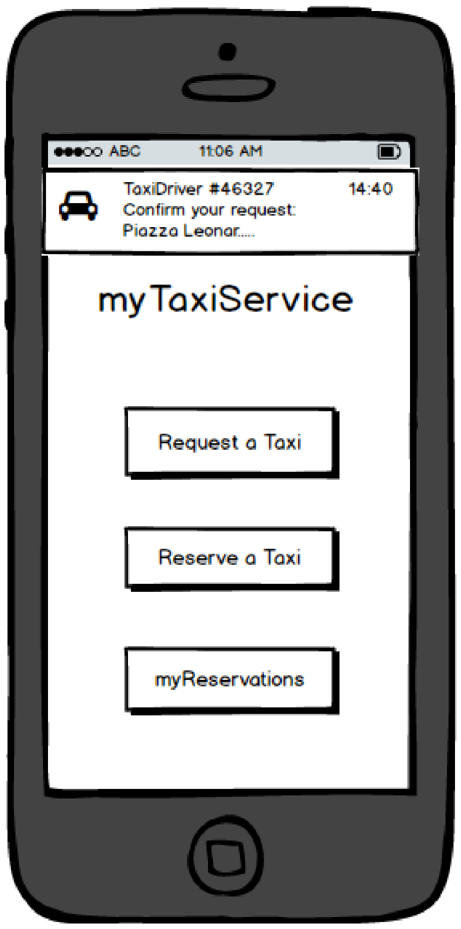
4 User interface design

These are some mockups for the user interface, see the RASD for more mockups.

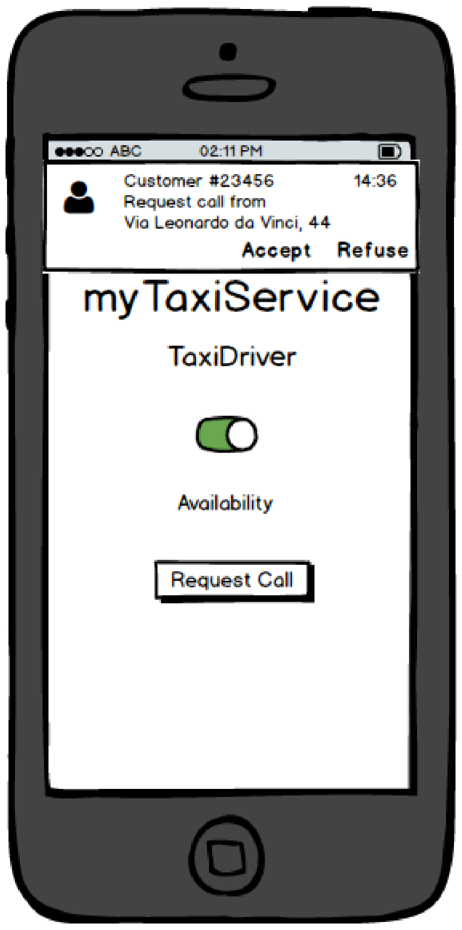
Invalid destination while reserving a taxi



Notification for a found taxi

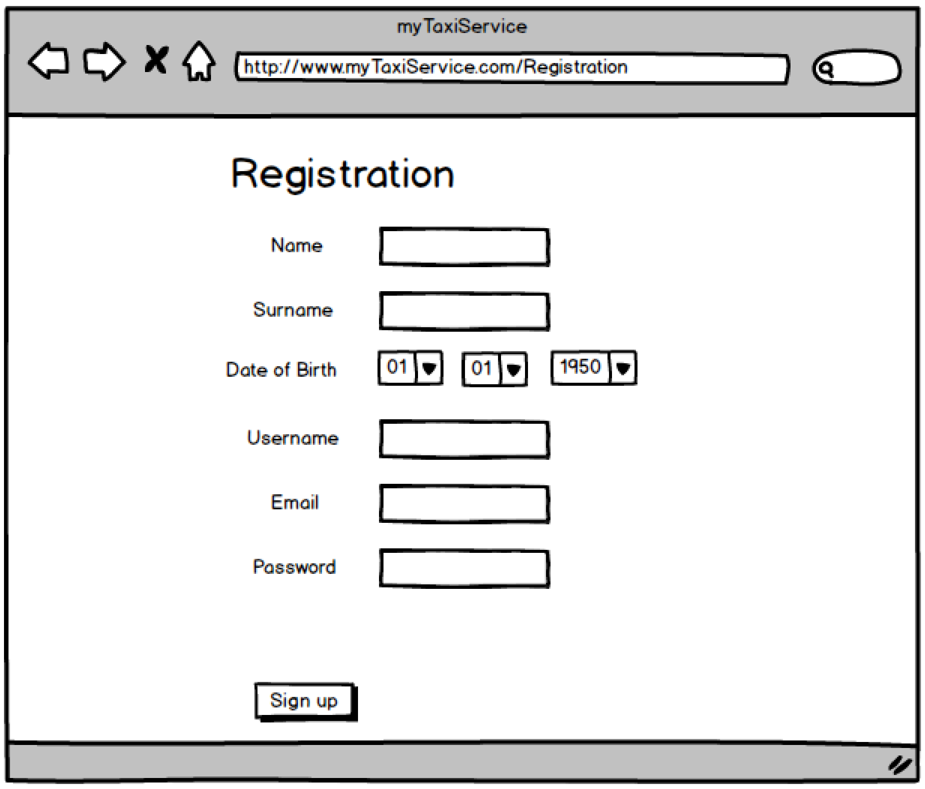


Notification for an incoming request

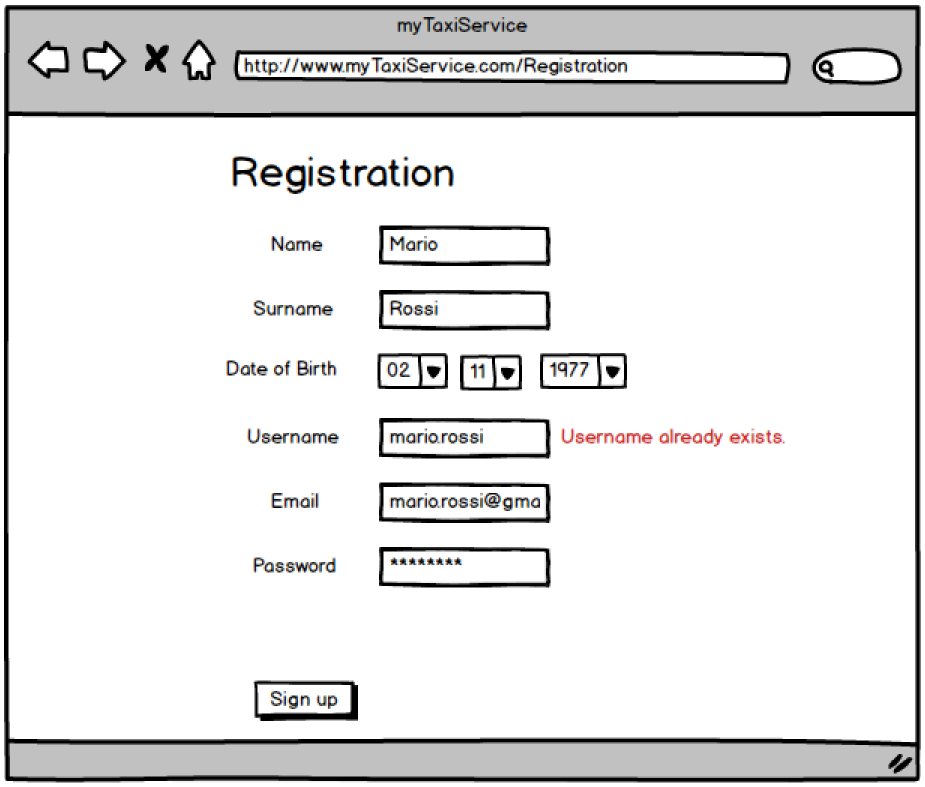


Customer #23456 14:36
Request call from
Via Leonardo da Vinci, 44

Registration form



Username already present in the system error in the registration form



Registration

Name	Mario
Surname	Rossi
Date of Birth	02 11 1977
Username	mario.rossi Username already exists.
Email	mario.rossi@gma
Password	********

5 Requirements traceability

In this section of the document are shown for each component, their respective requirement that have to fulfill.

Component	**Requirements**
Sign up component	The system has to provide a sign up functionally. The system has to provide two types of personal pages: one for the customer and one for the taxi driver.
Log in component	Log in
Customer component	See request Reservation Modify reservation Cancel reservation
Taxi Driver component	The system has to provide a function that allows the taxi driver to inform the system about his/her availability. The system has to provide a function that allows the taxi driver to confirm or not the request/reservation of the customers.
Request component	The system has to provide a function that makes possible to do a request.
Reservation component	The system has to provide a function that makes possible to do a reservation. The system has to provide a function that allows the customer to modify his/her reservation until 1 hour before the meeting.
Taxi Management component	The system has to provide a function that finds an available taxi 30 minutes before a reservation.
Notification component	The system has to notify the customer who wants a taxi, about the found taxi. The system has to notify the taxi driver when he/she has been assigned to a new customer.

6 References

- Slides of the course (Software Engineering 2)
- The UML website: http://www.uml-diagrams.org/