Politecnico di Milano

A.A. 2015-2016

Software Engineering 2 project:

MyTaxiService

Design Document

Xu Xiuli(10511854) Liu Yuqi(10511250)

4 December 2015

1 Introduction

1.1 Purpose

The aim of this Design Document(DD) for MyTaxiService application is to provide a basic 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 build.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 relationship between the modules.

1.2 Scope

As specified in the previous paragraph, in this document we will focus on the overall structure and architecture of the system, without going deeply into the details of the implementation. Only a small section of this document, in fact, will be dedicated to some guidelines for the implementations of the application's main algorithms.

Components, connectors, interfaces are instead the main participants of this document, but their scopes and interactions will be described only at a high-level. We will also explain the architecture styles adopted and the reasons behind them, trying to give a motivation for every choice taken.

It is important to understand that all the content of the document is platform independent and the various architecture components will be mapped onto real hardware and software components only further in the implementation phase.

Since we have already provided a bunch of mock-ups for the graphical user interface in the RASD, we will only redirect you to them without showing them again.

1.3 Definitions,Acronyms and Abbreviations

1.3.1 Definitions

Customer –Registered user that may demand a taxi ride

Taxi Driver – Employee of the taxi service with a driver account.

Guest – Users that are accessing to MyTaxiService's homepage not yet registered or not logged in

1.3.2 Acronyms

GUI – Graphical User Interface

DB – Database

RASD – Requirements Analysis and Specification Document

GPS – Global Positioning System

Java EE – Java Enterprise Edition

JDBC – Java Data Base Connectivity

JSON – JavaScript Object Notations

API – Application Programming Interfaces

JPA – Java Persistence API

Tier – It is a hardware level in a generic architecture

Layer – It is a software level in a generic software system

1.4 Reference Documents

This document refers to the following documents:

- MyTaxiService's RASD(By Liu Yuqi and Xu Xiuli)
- The IEEE standard 1016:Software Design Specification

1.5 Document Structure

Section 1 – Introduction: Introduce this document in relation to the MyTaxiService system.

Section 2 – Architectural Design: explains in details the architecture and design of MyTaxiService system, along with the chosen patterns and components identification.

Section 3 – Algorithm Design: shows a possible high-level implementation of some relevant application algorithms.

Section 4 – User Interface Design: shows indicatively how the user interface will look like.

Section 5 – Requirements Traceability: Shows how the requirements

specified in the RASD have been satisfied in the design phase.

Section 6 – References: Hours of works, software and tools used and others external references.

2 Architectural Design

2.1 Overview

we considered to develop our application using Java Enterprise Edition (JEE).

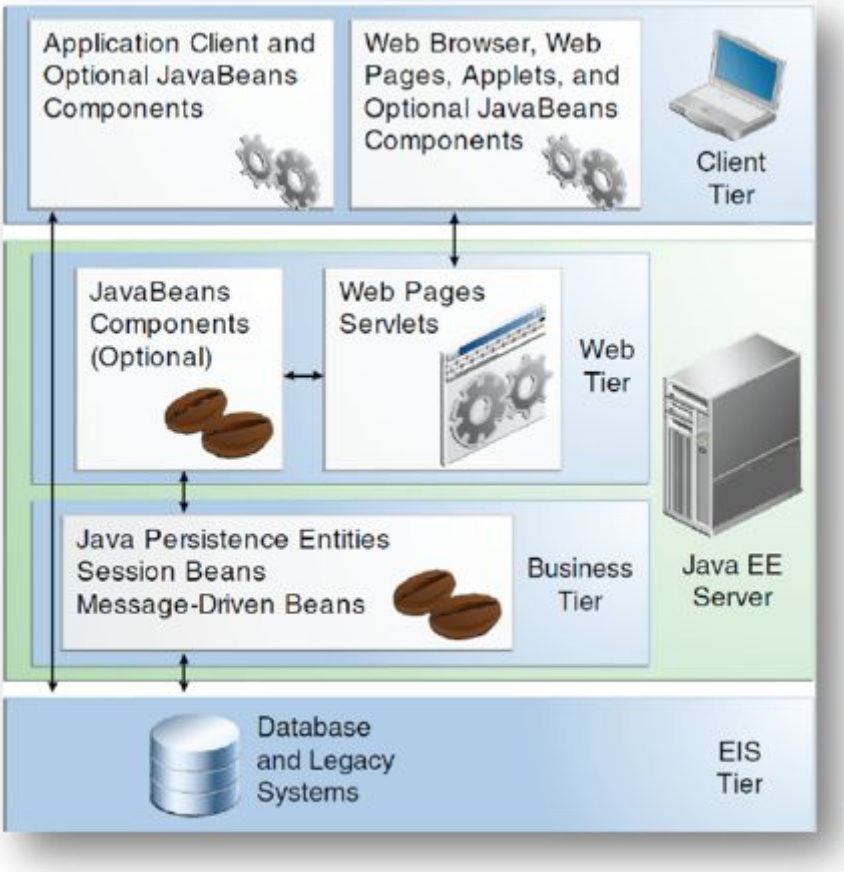
This will also be useful to satisfy important Non-Functional Requirements such as scalability, portability, availability, reliability and so on.

The applications of myTaxiService (the web application and the mobile one) will be large-scale, multi-tiered, scalable, reliable and the network will be secure.

The application developing takes as reference the standard of Java Enterprise Edition 7 (JEE7), the last release available now.

We will use a three-tier physical architecture mapped on four logical layers, as the standard of JEE.

Here is the general schema of the architecture:



Now we will see in a deeper level of detail the meaning of each layer:

- **Client Layer**: it contains Application Clients and Web Browsers and interacts directly with the actors (Customers and Taxi Drivers). In our application, the Client can access via browser (web application) or via smartphone (mobile application).
- **Web Layer**: it contains the Servlets and Dynamic Web Pages that needs the elaboration. This tier receives the requests from the Client layer and forwards the pieces of data collected to the Business Layer.

- **Business Layer**: it contains the application logic (with the Java Beans and the Java Persistence Entities). This will permit the communication between the System of mTS and the target users (Customers and Taxi Drivers).

2.2 Selected Architectural styles and patterns

2.2.1 Three-Tier architecture

The selected architectural style is multitier based on Java Enterprise Edition implementation of this acrhitectural style.

In what follows, the advantages of this architecure and reasons for the selection aregoing to be explained.

The main benefits of the N-tier architectural style are:

Maintainability - Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.

Scalability - Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.

Flexibility - Because each tier can be managed or scaled independently, flexibility is increased.

Availability - Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Java Enterprise Edition is designed to develop large-scale and multi-tiered applications that are scalable and meet reliability conditions and are secure at the same time.

According to RASD, considering these and all the other functional and especially the non-functional requirements (RASD 1.4, part 3.6), it could be concluded that JEE is more than acceptable solutions in terms of these requirements.

MyTaxiService is going to be a multitier application that is also scalable (offer service to thousands of customers at the same time), but reliable and secure at the same time, maintaining high availability.

Server side needs to meet the special non-functional requirements in terms of security (RASD 1.4, part 3.6.4.3) – so the server side needs to be split into web, business logic and database part. Java EE offers the separate client, web, business and database tier, which is exactly what is needed. So, this will give ability to place firewalls between each two parts and make application secure and meet the security requirements.

So, in what follows, the core concept of JEE is going to be explained and the overall idea of the multitier implementation using JEE in terms of this application.

Multitier architectural model, in this case, consists of:

-Client tier that is running on the client machine. It contains Application

Clients and Web Browsers and it is the layer that interacts directly with the actors. The client machine could be either mobile phone running application or web browser or personal computer running web browser in this case. Taxi drivers, according to RASD 1.4, part 2.1. must use mobile application.

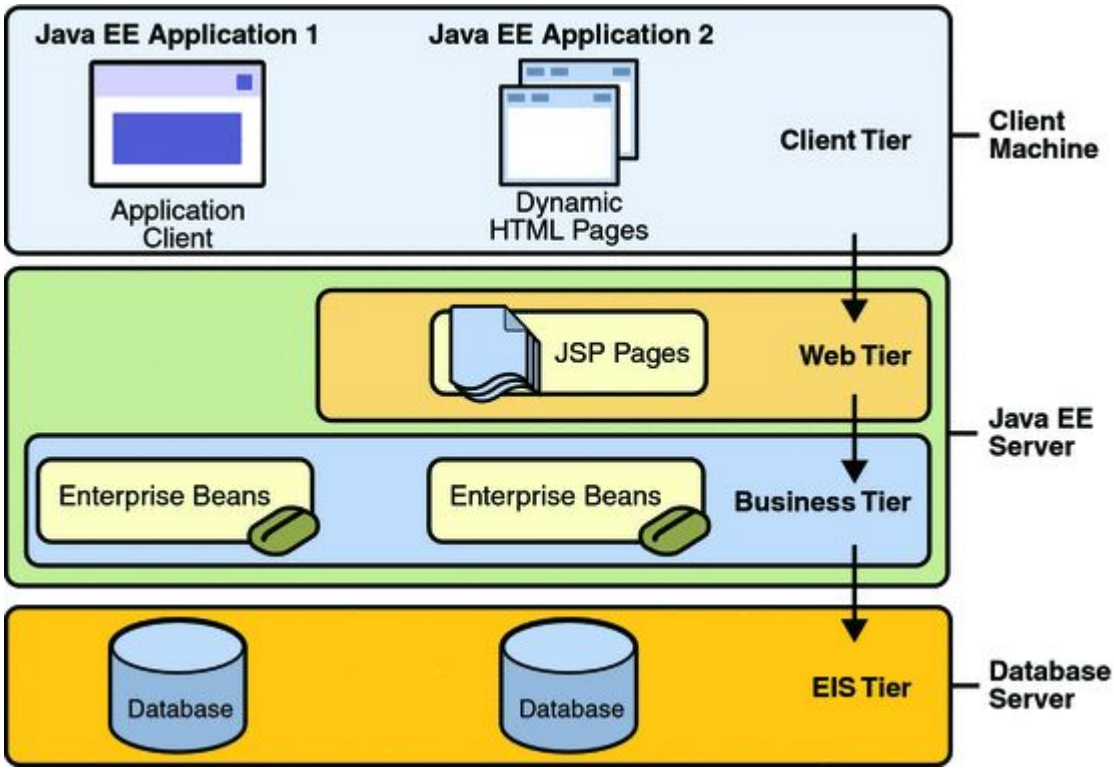
-Web tier, running on the Java EE server. It contains the Java Server Paged. This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier.

-Business tier, running on the Java EE server. It contains Java Beans, that contain the business logic of the application and Java Persistence Entities.

-Enterprise information system (EIS) running on the database server, consisting of data sources, to be more precise, databases and stores the data that needs to be retrieved and manipulated.

The server part is going to be run on a more powerful machine than a client – a high performance PC, according to RASD 1.4, part 3.6.2.1.

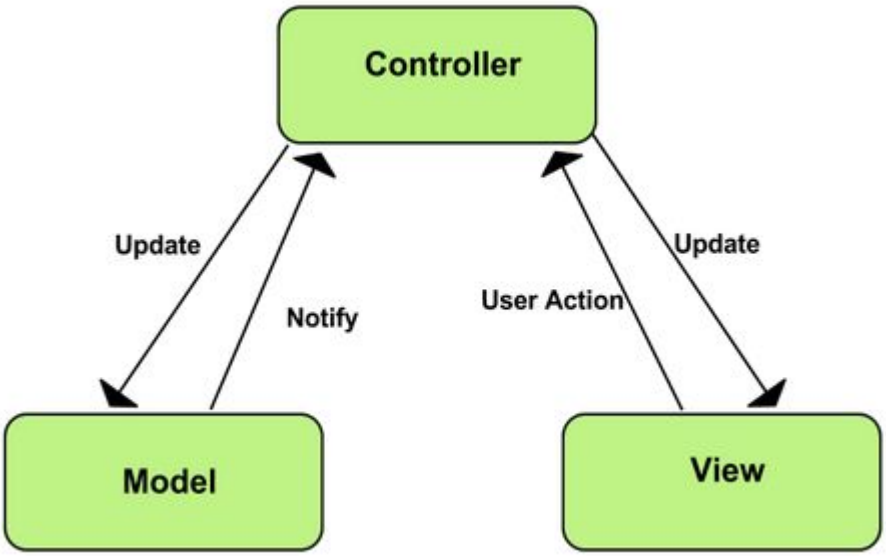
The illustration of the architectural style previously described, could be viewed below.



2.2.2 Model-View-Controller Pattern

MVC is a standard pattern that separates the user interface (View) and the business rules and data (Model) using a mediator (Controller) to connect model to the view.

The main benefit is the separation of concerns. Each part of the MVC takes care of its own work: the view takes care of the user interface, the model takes care of the data, and the controller sends messages between both of them.



In MyTaxiService case, this pattern is going to be implemented on client side of the application – either a web or mobile app.

All the forms and pages that are used to interact with users (Register form, Login form, Edit profile form, Request taxi form etc.) belong to View. As users take actions – click on buttons – these forms send user gestures to controller. Controller maps user actions to model updates. The Controller classes are going to be defined for these forms that are previously mentioned. User actions trigger the state change. Model encapsulates application state and responds to state queries. In this case, model will deal with sending requests to a server and getting the data from the server. When the model gets data, it changes according to the data received, so the model will notify the view about the changes, so the view could render the model.

The view consists of classes related to user interface, such as buttons,

text fields etc. When user takes some action, controller takes its role. Controller requests model state change, and in this case, model classes communicate with server side.

2.3 High level components and their Interaction

myTaxiService's system is composed by three main components: DBMS, Web

Server and client application. The client application provides the UI through

which end users can access the application's services. These requests are for-

warded to the Web Server which is in charge of providing a response, eventually

querying the Database in the DBMS for information. The Web Server is also

responsible for answering to the API calls coming from external applications.

The DBMS stores all the information of the end users's accounts, the active

requests and reservations and the current state of the queues of every zone.

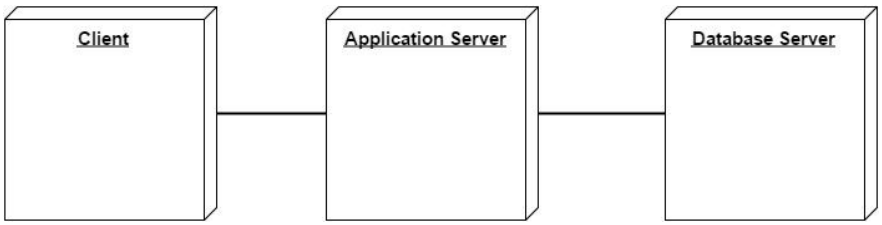
2.4 Components view

2.5 Deployment view

In this paragraph we describe, using a simple deployment diagram, how the components of our system are mapped to hardware components, representing on which hardware component each of our software components is located or executed.

Each node of the diagram represents “something” on which a software component can be located: usually it represents an hardware component (device), such as a sensor, a server or a mainframe.

Two nodes can be connected graphically using a segment that represents a communication path (connection) between the two components.



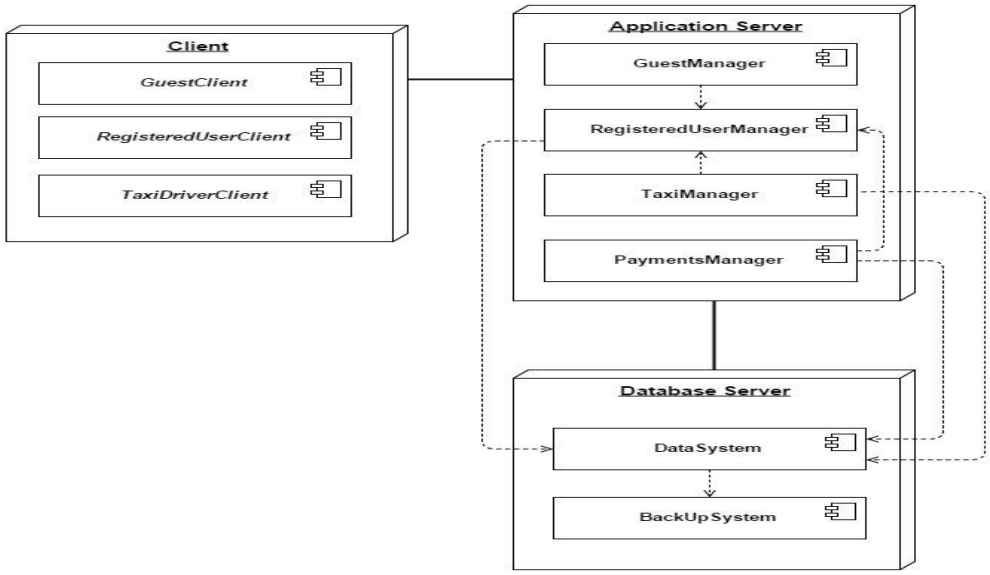
It is possible, and usually very useful, to integrate the component and the deployment diagram, in order to make more understandable the distribution of the different components on the nodes of the system.

In our case, we have on the Client the client components, one for each

actor involved in our system. The majority of the components are situated on the Application Server, that represents the core of the application. Here is where all the commands are received, interpreted and executed, all the tasks are satisfied and where the whole application is managed and coordinated.

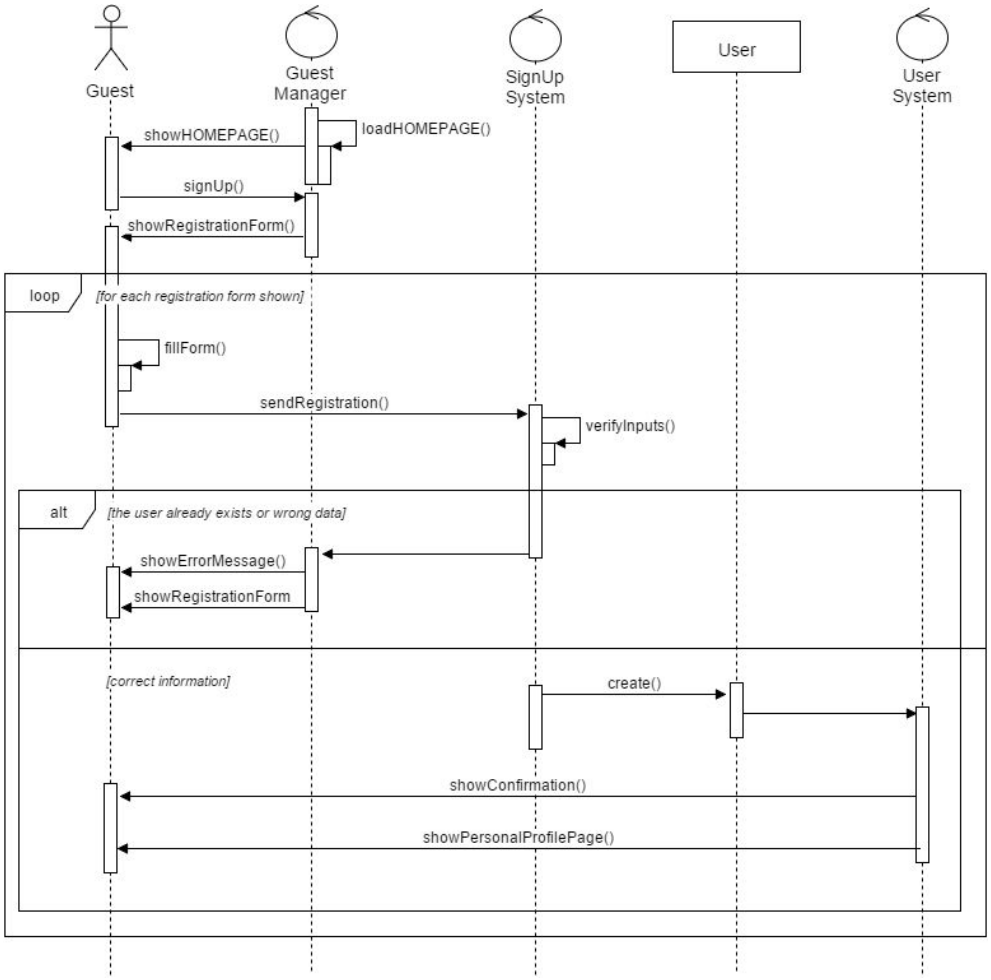
On the Database Server reside the components involved in the data management: the data system, containing all the main data and information of the application, stored in a relational database, and the back-up system, containing and managing a copy of the primary database, updated automatically by the system.

Reported below it's the integration between the two diagrams proposed in this paragraph and the previous one

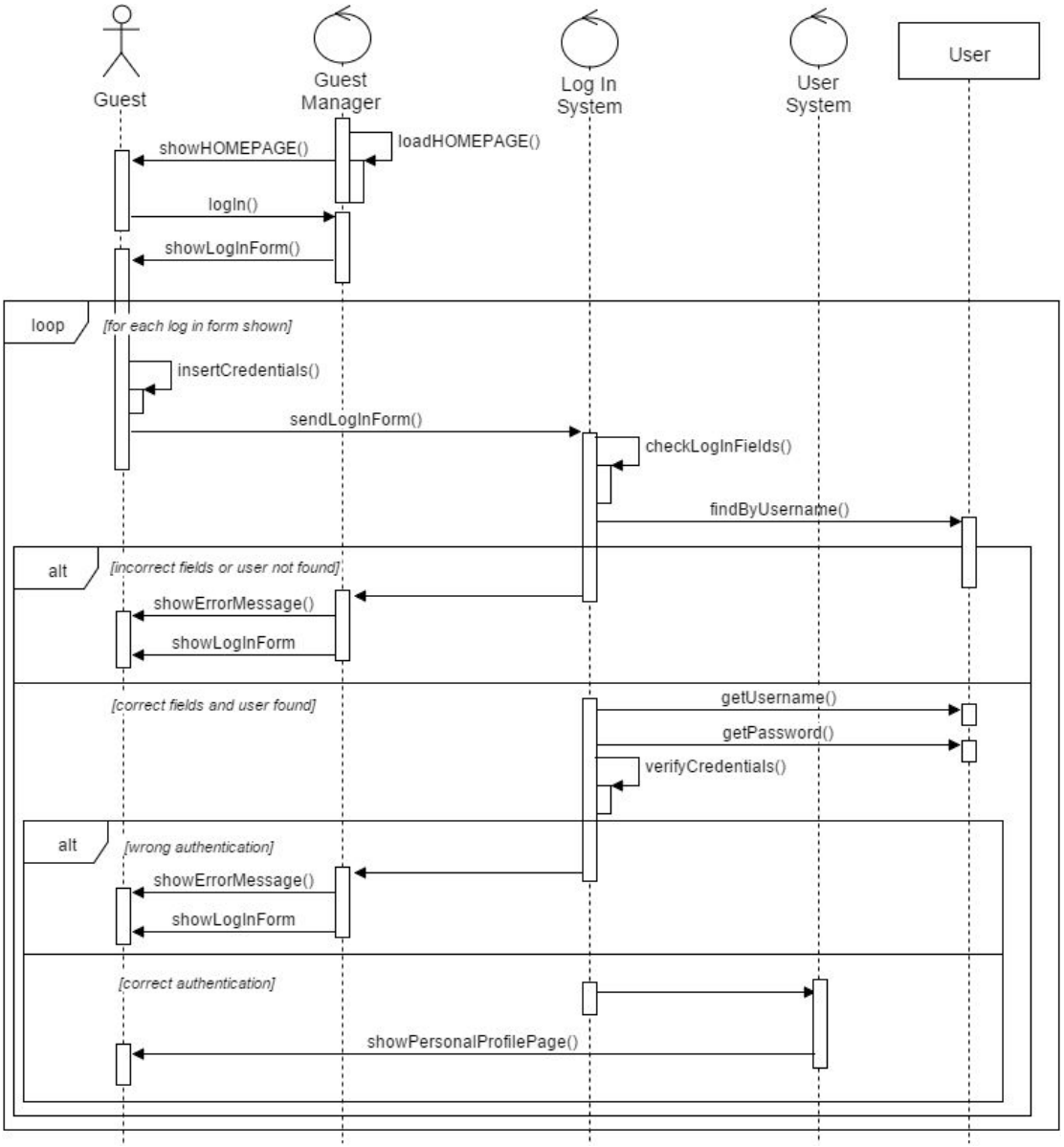


2.6 Run-time view

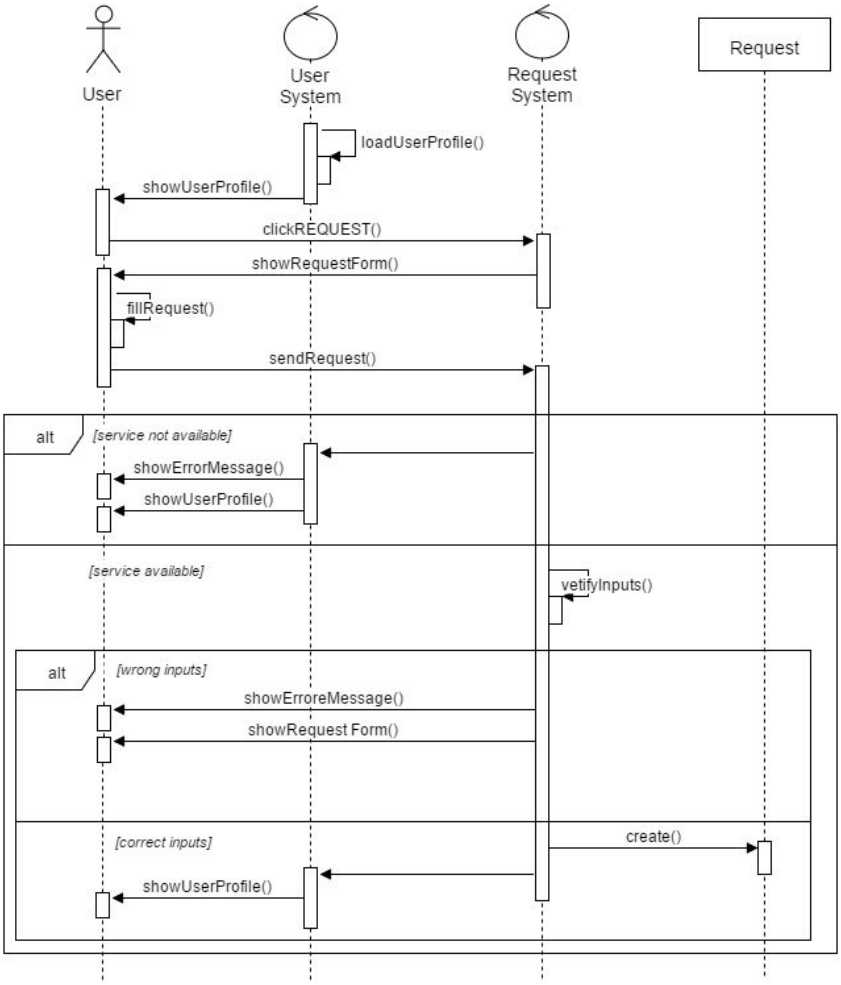
2.6.1 Sign up



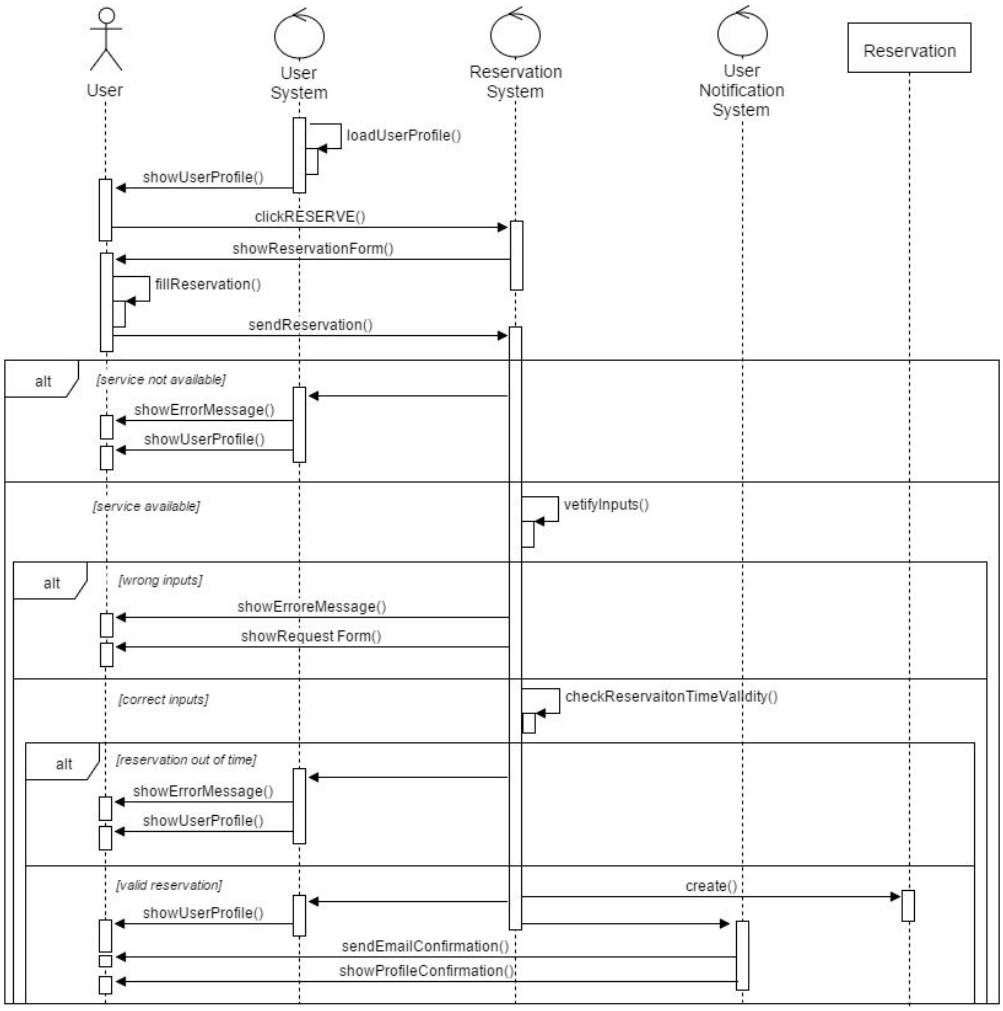
2.6.2 *Log in*


Guest
Guest Manager
Log In System
User System
User
showHOMEPAGE()
loadHOMEPAGE()
logIn()
showLogInForm()
loop
[for each log in form shown]
insertCredentials()
sendLogInForm()
checkLogInFields()
findByUsername()
alt
[incorrect fields or user not found]
showErrorMessage()
showLogInForm
[correct fields and user found]
getUsername()
getPassword()
verifyCredentials()
alt
[wrong authentication]
showErrorMessage()
showLogInForm
[correct authentication]
showPersonalProfilePage()

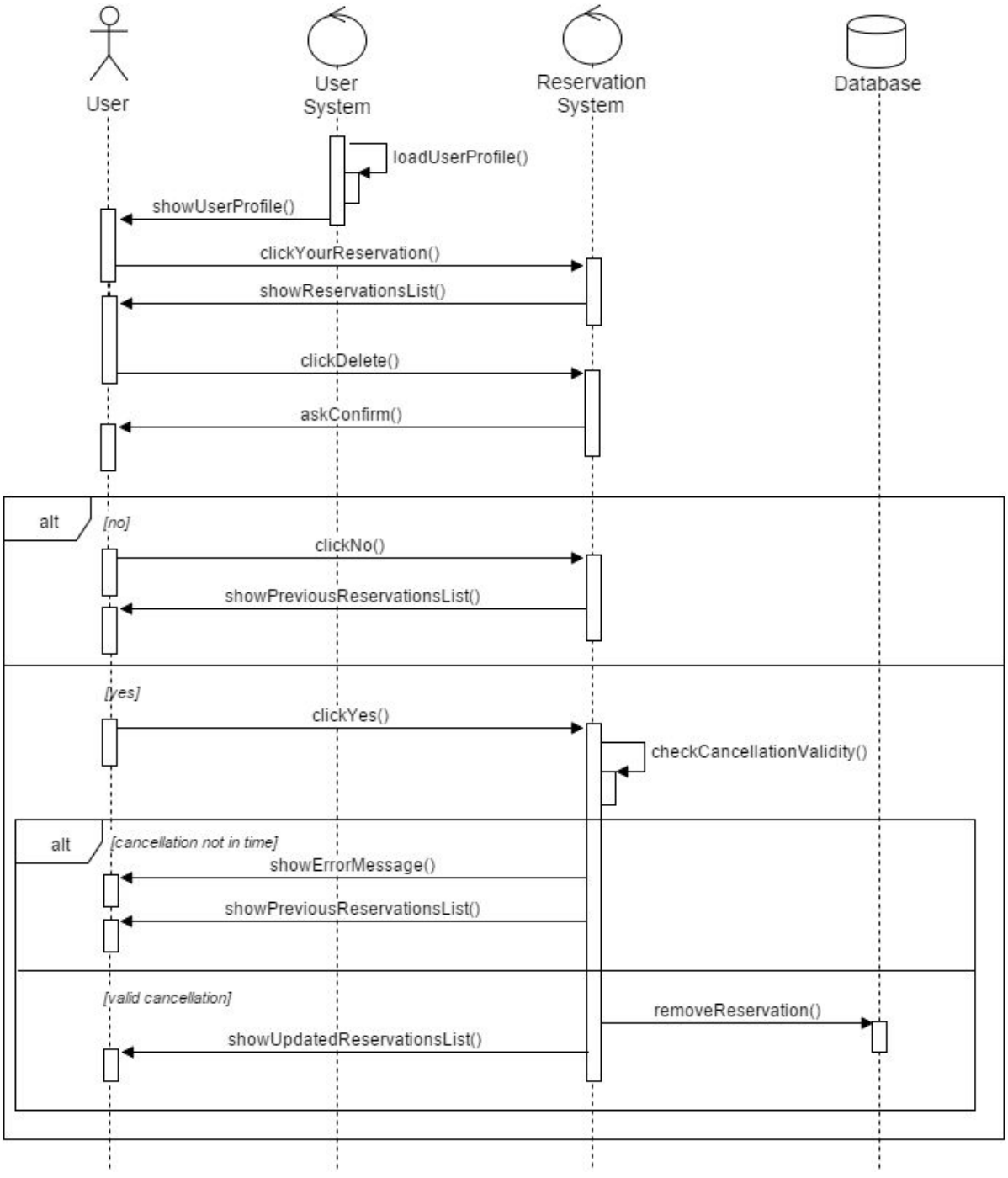
2.6.3 *Request a taxi*



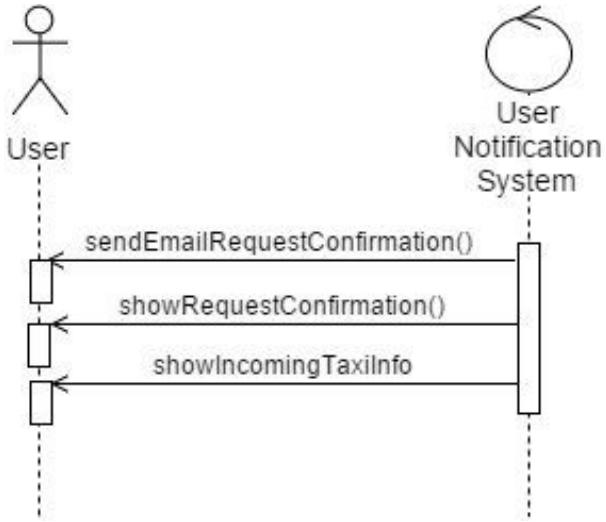
2.6.4 Reserve a taxi



2.6.5 *Delete a reservation*



2.6.6 *User request confirmation*



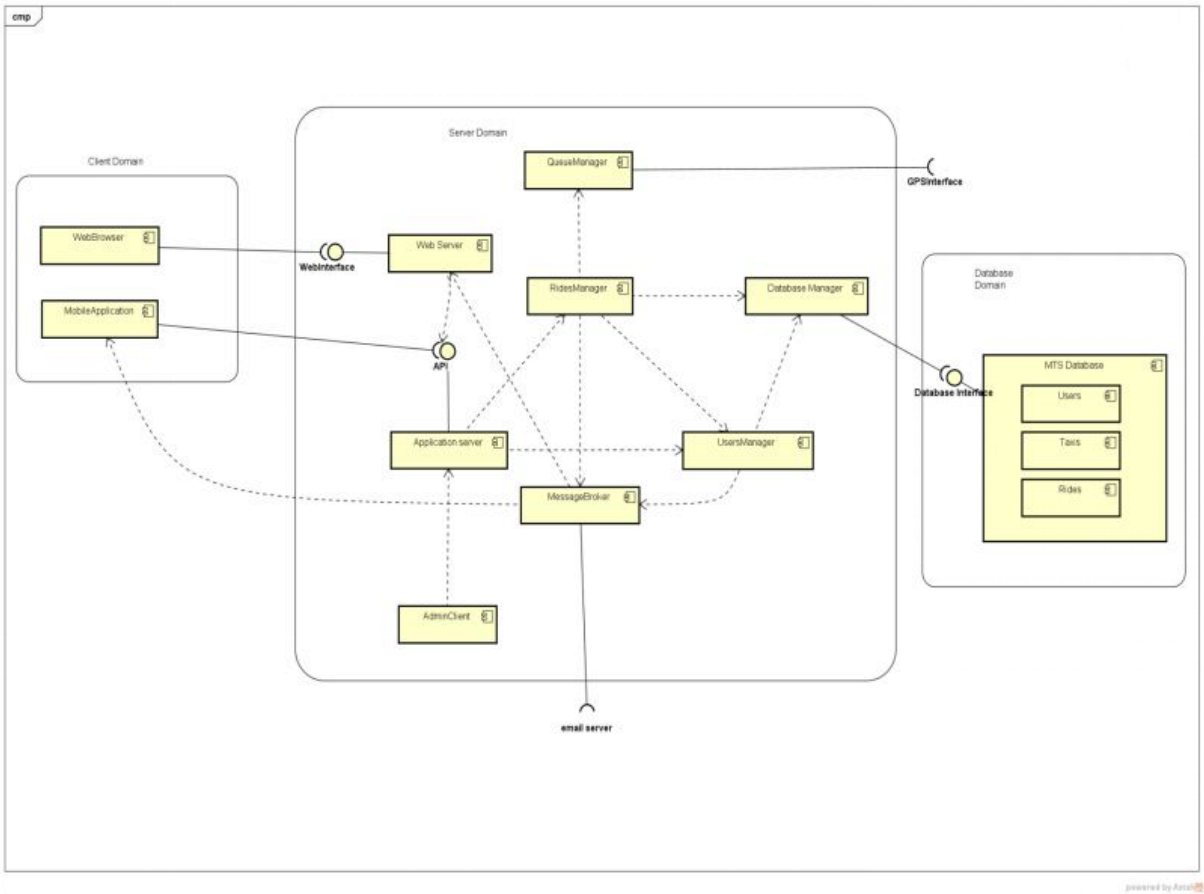
2.7 Components Interfaces

We now try to explain more deeply how the different components of our architecture are interfaced with each other. The client communicates with the server of our application through a web browser, following the HTTP (Hypertext Transfer Protocol). We chose this transfer protocol, because it's probably the most used protocol for the communication of information on the web, especially in a client-server application as the one we decided to adopt.

Almost all the components we identified are installed on the application server, so they are located on the same machine in order to facilitate the direct communication between them.

As for the interface between the application server and the database server, we chose a simple implementation: for now, these two servers are in the same location, so they are connected through a local network. The information are retrieved from the database simply using SQL queries.

We're considering for the future a better and more secure implementation, using a remote database, accessed through the SQL*Net networking software, and the cloud for the back-up.



3 Algorithm Design

In order to execute specific operations and accomplish its tasks, the managers of the application exploit dedicated algorithms.

We decided to focus on two of the most important algorithms that are part of our project: the first algorithm has the objective of managing the taxi queues in each zone of the city, while the second one is the well-known Dijkstra algorithm, used to compute the shortest path for a taxi in order to reach the location where the passenger required the taxi. This shortest path is then communicated to the taxi driver on his on board mobile app.

Management of taxis queues:

We use the concept of “queue” in the standard definition. A queue is a data structure based on a FIFO (“First In First Out”) mechanism: this means that the first element inserted in the queue is the first one that goes out. For our application, we imagine that the elements of the queues are all the taxi of the city, each one identified by its unique code. In the application database, each taxi is also associated with a number, that obviously can change, representing its position in the queue and a letter identifying the zone in which the taxi is located.

There are different queues, one for each zone of the city, and each zone is identified by a different name (Zone A, Zone B, Zone C and Zone D).

When a request arrives, the first taxi in the queue of the requested zone is notified: if it's occupied, it is move to the bottom of the queue and the request is forwarded to the following taxi; otherwise, if the taxi is available, it accepts the request and it's also moved to the bottom of the queue.

So, as for the algorithm, we have a function enqueue(Q,x) that inserts the element x in the queue, in the last position. This function is useful at the beginning of the day, when the application starts and all the queues are created or when a taxi needs to be moved to the bottom of the same queue of another queue (this can happen when a taxi ends its ride in a different zone with respect to the starting zone).

The function dequeue(x) removes the first element x (the first taxi) and returns it.

The function isEmpty(Q) returns a boolean specifying whether a queue is empty: considering our assumption ("There must be always a taxi in every queue") listed in the RASD document, this function will always return false.

The function front(Q) returns the first element of the queue, that is, in our case, the first taxi in the queue. We also define two important attributes that can help in the execution of the operations on the different queues: Q.head represents the beginning of the queue and Q.tail indicates the position of the next element (where a new element

will be inserted in queue). So, the elements of the queue are in the positions Q.head, Q.head+1,…, Q.tail-1. The queue is managed according to a circular orderd: this means that if we have a queue with n-1 elements (index from1 to n), after position n, we have position 1. When Q.head = Q.tail the queue is empty, while if Q.head = Q.tail = 1 the queue is full.

Here a simple pseudocode of the algorithm we have just described:

```
ENQUEUE(Q,x)
1  Q[Q.tail] = x
2   if Q.tail == Q.length       // if we are at the end of the array
3    Q.tail = 1          // we start again from the first position
4   else  Q.tail = Q.tail + 1

DEQUEUE(x)
1   x = Q[Q.head]
2   if Q.head == Q.length       // if we are at the end of the array
3     Q.head = 1                // we start again from the first position
4   else  Q.head = Q.head + 1

IsEMPTY(Q)
1  if Q.head == Q.tail
2    return true
3  else  Q.tail = Q.tail + 1

FRONT(Q)
1  return Q[Q.head]
```

Shortest path for a taxi:

In order to find the shortest path between two locations, we decided to use a well-known algorithm: Dijsktra's algorithm. We try to reduce the real-world situation to a simple directed graph, in order to execute the

algorithm: in our case, each node of the graph represents a street (identified by a progressive number according to alphabetical order) and each arc represents a path connecting two nodes. The cost of each arc is the time (expressed in minutes) needed to go from a node to the other.

Here the definition of the algorithm and the pseudocode describing all the steps:

Given a directed graph G(N,A), with n nodes and m arcs, a cost cij for each arc (i,j) connecting node i to node j (with cij = +∞, if (i,j) $\notin$ A) and a node s $\in$ N, the Dijkstra's algorithm allows us to find the shortest paths from s to all the other nodes of the graph.

```
BEGIN
      S := {s};      // S = subset of nodes already considered
      L[s] := 0;     // L[j] = cost of a shortest path from s to j
      pred[s] := s; // pred[j] = predecessor of j in the shortest path from s to j

      WHILE |S| ≠ n DO
      select (v,h) ∈ δ+(S) = {(i,j) : (i,j)∈A, i∈S, j∉S} such that
            L[v] + cvh = min {L[i] + cij : (i,j)∈ δ+(S)};
      L[h] := L[v] + cvh;
      pred[h] := v;
      S := S ∪ {h};
      END-WHILE
END
```

4 User Interface Design

We provide in the following pages some mockups representing how the user interfaces of our system will look like.

In the RASD document we have already provided few mockups representing our idea of the structure of the most important application pages. However, those mockups represented our initial idea of the interfaces and were very essentials and schematic.

We now want to show how our application will really be and how all the functionalities offered can be exploited. We represent the web application interface for the guests and registered users and the on board mobile app of each taxi driver.

In a near future, a mobile application for each kind of smartphone (Android, IOS, Windows Phone) is planned to be developed and implemented, but for now the service is offered only through the web and can be exploited (on PCs, smartphones and tablets) using any web browser and an internet connection.

The following mockups are thought to be browsed sequentially, in order to correctly visualize the flow of the navigation and the consequences of each action or operation.

Obviously, not all the possible functionalities and pages are show,

because honestly this would have been too long; however, we have drawn a remarkable number of mockups showing almost all the functionalities offered.

5 Requirements Traceability

Goals	Components
Goal 1: Register as a User	
[1] Only registered customers can request a taxi ride.	Web Server Application Server (Three tier architecture)
[2] Using phone number or email address to register as a user of the taxi service	Web Server Application Server (Three tier architecture)

Goal 2: Log into the taxi service using the mobile phone number	Web Server Application Server (Three tier architecture)
[1] Customers should be able to access the service through both the web and the mobile application	Web Server Application Server (Three tier architecture)
[2] The system should allow the log out functionality.	Web Server Application Server (Three tier architecture)
[3] Only registered customers can access MyTaxiService's services.	Web Server Application Server Users Manager (Three tier architecture)

Goal 3: Request a taxi	
[1] Customers must insert a valid origin location in order to request a ride.	Rides Manager
[2] The system will not allow more than a request if the previous one (either request or reservation) has not been accomplished yet.	Web Server Application Server (Three tier architecture)

Goal 4: Reserve a taxi by specifying the origin and the destination of some rides	
[1] The system allows reservations only 2 hours before the time and date specified by the customer.	Rides Manager
[2] The system should allow taxi reservations for a specific path communicated by the customer.	Rides Manager
[3] The system must not allow overlaps between reservations made by the same customer.	Rides Manager
[4] The system will assign a taxi driver for the reserved ride 10 minutes before the time and date specified by the customer.	Rides Manager

Goal 5: Checking the information about the requested taxi	
[1] After a taxi driver accepted a taxi request from a user, then the server will send the information about the taxi and the driver to the customer(user).	Web Server Users Manager (Three Tier Architecture)

Goal 6: Inform the system about the taxi's status	
[1] Taxi drivers should be able to communicate their current availability state to the system.	Rides Manager Web Server
[2] Taxi drivers must be able to log in the mobile application with preassigned credential and be identified as drivers.	Rides Manager Web Server
[3] At the end of their worksheet, taxi drivers must be able to log out of the mobile application in order to communicate to the system that they are no longer active.	Rides Manager Web Server

Goal 7: Confirm or refuse an request or a reservation	
[1] After receiving an incoming request, the taxi driver should be able to either confirm it or not.	Rides Manager Web Server Users Manager

Goal 8: Answer the request by informing the passenger the code of a incoming taxi	
[1] Customers must receive the taxi code in order to be able to recognize its driver.	Rides Manager Users Manager
[2] Customers must receive the taxi drivers' contact number after the system has paired them	Rides Manager Users Manager Web Server
[3] Customers must receive the taxi code in order to be able to recognize its driver.	Rides Manager Users Manager

Goal 9: Automatically distributes the taxi in the various city zones(based on the GPS)	
[1] The system should be able to allocate the taxis to different city zones according to the designed algorithms based on the GPS.	Application Server Server Manager

Goal 10: Manage the taxi queues of different zones in the system	
[1] The system should be able to manage the taxi queues according to the status of the taxis	Server Manager Rides Manager

6 References

Here is a short list of the references for this Design Document:

- Slides of the Software Engineering 2 course (from the Beep Platform)
- Design Document Template (from the Beep Platform)
- Software Engineering: Principles and Practice (Hans Van Vliet)
- UML Distilled (Martin Fowler)
- Introduction to Operations Research (Frederick S. Hillier, Gerald J. Lieberman)
- Wikipedia (https://www.wikipedia.org/)