Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Design Document
myTaxiService

Student: Federico Gatti [Matricola: 852377]
Student: Luca Foochetta [Matricola: 792935]

AA 2015–2016

Contents

Chapter 1

RASD Revision

In this section are described the corrections of the RASD.

1.1 State Diagram

State Diagram of the RASD document said that a taxi can't change its status from *Busy* or *Out of Service* to *Transition* but only to *Available*. This is incorrect to obtain a correct System. In fact a taxi will have an *Available* status only if it is in the correct zone of competence and so in the queue of the corresponding zone; but if a taxi decided to change its status, for example from *Busy,* but it isn't in its competence zone it must change its status not in *Available* but into *Transition* and it isn't adding in the queue.

Also the diagram in the RASD shows that from *Transition* is not reachable *Busy*, this is an error, in fact the RASD requirements said:

- The system provides a command to change taxi's status from *Available/Transition* to *Busy* when the taxi picks up a passenger

In the next figure you can see the correct diagram

StateMachine1::StatechartDiagram1

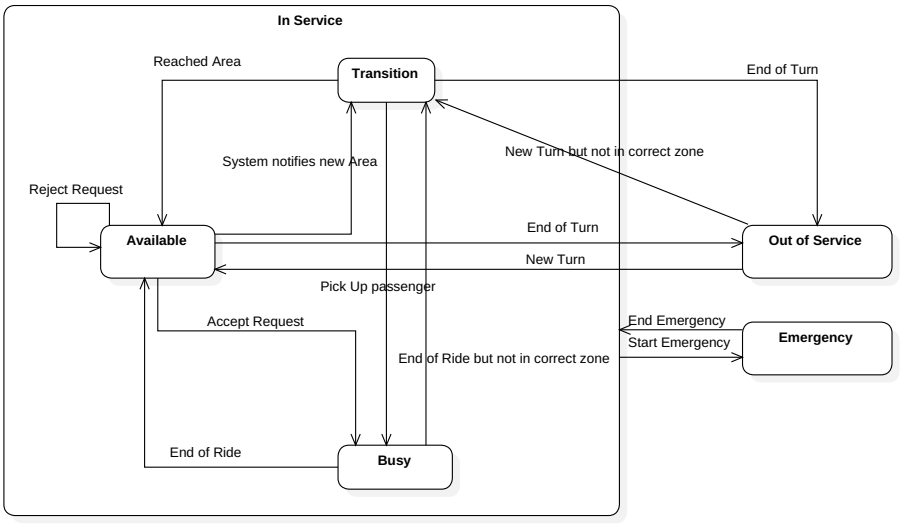


Figure 1.1: State Diagram

Chapter 2

Introduction

2.1 Purpose

The purpose of the *Design Document* is to give a detailed description, analysis and specification of the software and hardware structure for the *myTaxiService* system. This document, starting from the analysis performed in the RASD, will explain the architectural structure of the system using a top-down method.

The main objective of the *Design Document* is to achieve good understanding among analysts, developers, testers and customers. It is also aimed to be a solid base for project planning, software evaluation and possible future maintenance activities. Therefore this document is primarily intended to be proposed to the stakeholders for their approval, to the analysts and programmers for the development of the project and to the testing team the validation of the first version of the software.

2.2 Scope

The aim of the project is to create a software, called *myTaxiService*, which can manage the queue of the taxi requests in a city.

The system is composed by four parts:

- Two Front-End applications used by the passengers called PMA and PWA
- One Front-End application used by the Taxi Drivers called TMA
- One back-end sub-system called QTM

The system will be able to suggest the best distribution of the taxi in the city zone to maximize passengers' satisfaction and the quality of the service.

A passenger who sends a request, using a Web application or Mobile Application, can see from the application the *waiting time* and the code of the taxi that accepted the request. The passenger position can be determinate from GPS or if GPS information are incorrect, or aren't available, the passenger can insert manually the information of the location.

An authenticated passenger can reserve a taxi by specifying the origin and the destination of the ride. In this case the passenger must place the reservation at least two hours before the ride. The reservation request will be sent like a normal request ten minutes before the specified time. An authenticated user can always delete his reservation or modify it.

A Taxi driver can accept or reject a request.

2.3 Definitions, Acronyms, Abbreviations

- PMA: Passenger Mobile Application
- PWA: Passenger Web Application
- PA: Passenger Application refers indiscriminately at PMA and PWA
- TMA: Taxi Mobile Application
- RASD: Requirement Analysis and Specification Document
- QTM: Queue and Taxi Manager
- DB: Data Base
- DBMS: Data Base Management System
- API: Application Programming Interface

2.4 Reference Documents

- Template for The Design Document
- Wikipedia
- Graphs and Network Flows
- Fundamentals of Database System

2.5 Document Structure

This document is drawn up according to the Template for The Design Document and is composed of five sections and appendix.

1. The first, this one, section gives a general description of the document and brief information about the system and the purpose of the design.

2. The second section provides, using the UML diagrams, a detailed description of the system, highlighting in the division of the system in its sub-systems and how they interact each other. Starting from an high level description of the system, the chapter will explain in detail what are the sub-components.

3. The third section explains the main algorithms which are performed for the correct operation of the system. The algorithms is described using a pseudocode: this will allow the developers to use the programming language that they prefer

4. The fourth section describes with a UX diagram the users interaction with the system

5. The fifth section lists, using a table, the components which perform the functional requirements described in the previous document (see section 3.2 of the RASD)

Chapter 3

Architectural Design

3.1 Overview

This chapter shows you the different parts of the *myTaxiService* System. Starting from an high level description of the system, the chapter will explain in detail what are the sub-components ah how they interact each other.

3.2 High level components and their interaction

The system, as it is described in the RASD and touched upon beforehand, is composed by four essential components. Three are Front-End applications and the last is a Back-End sub-system. In detail:

- Two Front-End applications used by the passengers called PMA and PWA. This applications provide an user friendly interface that allow the Passengers to communicate their action to the system. The PMA can be installed into all mobile devices and the PWA can be run in all browsers.
- One Front-End application used by the Taxi Drivers called TMA. This application provides an user friendly interface that allow the Taxi Driver to answer at the passengers' requests and changes taxi's status.
- One back-end sub-system called QTM. The QTM isn't only the Server application, but it contains also the Web component and the Database. QTM has the tasks to manage all the requests and reservations that come from the PA and correctly forward it to the TMA. QTM has also the assignment to manage and store the data in the DB. The DB contains all the information about the taxis, registered passengers, reservations and requests. QTM can be decomposed in another three main components.

1. **Web Tier**: is the tier that receives the requests from the PA and TMA and, after an elaboration of these requests, forwards it to the Application tier. The Web tier also

has the task to submit the answers or requests coming from the Application Tier to the correctly Taxi or Passenger Application. The Web tier responds to the clients using xml files: this allow to handle in an indiscriminately way the Mobile and Web Application.

2. **Application or Business Tier**: is the tier where is implemented the logic of the system. The application layer receives the clients' requests by the Web tier and, after an elaboration of the requests, submits the requests to the taxis using the Web tier. The Application layer is the only layer that can obtain data from the DB.

3. **DBMS and Database**: is the tier that contains and manage all the data of the system.

The system uses Google Maps APIs to obtain all the taxis' position and, when is available, to obtain the passengers' positions at the moment of the requests.

In the figure 3.1 is represented the system described before.

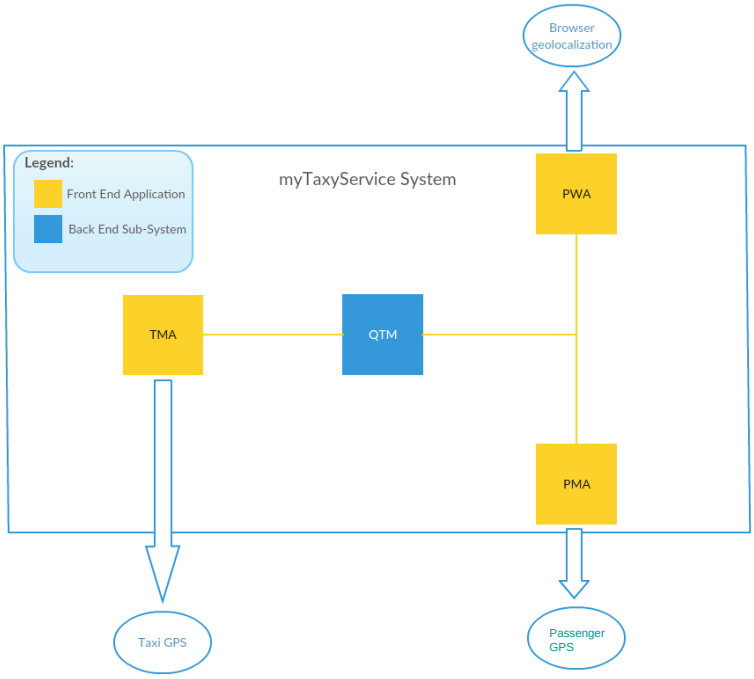


Figure 3.1: High level diagram of the system

3.3 Selected architectural styles and pattern

In this section is described the architectural choices made for the system and illustrates the main design patterns used to develop it.

All the subsections start with a little theoretical presentation of the architecture or the pattern, after that is explained the reason of this choice.

3.3.1 Client - Server

The client–server model of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

The Client-server characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services. Servers are classified by the services they provide.

Clients and servers exchange messages in a *request–response* messaging pattern: The client sends a request, and the server returns a response

3.3.1.1 Our system

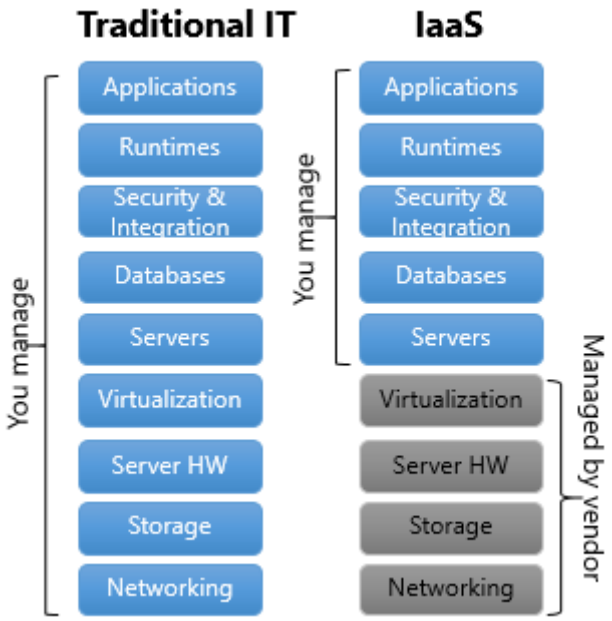
Considering the fact that *myTaxiService* is a distributed system and there are several clients who require a service, the Client - Server architecture seems to be a good solution. Another reason for using this architectural style is that some operations can be performed locally (for example the queue management), so there is the necessity to have a server that provide this services.

The fact that the application logic cannot be run locally suggest us that Peer-to-peer architecture is a worst solution.

3.3.2 IaaS

Considering the fact that the numbers of the requests during a day is very variable we prefer to use a Cloud solution for the system. An *IaaS* (Infrastructure as a Service) solution allow us to maintain a *Client-Server* architecture but with an high scalability and a cost management that depending only to the number of requests. The most different between an *IaaS* an a in-house solution is that the Java EE is going to be executing on top virtualized machines, but the architecture doesn't change, except the fact that the components that we want to scale should be stateless to allow the creation of replica. A hypervisor runs the virtual machines as guests. Pools of hypervisors within the cloud operational system can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements. IaaS clouds often offer additional resources such as a virtual-machine disk-image library, raw block storage, file or object storage, firewalls, load balancers, IP addresses, virtual local area networks (VLANs) and software bundles. IaaS-cloud providers supply these resources on-demand from their large pools of equipment installed in data centers.

In the next figure you can see the main difference between a Traditional IT and an *IaaS*.



3.3.3 Multilayered architecture

A multilayered software architecture is a software architecture that uses many layers for allocating the different responsibilities of a software product. The layer is a logical division of the tasks in the system.

3.3.3.1 Common layers

In a logical multilayered architecture for an information system with an object-oriented design, the following three are the most common: Presentation, Application and Data access

3.3.4 Multitier architecture

Multitier architecture (often referred to as n-tier architecture) is a client–server architecture in which presentation, application processing, and data management functions are physically separated.

N-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application.

The most widespread use of multitier architecture is the three-tier architecture.

3.3.4.1 Three-tier architecture

Three-tier architecture is a client–server software architecture pattern in which the user interface (presentation), functional process logic (business rules), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.

Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology. For example, a change of operating system in the presentation tier would only affect the user interface code.

This choice is indispensable to obtain a system that can be easy modifiable. A change of one tier doesn't impact the others.

3.3.4.2 Three-tier architecture components

In the three-tier client-server architecture, the following three layers exist:

- **Presentation layer** This provides the user interface and interacts with the user. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access.When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.

- **Application layer** (business logic). This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources.

- **Data server**: This layer includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. This layer handles query and update requests from the application layer, processes the requests, and sends the results. The data access layer should provide an API to the application layer that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms.

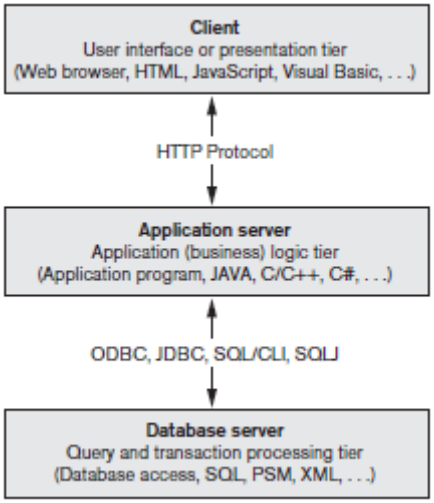


Figure 3.2: Three-tier architecture

3.3.5 MVC

Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

3.3.5.1 Components

The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface. The *model* directly manages the data, logic and rules of the application. A *view* can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the *controller*, accepts input and converts it to commands for the *model* or *view*

3.3.5.2 Interactions

In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.

- A *controller* can send commands to the model to update the model's state. It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).
- A *model* stores data that is retrieved according to commands from the controller and displayed in the view.
- A *view* generates an output presentation to the user based on changes in the model.

This kind of solution allow to change one component without the necessity to modify the others.

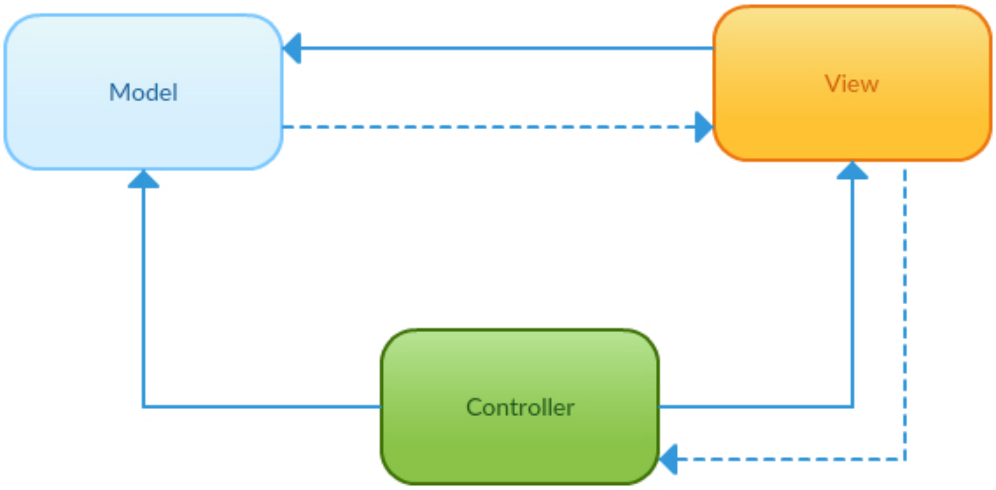


Figure 3.3: MVC pattern representation

3.3.6 Architectural choices

Considering the decision to use an *IaaS*, the responsibility to subdivide the logic layers into physical tier is a cloud provider's decision: for this reason we doesn't matter how they are going to handle it.

We can speculate that the cloud provider will be using a Client - Server architecture with three or four tier. In the next figure is represented a possible three tier solution

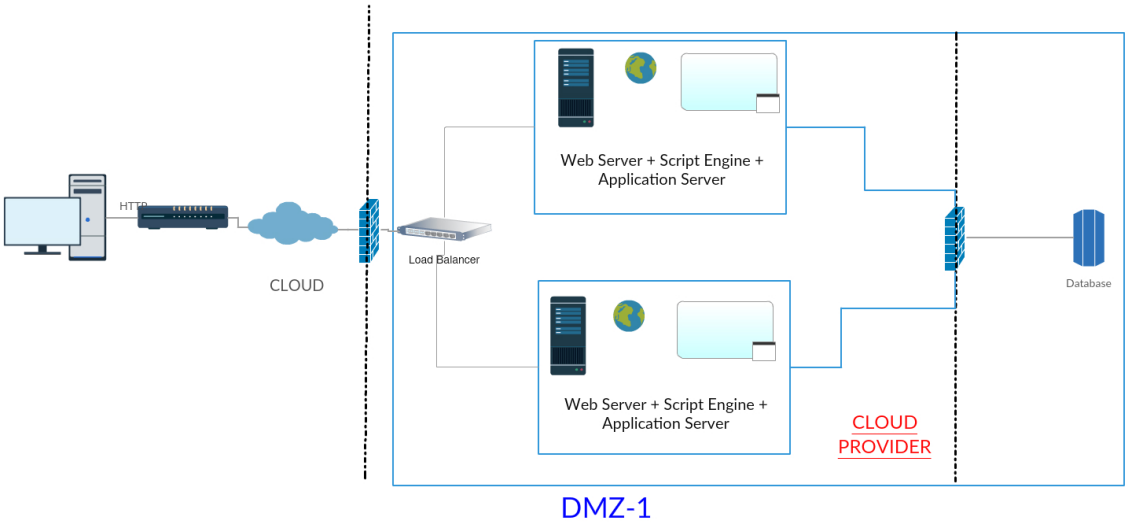


Figure 3.4

The choice that we have to take is the general system's architecture and we have also

to specify the system's logic application. The better solution is not to make clear division where each tier correspond to one layer, but instead using an hybrid solution of this kind:

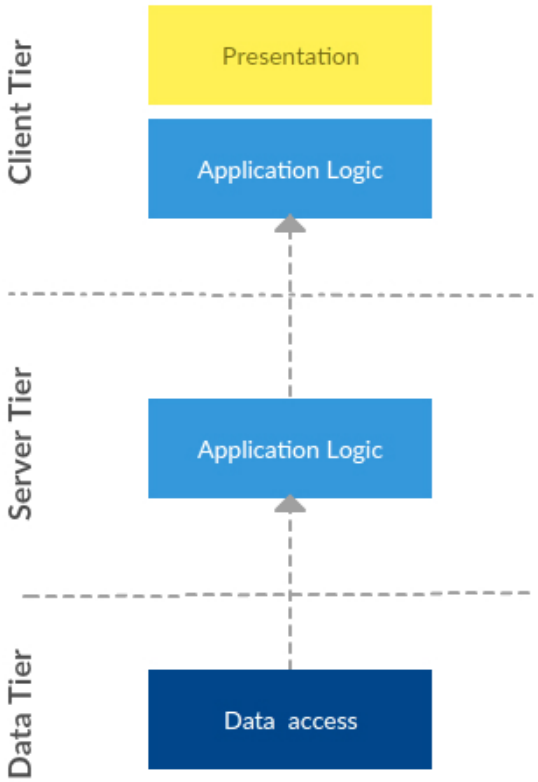


Figure 3.5: Layer and tier division

The presence of logic application inside the client tier makes it a *Fat Client*. This is because the client needs application features to use API Google Maps to locate the passenger or the taxi, and the client could also check input data (for example if it is filled the position's field when a Passenger makes a request).

Cloud Provider allows high level of scalability to the system using virtual machine that aren't managed directly by us; for this fact we doesn't matter what type of architecture (for example server farm with multiple application server) the provider uses to implement our system.

To maintain a copy of the data system is expected a daily back up, this make the system robust to data loss and corruption.

3.3.7 Security

Considering the fact that the System manages personal data is necessary to implement some security features. First of all encrypted sessions are necessary: these can be implemented using HTTPS. Another level of security is implemented using the architecture of

the system; Client can only communicate with the logic unit by the web server that can delete all possible intrusions using a firewall or other software.

The system must use methods of authentication to access at the logic application and data.

3.3.8 Commercial choices

In this section are showed some possible commercial choices for the system. Each section contains a little description of the component in which are described the main features of the platform.

This suggestion can be used for the future developers for the implementation of the system.

3.3.8.1 Application Server

One possible chioce for the application server is Java Enteprise Edition.

The **Java™ Platform**, Enterprise Edition (Java™ EE) is a widely used enterprise computing platform developed under the Java Community Process. The platform provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications.

The Java Enterprise Edition reduces the cost and complexity of developing multitier, enterprise services. Java EE applications can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures.

In the system design ca be used different Java EE APIs for our scopes, some of this are:

1. **Servlet** is an object that receives a request and generates a response based on that request
2. **JavaServer Faces** (JSF) is a Java specification for building component-based user interfaces for web applications. JSF 2 uses Facelets as its default templating system
3. **Enterprise Java Beans** (EJB) provides an architecture to develop and deploy component based enterprise applications considering robustness, high scalability and high performance
   (a) Simplified development of large scale enterprise level application
   (b) Application Server/ EJB container provides most of the system level services like transaction handling, logging, load balancing, persistence mechanism, exception handling and so on. Developer has to focus only on business logic of the application
   (c) EJB container manages life cycle of EJB instances thus developer needs not to worry about when to create/delete EJB objects

4. **Java Persistence API** (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database.

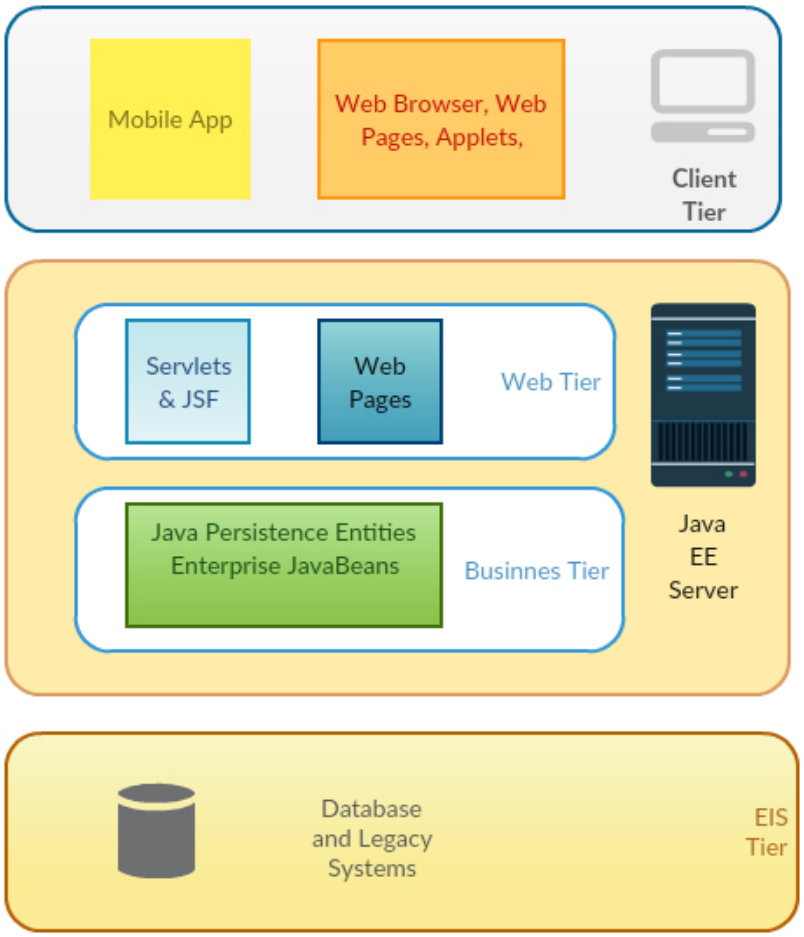


Figure 3.6: J2EE in client server architecture

3.3.8.2 Mobile Client application

The mobile application can be developped using **Ionic framework**.

Ionic is a complete open-source SDK for hybrid mobile app development. Built on top of AngularJS and Apache Cordova, Ionic provides tools and services for developing hybrid mobile apps using Web technologies like CSS, HTML5, and Sass.

3.3.8.3 Web Server

For the Web Server is chose **Apache HTTP Server**. Apache supports a variety of features, many implemented as compiled modules which extend the core functionality. Popular compression methods on Apache include the external extension module, mod_gzip, implemented to help with reduction of the size (weight) of Web pages served over HTTP. ModSecurity is an open source intrusion detection and prevention engine for Web applications. Apache logs can be analyzed through a Web browser using free scripts.

Apache features configurable error messages, DBMS-based authentication databases, and content negotiation. It is also supported by several graphical user interfaces (GUIs).

It supports password authentication and digital certificate authentication. Because the source code is freely available, anyone can adapt the server for specific needs, and there is a large public library of Apache add-ons

3.3.8.4 DBMS

One of the best solution for the the DBMS is **MySQL** is an open-source relational database management system (RDBMS). There are several motivation for use this platform, the mains are

- MySQL can be run on cloud computing platforms such as Amazon EC2.

- High availability using *MySQL Fabric.* MySQL Fabric is open-source and is intended to be extensible, easy to use, and to support procedure execution even in the presence of failure, providing an execution model usually called resilient execution.

- Backup software: MySQL Enterprise Backup is a hot backup utility included as part of the MySQL Enterprise.

3.4 Component view

The Component Diagram has the aim to describe the sub-components that composed the system. Obviously all the components aren't marked but only the mains; maybe some of these components can be decomposed in other sub-components, but the intention is to give an high level idea of the System and allow the future developers to have a starting point maintaining a degree of freedom to develop all the classes in the way that they consider more appropriated.

The next figure illustrates a representation of the system

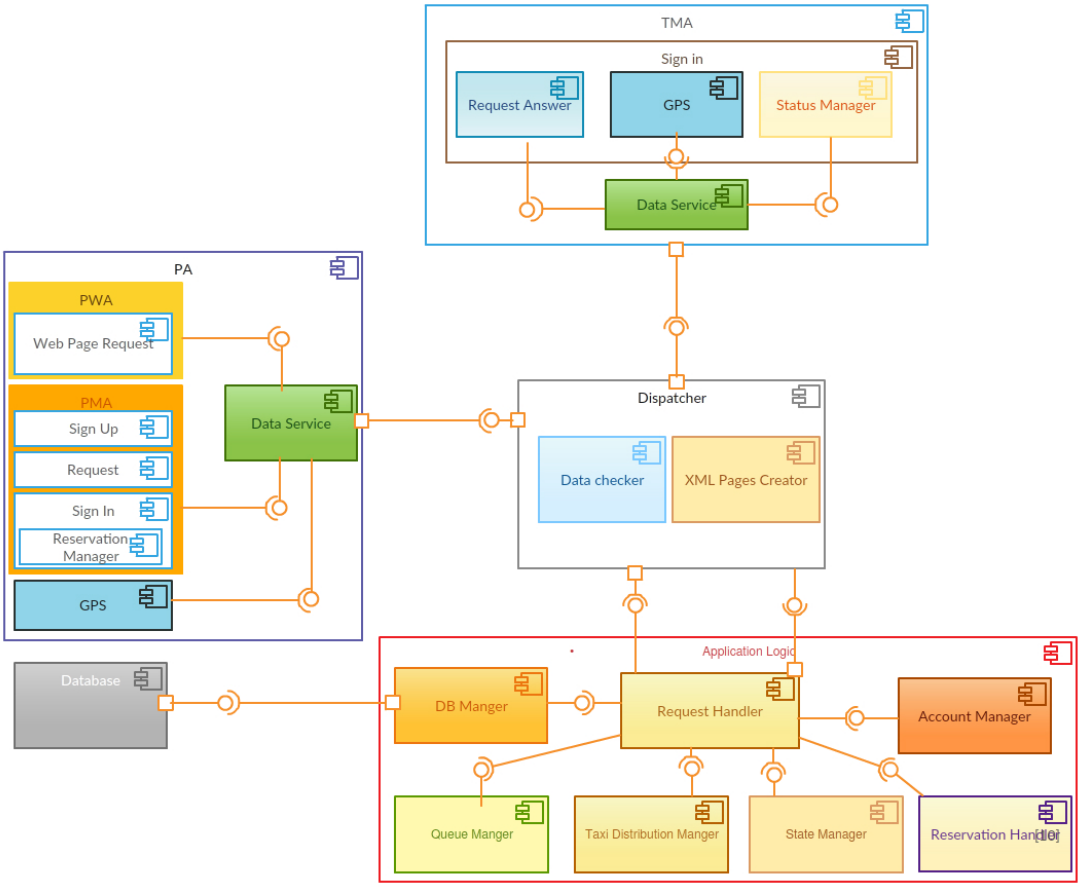


Figure 3.7: Component Diagram

3.4.1 Sub-Components

Application Logic is the component in which resides the application logic of our system:

- **Request Handler** takes all the requests that come from the Dispatcher and forward them to the correct components inside the Application Logic; it provides internal component communication and also sends the answers from the Application Logic to the Dispatcher. Finally it provides the communication from the Application logic and the Database
- **Queue Manager** handles all the queue of the system according to the RASD requirements
- **Taxi Distribution Manger** has the task to maintain a optimal taxis' distribution in the city
- **State Manger** has the task to maintain consistent the taxis' status in the system according to the RASD requirements, both regarding the passengers' requests and taxis' answer and request of status change. Status Manager also provides the linked actions regarding a status changement: for instance, when a taxi had accepted a

request but changed its state into *Emergency* the Status Manager must forward a new request to the system

- **Reservation Handler** has the task to handle all the request coming from the passengers, the storage of all the reservations, and it also doesn't allow multiple rejections from the same taxi for the same request and implements the request answer *time out*
- **Account Manager** has the task to handle all the Passengers' and Taxi drivers' account, in particular checks data at the moment of sign in to verify if already exists an account for that user. In addiction Account Manger avoids that an already registered user can perform another registration with the same data.
- **DB Manage**r provides the communications between the Application Logic and the Database. DBM takes all the data requests coming from all the Application Logic components forwarded by the Request Handler and creates query to interrogate the DB, once obtain the answer sends it to the Request Handler.

Dispatcher has two different tasks:

1. Take all the requests that come from the client, send it to the data checker and analyzes the output; if the output is an error message discards the request and notifies the client, else forwards the output to the Application Logic
2. Take all the answers that come from the Application Logic, send them to the XML Pages Creator, which will forward the output to the correct Client
   (a) **Data checker** checks that the incoming message is correct and complete; if it's ok, the output is a formatted request; otherwise Data checker sends an error message
   (b) **XML pages creator** creates the XML files to send to the client using the data coming from the application logic

PA is the Passenger Application and is composed by three sub-system :

1. PWA is the web application. The only sub-component that we find in this application is the Web Page Requster (WPR). WPR has the task to contact the web server to require the XML pages, after that the browser interprets them to show an UI to the customer.
2. PMA is the mobile application. Differently from PWA the PMA need one sub-component for each feature that the Passenger can perform, so in this sub-component we find:
   (a) **Sign Up** allows passengers to register into system
   (b) **Sign In** allows passengers to sign in into the system

(c) **Request Creator** allows passengers to place a request

(d) **Reservation Manger** is a Sign In sub-components and allows passengers to place and manage his reservations (according with the RASD requirements)

3. **Data Service** has the tasks to take all the outcoming requests and address them to the web server; it is the component that listen to an answer of the web server, also can check that all the fields are filled out.

4. **GPS** provides the localization of the passenger client

TMA is the Taxi Mobile Application, like PMA, it doesn't need to require XML pages to allow a Taxi Driver to perform an action after the **Log In**, but there is a sub-component for each feature, in detail:

1. **Request Answer** allows Taxi driver to answer for and incoming request

2. **Status Manger** allows Taxi Driver to change his state

3. **Data Service** has the tasks to take all the outcoming requests and address to the web server, is the component that listen to an answer of the web server, also can check that all the fields are filled out.

4. **GPS** provides the localization of the taxi client

Database contains and manages all the data system

3.5 Deployment view

A deployment diagram in the Unified Modeling Language models the physical deployment of artifacts on nodes.

The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have sub-nodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.

There are two types of Nodes:

1. Device Node

2. Execution Environment Node

Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones.

An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.

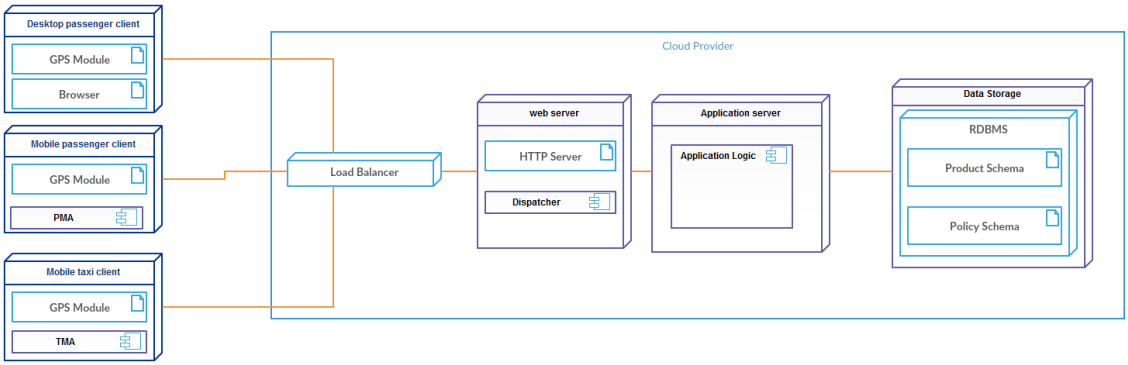
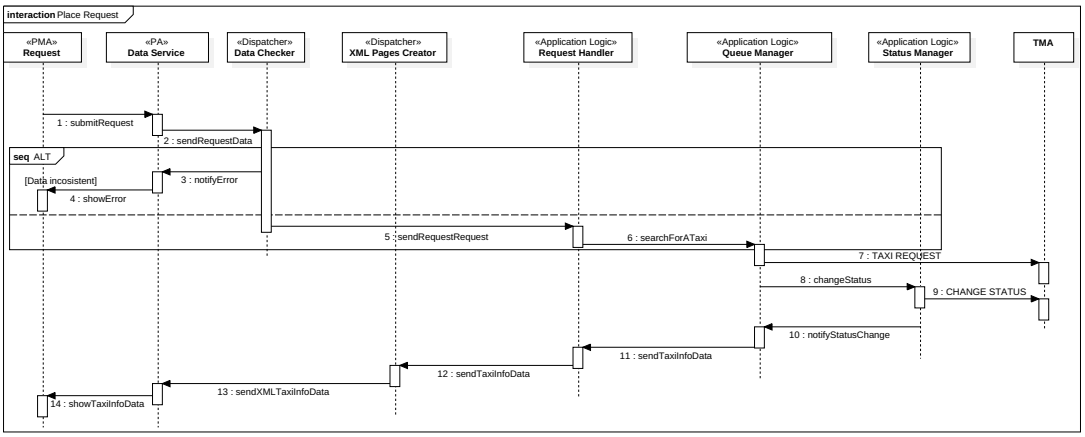


Figure 3.8: Deployment diagram

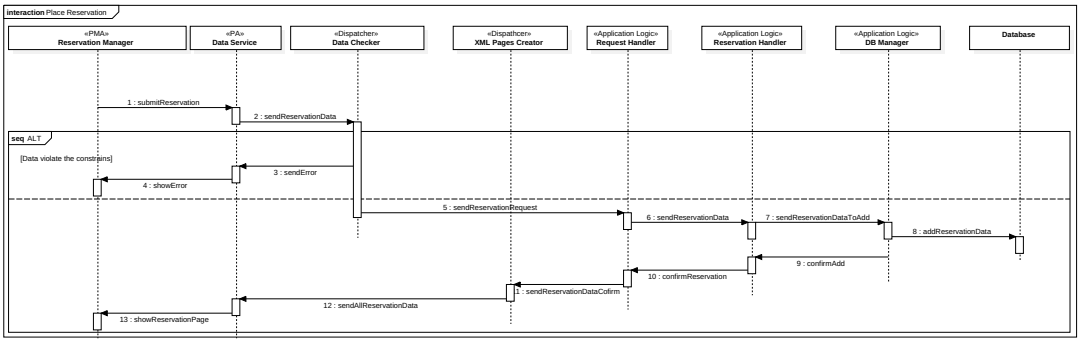
3.6 Runtime view

This chapter shows the interaction between the sub-components that composed the system

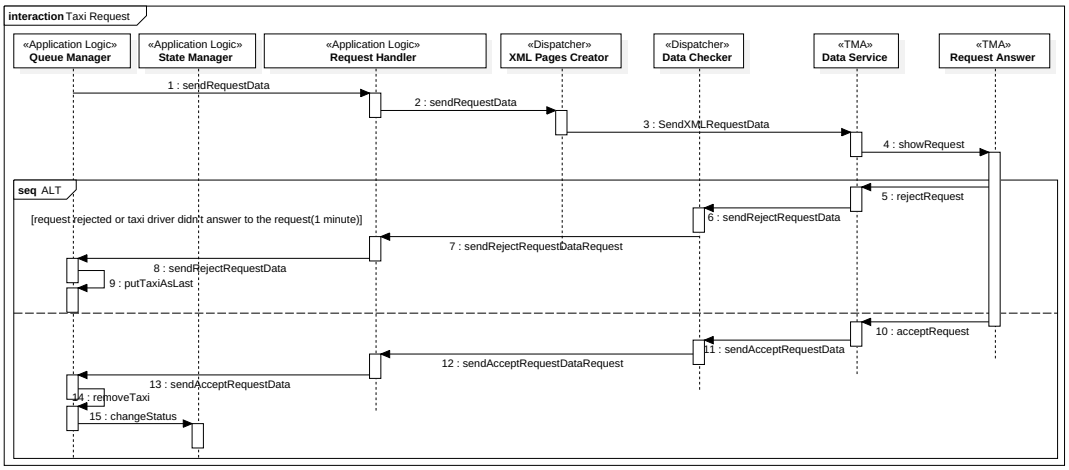
3.6.1 Place Request



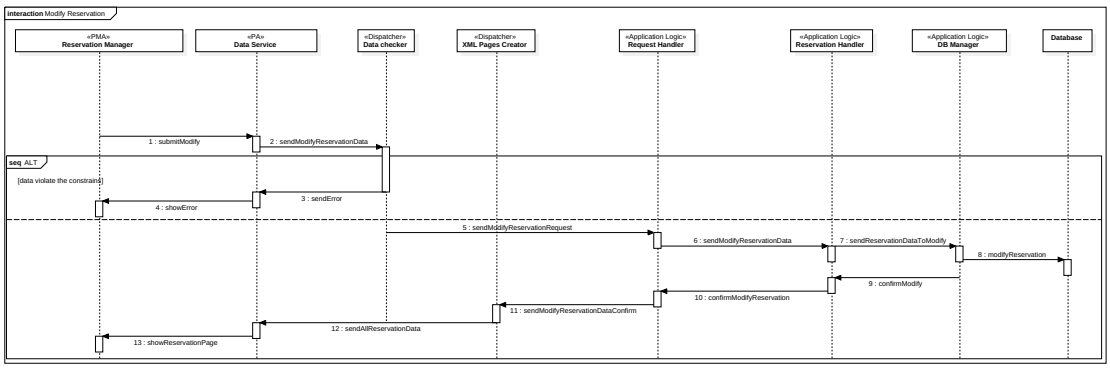
3.6.2 Place Reservation



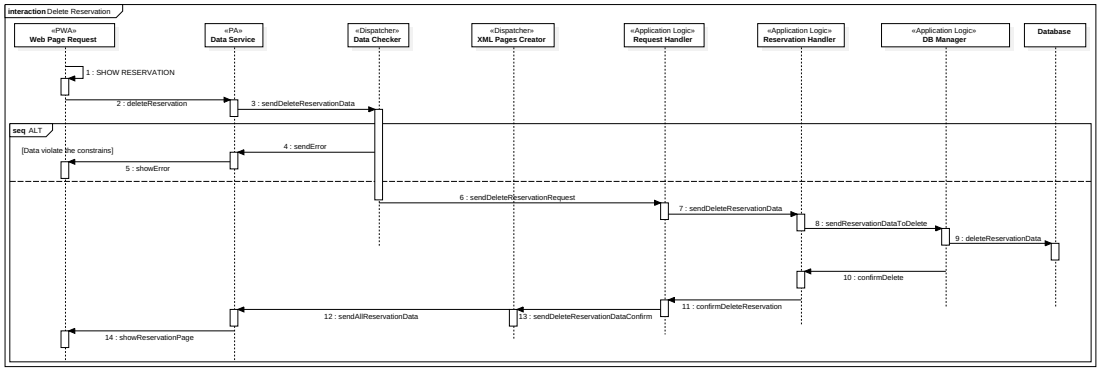
3.6.3 Taxi Request



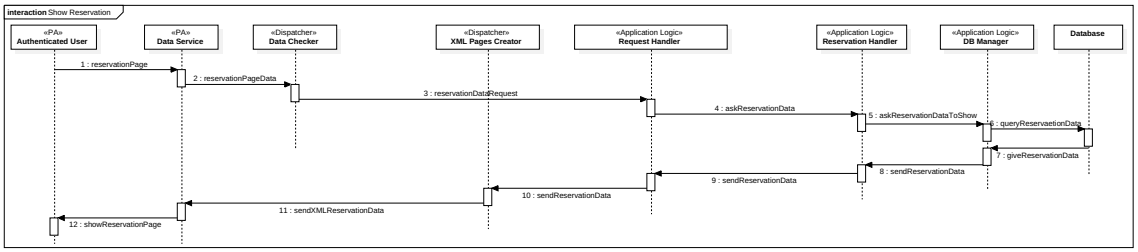
3.6.4 Modify Reservation



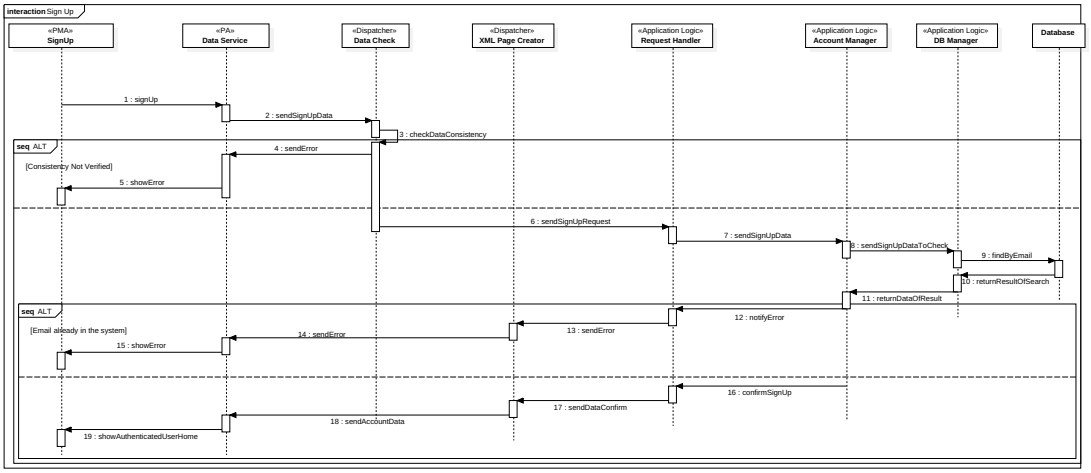
3.6.5 Delete Reservation



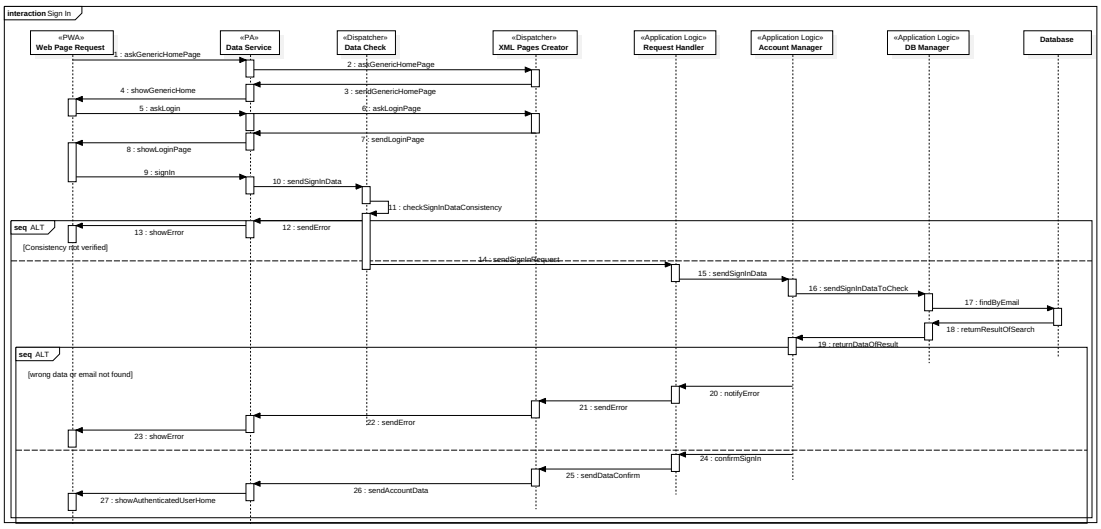
3.6.6 Show Reservation



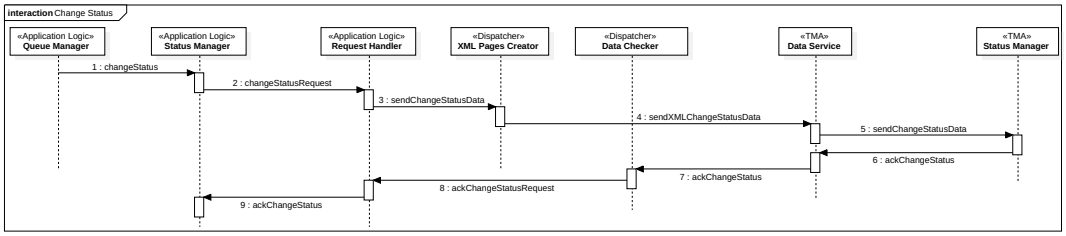
3.6.7 Sign Up



3.6.8 Sign In



3.6.9 Change Status



Chapter 4

Algorithm Design

In this chapter are described the mains system's algorithms. All the algorithms are subdivided for components.

Data:

- Z identifies the Set of the Zone
- N identifies the number of *Available* Taxis in the city
- $a_z$identifies the queue of *Available* Taxis in the zone z
- $t_z$identifies the taxis that are reaching z,taxis that have $Transition$ status, so this taxis aren't in any queue
- $q_z$identifies the queue of the zone z
- r_z identifies the number of taxis requested in the zone z to obtain a fair distribution
- $m_z$is the number of request per minute (in the RASD this value was called r_z)
- $T \epsilon [0, 100]$ identifies the tolerance of the system. A good T can be 25.
- $T(r_z)$ is the value $r_z$considering the tolerance

The number of taxis, according to the RASD, in the zone z must be $r_z = \frac{N m_z}{\sum_{z=1} m_z}$.

$T(r_z) = r_z - \frac{r_z * T}{100}$

4.1 Taxi Distribution Manager

Taxi Distribution Manger has the task to maintain a optimal taxis' distribution in the city. The algorithm must be able to take taxis from the zones that have a surplus of taxis and relocate them in the zones that have a deficiency.

To simply the representation and the calculation in this section we suppose that in the system there aren't taxi in *Transition*, this means that a taxi can reach immediately a zone when the system notify it.

4.1.1 Minimum cost flow

The first algorithm uses as cost the distance between two zone. The cost to reach an adjacent zone is 1.

A network flow is specified by a digraph $G = (Z, A)$; each node is a zone and for each node $i \epsilon Z$ we associate a value b_i (called balance of the node); with each $arc(i, j)$ we associate the unit cost c_{ij} and a capacity u_{ij} limiting the maximum quantity of flow that can transit. Balances at nodes regulate the flow on the network. A node with balance $b_i = 0$ is called a transshipment node because it neither requires nor provides any flow; a node with $b_i < 0$ called an *origin node* (or *source*) because it provides some flow; a node with $b_i > 0$ is called a *destination node* (or *sink*) because it requires some flow.

The *minimum cost flow problem* consists in determining the flow on the arcs of the network so that all available flow leaves from sources, all required flow arrives at origins, arc capacities are not exceeded and the global cost of the flow on arcs is minimized. By employing the usual flow variables x_{ij} on arcs, we obtain the following problem's formulation:

$$min \sum_{(i,j)\epsilon A} c_{ij} x_{ij}$$

$$-\sum_{(i,j)\epsilon FS(i)} x_{ij} + \sum_{(j,i)\epsilon BS(i)} x_{ji} = b_i \forall i \epsilon Z$$

$$0 \leq x_{ij} \leq u_{ij} \forall (i,j) \epsilon Z$$

Let us consider a flow $\bar{x}$ being feasible for the problem. we can define the residual graph $G_R(\bar{x})$ (see 4.1.1.4) associated with it. The residual graph provides the information needed to evaluate the possible flow variations. In the minimum cost flow case the "cost" component is essential to decide whether a flow variation is advantageous, therefore with each arc of the residual graph we can associate a *residual cost* g_{ij} quantifying the effect on the objective function of a unitary variation of the flow corresponding to the residual graph's arc:

$$g_{ij} = \begin{cases} c_{ij} & if (i,j) \epsilon A^+(\bar{x}) \\ -c_{ij} & if (i,j) \epsilon A^-(\bar{x}) \end{cases}$$

A route on the residual graph corresponds to a possible variation of flow $\bar{x}$, a positive variation for the flow variables corresponding to arcs of $A^+(\bar{x})$ and a negative variation for the flow variables corresponding to arcs of $A^-(\bar{x})$.

The only way of modifying the solution without violating the flow conservation constraints is to vary the flow on a cycle (or on a set of cycles) by an identical quantityθ. In fact, a flow variation on a route which does not close into itself would immediately bring to a violation of the flow conservation constraints which were satisfied by the original flow $\bar{x}$. Considered a cycle C, the maximum practicable flow variation is given by:

$$\theta = min\{\{u_{ij} - x_{ij}^{-} : (i,j) \epsilon A^+(\bar{x}) \cap C\}, \{x_{ij}^{-} : (i,j) \epsilon A^-(\bar{x}) \cap C\}\}$$

Note that, due to how the residual graph has been built, $\theta > 0$. The flow is updated according to the following rules:

$$x'_{ij} = \begin{cases} \bar{x_{ij}} + \theta & if(i,j)\epsilon A^+(\bar{x}) \cap C \\ \bar{x_{ij}} - \theta & if(i,j)\epsilon A^-(\bar{x}) \cap C \\ \bar{x_{ij}} & else \end{cases}$$

The variation of the objective function is the following:

$$\sum_{(i,j)\epsilon Z} c_{ij}x'_{ij} + \sum_{(i,j)\epsilon Z)} c_{ij}\bar{x_{ij}} = \sum_{(i,j)\epsilon C} g_{ij}\theta$$

This means that there is an improvement in the solution value only if cycle C has sum of negative residual costs. We can try the reverse as well, i.e., if there exist no negative cycles in the residual graph, then the flow is not improvable, as suggested by *flow decomposition theorem*.

A flow x is called a circulation if the quantity of flow coming into each node is equal to the quantity of outgoing flow, that is if:

$$-\sum_{(i,j)\epsilon FS(i)} x_{ij} + \sum_{(j,i)\epsilon BS(i)} x_{ji} = 0 \forall i \epsilon Z$$

A circulation x is said to be *simple* if arcs (i,j) in which $x_{ij} > 0$ detect a cycle. Note that, because of the flow conservation, the quantity of flow on the arcs of a simple circulation is constant.

We can provide a simple solution algorithm for the minimum cost flow problem which is based on the next flow update until the residual graph contains no negative cycles. The algorithm receives as input the network G and a feasible flow. The algorithm internally uses a procedure detecting negative cycles on the residual graph. This procedure can be efficiently carried out on the basis of shortest-path computation, as explained in the next paragraph. The flow update is performed according to the rules discussed above.

Algorithm 1 Minimum cost flow

```
1: procedure NegativeCycleElimination(G, x)
2:     Gr ← ResidualGraph(x)
3:     while exist negative cycle C in Gr do
4:         UpdateFlow(x, C)
5:         updateResidualGraph(x)
6:     end while
7:     return
8: end procedure
```

4.1.1.1 Detection of negative length cycles

There are different ways to recognize negative length cycles.

Let C be the maximum absolute value of arc lengths; evidently, if there are no negative length arcs, it will never be possible to have labels $d[j] < -(n-1)C$. As soon as, in the iterations of the algorithm, such eventuality takes place, we can stop. The negative length cycle can be reconstructed by backtracking along predecessors P starting from j, until j is not reached again. Negative length cycles can also be detected in a more efficient way.

By using the SPT_L algorithm implemented with a queue, we know that a node can be introduced in the rear no more than $n-1$ times, if there are no negative cycles. Therefore, as soon as a node is introduced into Q for the nth time, this means that a negative cycle is present. So, with a complexity of $O(nm)$ we detect negative length cycles. Nodes still being in queue Q after $n-1$ iterations of the algorithm are part of negative cycles. In order to reconstruct the cycles, it suffices to go up the predecessors of the nodes present in Q.

4.1.1.2 Feasible starting flow

In order to apply the algorithm we need a method enabling to generate a feasible starting flow, provided that one exists. The computation of a feasible flow can be performed in polynomial time by means of a maximum flow algorithm on an appropriate graph. Let us consider an auxiliary graph $G'=(Z \cup \{s,t\}, A')$ in which A, in addition to all arcs of A, also includes arcs connecting source s to nodes$i$ having negative balance (flow sources in G) and arcs connecting nodes having positive balance (flow destinations) to sink t. Arcs of type (s,i) have capacity$u_{si} = -b_i$, whereas arcs(i,t) have capacity $u_{it} = b_i$. All other arcs have unchanged capacity with respect to the original graph G. At this point node balances are set at zero. A feasible flow (if one exists) can be found by solving a problem of *maximum flow* (see 4.1.1.3) from s to t. If the value of the maximum flow is equal to $-\sum_{i:b_i<0} bi$, then there exists a feasible flow, and it is precisely given by the maximum flow which is found.

We can use the Edmonds-Karp algorithm for solving the maximum flow problem (complexity $O(m^2n)$)

4.1.1.3 Maximum flow

Given a digraph $G = (N, A)$ with u_{ij} capacities associated with arcs, the maximum flow problem consists in finding the maximum amount of flow that can be sent from node $s \in N$ (called source) to node $t \in N$ (called sink), while respecting arc capacities as well as the fact that flow must not be dispersed at intermediate nodes. The problem's variables are the quantities of flow sent to every single arc: x_{ij}, for each$(i,j) \in A$. The problem can be formulated as follows:

$$max \sum_{(s,j)} x_{sj}$$

$$0 \leq x_{ij} \leq u_{ij} \forall (i,j) \epsilon A$$

$$-\sum_{(i,j)\epsilon FS(i)} x_{ij} + \sum_{(j,i)\epsilon BS(i)} x_{ji} = 0 \forall i \epsilon N \setminus \{s,t\}$$

4.1.1.4 Residual Graph

Given a graph $G = (N, A)$ with capacity on the arcs u and a feasible flow $\bar{x}$, we define the residual graph $GR(\bar{x}) = (N, A(\bar{x}))$, where arcs are defined as follows:

Algorithm 2 Maximum Flow

```
1: procedure AugmentingPaths(G, x)
2:     x ← 0
3:     while P[t]!=0 do
4:         G_r ← Residual_graph(x)
5:         Search(G_r, s)
6:         if P[t]!=0 then
7:             AugmentFlow(P_st, x)
8:         end if
9:     end while
10: end procedure
```

$A(\bar{x}) = A^+(\bar{x}) \bigcup A^-(\bar{x})$,
$A^+(\bar{x}) = (i,j) : (i,j) \in A, x\bar{}_{ij} < u_{ij}$,
$A^-(\bar{x}) = (i,j) : (j,i) \in A, x\bar{}_{ij} > 0$.

4.1.1.5 Complexity Analysis

Now let us analyze the complexity of the negative cycle elimination algorithm which was performed without adopting any specific rule for the choice of the negative cycle to be removed, in case there are more than one alternative. We suppose that all costs and capacities of network G are non-negative integers and we call $Cmax$ and $Umax$ the maximum cost and the maximum capacity of the graph arcs, respectively. Since the choice of the initial feasible solution is not in the least dictated by cost criteria, at the beginning in the worst case the cost of the provided solution is given at most by $mCmaxUmax$. On the basis of a similar reasoning, we may suppose as well that the optimal solution in the extreme case has cost 0. Observing that at each iteration of the algorithm the flow varies by at least one unit and that this induces a decrease in the value of the objective function by at least one unit, we will have to perform, in the worst case, $O(mCmaxUmax)$ iterations. Knowing that negative cycles can be recognized in $O(nm)$, the algorithm's complexity is $O(nm^2CmaxUmax)$. It can be demonstrated that, in case of presence of multiple negative cycle, if the algorithm chooses the cycle with the minimum average (between the sum of cost cycle and the number of the node) the algorithm complexity is polynomial, in particular the complexity is $O(nmlog(n)log(nC))$.

4.1.1.6 Our implementation

The graph is a *complete bipartite graph* where the starting node s is linked with all the nodes i that represented the zone with a surplus of taxi. The final node t is reachable from all the nodes j that have a lack of taxi.

All the zones i that have a surplus of taxis are a node with $b_i = -(a_i - T(r_i))$

All the zones$j$ that have a deficit of taxis are a node with $b_j = r_j - a_j$

The capacity of each arcs (s,i) is equal at the difference between the *Available* taxi in

the zone and the taxi request in the zone $a_i - T(r_i)$, and the cost is equal to zero.

The capacity of each arcs (i, j) is equal to the capacity of (s, i), and the cost is proportional at the distance between i and j.

The capacity of each arcs (j, t) is equal to $r_j - a_j$, and the cost is equal to zero.

An example is illustrated in the document **minimumCostFlowExample.pdf**

4.1.2 Other possible solution

Observing the previous solution we notice that the algorithm doesn't work very well in some cases, for example consider the next graph

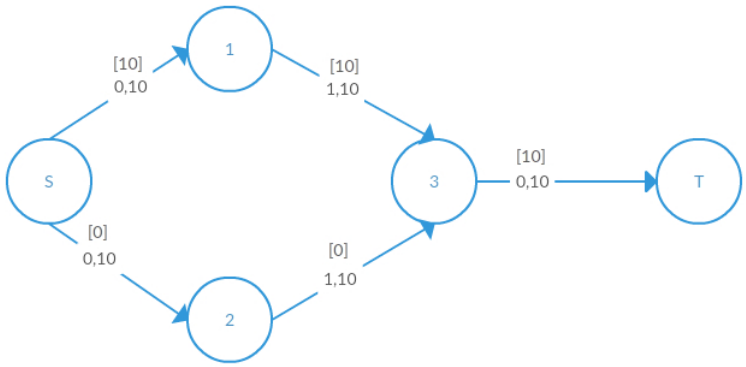


Figure 4.1: The new graph

We notice that this is a correct solution of the minimum cost flow (in this case correspond with the maximum cost flow) but using a common sense we can say that isn't the optimal solution for the real world. In fact, a better way to solve this situation is transport five taxis from node i={1,2} to the node $j = \{3\}$, in this way we can conserve some taxis in the zone for the next reservations.

However the algorithm suitable to solve the problem with that constrains is reasonably exponentially in complexity, therefore it is not convenient to be used.

4.2 Queue Manager

Algorithm 3 Select an Available taxi from the queue

```
1: procedure SelectTaxiForARequest(Z, z, r)
2:     S ← z                                          ▷ S is a Set of Zone
3:     Z ← Z − S
4:     while a_S is empty Z is not empty do
5:         S ← adjacent(S)
6:         Z ← Z − S
7:     end while
8:     if a_S is empty then
9:         add Request in the queue of request
10:    else
11:        select the first t taxi from the queue a_S
12:        remove t from a_S
13:        SendRequest(r,t)
14:    end if
15: end procedure
```

4.3 Reservation Handler

Algorithm 4 Update the number of available Taxi in the city

```
1: procedure updateTaxiAvailableNumber(oldStatus, newStatus)
2:     if newStatus == Available && oldStatus != Transition then
3:         N = N + 1
4:     else if oldStatus == Available && newStatus != Transition  then
5:         N = N - 1
6:     end if
7:     t.zone() = z
8:     sendCommunication(t)
9: end procedure
```

Algorithm 5 Count the number of Answer

```
1: procedure countNumberOfAnswer(t, r)
2:     if t has already answer to request r then
3:         t.accept(r)
4:         sendInfo(r,t)
5:     else
6:         sendRequest(r,t)
7:     end if
8: end procedure
```

Algorithm 6 Answer Time Out

```
1: procedure TimeOut(t, r)
2:     startTimeOut()
3:     if t no Answer before 1 minute then
4:         t.reject(r)
5:     else
6:         sendInfo(r,t)
7:     end if
8: end procedure
```

4.4 State Manager

Algorithm 7 Change Status in Available

```
1: procedure changeStatusInAvailable(t)
2:     t.setStatus(Available)
3:     insertTaxiInQueue(t,t.getZone())
4:     updateTaxiAvailableNumber(t.oldStatus(),t.status())
5: end procedure
```

Algorithm 8 Change Status in Busy

```
1: procedure changeStatusInBusy(t, N)
2:     if t.status == Available || t.status == Transition then
3:         t.setStatus(Busy)
4:         deleteTaxiFromAQueue(t,t.getZone())
5:         updateTaxiAvailableNumber(t.oldStatus(),t.status())
6:     else
7:         textbfReturnError
8:     end if
9: end procedure
```

Algorithm 9 Change Status in Emergency

```
1: procedure changeStatusInEmergency(t)
2:     if t.getConfirmedRequest() is not empty then
3:         deleteRequest(r)
4:         sendRequest(r,r.getZone())
5:     end if
6:     deleteTaxiFromAQueue(t,t.getZone())
7:     t.setStatus(Emergency)
8:     updateTaxiAvailableNumber(t.oldStatus(),t.status())
9: end procedure
```

Chapter 5

User Interface Design

Into this paragraph we want to discuss about the user experience [UX] that our system offer to our user that can be both taxi driver or client. We decide to use the UX diagram despite mockup because we can show the experience of all the user on a better way. We use some stereotypes to identify what each class must represent on our web site. The stereotypes are:

- «screen» It represent the html page of our site.

- «screen compartment» With this stereotype we represent a component of a normal page.

- «input form» This stereotype represent a field that can be filled by the user and can be submitted by the click of button.

- The classes that haven't the stereotype represent the information that are show through the page or the component of the site.

Model::Main

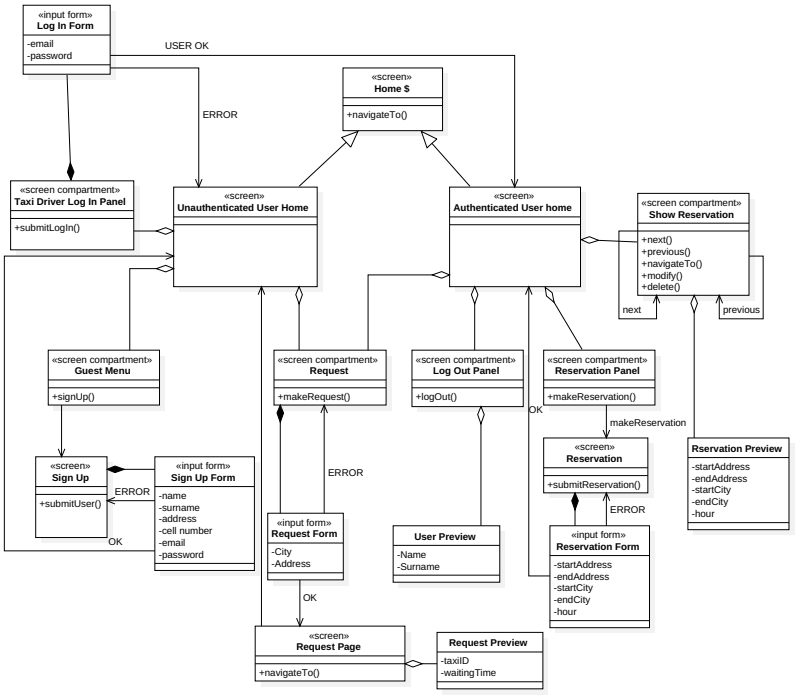


Figure 5.1: Passenger UX

Model1::ClassDiagram1

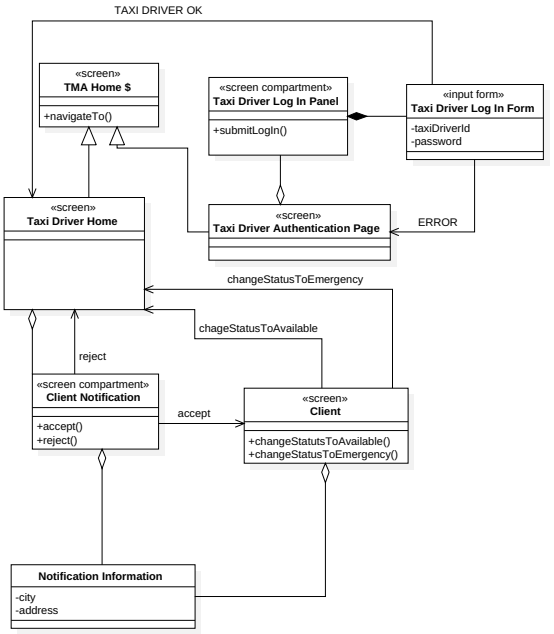


Figure 5.2: Taxi Driver UX

Chapter 6

Requirements Traceability

In this chapter are described using a contingency table the relationship between the RASD requirements and the component of the DD. The table explained which components perform a requirements. The components acronyms are listed below. For the requirements refer to the RASD.

- Queue Manger (QM)
- Taxi Distribution Manger (TDM)
- Status Manger in the Application Logic (SM)
- Request Handler (RH)
- Account Manger (AM)
- Data Base Manger (DBM)
- Data Checker (DC)
- XML Pages Creator (XPC)
- Web Pages Request Unit (WPR)
- Sign Up (UP)
- Sign In Passenger Mobile Application (IN)
- Request Unit (RQ)
- Reservation Manger Passenger Mobile Application (RMP)
- Request Answer (RA)
- Status Manager Taxi Mobile Application (STM)
- Sign In Taxi Mobile Application (INT)
- GPS for PA and TMA

	QM	TDM	SM	RH	AM	DBM	DC	XPC	WPR	UP	IN	RQ	RMP	RA	STM	DS	INT	GPS
1					X	X	X		X	X						X		
2					X	X	X		X		X					X		
3					X	X	X									X	X	
4		X	X					X										
5				X										X				
6	X		X	X														
7				X		X	X				X		X					
8				X														
9			X															
10			X															
11			X	X			X								X		X	
12																		X
13	X		X															
14	X	X																
15								X										
16			X	X														
17				X														

Table 6.1: Requirements/Components table