myTaxiService

-

Design Document

Davide Cremona (matr. 852365), Simone Deola (matr. 788181)

December 2, 2015

Contents

Chapter 1

Introduction

This chapter is intended to give an overall description of this document, it contains various sections like:

- **Purpose:** in this section is described the purpose of this document.
- **Scope:** in this section is described the scope of the myTaxiService system.
- **Definitions, Acronyms, Abbreviations:** here are listed all the definitions, all the acronyms and all the abbreviations that the reader will encounter in this document.
- **Reference Documents:** here are listed all the documents that the reader may need to read to better understand what is written in this document.
- **Document Structure:** here is explained the internal structure of this document, giving a fast description of each chapter.

1.1 Purpose

The MyTaxi Service Design Document is intended to provide an explanation of how the system has been designed. It's also destined to give to the software development team an overall guidance to the implementation of the architecture of the software project. These goals are chieved by describing:

- The system architecture;
- Architecture hig-level components;
- Interaction between components;
- Patterns and Styles used to design these components.

1.2 Scope

The scope of myTaxiService is to simplify the interaction between Taxi Drivers and Customers. The main goal is to provide a system to request a taxi and ensure that this will arrive in a small amount of time. Also, customers can reserve taxis for a future ride. The system also ensures a fair management of taxi queues that are divided in city zones in orded to reduce waiting times and to provide a fair division of work between Taxi Drivers.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Customer: a Customer is the end-user (a taxi customer) that makes request or reservations to use taxis.
- Taxi Driver: a Taxi Driver is a driver of a taxi. He can receive requests and accept or decline them.

1.3.2 Acronyms

- RASD (or R.A.S.D.): is the Requirement Analysis and Specification Document.
- DD (or D.D.): is the Design Document (this document).
- MVC (or M.V.C.): is the Model-View-Controller software design pattern.
- DBMS (or D.B.M.S.): it's the Data Base Management System.
- UI (or U.I.): it's the acronym for User Interface.

1.3.3 Abbreviations

Definitions Acronyms Abbreviations section

1.4 Reference Documents

You can refer to the Requirement Analysis and Specification Document for a better understanding of what is described in this document (myTaxiService R.A.S.D. Link).

1.5 Document Structure

This document is divided in 6 main Chapters:

- Introduction: this chapter is used to give a general overview of this document (purpose, references etc..).

- Architectural Design: in this chapter is described the general architecture of the system and his components. It's also described how the components interacts with each other and the Design Patterns and Architectural Styles used to design them.
- Algorithm Design: here is described the most relevants algorithms used to create the system.
- User Interface Design: here is described the end-user interface of the mobile and web applications.
- Requirements Traceability: here is described how the requirements described in the RASD document are implemented in the various components of the system.
- References: in this chapter, there are useful references to external resources that helps to understand this document.

Chapter 2

Architectural Design

In this chapter there will be described the architecture of the system and the various components that will compose the system. Here it's also described how these components will interact to each other to perform the various tasks that the system have to do to provide to the users the functions described in the RASD document. Finally, there will be described what architectural styles and patterns have been used to design the entire system.

2.1 Overview

MyTaxiService system architecture has been designed as a 3-tier architecture (Data Tier, Logic Tier and Presentation Tier) to facilitate the development of the system using the MVC software design pattern. In fact the three tier represents in some way the Model (Data Tier), the Controller (Logic Tier) and the View (Presentation Tier) of the software.
The architecture of the system is described by the following figure:

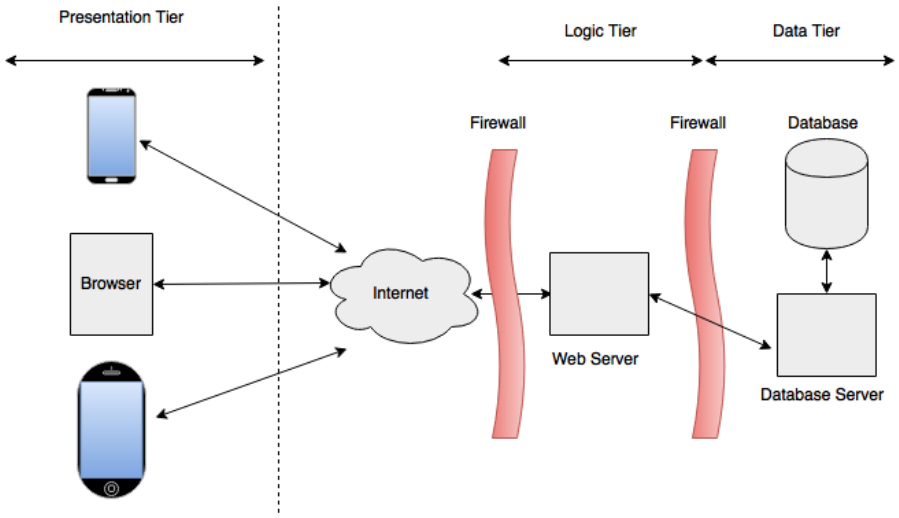

Figure 2.1: System Architecture

The components of the system are divided in the three tiers:

- In the Presentation Tier there are imoplemented classes and the objects that compose the view part (what the user see). From this tier users can perform actions to query the system and use the web and mobile applications.
- The Logic Tier is the place where are implemented classes and objects that compose the controller (actual logic of the system). This part is in charge of taking requests by the users applciations and perform changes on the data. This part is also in charge to notify the applications (presentation tier) of the data changes.
- The Data Tier is the place where are implemented the classes and the object that interacts directly with the database. These objects represents the entities of the database and are able to perform creation, modification and deletion of database records.

2.2 High Level Components and Their Interaction

In this section each Tier is decomposed in his high level components. For each component is given a short description of his purpose and how the component interacts with the others components.

2.2.1 Data Tier

Ther Data Tier is the part of the system that include the database and all the classes and the objects that will interact with it. It's composed by these components:

- **Database:** it's the data structure where all the system data are stored. The management of this component is delegated to the DBMS. The DBMS will provide a query language to extract data from the database.

- **Database Manager:** this component represents the Model of the MVC design pattern, it will include all the objects and the classes of the Data Model. This component also contains all the classes that are in charge to perform queries to the DBMS in order to make the data available to the Logic Tier.

2.2.2 Logic Tier

In this Tier there are implemented all the classes and the objects that are used to logically organize the requests and the responses from/to the Presentation Tier and the Data Tier. This Tier represents the Controller of the MVC software design pattern and it's composed by these components:

- **IO Manager:** This component is in charge to collect all the requests from the myTaxiService clients and to forward them to the correct component of the system by calling other components functions. It's also in charge to collect messages (like notifications, error messages...) from the internal components and to forward them to the correct user.

- **Requests an Reservations Manager:** This component receives from the IO Manager component requests that concerns requests for taxis and for reserve a future taxi ride. It's in charge of check the validity of the data that receives and, if it's the case, to call functions of the Database Manager to insert these requests into the database.
  Another task of this component is to continuously check if there are some requests or reservations that are unhandled in the database and to allocate a taxi that has to be selected using the Zones Manager. Once a taxi is allocated, then a notification to the Taxi Driver is sent through the IO Manager.

- **Zones Manager:** This component is in charge to manage the zones of the cities. It receives GPS updates of Taxi Drivers from the IO Manager and compute them to update the zone queues of the relative city. This component also orders Taxi Drivers queues in function of their priority and provide some useful functions (for instance: get the first available Taxi Driver).

- **Profile Manager:** This component receives requests by the IO Manager that concerns the visualization or the modification of the users profiles. It checks if the modifications can take place and then calls Database Manager functions to modify the database entries. Once a modification is performed, then this components will send a notification to the user by calling the right IO Manager function.
  Another task of this component is to provide registration and login functions to the users interacting with the Database Manager.

2.2.3 Presentation Tier

The purpose of this tier is to make the data and their changes visible to the user. Presentation Tier is also in charge to make the system functions available to the users according to their permissions (see the RASD to read about functionality differences between Customers and Taxi Drivers). This Tier represents the View of the MVC design pattern and it's composed by these components:

- **Web Application UI:** this component contain all the objects that the users need to comunicate with the system, via web browser.
- **Mobile Application UI:** it contains all the necessary components to comunicate with the system via mobile devices.

2.2.4 Schema

To give a visual description of what is written so far it's provided a schema of the system:

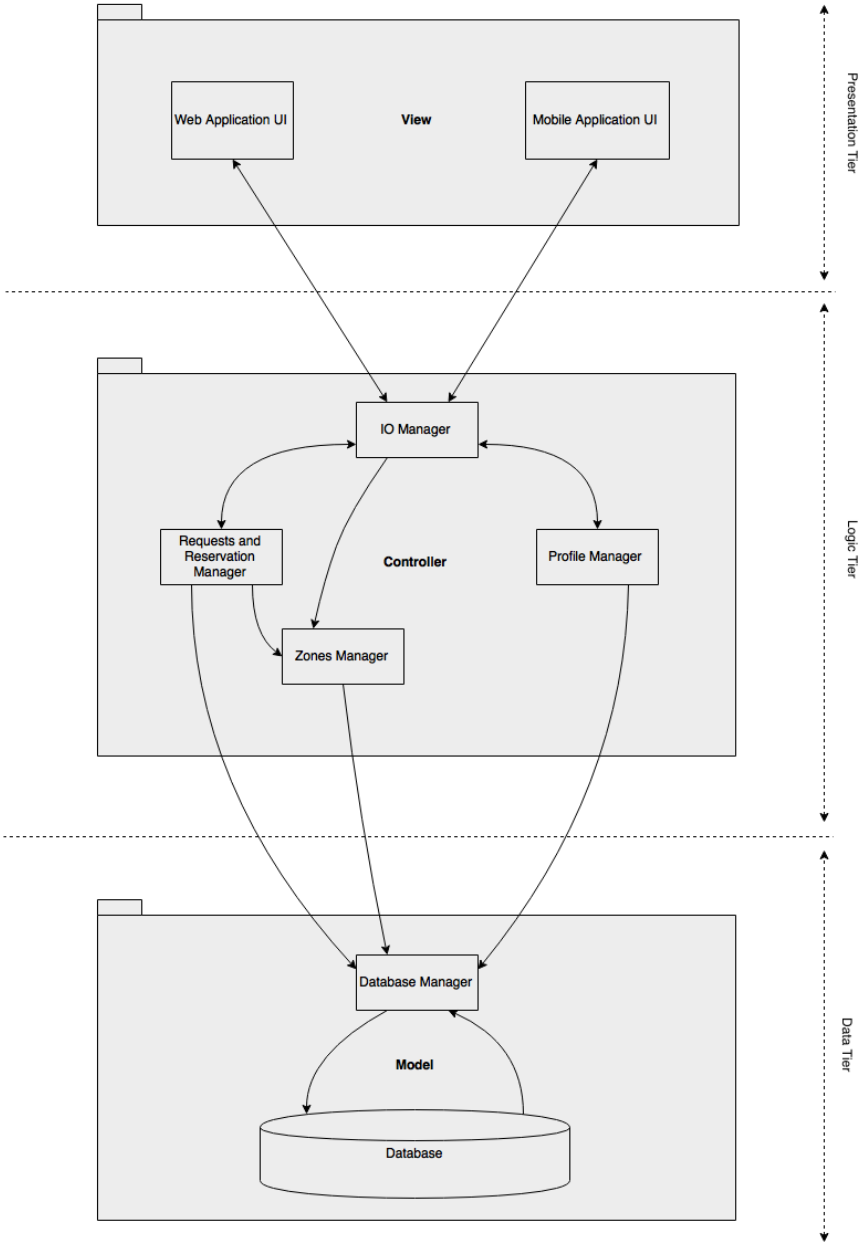

Figure 2.2: High Level Components and Interactions

2.3 Component View

"Per ogni componente individuato nella sezione precedente, definiamo le classi principali e le descriviamo."

On this section we take the component described in the previous section and give a more detailed description. For each component we describe the main classes to give a general overview of how the system is going to be implemented.

2.3.1 Data Tier

This part represent the model of the system (using a MVC architecture), so the class we use in this section was the representation of the information that must be stored to use the system.

Database

the database is the data structure we don't specify the classes used by this structure, all the information we store are managed by the Database Manager.

Database Manager

The main classes used to represent the stored data of our application are:

- Registered User: this class contains the information about the registered user.
- Taxi Driver: this class extends the Registered User class and represent the information about the taxi drivers
- Customer: this class extends the Registered User class and represent the information about the customers
- Zone: this class represent the information about the zones. On this class we store also the queue of the taxi driver that are available on this specific zone.
- Ride: this class contain the general information that Ride and Request share. This abstract class follows the pattern State to represent the state of a ride. All Ride start in the state NOT_HANDLED, then can pass in state HANDLED, then from this state can pass in the state NOT_HANDLED.
- Request: this class extends Ride and represent the information of the requests for a ride.
- Reservation: this class extends Ride by adding information about the starting and the end position, and the time of the meeting. Adding this information we can represent the reservation in our system.

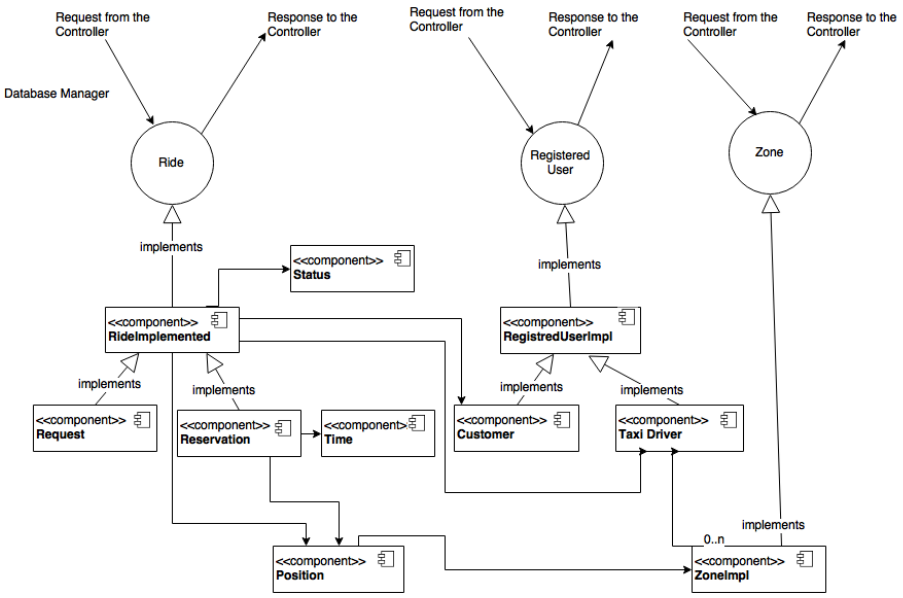

Figure 2.3: Data Manager Structure

2.3.2 Logic Tier

Here are described in short the various sub-components that are included in the Logic Tier (Model package).

IO Manager

This component will be implemented using the Command Pattern (see Command Pattern on Wikipedia). It will contain the following main classes, in general for each functionality of the UI there is a Message class that, for the sake of brevity, are not all listed:

- **Message Type** (Interface) **:** this sub-component is the Interface which is extended by each Message class.
- **Message Types:** this sub-component is an enumeration in which each enumeration entry returns an implementation of the Message Type Interface.
- **Message Types Map:** this sub-component is an Hash Map that associates the name of the elements of the enumeration, the enumeration element identified by that name.
- **Messages Handler:** this sub-component receives messages from the View classes. It selects the right message to send by processing the message name and selecting (through the Message Type Map) the right Message Class.

- **Request Message:** This sub-component is in charge to call Requests and Reservations functions to add a Taxi Request in the myTaxiService database.
- **Reservation Message:** This sub-component is in charge to call Requests and Reservations functions to add a Taxi Reservation in the myTaxiService database.
- **Customer Notification Message:** This sub-component is used to send notifications to a Customer. These notifications are used mainly to confirm to a Customer that his request has been handled.
- **TaxiDriver Notification Message:** This sub-component is used to send notifications to a Taxi Driver. These notifications are mainly used to advise a Taxi Driver that he's been selected for a certain ride request.
- **Login Message:** This sub-component is in charge to forward login requests from users to the Profile Manager.
- **Registration Message:** This sub-component is in charge to forward registration requests from users to the Profile Manager.
- **Ok Message:** This sub-component is used to send success notifications to a specific user.
- **Error Message:** This sub-component is used to send error notifications to a specific user.
- **End Ride Message:** This sub-component is in charge to call the functions of the Requests and Reservations Manager that can set as ended a ride. An End Ride Message is sent by the client applications to signal that a ride is ended.
- **Profile Modification Message:** This sub-component is in charge to call the functions of the Profile Manager that can modify a user profile.

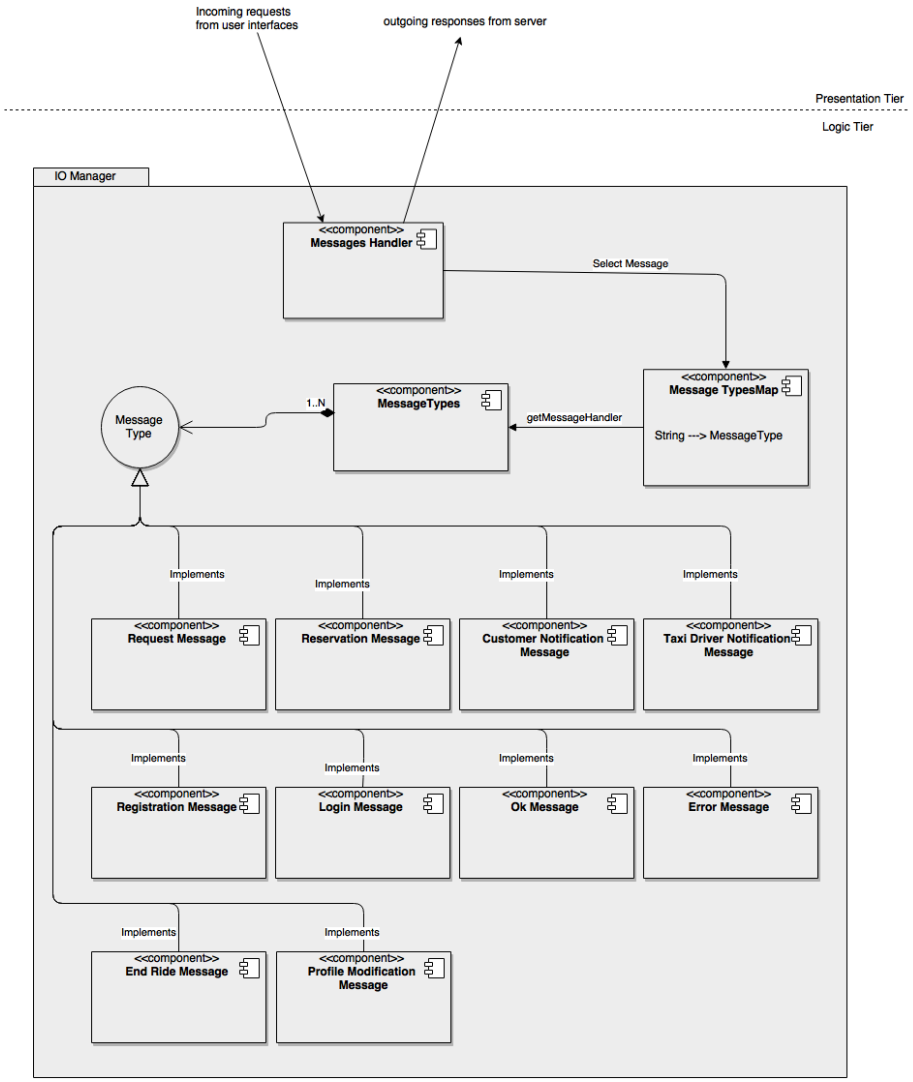

Figure 2.4: IO Manager Structure

Requests and Reservations Manager

This component contains all the classes that are in charge to add/remove ride requests and ride reservations. In this component are also contained classes that check the database to find unhandled requests or reservations that has to be handled. Here are contained these main classes:

- **Taxi Request Adder:** This sub-component is in charge to create the data for a new request entry in the database and send them to the Data Tier calling the Database Manager function that add a new Ride in the Database. These data are Customer ID, Customer Position, Customer Zone, Request Time and Date.

- **Taxi Reservation Manager:** This sub-component is in charge to create the data for a new taxi reservation entry in the database and send them to the Data Tier calling the Database Manager function that add a new Reservation in the Database. These data are Customer ID, starting position, ending position, request time and date.
  This sub-component is also in charge to send data about the Reservations to the Customers and to delete one Reservation if it can be done.

- **Taxi Allocator:** This sub-component is in charge to check if there are reservations or ride requests that are not been handled yet and handle them. This means that Taxi Allocator have to communicate with Zones Manager (to get the first available Taxi Driver), with the IO Manager (to notify Customers and Taxi Drivers) and with the Database Manager (to get reservations and requests to allocate).

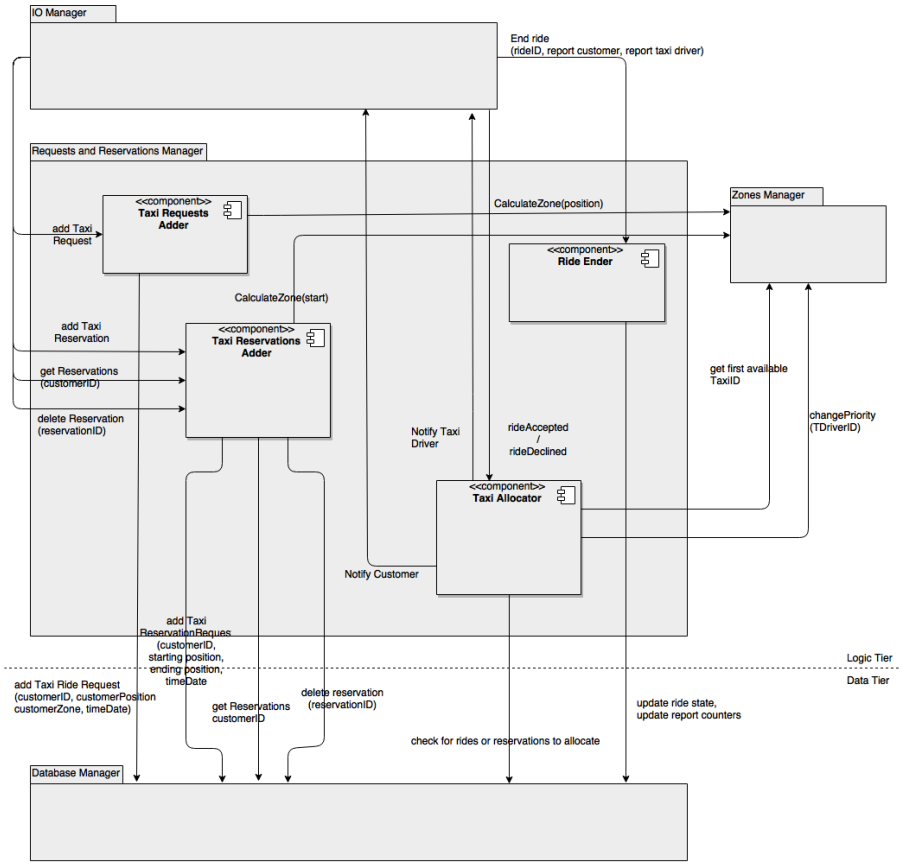

Figure 2.5: Requests And Reservations Manager Structure

Zones Manager

This component contains all the sub-components needed to manage zones and taxi queues. It provide some functions to get avaialables Taxi Drivers and to notify that a Taxi Driver has been chosen for a ride.

- **Zone Calculator:** The only function of this sub-component is to return the zone associated to a certain position.
- **Queues Manager:** This sub-component is in charge to keep the queues updated and to provide functions to select Taxi Driver from them.
- **GPS Updater:** This sub-component is in charge to receive GPS updates from IO Manager and to ask to the Database Manager to update them.

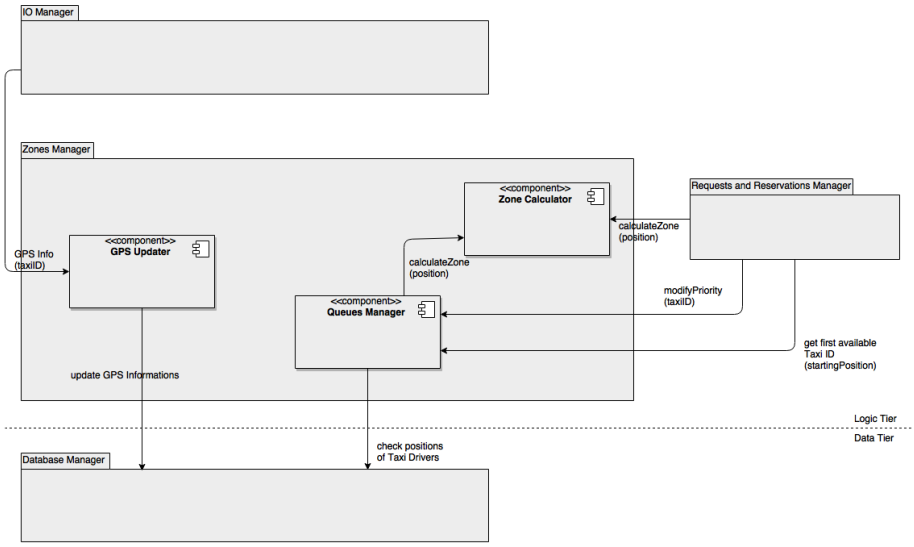

Figure 2.6: Zones Manager Structure

Profile Manager

This component have to check if a registration or a login request is legal or not and if it is, then the relative function of the Database Manager has to be called.

- **Registration Manager:** This sub-component is in charge to accept registration requests from IO Manager and, check the credentials using the Credential Checker and then if it's the case register a new user using the functions provided by the Database Manager.
- **Login Manager:** This sub-component is in charge to accept login requests from IO Manager, check the credentials using the Credential Checker and then if it's the case set a user as logged in using the functions provided by the Database Manager.
- **Profile Modifier:** This sub-component is in charge to accept profile modification requests from the IO Manager, check the new credentials and, if it's the case, change the profile credentials using the functions provided by the Database Manager.
- **Profile Viewer:** This sub-component is in charge to provide to the IO Manager information about an user.
- **Credential Checker:** This sub-component is in charge to check the credentials provided by the others sub-components. Here are implemented all the rules about the acceptance of users credentials (like password composition policy etc...).

- **Profile Security:** This sub-component is in charge to retrieve a password if the customer asks for it. It checks if the email is correct using the Credential Checker and also checks if the email is registered in the system database. Then, this sub-component, sends an email to the user with his password.

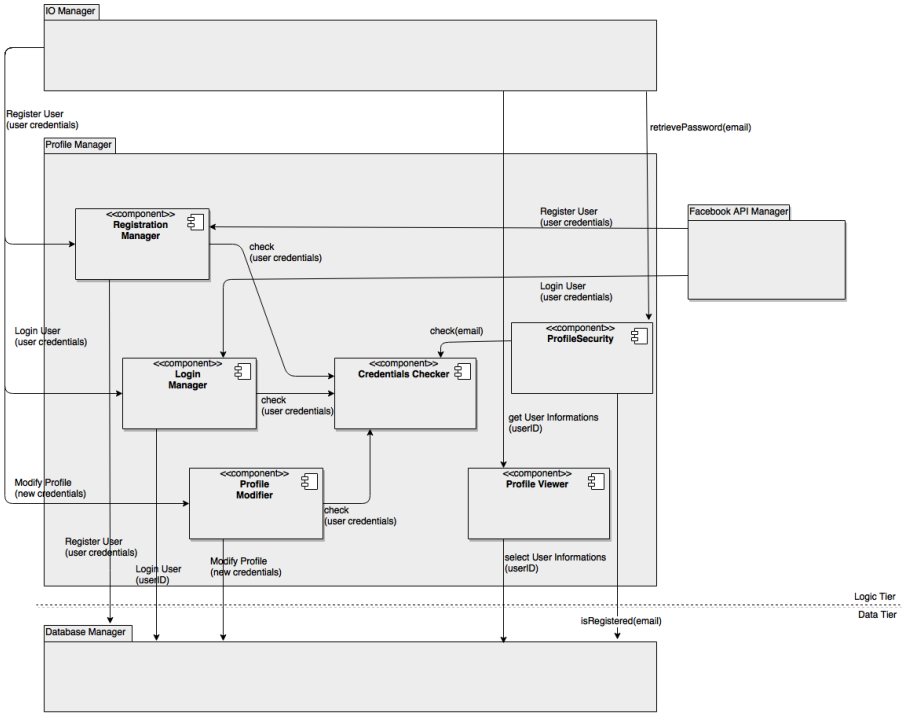

Figure 2.7: Profile Manager Structure

2.3.3 Presentation Tier

On this tier the two main components that we find are the Web Browser User Interface and the Application User Interface. This two components are different for programming language and it will run on different devices. But, this two component, must comunicate with the controller via the same set of class that must send and read data in the same way. The idea was that the view of the UI can change to be optimized for the device, but the communication with the Logic tier must be the same. So each UI have two kind of classes, the first kind of classes are the classes of visualization that must follow the organization shown on the mock object of the RASD. For example on the RASD we describe all the pages and all the input form of the two kind of application. The second class of classes must contain a class that can sent message to the Logic Tier and one that can receive its responses. We call the first class CommandSender and the second one ResponseReciver.

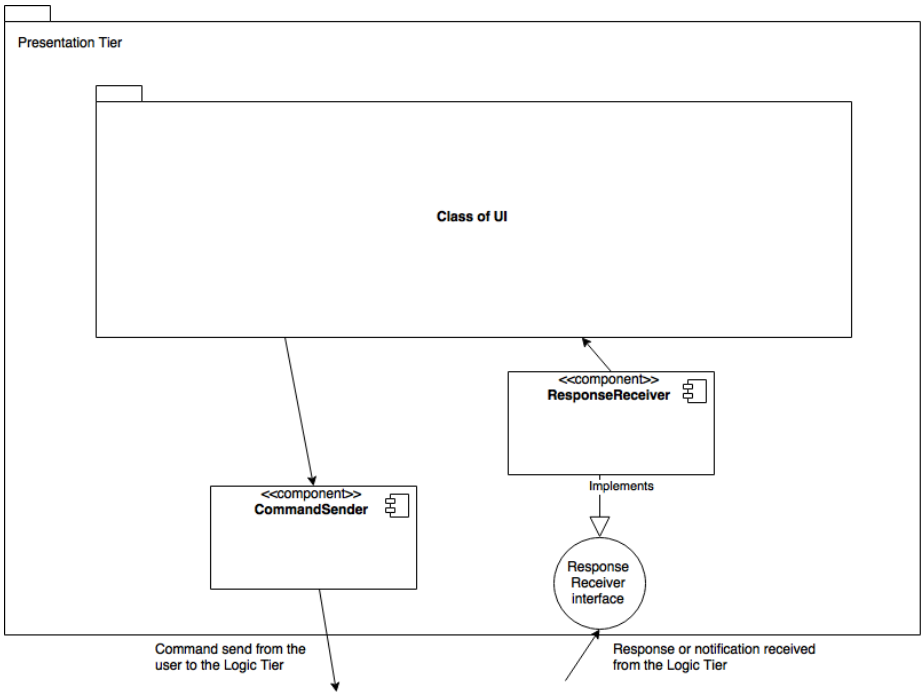

Figure 2.8: Presentation Tier Structure

2.4 Deployment View

descrizione

The main hardware devices involved in this system are the following:

- Web Server: on this server we have the logic tier of our application, this tier contain the logical part of the system, so this devices must have the higher computation power between the devices. It offer via HTTP the connection from the user, and access to the database via XMPP (communication protocol based on XML typical used for inter-server communication).

- Data Base Server: on this server we have the data tier of our application. This device must store on its memory all the data of our system. So, to avoid problem, the memory of this device must be suitable for store all the information. This device must be connected to the web server for the purpose of make available to the logic tier the information that the data tier store. the communication between this two takes place in XMPP.

- User Devices: this kind of devices are divided into Mobile Devices and Computers. On this kind of devices we have the presentation tier of our application. The kind of component we use on this devices depends on the type of devices

and the OS installed (as we have described on the chapter Software Limitation in the RASD) . We can't know the hardware specification of this devices so the presentation tier must be lighter as possible. the communication between this devices and the Web Server takes place in HTTP between the internet.

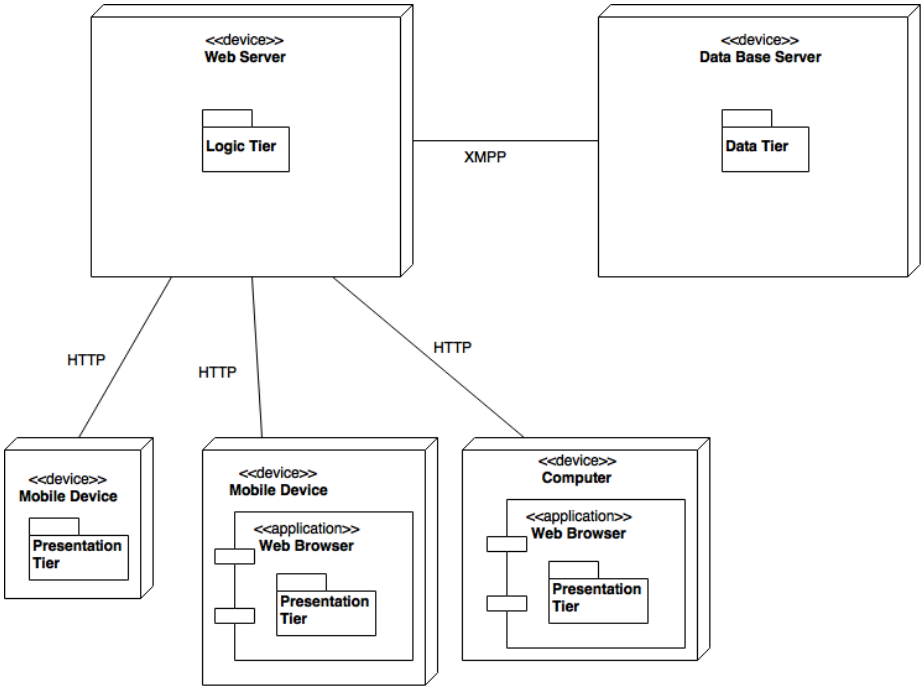

Figure 2.9: Deployment Diagram

2.5 Runtime View

In this section is described the role of the components for each scenario of the RASD (Section 3.3 of RASD)

2.5.1 Registration Through the Creation of a New myTaxiService Account

Runtime View relative to the Scenario S.1 of the RASD.

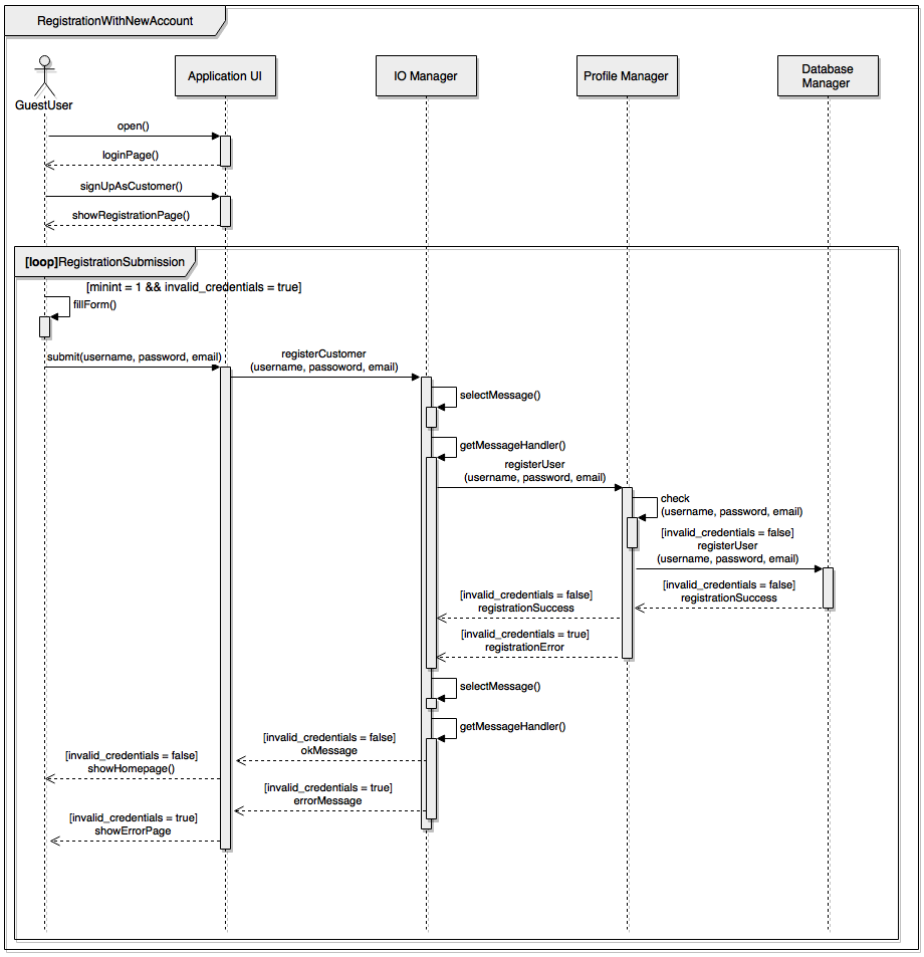

Figure 2.10: Registration with New Account Runtime View

2.5.2 Registration Using Facebook Account

Runtime View relative to the Scenario S.2 of the RASD.

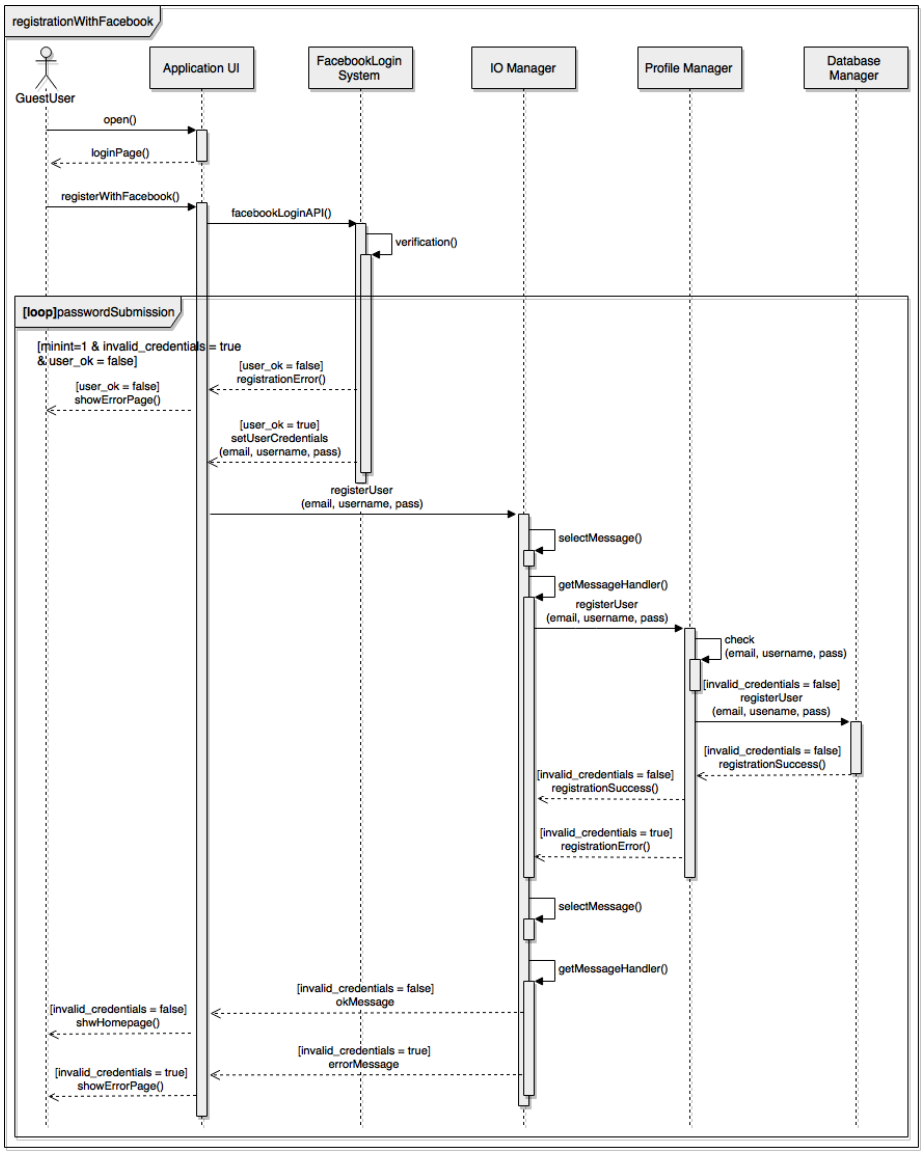

Figure 2.11: Registration with Facebook Account Runtime View

2.5.3 Taxi Driver Registration

Runtime View relative to the Scenario S.3 of the RASD.
The registration of a Taxi Driver is quite similar to the registration of a Customer. For the sake of brevity, this kind of registration is not modeled with a Sequence Diagram because it's sufficient to take the "Registration Through The Creation Of A New myTaxiService Account" and substitute the "registerUser(username, password,

email)" call with a "taxiDriverRegistration(username, password, email, license, taxiNumber)" call.

2.5.4 Registered User Login

Runtime View relative to the Scenario S.4 of the RASD.

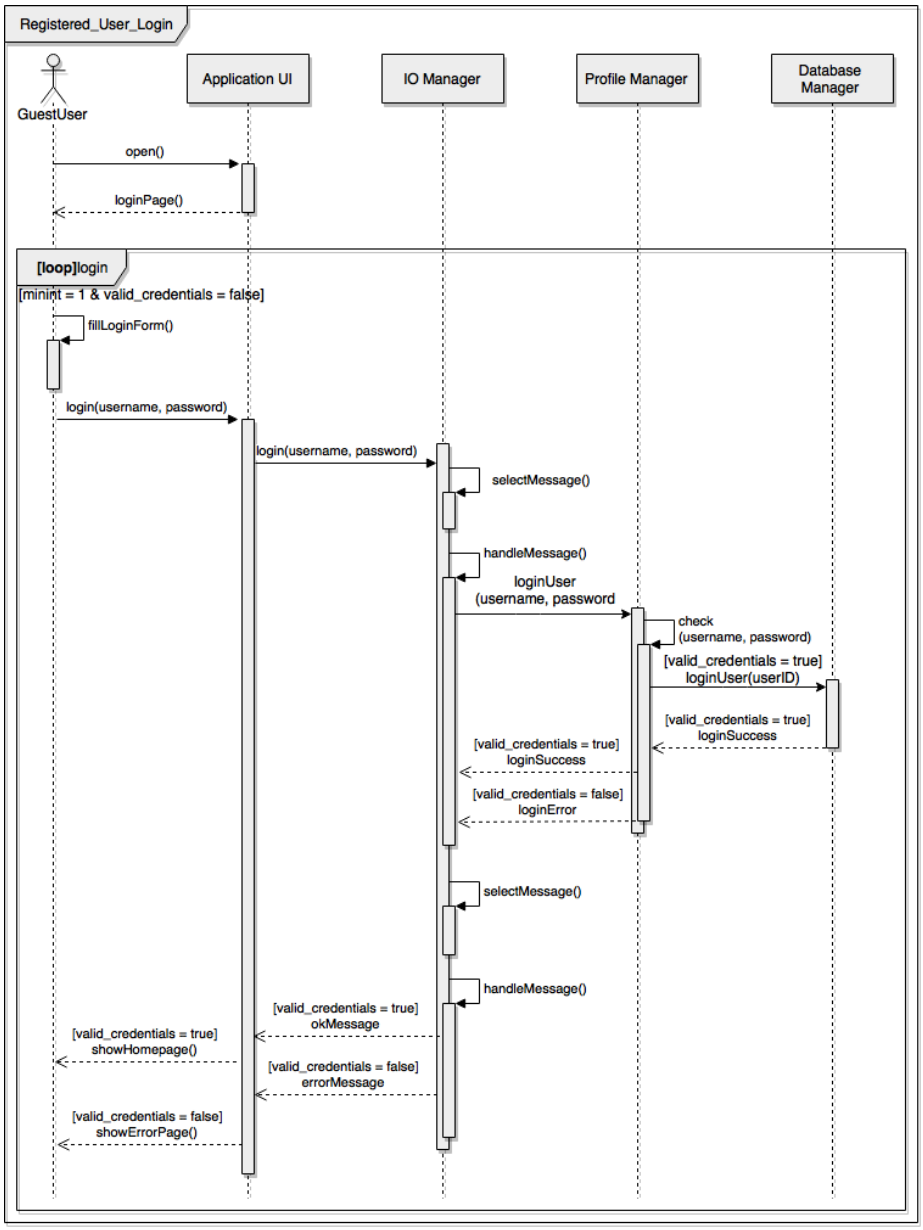

Figure 2.12: Registered User Login Runtime View

2.5.5 Customer Facebook Login

Runtime View relative to the Scenario S.5 of the RASD.

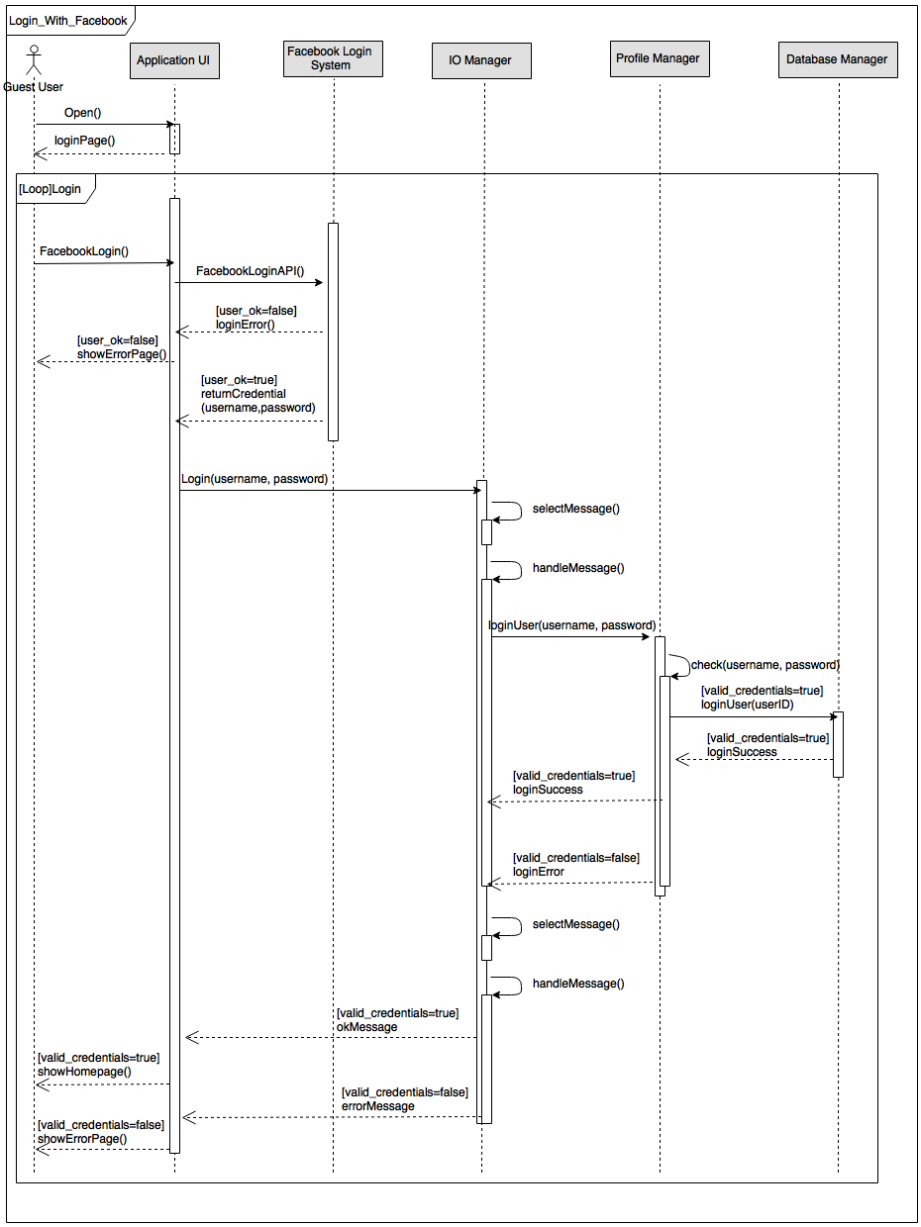

Figure 2.13: Registered User Facebook Login Runtime View

2.5.6 Profile Modification

Runtime View relative to the Scenario S.6 of the RASD.

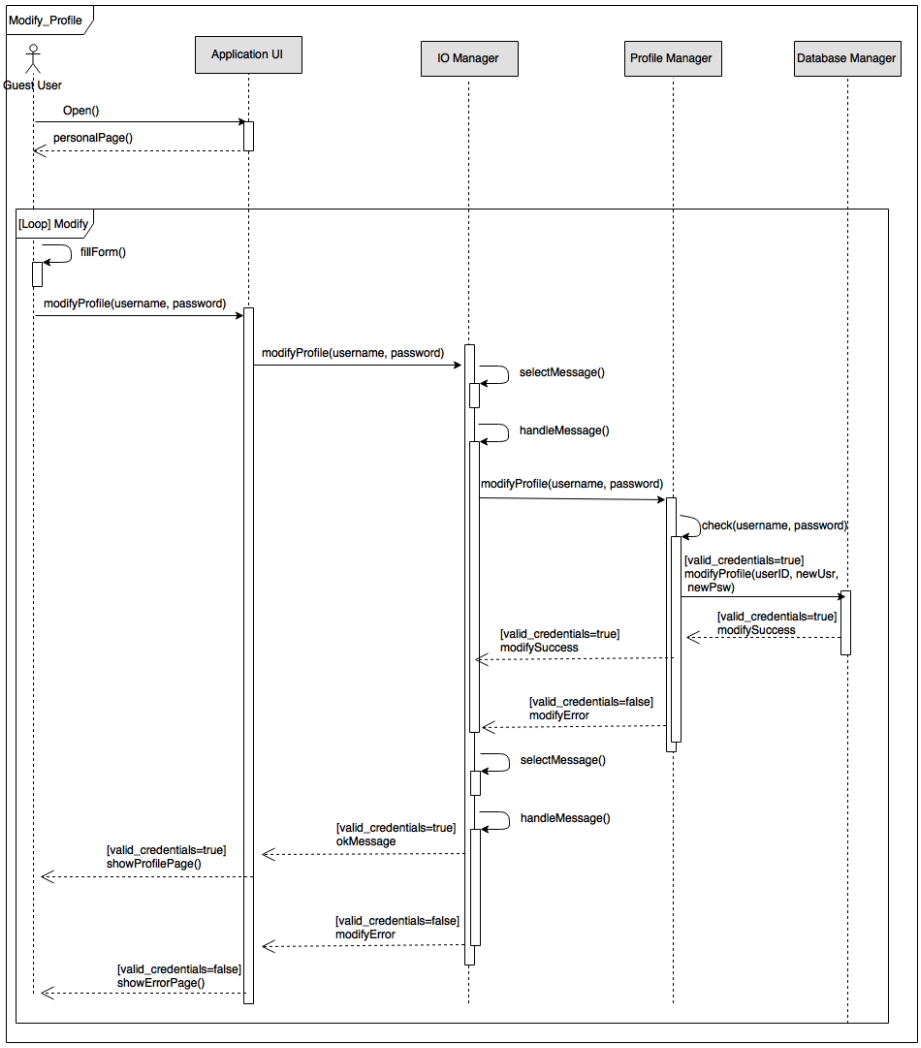

Figure 2.14: Modify Personal Profile Runtime View

2.5.7 Password Retreival

Runtime View relative to the Scenario S.7 of the RASD.

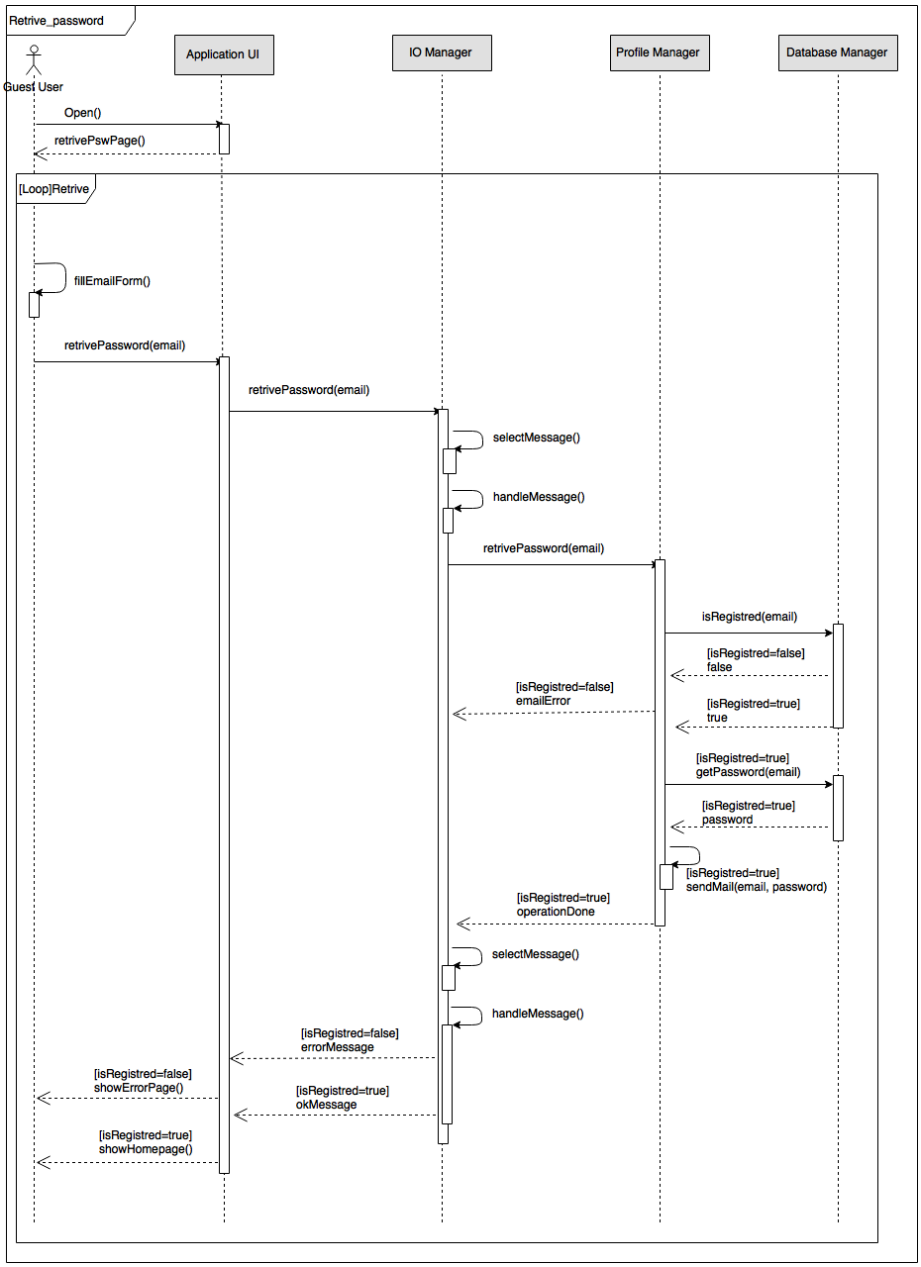

Figure 2.15: Password Retriving Runtime View

2.5.8 Taxi Driver Reporting

Runtime View relative to the Scenario S.8 of the RASD.

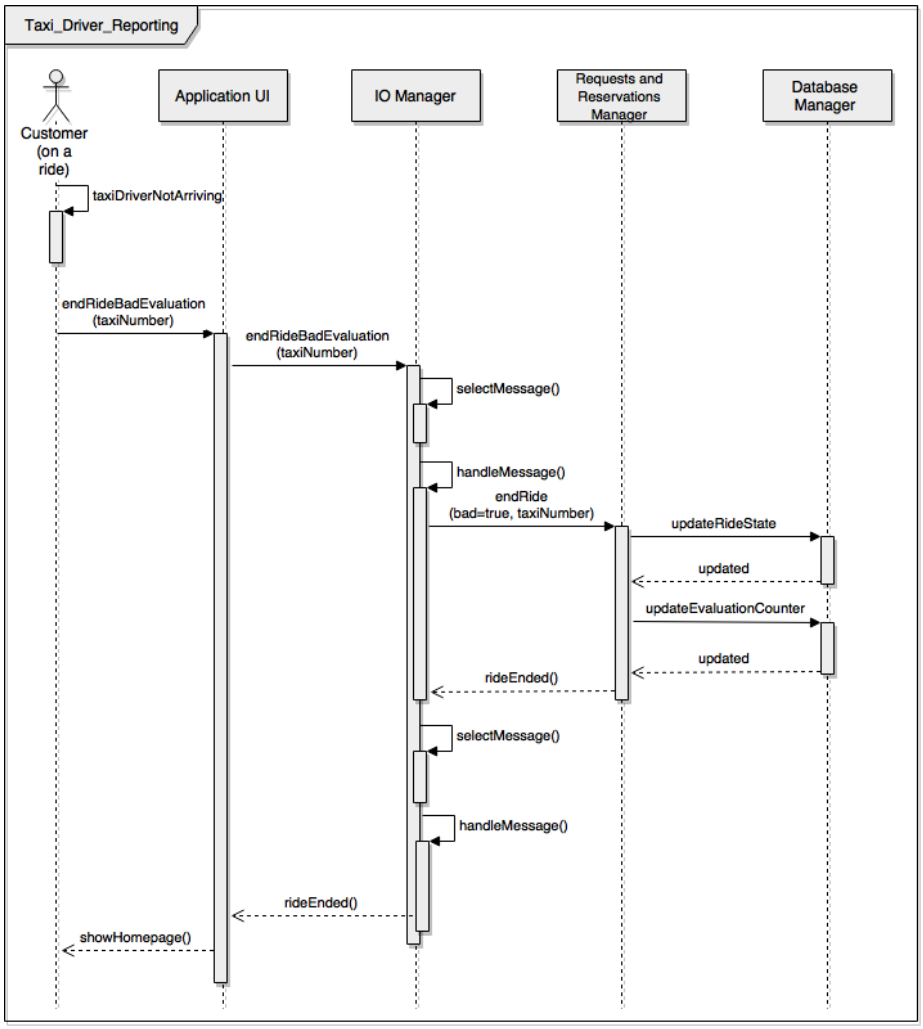

Figure 2.16: Taxi Driver Reporting Runtime View

2.5.9 Customer Reporting

Runtime View relative to the Scenario S.8 of the RASD.

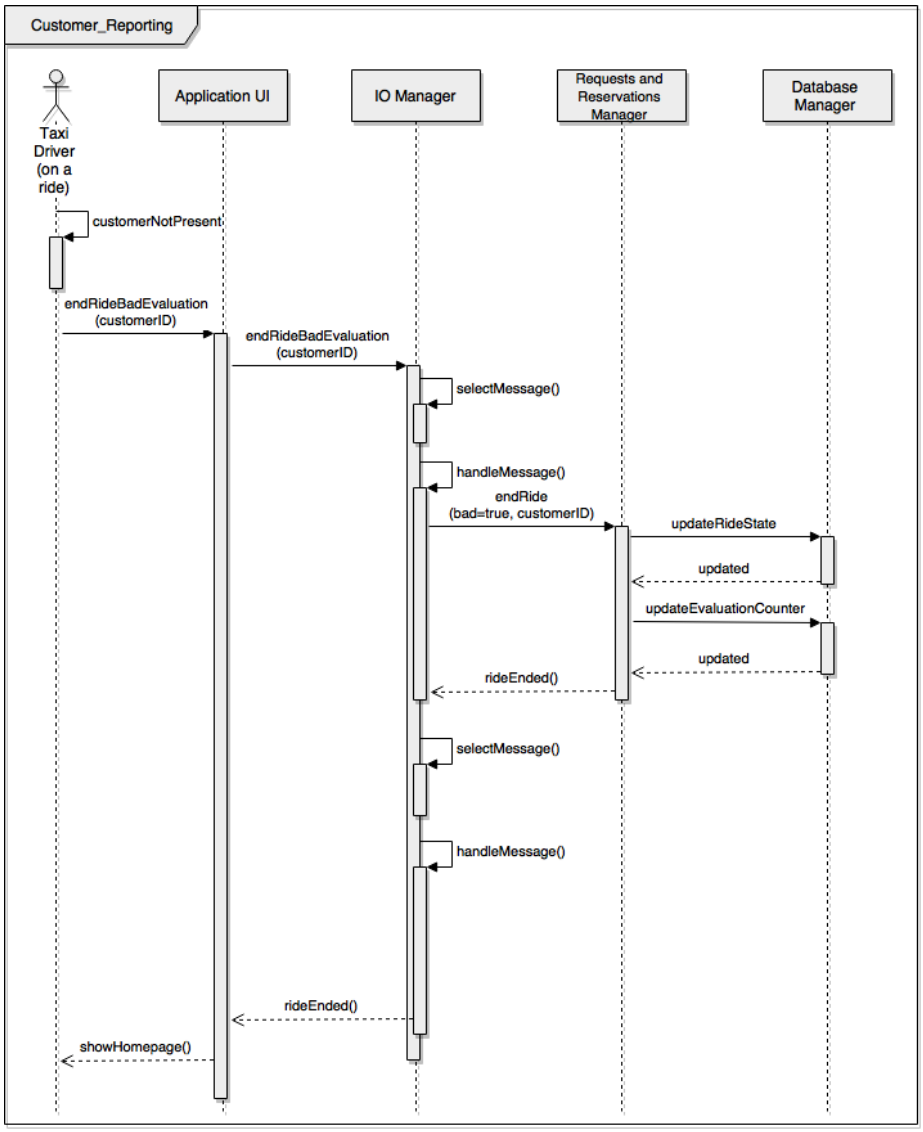

Figure 2.17: Customer Reporting Runtime View

2.5.10 Request A Taxi

Runtime View relative to the Scenario S.9 of the RASD.

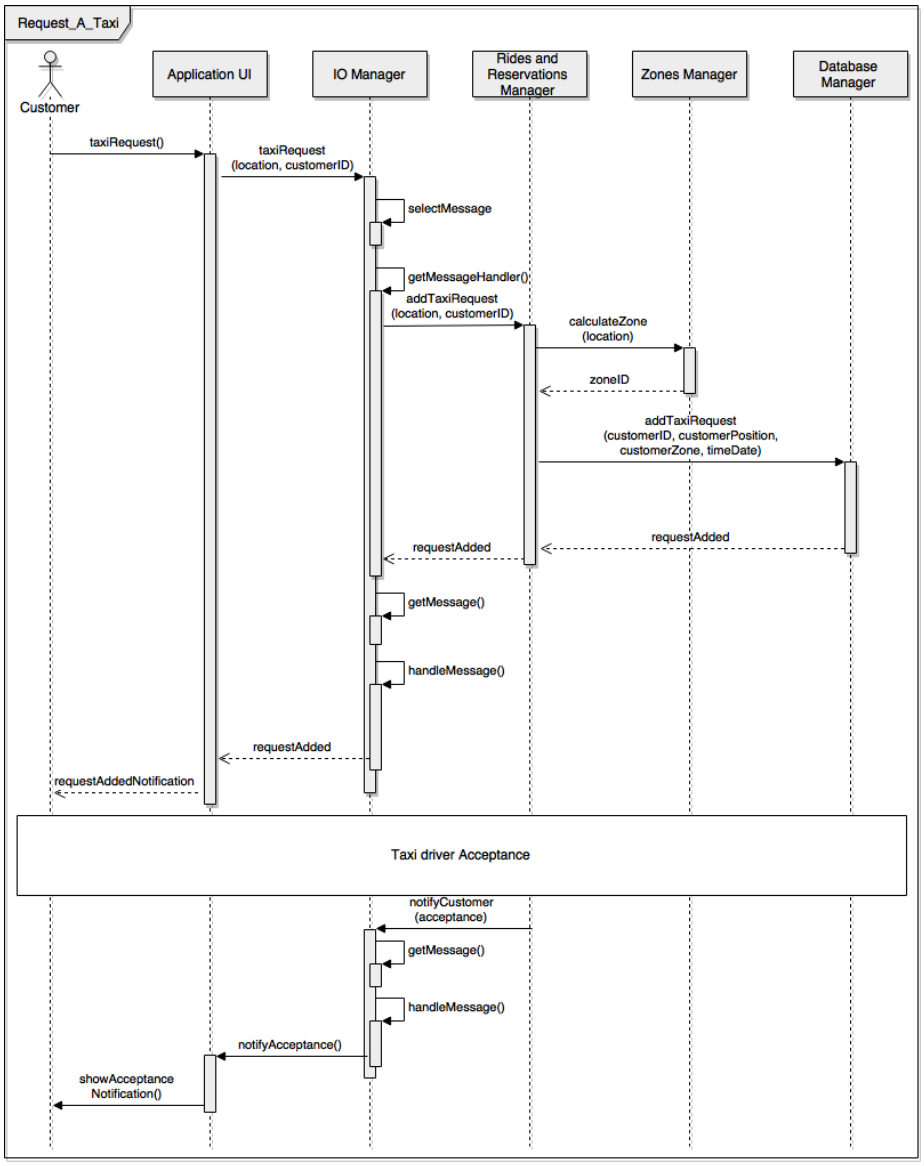

Figure 2.18: Request a Taxi Runtime View

2.5.11 Reserve A Taxi

Runtime View relative to the Scenario S.10 of the RASD.

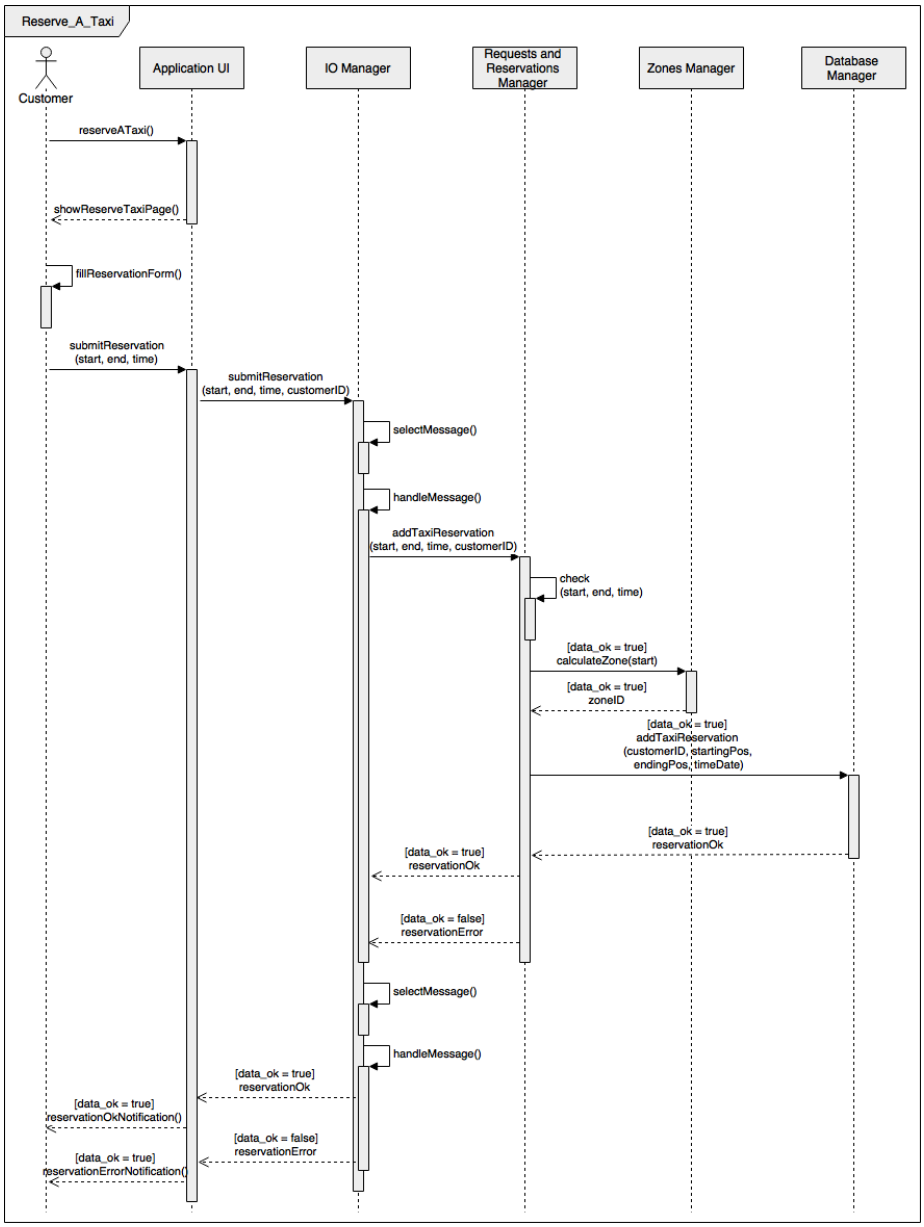

Figure 2.19: Reserve a Taxi Runtime View

2.5.12 Delete A Reservation

Runtime View relative to the Scenario S.11 of the RASD.

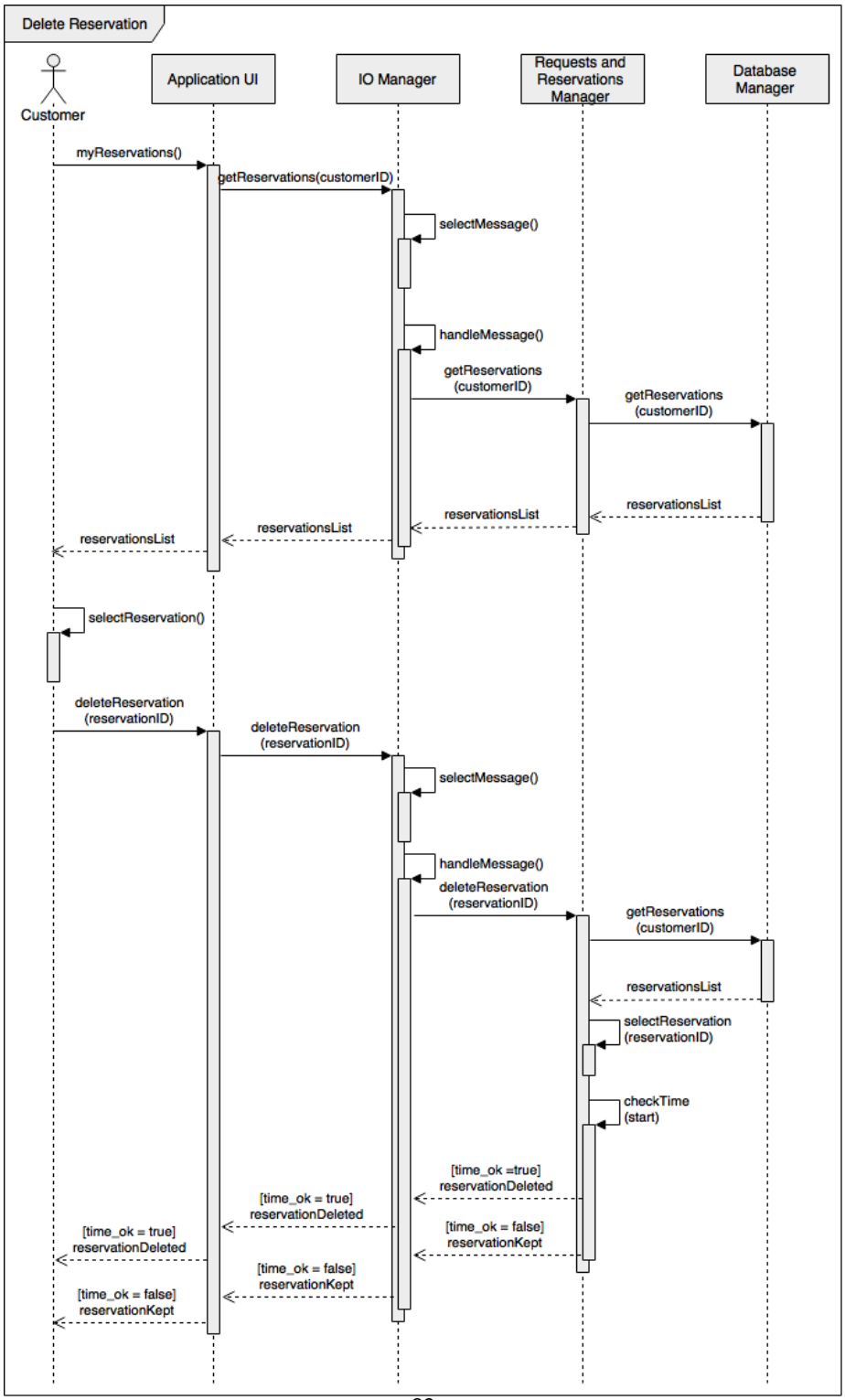

Figure 2.20: Delete one Reservation Runtime View

2.5.13 Accept Or Decline a Ride Request

Runtime View relative to the Scenario S.12 of the RASD.

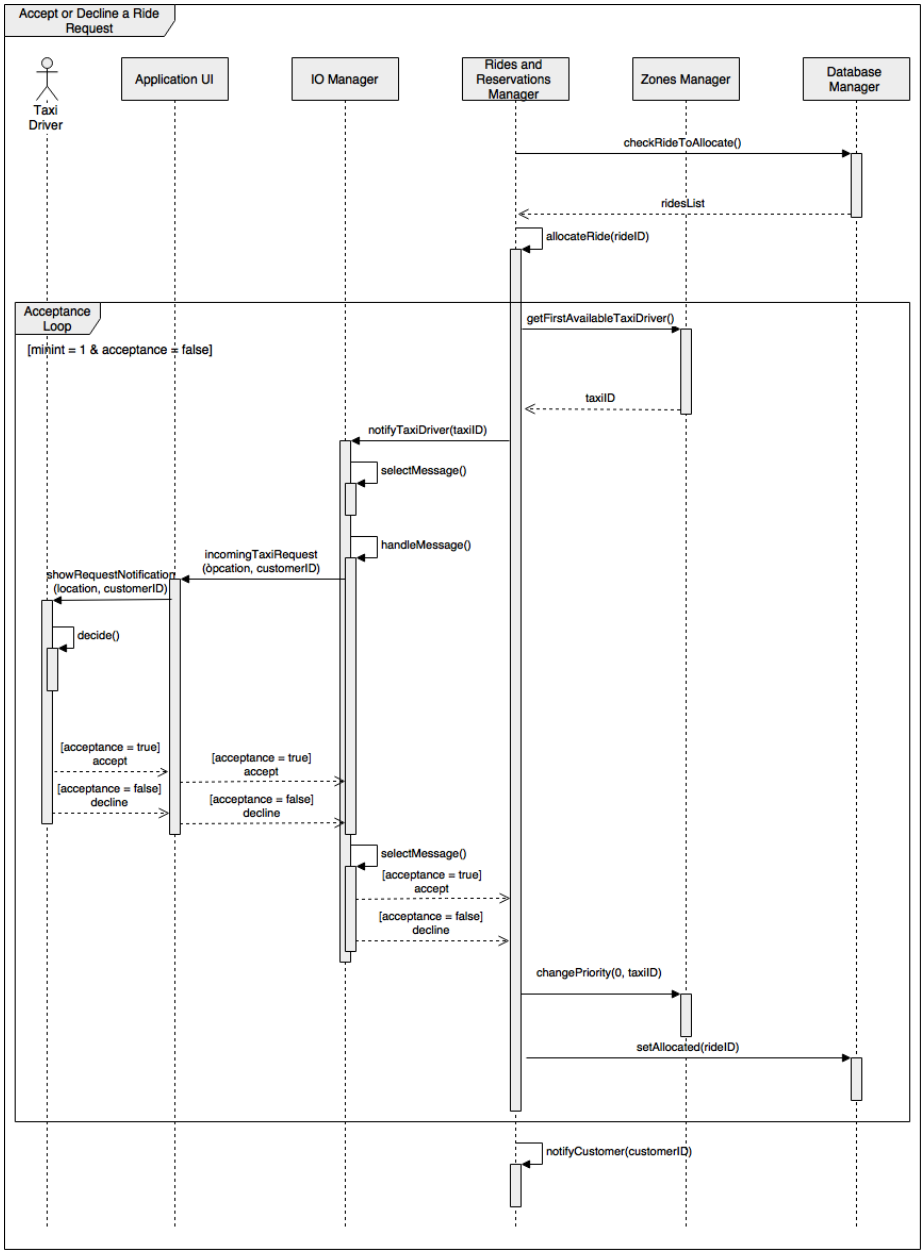

Figure 2.21: Acceptance or Declination of a Ride Request Runtime View

2.5.14 Taxi Driver Availability Notification

Runtime View relative to the Scenario S.13 of the RASD.

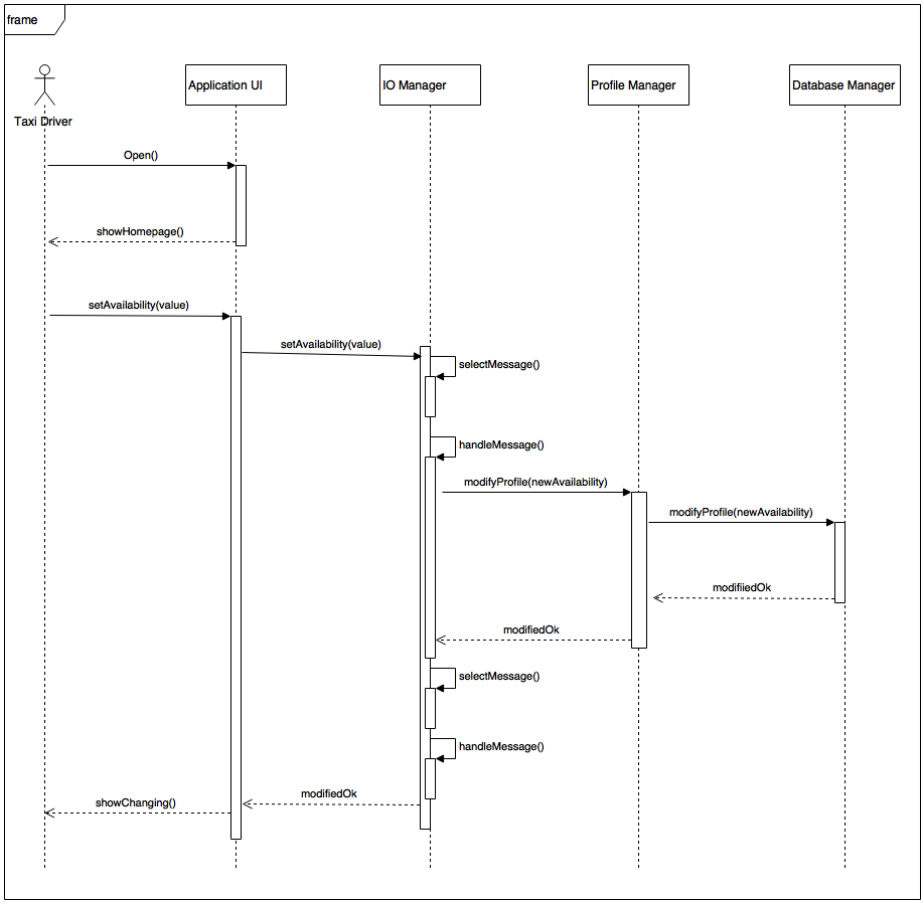

Figure 2.22: Availability Modification Runtime View

2.6 Component Interfaces

descrizione

2.7 Selected Architectural Styles and Patterns

Here are described all the main decisions about Architectural Styles and Patterns.

2.7.1 Architectural Styles

- **Distribution System:** For the distribution of the system we have selected a Client-Server (3 tier) Architectural Style. All the informations about this architectural style are in the "Multitier Architecture" page of Wikipedia (link).
- **Structure:** The structure that we have selected for myTaxiService System is a Layered Structure (Presentation-Logic-Data) and each Layer have a Component-Based Architecture, in fact, each Layer is divided in Components and Sub-Components to ensure the system modularity and scalability.
- **Messaging:** To manage messages between Clients and Server we use a Message-Oriented Middleware that is implemented in the IO Manager in the Logic Tier. This architectural style ensures that the messages are handled in the right way and that the logic is insulated from the client application.

2.7.2 Patterns

- The main pattern we use is the **MVC pattern**. We choose this pattern because of all the advantages it involves. For example the separation between the three main layer makes the system compatible with possible future changing. The fact that the View is divided from the rest of the system gives the possibility to implement different kind of UI, that is required for this system. Also, the separation of the three part of the system make possible a Parallel development that speed-up the creation of the system. And so on.
- The previous pattern is implemented on a **three-tier architecture**. This architecture splits the tre part of MVC on three different tier. The Model part is implemented on the Data Tier, the Control part is implemented in the Logic Tier and the view part is implemented in the Presentation Tier. This pattern is the best choice if you choose to implement a MVC pattern.
- The IO Manager, implement a **Command Pattern**. We choose this pattern to handle the command sent to the server because it makes the communication simpler and make the system scalable. For example if we want to implement a new command that the user can do, we just add a new command class and make it handable for the Message handler class. At this point the previous system still working and if the user want to use the new functionality should only update his front end application.
- The Ride class implement the **State Pattern**. We choose this pattern because of the nature of the class that change different state in his lifecycle.

2.8 Other Design Decisions

All the main Design Decisions are well explained in the previous sections.

Chapter 3

Algorithm Design

This section is intended to better explain some functions and algorithms that can be ambiguous in the previous presentation of our system.

Algoritmi speciali usati nel nostro sistema, uno per uno descritti con pseudocodice o flow chart

algoritmi che potremmo mettere:

- quello per selezionare il messaggio e gestirlo (IO Manager)

  1)procedure selectMessage()

- taxi allocator (selezione delle ride non gestite e gestione)
- queues manager (check positions e ordinamento delle code)
- modify priority –¿ recall della funzione di ordinamento del queue manager
- credentials checker (check email, password)

Algorithm 1 Descrizione Algoritmo

```
1:  procedure EXAMPLE(par1, par2)                    ▷ example algorithm
2:      par1 ← 1
3:      if i ≥ maxval then
4:          i ← 0
5:      else
6:          if i + k ≤ maxval then
7:              i ← i + k
8:          end if
9:      end if
10: end procedure
```

3.1 IO Manager Algorithms

Here we describe some functions and algorithms relatives to the IO Manager component.

3.1.1 Select Message

The select message procedure have to select the right message handler for an incoming/outgoing message. It's described by the following pseudocode:

Algorithm 2 Select Message Procedure

1: **procedure** SELECTMESSAGE($rawMessage$) ▷ rawMessage is an xml (or other markup language like json) message
2: $messageHandler \leftarrow messagesTypeMap.getMessageHandler($
3: $rawMessage.type)$
4: $returnmessageHandler.handleMessage(rawMessage)$
5: **end procedure**

Note on handleMessage(): every single message class that is present on the IO Manager component, can handle incoming messages or outgoing messages and the implementation of handleMessage() for each of this sub-components is different depending on the message that has to be handled.

3.2 Requests and Reservations Manager

Here we describe some functions and algorithms relatives to the Requests and Reservations Manager component.

3.2.1 Check Rides To Allocate

The checkRidesToAllocate procedure is invoked by a scheduler every 2 minutes. This procedure has to check if there are rides or reservations that has to be handled and to put them into the ride-handling stack.

Algorithm 3 Check Rides to Allocate

procedure CHECKRIDESTOALLOCATE
2: $tempList \leftarrow DatabaseManager.getReady()$ ▷ get all the rides that need to be handled
for all $element$ in $tempList$ **do**
4: $allocateRide(element)$ ▷ handle the single ride
end for
6: **end procedure**

3.2.2 Allocate Ride

Allocate Ride starts a new Thread to perform the handling in parallel with the others operations.

Algorithm 4 Allocate Ride

```
    procedure ALLOCATERIDE(ride)
        taxiID ← 0                                              ▷ initialization
3:      repeat
            taxiID ← ZonesManager.getFirstAvailableTaxiDriver()
            msgTD ← createMessage("TaxiDriverNotif", taxiID)
6:          response ← IOManager.selectMessage(msgTD)
            ZonesManager.changePriority(0, taxiID)
        until response == decline
9:      DatabaseManager.setAvailability(taxiID, false)
        msgC ← createMessage("CustomerNotif", ride.CustomerID)
        IOManager.selectMessage(msgC)
12: end procedure
```

3.3 Zones Manager

Here we describe some functions and algorithms relatives to the Zones Manager component.

3.3.1 Check Positions

This procedure is invoked by a scheduler every 2 minutes and updates the zones of every Taxi Driver.

Algorithm 5 Check Position

```
    procedure CHECKPOSITIONS
        availableDrivers ← DatabaseManager.getAvailableTDs()
    end procedure
```

Chapter 4

User Interface Design

We have already describe all the main interfaces that the application can show on the chapter 3.1.1 of the RASD. On that chapter we have already describe also the input and the action that each user can do, and show a possible graphic layout of the pages using some mock object. On this chapter we focus on the navigation between this pages and we describe in a more specific way the input form. To do that we use a UX diagram UML. With this diagram we give a global overview of all the possible pages with the relative forms.

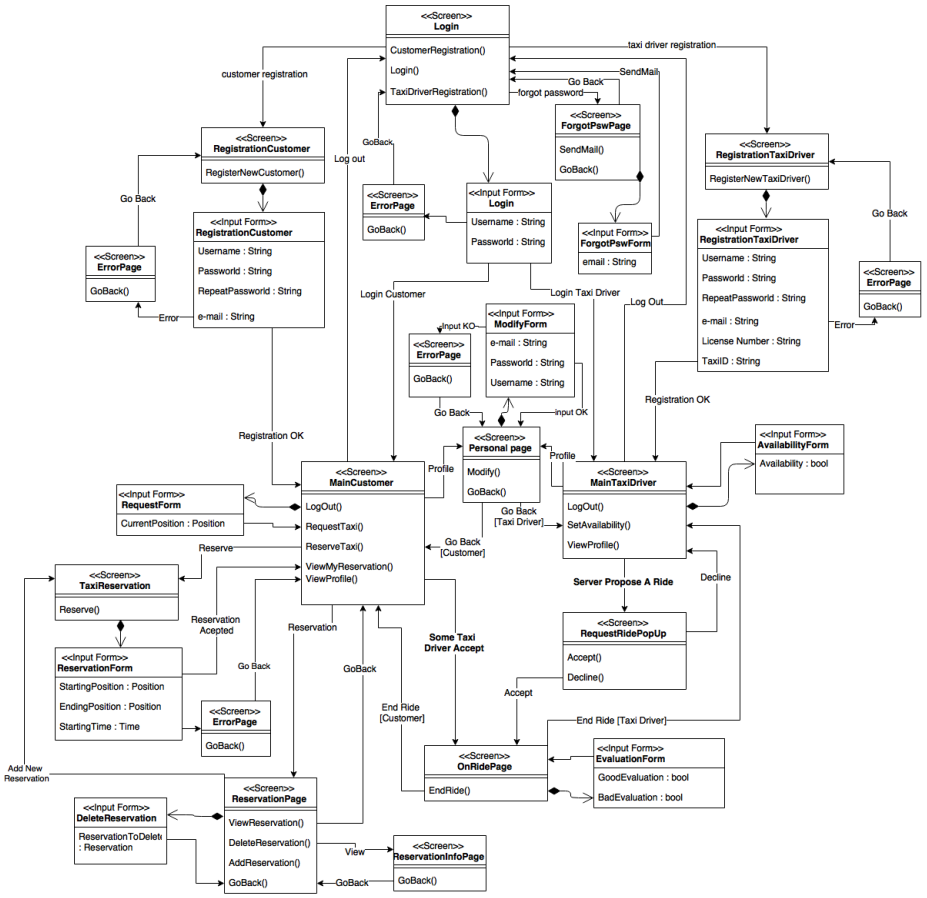

Figure 4.1: UX Diagram UML

Note: We use [Customer] or [TaxiDriver] on the transition to identify the return page if you are respectively logged as a customer or a taxi driver. We do this because the incoming pages have no differences if you are logged as a customer or as a taxi drivers. Note: The transition with the bold text is transition that not depends on the input of the user. So, this kind of translations are, for example, pop up that is showed when on the system something is happen. E. g. a taxi driver that is on his main page can pass on the RequestRidePopUp screen without make any move, simply the pop up is showed when the system ask to the taxi driver if he want to take that ride.

Chapter 5

Requirements Traceability

I requisiti funzionali (nel RASD) come si tracciano all'interno del nostro sistema? I requisiti non funzionali come?

Chapter 6

References

Referenziamo ogni articolo, sito, libro eccc. esterno usato per scrivere il documento