Design Document

Beretta Carolina 852650
Brizzolari Cecilia 852399

December 4, 2015

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide a description of the architecture of the myTaxiService system, going from a high-level representation into a more specific one, underlying the main architectural components and their interaction. Furthermore it provides guidelines for the implementation of the most critical algorithms of the system and user interface design.

This document is meant for project managers and developers.

1.2 Scope

The scope of this phase of the project is to design a system able to optimize the taxi service of the city that satisfies the requirements written previously in the Requirement Analysis and Specification Document.

Such program will need different user interfaces: for this reason, it is expected that the program has the model and the controller parts cleanly separated from the view part. This last piece of code should be able to be interchanged depending on the device without any lapse in the functioning of the service.

1.3 Definitions, Acronyms and Abbreviations

In order to avoid any ambiguity, a list of definitions and acronyms is provided.

Guest: generic unregistered person

User: generic registered person, who is logged into the system

Passenger: user registered as a passenger, who will be able to book a taxi through both the mobile application and the website

Driver: user registered as a taxi driver, who will be able to use the system functions only through the mobile application

Call: immediate booking, the request must be fulfilled as soon as possible

Reservation: advanced booking

Zone: part of the city in which the customer and the taxi is situated. The city is defined as the disjoint union of all the zones

Queue: a list of available taxis in a specific zone, ordered from the one who has been available the longest to the one one who has just finished carrying a customer around

Passenger Application: indicates both the mobile application and the website passengers can use to access the functionalities of the system

MyTaxyApp: mobile application for passengers

MyTaxyDriver: mobile application for drivers

ETA: estimated time of arrival

UX : user experience

1.4 Reference Documents

- Template for the design document provided by the teacher
- Requirement analysis and specification document(RASD) for the myTaxiService system
- Slides on architectural styles of the Software Engineering 2 course
- Component and deployment diagrams: http://www.diit.unict.it/users/gdimodica/IngSoft-Slides/UML10-Component-DeploymentDiagrams.pdf
- Google Maps Distance Matrix API: https://developers.google.com/maps/documentation/distance-matrix/intro

1.5 Document Structure

The document is divided mainly into four parts:

- **First Section:** describes the architecture of the system following a top-down approach.
- **Second Section:** defines the most relevant algorithms to implement
- **Third Section:** provides an overview of how the user interfaces will look like, and how users will be able to access the functionalities of the system through them.
- **Fourth Section:** explains how the requirements identified in the RASD map into the design decision of this document

2 Architectural Design

2.1 Overview

This chapter deals with the structure of the program: which are the high-level components, which architectural style is chosen to host them and how the low-level components of the system interact with each other and the client applications.

Everything will be explained through UML diagrams, specifically component diagrams, deployment diagrams and sequence diagrams. The chapter will also include specifications about the methods of the interfaces and the patterns and architectural styles that should be used in developing the system.
The system will have a three-tier architecture where presentation, application processing, and data management functions are separated as follows:

- **Presentation tier**(client front-end): layer directly accessible by the user through the website or mobile application, allows users and guests to interact with the system and access its functionalities.
- **Business tier:** manages the processes and the logic behind our system and represents the server software, that provides the core system logic and implements the business process and services. This layer is also in charge of processing data between the other two layers.
- **Data tier**: stores and retrieves data from the database: it is composed by the database management system and the database itself.

2.2 High Level Component and Interaction

Figure 1 shows the component structure of the project. It is mainly divided in three parts: a database manager, the system and the client applications (the passenger one and the driver's).

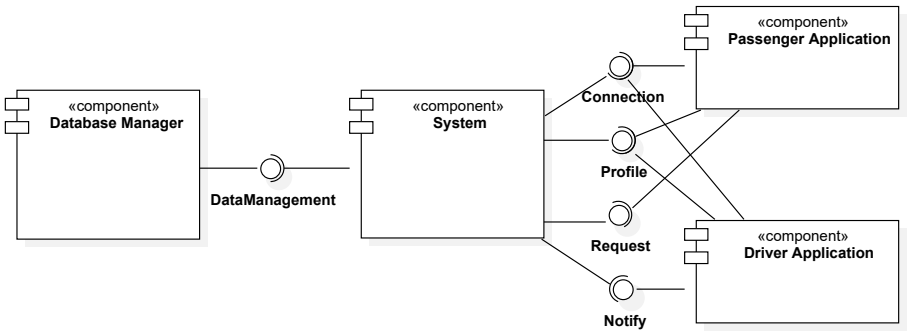


Figure 1: High-level Component Diagram

The Database Manager offers the *DataManagement* interface to let the system read and write data, while the System provides three distinct interfaces for the communication with the clients.

These three interfaces are implemented by different parts of the system, and thus offer different functions.

- *Connection* deals with setting up the user's account, letting the user register or login in order to exploit the other services.
- *Profile* is used to manage profile information: both client applications are able to modify the information contained in the owner's account, to delete an account and, in the case of the driver application, the user is able to change his status.
- *Request* is required only by the Passenger Application, because it is connected with the functions *call taxi*, *reserve taxi* and *active request*, exclusive of the passenger's app.

The last interface (*Notify*) is provided by the Driver Application and is required by the System. It is used by the latter to inform a driver of a pending call by a potential client and to get the answer processed by the Driver Application on the driver's input.

2.3 Component View

The following diagram (figure 2) illustrates all the low-level components, where they are located and exactly how they are connected.

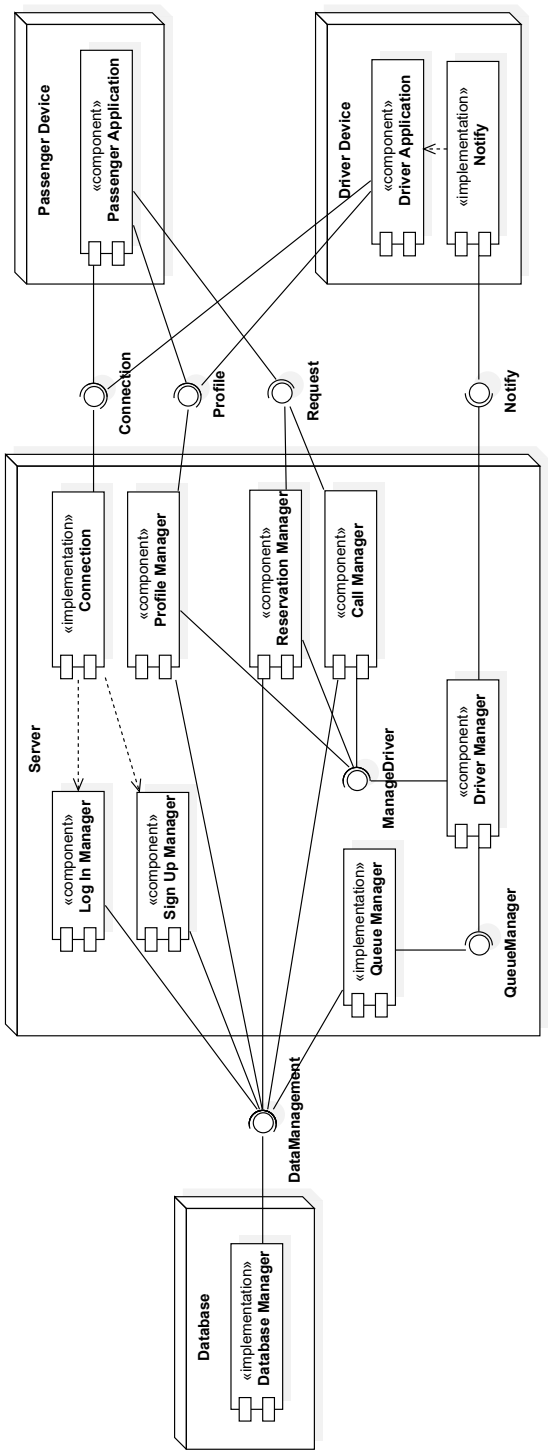


Figure 2: Component Diagram

The server has eight components, six of which are used to receive signals from the clients while the other two are deeper within the system and carry out support operations. There are also four more components, one associated with the Database, one for the Passenger Device and two for the Driver Device.

"Connection", "Profile Manager", "Reservation Manager" and "Call Manager" are only the implementations of the interfaces *Connection*, *Profile* and *Request*, plus some some classes to better distribute the code weight. The first one is also aided by two components, "Login Manager" and "Sign Up Manager", which will take care of the registration phase and the setting up of the information transfer; they will need to manage all of it through a security protocol, to prevent the data to be damaged.

The Driver Manager, belonging to the second category, is the one in charge of getting in touch with the driver's application and of managing everything related to the drivers, such as the forwarding of requests and the change of status. This component is also the only one which isn't connected directly to the Database Manager, but has instead to pass through the Queue Manager. This is because the Queue Manager is the component which deals with the update and the synchronization of the queues, and a driver's status cannot be changed without it having an effect on the queue. Also, the role of finding an eligible driver is left to the Queue Manager, for the same reasons described above.

The Database Manager is the component which makes sure that the database remains consistent and isolated. It also runs the queries asked to be performed by the other components; this component needs to be supported by a security protocol too, since being the implementation of the *DataManagement* interface makes it the one with the most number of request from the other components.

The Passenger Application component deals with the listening and forwarding of the customer's demands; it only shows and gathers information on two different interfaces, one for the mobile version of the application and the other for the browser version, without manipulating it. It is connected to three interfaces to request the right services and to get the messages needed in return, displaying it to the customer's device.

The last two components are the ones relegated to the driver device, "Driver Application" and "Notify". The former works roughly like the Passenger Application, only without the functions provided by the *Request* interface, without the web browser view but with a new functionality (*change status*). The latter, on the other hand, provides and implements the *Notify* interface, which receives call requests forwarded by the system and gets back the driver's answer.

2.4 Deployment View

The deployment view in figure 3 shows how the executable files are deployed inside a given architectural node, while displaying what these nodes actually are.

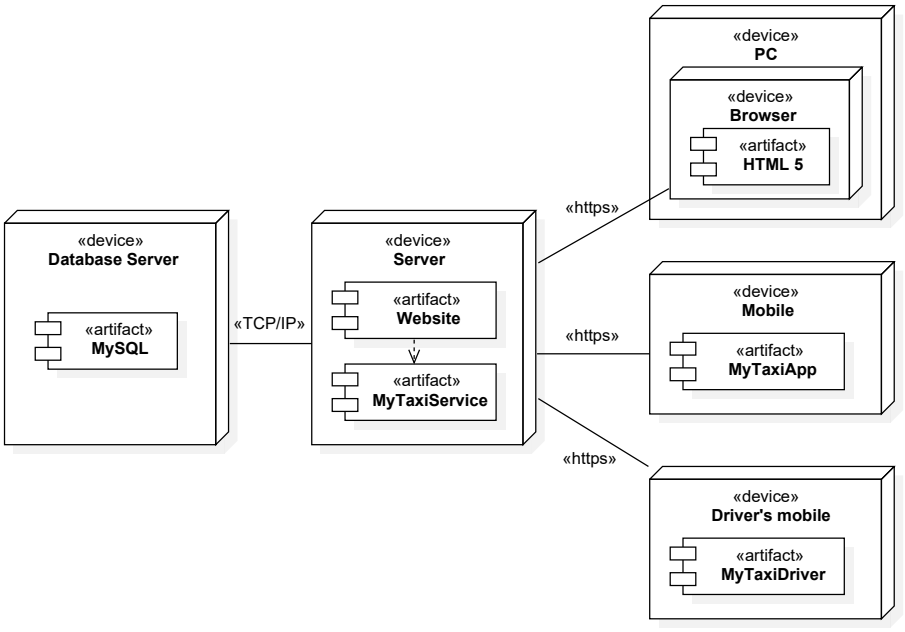


Figure 3: Deployment Diagram

The architecture is divided in five different nodes, two of which represent the two different devices a potential passenger may use.

The central one is the Server farm, which is the physical counterpart to what has been called "the system". It hosts the artifact *myTaxiService*, which is the entire running program that provides the services requested by the customers. It also hosts the website, since the architecture is a three-tier and not a four-tier which includes a Web Server specific for this kind of artifact.

The other server device is the Database Server, that has MySQL running on it to perform queries on the data stored; it is connected to the Server through a TCP/IP protocol.

The client devices are represented by three different nodes:

- a PC device, which inside has a Browser device that, in turn, runs the latest HTML artifact; it shows the Passenger the application through connection with the website.

- a Mobile device, associated to every kind of user which has installed the artifact *myTaxiApp* on it; it doesn't need the aid of an artifact provided by the Server to show the functionalities but still need to be connected to it to carry them out.

- a Mobile device, called "Driver's mobile", owned by a licensed and verified taxi driver, containing the application *myTaxiDriver*; it works like the artifact *myTaxiApp*, only with different functionalities and permissions.

2.5 Runtime View

This paragraph illustrates exactly how the components cooperate with each other every time one of the client applications selects a function. The description of these interactions is made mostly with the aid of sequence UML diagrams.

To avoid redundancy, all functions that can be performed by both the driver and the passenger will only be shown once, using the Passenger Application component.

2.5.1 Sign Up

When a generic user asks to register, the Passenger application communicates this action to the Connection component through the Connection interface. Then, the same request gets forwarded to the Sign Up Manager, which is the only one who can create a new account in the database through the DataManagement interface implemented by the Database Manager.

There are three check phases: the first one is about the validity of the input, for example if a password meets the requisites or if a username is long enough; the second one is meant to look into the database and find out if there's already an account with the same email and/or username as the ones presented in the form; and the last one verifies if the activation link is still valid or if it has passed the seventy-two hours mark.

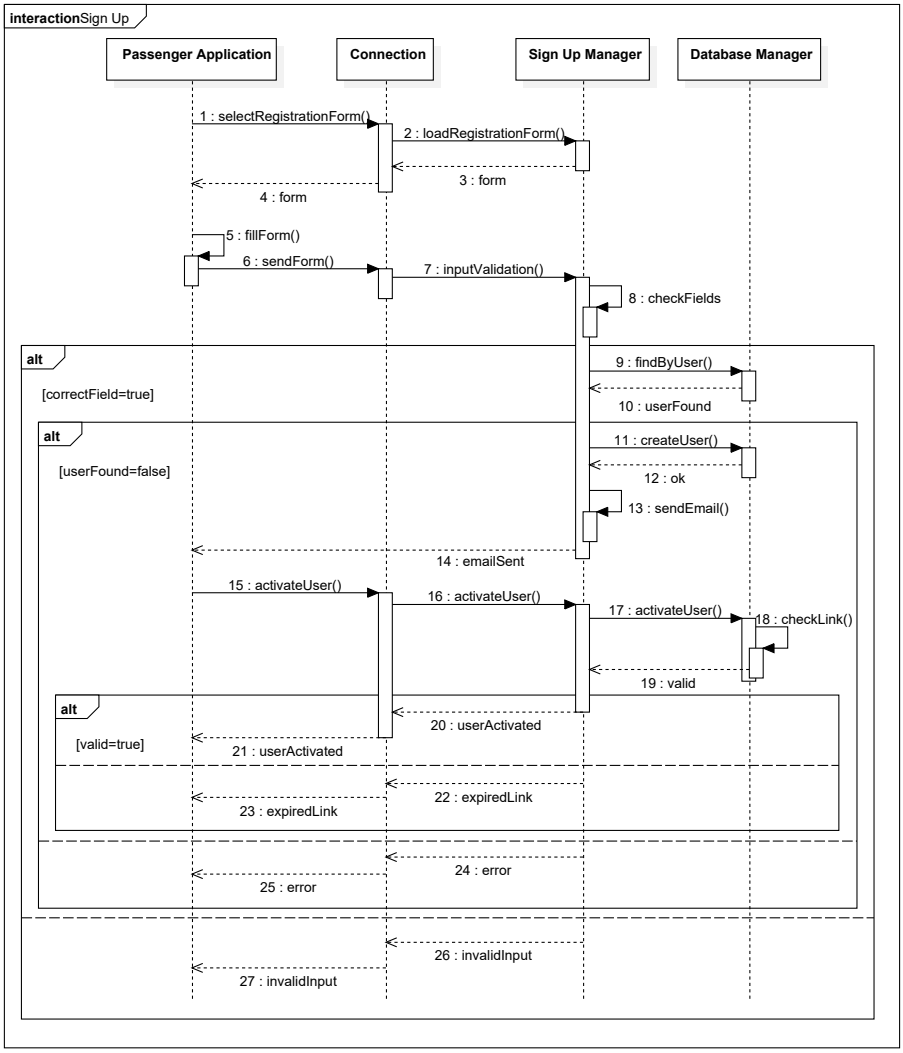


Figure 4: Sequence Diagram: Sign Up

2.5.2 Login

The Login function works a lot like the Sign Up function, with minor differences in the components used but some more in the interaction with the database.

As soon as the user chooses the *Login* function, the correct form is loaded; the user only has to insert his credentials and press the button to send them. The client application uses the *Connection* interface and implementation to get to the Login Manager, which executes two checks: the first on the syntax of the credentials while the second one, happening afterwards, looks for the username

in the database through the Database Manager, while comparing the password given by the one saved in the database.

If both checks return a true value, the home page of the account is loaded by the Passenger application, showing the eligible functions. Otherwise, an error message is sent back.

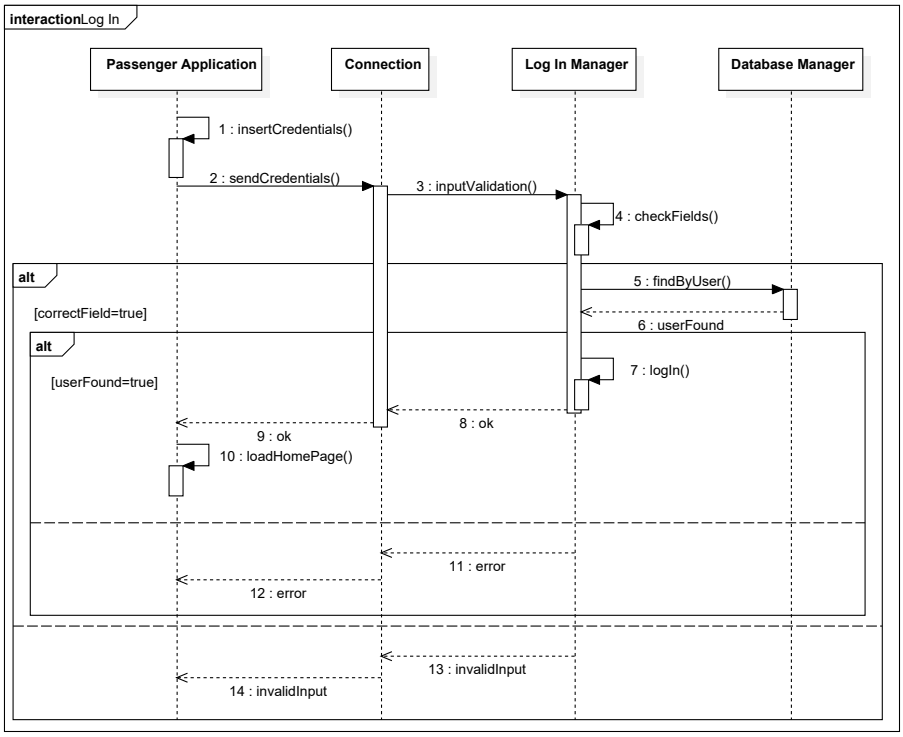


Figure 5: Sequence Diagram: Login

2.5.3 Update profile

To update the info of an account, the client application has to request the form containing all previous credentials to be recompiled by the user. So the Passenger Application calls the update function presented in the *Profile* interface which triggers the interaction between the Profile Manager and the Database Manager, returning the above mentioned form. After the form has been filled, it gets forwarded again to the Profile Manager, which performs only the syntax validity check before sending the new info to the Database Manager.

The account information can either be updated successfully, or the system can return an error message to the Passenger Application.

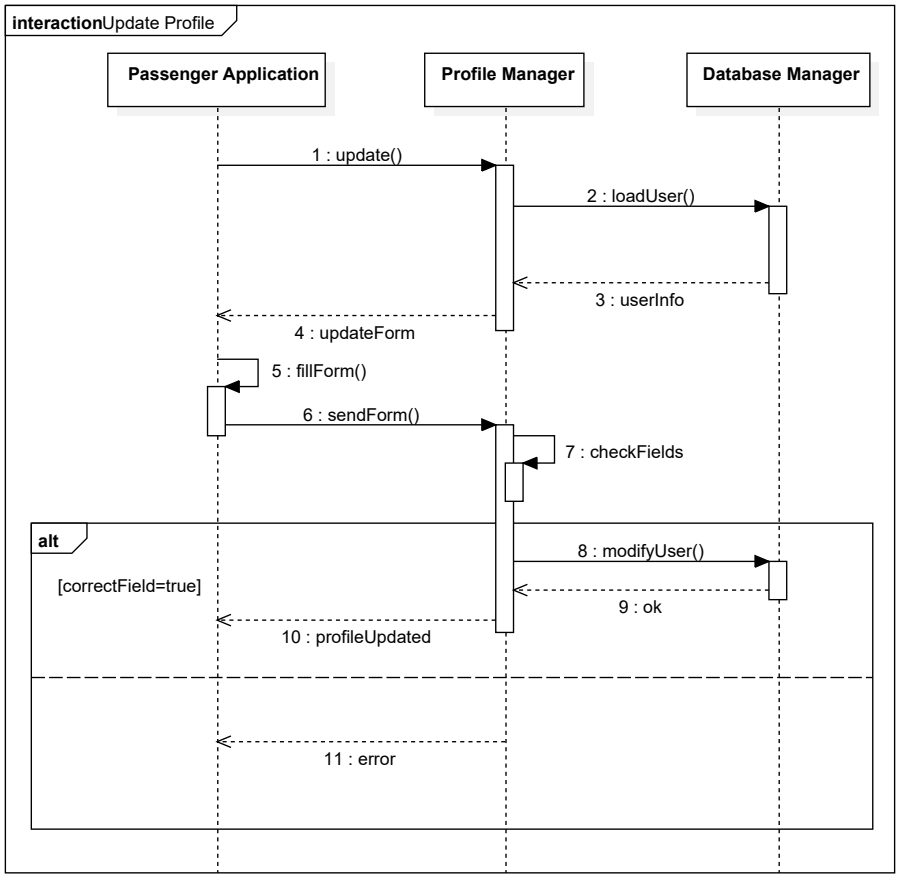


Figure 6: Sequence Diagram: Update profile

2.5.4 Delete profile

The deletion of an account involves three components: the Passenger Application, the Profile Manager along with its interface, and the Database Manager, this too with its interface.

In this function, the required form needs only the password to be written; this form gets forwarded to the Profile Manager, which loads the user from the database and checks if the password is correct. If the method returns a true value, the Profile Manager asks the Database Manager to delete all the information pertaining the account, and informs the user of the completed operations as soon as it receives the ok from the Database Manager.

After that, the Passenger Application loads the homepage for guests.

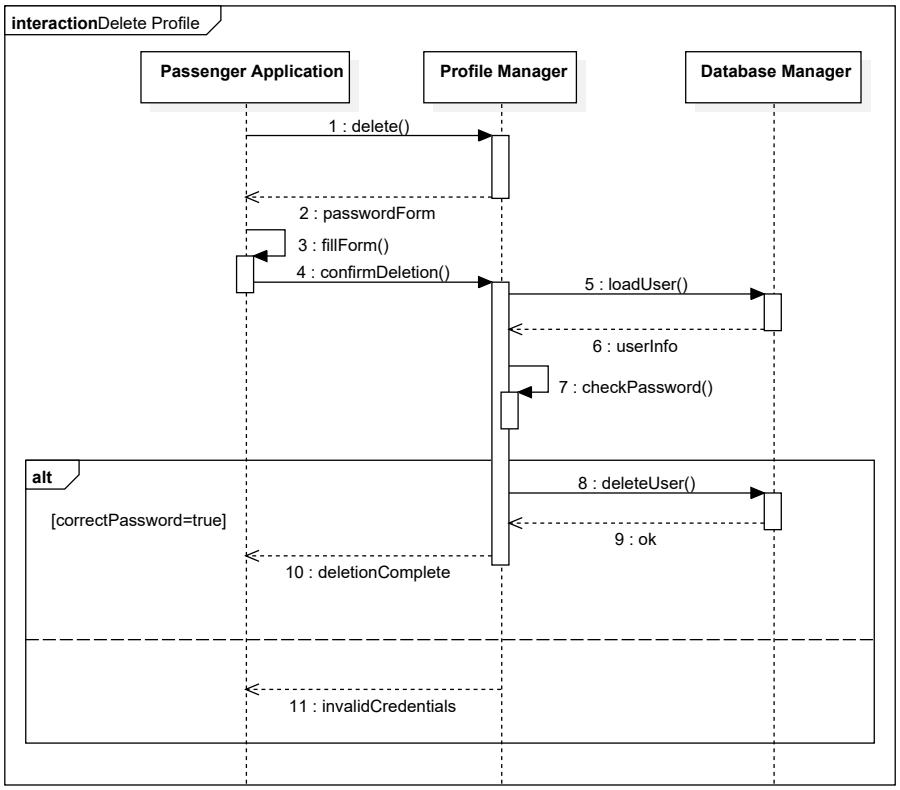


Figure 7: Sequence Diagram: Delete profile

2.5.5 Call taxi

The Passenger Application can send a call taxi request through the *Request interface*, implemented by the Call Manager, which returns a simple form to be filled: it only needs the position in which the passenger wants to be picked up and the number of passengers to be expected. The position can be taken automatically by the Passenger Application through exploiting the GPS on the mobile phone, or can be written manually by the user.

After the form is sent to the Call Manager, this component checks if the fields are valid and, if this is the case, proceeds to ask the Database Manager to store all call information in the database and the Driver Manager to find an eligible driver. If the driver is found, the Call Manager informs the Passenger Application that the call has been submitted correctly and all the relevant call info; otherwise, the Call Manager asks the Database Manager to delete all previous information and sends to the user a "Taxi not found" message.

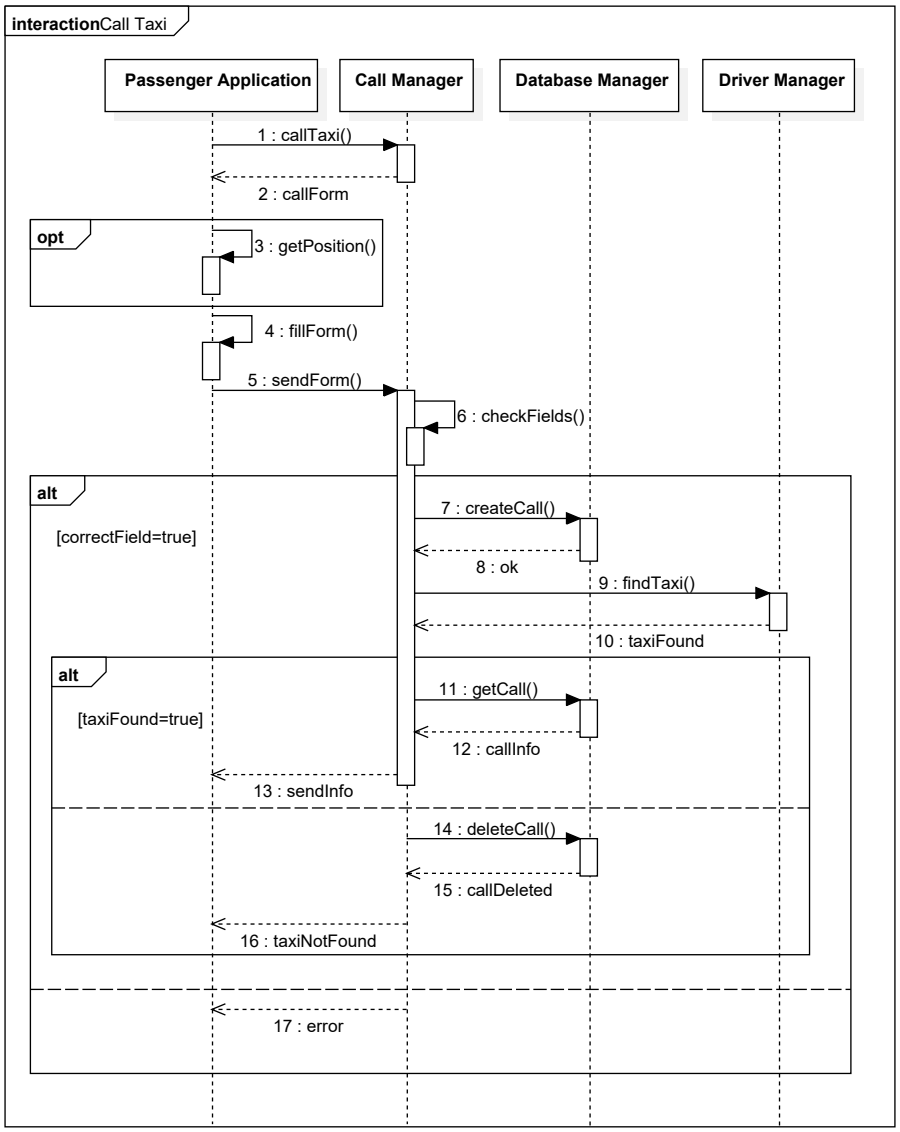


Figure 8: Sequence Diagram: Call taxi

2.5.6 Reserve taxi

The reservation of a taxi only consists, at first, in the creation of a Reservation instance and the storage of information in the database; after that, it is the Reservation Manager's job to find reservations whose waiting time is almost up, ask the Driver Manager to find a driver and send a reminder of the reservation to the passenger.

The Passenger Application, through the *Request* interface, asks the Reservation Manager for the correct form, waits of the user's input and then sends the compiled form back; the Reservation Manager checks the validity of the fields, which are position to be picked up from, time of meeting and expected passengers, and proceeds to trigger the creation of a new reservation in the database from the Database Manager. After that, the components only return messages pertaining the result of the operations.

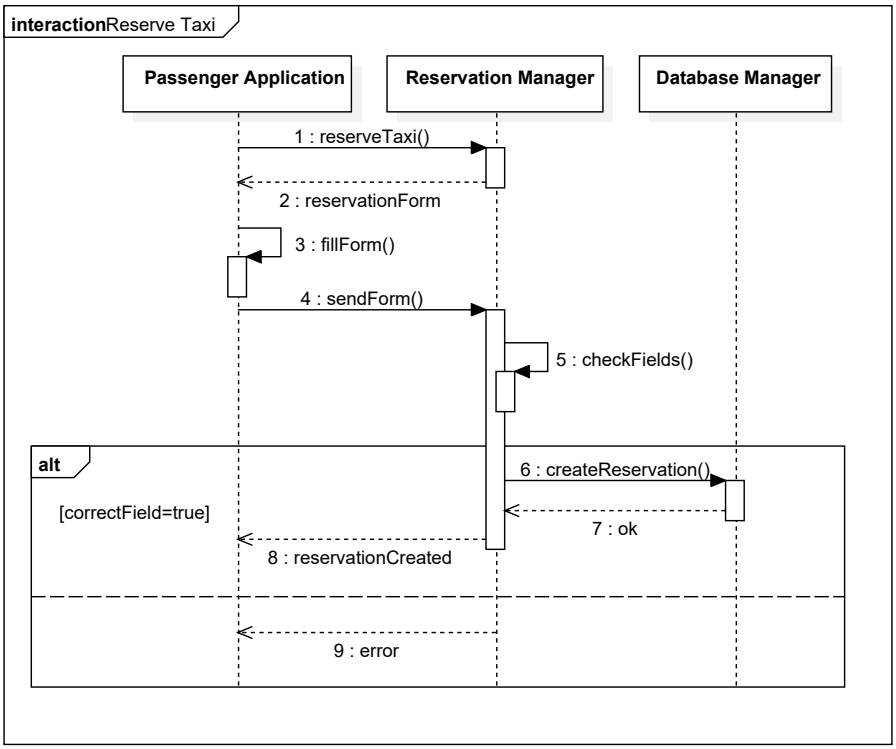


Figure 9: Sequence Diagram: Reserve taxi

2.5.7 Active request

A passenger can also see the status of an active request he/she may have. In this case the Passenger Application interacts with the Call Manager which, in turn, cooperates with the Database Manager in order to get the information stored under his/her username. If the value returned doesn't point to anything, then the Call Manager informs the Passenger Application that there's no active request and the latter shows the message to the user; otherwise, the informations regarding the active call are shown.

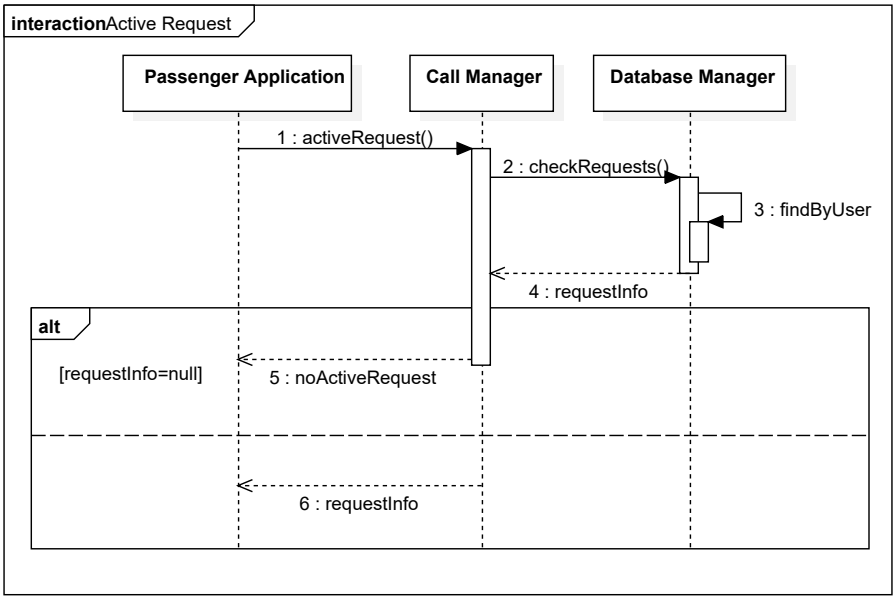


Figure 10: Sequence Diagram: Active requests

2.5.8 Update status

Exclusive of the Driver Application is a function to update one's status from *offline* to *available* and viceversa. Such function involves several components: the Driver Application, the Profile Manager, the Driver Manager, the Queue Manager and, at last, the Database Manager. This happens because in order to modify a driver's status the system has to rearrange the queue pertaining the zone the driver was in.

The Driver Application gets the input from the driver, calls the Profile Manager which, in turn, forwards the demand to the Driver Manager. The latter can ask the Queue Manager to either add a driver to a given queue or to remove it. In the first case, the Queue Manager will tell the Database Manager to set the driver's status to *available* and then update the queue, while in the second case the driver's state will be set to *offline* and the driver removed while updating the queue.

The process ends when the Driver Application receives the message that the status has been updated.

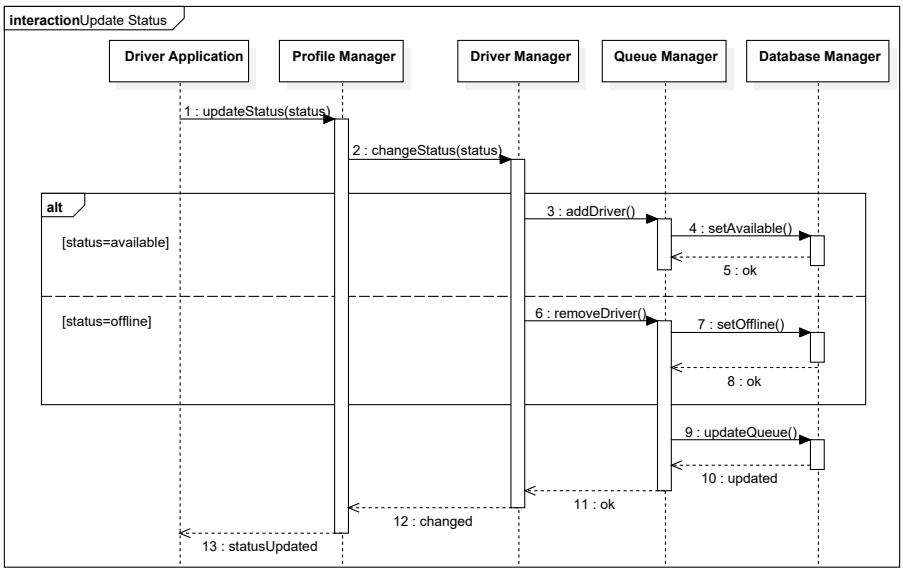


Figure 11: Sequence Diagram: Update status

2.5.9 Allocate taxi

This last diagram represents the process the system undergoes while finding a driver that will take a passenger's request. The algorithm describing this whole interaction is better explained in chapter 3 of this document.

The components involved are the Driver Manager, the Queue Manager, the Database Manager, the *Notify* interface with its implementation and the Driver Application.

The process is triggered when the Call Manager asks the Driver Manager to find an eligible taxi: the former forwards the demand to the Queue Manager, which selects the queue belonging to the correct zone and starts iterating it. After the first driver has been selected from the Database Manager and his/her info have been returned to the Driver Manager, this component can notify the Driver Application through the *Notify* interface and implementation of the pending call. Depending on the driver's answer, the Driver Manager asks the Queue Manager to update the queue in two different ways: it either removes the driver from the queue and sets his/her status to *busy* if the call has been accepted, or it moves the current driver to the bottom of the queue, and the process starts again.

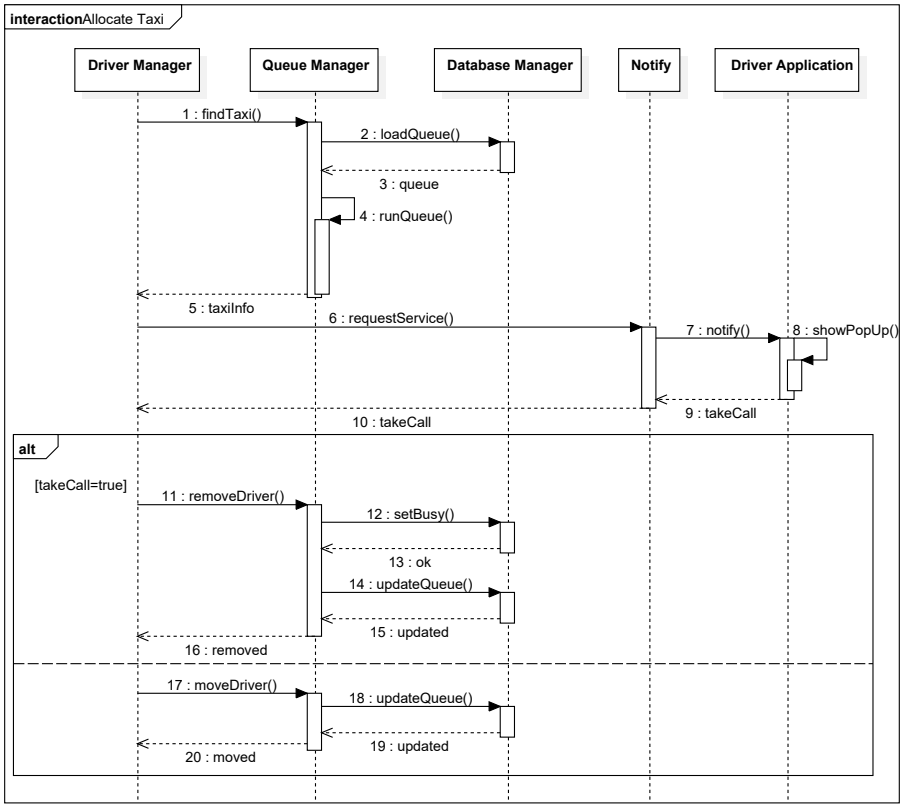


Figure 12: Sequence Diagram: Allocate taxi

2.6 Component Interfaces

In the component diagram there are six visible interfaces. Each of them offer different methods to the other components, illustrated in the following tables.

Connection
Form: selectRegistrationForm()
String: sendForm(Form)
String: sendCredentials(Form)
Boolean: activateUser(String)

Table 1: Connection interface

Profile
Form: update(String)
Form: delete(String)
Boolean: sendForm(Form)
Boolean: confirmDeletion(Form)
Boolean: updateStatus(Int, String)

Table 2: Profile interface

Request
Form: callTaxi()
Form: reserveTaxi()
String: sendForm(Form)
String[]: activeRequest(String)

Table 3: Request interface

Notify
String: notify()

Table 4: Notify interface

ManageDriver
Boolean: findTaxi()
Boolean: changeStatus(Int, String)

Table 5: ManageDriver interface

QueueManager
Boolean: addDriver(Int)
Boolean: removeDriver(Int)
Boolean: moveDriver(Int)
Driver: findTaxi()

Table 6: QueueManager interface

DataManagement
Boolean: findByUser(User)
Boolean: createUser(User)
Boolean: activateUser(String)
User: loadUser(String)
Boolean: modifyUser(User)
Boolean: deleteUser(User)
Boolean: createCall(Call)
Call: getCall(String)
Boolean: deleteCall(Call)
Boolean: createReservation(Reservation)
Request: checkRequests(User)
Boolean: setAvailable(Driver)
Boolean: setBusy(Driver)
Boolean: setOffline(Driver)
Boolean: updateQueue(Queue)
Queue: loadQueue(Zone)

Table 7: DataManagement interface

2.7 Architectural Styles and Patterns

The architectural styles and patterns used for the development of the code are the following:

Client-Server

The system is designed with the classical client-server approach. Clients will follow the thin client paradigm, since all application logic and computation is on the server and they will just have the presentation layer.

Service Oriented Architecture

This style is used in the communication between the server and both users and guests. In fact users and guests will access the system's functions through component interfaces, independently from their implementation. Moreover the system itself relies on external services such as Google Maps for the computational of traveling time and on the service provided by the city for the driver license validation.

MVC

The system will strongly exploit the MVC pattern. In fact users and guests will only be able to access the view and not the model directly, for security reasons and to support the system's portability.

3 Algorithm Design

This section contains guidelines for the implementation of the algorithms of the system. The analysis will focus on the most complex and significant ones.

3.1 Queue Management

As specified in the class diagram of the myTaxiService system, each zone will have a queue of taxi drivers. The queue will contain only drivers whose status is set to available and will be used to select them for request forwarding. It is important to stress that drivers can also be located outside the city zones, as the reservation constraint of maximum distance is a 50km radius. When a driver is added to a queue, he/she will always be added at the end of the queue, i.e pushed into the queue.

The system will not have a generic algorithm that only aims at managing the different queues associated to the zones. Instead the queues will be updated each time one of the events described in the following sections occurs.

3.1.1 Driver's status changed

As soon as a driver updates his/her status the queues have to be updated accordingly:

1. **Offline to Available:** if the driver is in a zone Z, the system pushes him/her into the queue of Z, otherwise no update is done
2. **Available to Offline**: if the driver is in a zone Z, the driver is removed from the queue of Z, otherwise no update is done

3.1.2 Driver leaves or enters a zone

Each time a driver's position is updated, the queues need to be updated accordingly.

As soon as a driver leaves a zone, the queues are updated. There can be two different cases depending on the driver's new position:

1. Inside the city: the driver is removed from the queue of the previous zone and pushed in the queue of the current zone
2. Outside the city: the driver is removed from the queue of the previous zone

Another scenario is when the driver enters a zone Z from a position outside the city. In this case the system simply pushes the driver into the queue of Z.

3.1.3 Driver declines or accepts a request

A driver can decline a request willingly or have it automatically declined by the system when the timeout is triggered. In both cases the system removes the driver from the top of the current queue and pushes him/her back into it. On the other hand if the driver accepts the request his/her status is set to busy and he/she is removed from the queue he/she is currently in. This event is handled in the algorithm described in section 3.2.

3.2 Request Forwarding

This algorithm is run every time the system has to find a driver for a ride. If the request is a call the system runs the algorithm as soon as the request is accepted. On the other hand, if the request is a reservation the algorithm is run fifteen minutes before the time indicated in the reservation. A flowchart of the algorithm is shown in 13.

The algorithm uses two different timers: a global and a local one. The local timer represents the thirty seconds time limit the driver has to accept or decline a ride request. Once the local timer is set, if the driver accepts the request before the time runs out the algorithm follows the rightmost branch in 13 and terminates; otherwise, the request is declined and the algorithm selects a new driver.

The purpose of the global timer is to limit the research time in order to fulfill the functional requirement specified in the RASD. If the algorithm has not found a driver within five minutes, the search terminates and the passenger is notified of the failure as shown in 14.

The refuse list is a local data structure used by the algorithm to store the ID of drivers that have already declined the ride request. Request are forwarded only to eligible drivers, i.e. drivers that haven't already declined that request. Each time a request is declined the algorithm checks if there is at least one eligible driver in the current queue. The research is done in order starting from the top of the queue and going down to the end. As soon as an eligible driver is found the system select him/her as the next driver and forwards the ride request. On the other hand, if no eligible driver is found the algorithm switches the current zone to one of the neighboring ones.

The refuse list's purpose is to avoid cyclically sending the request to the same set of drivers, that have already declined it. The zones are of 2 km^2, thus it is likely that some zones will have low populated queues. In the worst case scenario where all drivers decline the ride request, the search, if limited to only the pick-up zone, would reach a dead end. Since the zones have a fairly small extension, selecting a driver from a neighboring zone will not significantly affect the traveling time, while it will avoid refusing a request and cut down the response waiting time for the passenger. This design decision demands the neighboring zones to be easily calculated, so it is convenient to have the zone's data structure to have a field containing adjacent zones. Due to the nature of the problem, the refuse list should not need any specific search algorithms

to be implemented. In fact, as assumed in the RASD, drivers that are using the *myTaxiDriver* app and have their status set to online are drivers that want to use the system to optimize their work. Therefore it is unlikely that a huge number of drivers will decline the ride request, making the refuse list growth rate a problem. Once the algorithm has switched from the pick-up point zone to a neighboring one, it will not go back to the original zone. However queues are assumed to be enough populated to avoid the situation in which all neighboring zones have been selected, and the request has been sent to all its drivers, before the global timeout.

3.3 ETA

The system will not have an algorithm for computing the ETA, but will rely instead on Google Maps. As soon as a driver has been assigned to the ride, the system will retrieve the driver's position and will use the service offered by Google Maps Distance Matrix API to compute traveling time between the driver position (origin) an the pick-up point of the ride (destination). The system will not specify the traffic_model field, as it will use the best guess option. Once the system has obtained the ETA, it will send all taxi's info to the passenger. The ETA however, will need to be updated every two minutes.

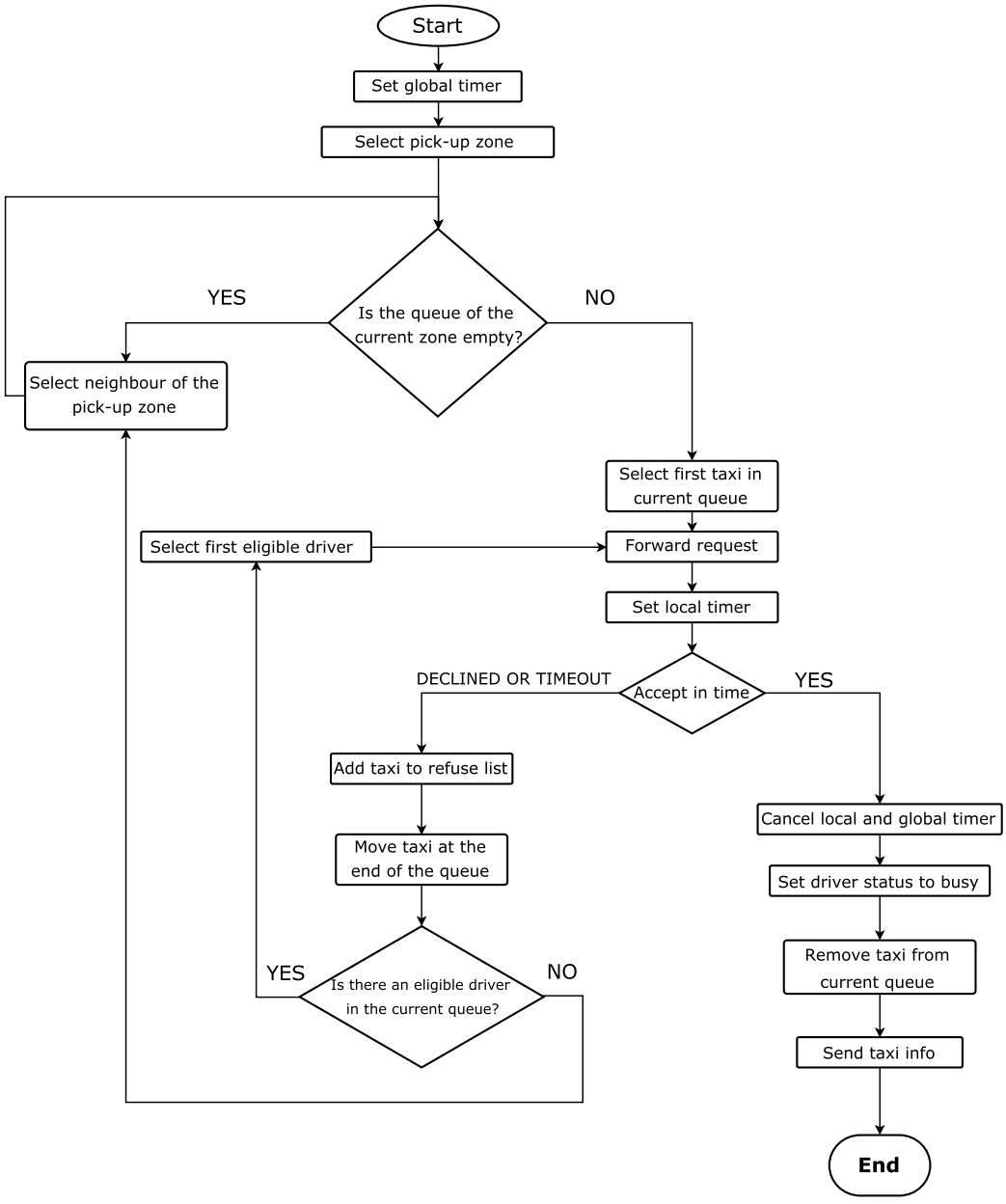


Figure 13: Request forwarding flowchart

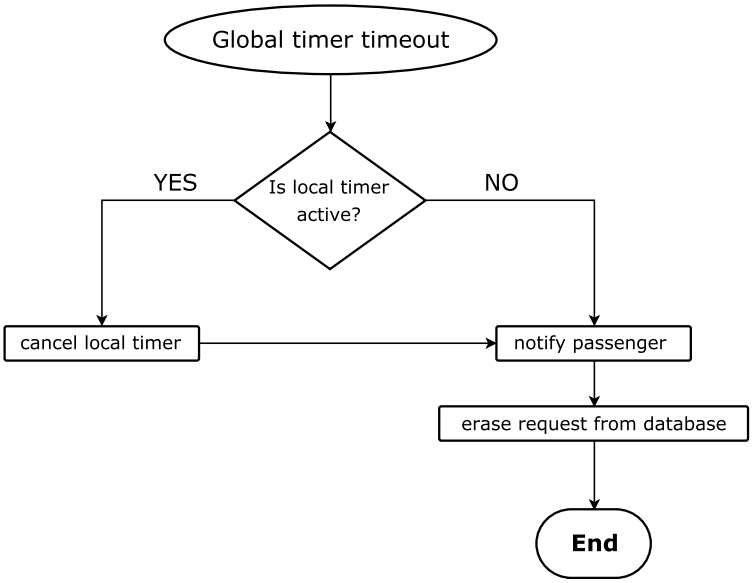


Figure 14: Global timer timeout

4 User Interface Design

4.1 User Experience

In order to give an overview of the interfaces users will interact with, an user experience (UX) diagram is provided. The diagram is a class diagram with appropriate stereotypes: <<screen>> and <<input form>>. While <<screen>> represents general pages, <<input form>> represents input fields that the user will fill and then submit to the system by clicking on a confirm button. Furthermore <<screen>>s can have a landmark **$,** used to indicate that the page is reachable from any other page of the application.
For the sake of clarity, the whole UX diagram has been split into three subdiagrams, one for each specific actor of the system. The purpose of these diagrams is to give a general view of the user experience and to illustrate at a high level how users and guests can access the functionalities offered by the system, without underlying the difference between the three applications. A more detailed specification and visual representation of how the different interfaces will be implemented is given in section 1.2. As stated in the RASD, drivers will be able to access the system only through a mobile application, whereas passengers can choose to use the mobile application or the web one. Nevertheless, potential drivers will also be able to register to the system through the web application, but to access the other functionalities they will have to download and install the *myTaxiDriver* mobile app.

4.1.1 UX diagram of guests

The UX diagram in figure 15 describes the interaction with the guest, by focusing on how the potential user may accomplish the log in and registration functionalities.
The first page shown to the guest is the *Guest Home*, where he/she can chose to either log in to the system or create a new account. The log in function takes the guest to the *Log in* page, where he/she can insert the needed information to log into the system. If the credentials are correct the guest becomes a user and is taken to his/her *User Home* page, otherwise an error message will appear on the same page.
The *Registration* page contains an input form that allows the guest to register to the system. When the guest sends the compiled form, the system checks the input data and if some data is missing or wrong an error on the same page is displayed. On the other hand, if the data provided by the guest is formally correct and no problem has been detected, the guest is notified of the successful registration and asked to activate the account with the activation link sent to his/her email account. After that the guest is redirected to the *Guest Home* page.

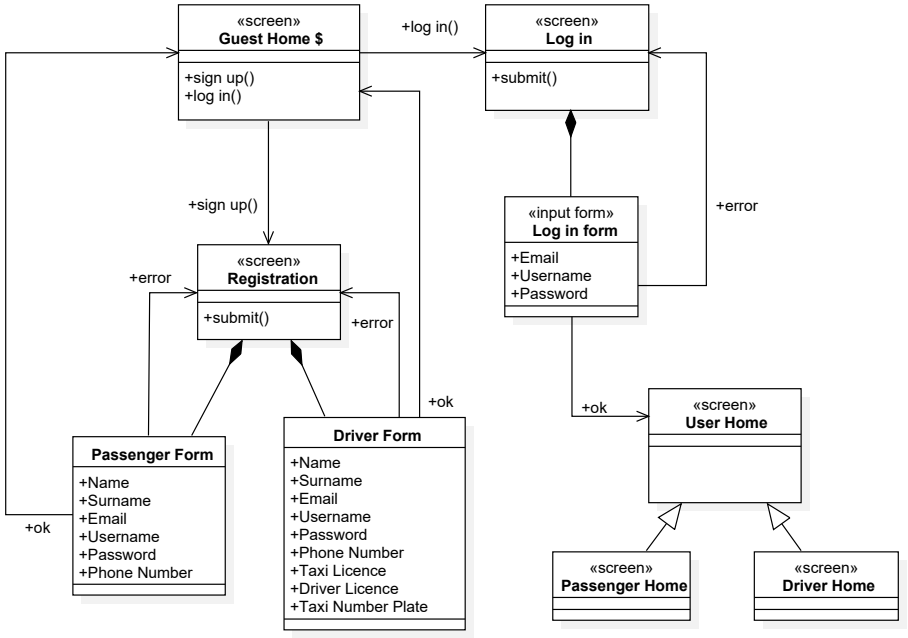


Figure 15: Guest registration and log in

4.1.2 UX diagram of passengers

The UX diagram in figure 16 describes the interaction with the passenger and the specific functionalities offered to this type of users.
The first page that is shown, as soon as the passenger logs in, is the *Passenger Home*. This page is reachable from all the other pages (as indicated by the landmark **$**) and allows the passenger to access all the functionalities of the system. The passenger can request a taxi through the *Call Taxi* page for an immediate booking, or the *Reserve Taxi* page for advance booking. In both cases the passenger must fill a form with the required information and submit it. If the request is accepted, i.e the data provided was successfully validated, the system notifies the passenger that the request has been accepted and starts looking for a taxi. If the booking was made through the Call Taxi page, as soon as a driver has been assigned to the ride request the system redirects the passenger to the *Active Request* page. In this page the passenger is able to see his/her active request and the related info. On the other hand if any problem with the input data is detected, an error on the same page is displayed.
The *My Profile* screen page shows passenger's information. Passenger can update their data by compiling a form or delete their account from the system.

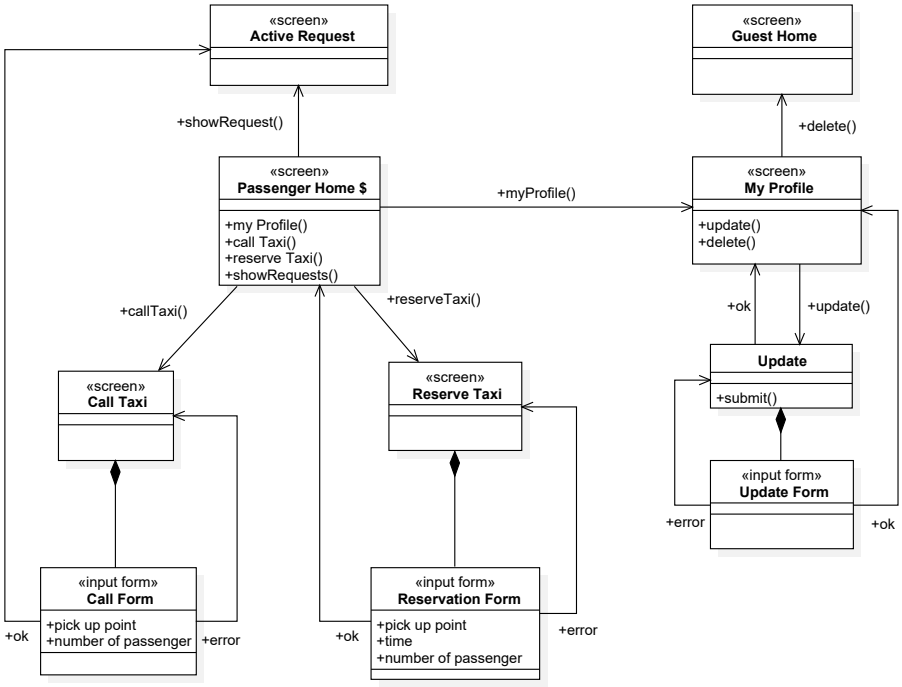


Figure 16: passenger application UX

4.1.3 UX diagram of drivers

The UX diagram in figure 17 describes the interaction with the driver and the specific functionalities offered to this type of users.
The first page shown after the log in is the *Driver Home* page. From here drivers can change their status or visualize their profile. The *My Profile* page behaves in the same way as the one for passengers, described in section 1.1.2. The *Change Status* page allows drivers to update their status (available, offline).
When the system forwards a ride request to a driver, the application alerts the driver with a a pop-up notification. After the request is accepted the application redirects the driver to the *Request Accepted* page. Here the driver can notify the system that he/she has picked up the passenger and after the end of the ride he/she can select the end ride option to return to the *Driver Home* and wait for another request.

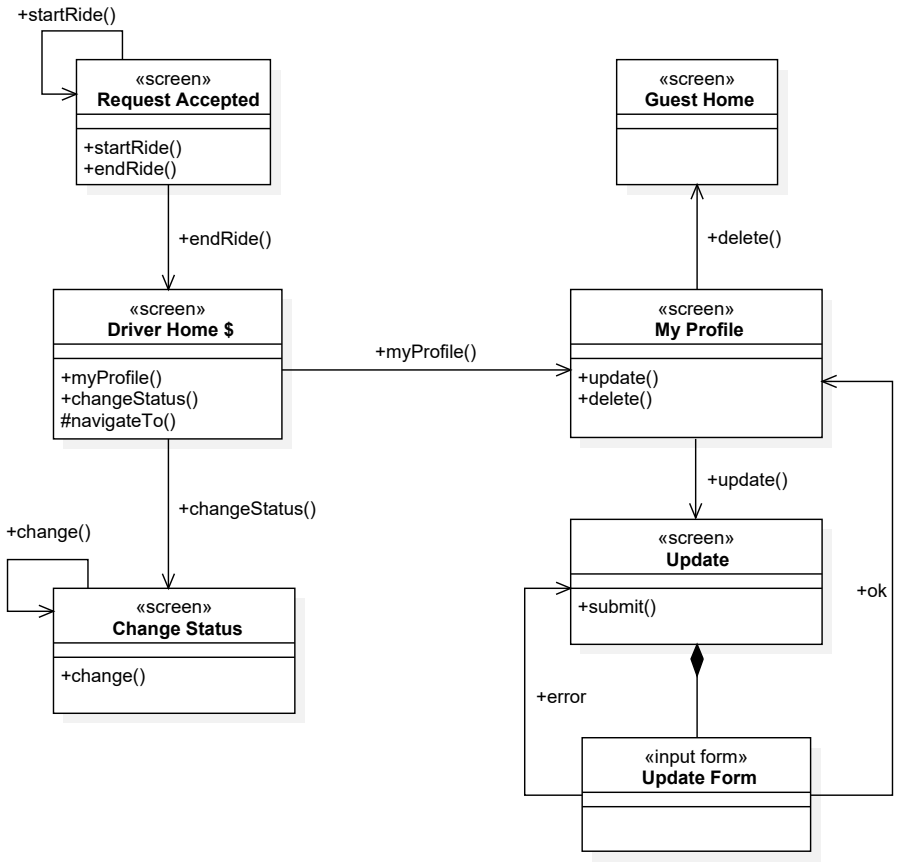


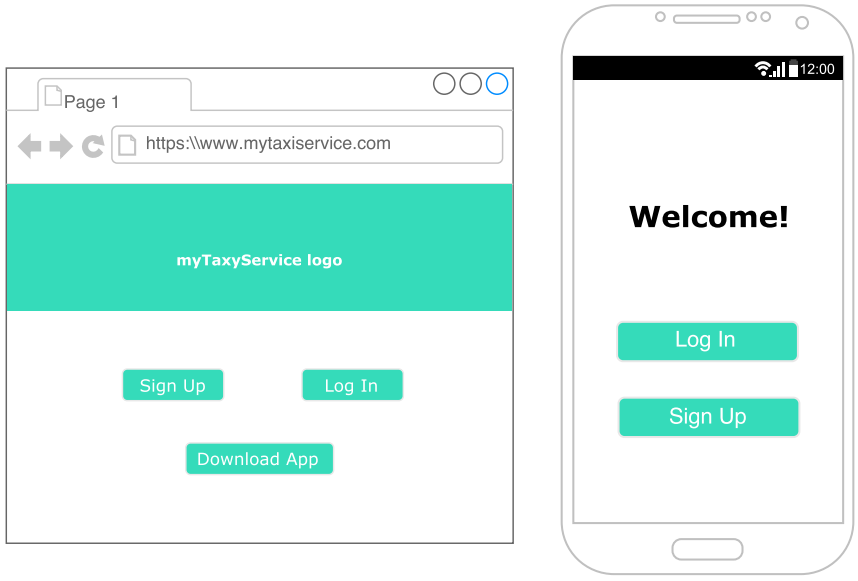
Figure 17: driver application UX

4.2 Mockups

In this section the mockups of both the web application and the two mobile applications are presented. It is important to stress that the two mobile applications will be different, one for the passengers (*myTaxiApp*) and one for the drivers (*myTaxyDriver*). Nevertheless, they will have a similar look and some functionalities will be in common. Both type of user will in fact be able to register to the system, log in and manage their profile. For this functionalities only a single mockup is provided, since the design will be the same.

4.2.1 Guest Home Page

Figure 18 represents the main page accessible to guests. Since the system is accessible from mobile devices too, the web application displays a link to the official download page on the store for the mobile apps.



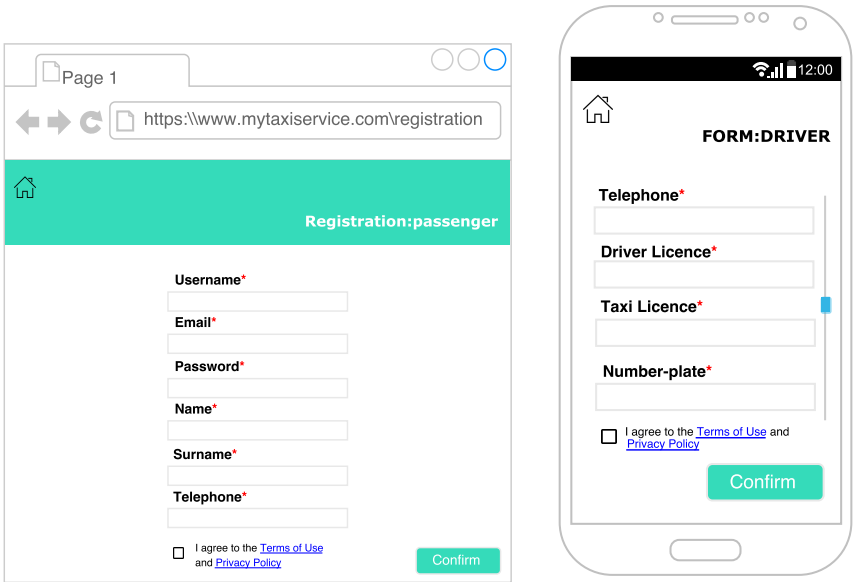
(a) Web (b) App

Figure 18: Guest home page screen

4.2.2 Registration Page

In order to subscribe to the system the guest needs to fill a form. If the guest accesses this functionality from the web application, the system first asks him/her to choose the type of account: passenger or driver. As the guest selects the account type he/she is interested in, the system loads the corresponding form.

Since the mobile application for drivers and passengers are two separate entities, this preliminary step is skipped, and the correct form is displayed immediately. In order to complete the registration the guest must fill all the mandatory fields, indicated by the red star, and agree to the Terms of Use and Privacy Policy. Once the guest clicks on the confirm button he/she submits the data to the system that proceeds to validate them. If any problem is detected an error is displayed and the guest needs to recompile the form. On the other hand, if the data is correctly validated a message is displayed to remind the guest that he/she needs to activate his/her account by clicking on the link provided in the email the system sent to the provided email account.



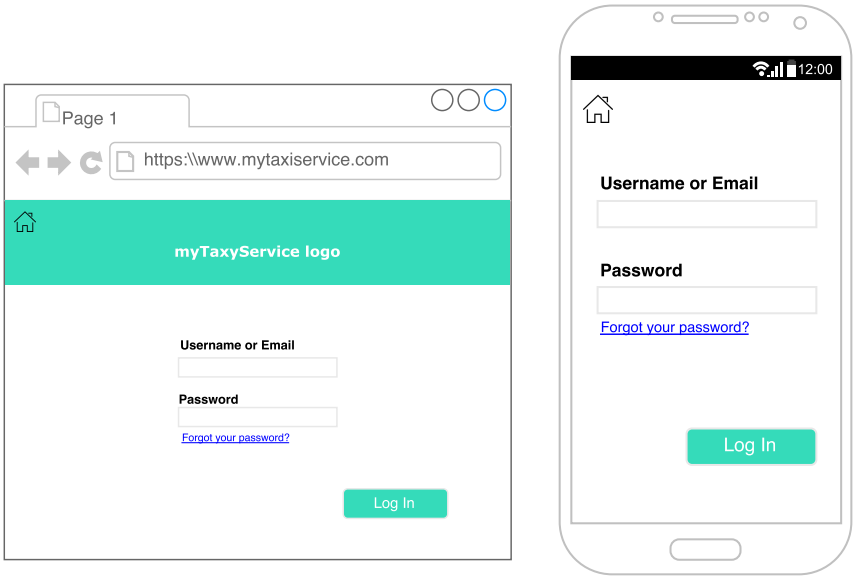
(a) Passenger form on website

(b) Driver form on mobile

Figure 19: Registration screen

4.2.3 Log In Page

The Log in page allows users to log in to the system. Users have to insert their username or email and associated password. If the credentials are wrong an error message is shown, otherwise the user is redirected to the home page. The system offers the possibility to retrieve the password, by clicking on the "forgot your password" link. If this option is selected, the system will ask the user's email account and send an email to it with a temporary password. The user can then use that password to log in to the system and will be asked to set a new password.



(a) Web (b) App

Figure 20: Log in screen

4.2.4 My Profile

Users are able to visualize their profile and their data. They can update it or delete it. If the delete option is chosen, the system asks the user to confirm the operation and insert the password before deleting the account.

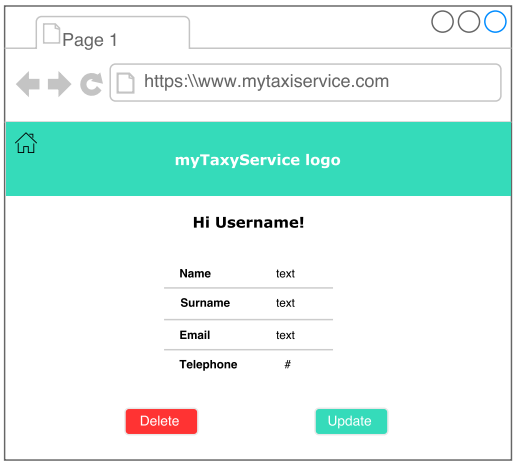
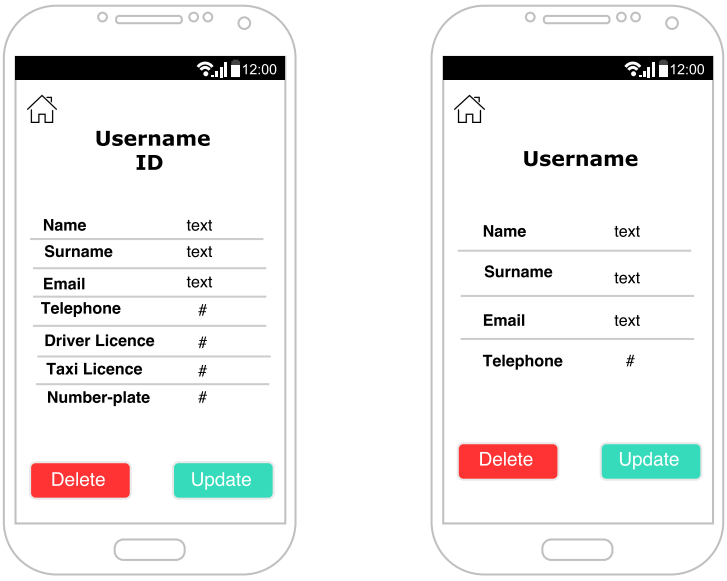


Figure 21: Passenger's profile on website

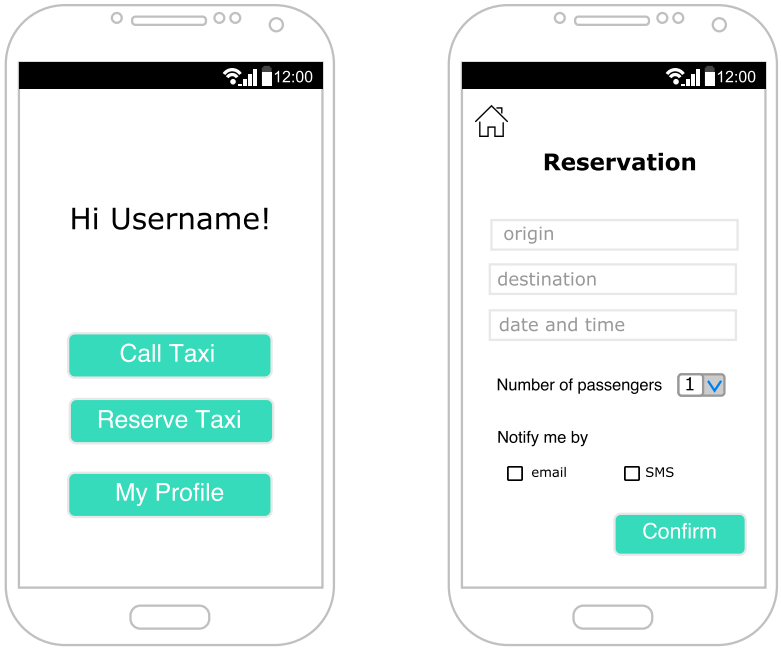


(a) Driver (b) Passenger

Figure 22: My Profile page on mobile app

4.2.5 Passenger Pages

From their home page, passengers can access all the functionalities offered by the system. In the *Call Taxi* page the passenger needs to specify the pick-up address and the number of passengers expected. As stated in the RASD, all taxis have a maximum capacity of four sets, therefore the passenger can choose the number of sets from a preset selection. If the passenger is using myTaxyApp he/she can choose to activate the localization via GPS(24b). The application shows the detected position and the passenger can then choose to use that position or manually insert a new one, whereas in the reservation page the passenger has to specify the pick-up point as well as the destination, the number of passengers, the time and date of the pick-up. In both pages in order to submit the data the passenger has to click on the confirm button.



(a) home page (b) Reservation

Figure 23: pages on the mobile app

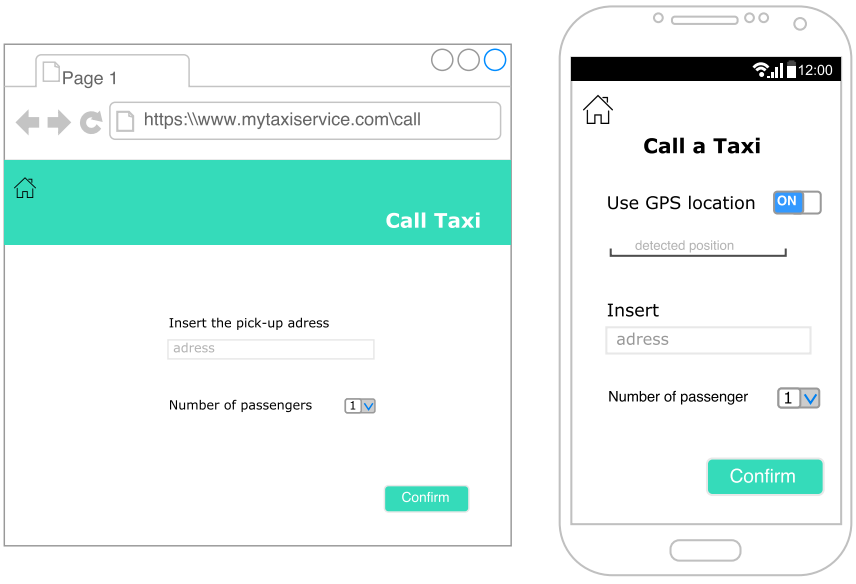
The active request page displays all the info of the current booking, as passengers are able to have a single active request at a time. The request's info are the following:

1. Status: indicates whether the system is still searching for a taxi (*pending*)

or a taxi has already been assigned to the ride (*on the run)*

2. Taxi ID: the ID of the assigned taxi driver, if the request is still pending this filed is set to unknown
3. Origin: the pick-up point
4. Destination: if the request was a reservation, the destination's address is displayed, otherwise this field is set to unknown
5. Date: displays the pick-up time for the reservation, and the time the request has been accepted for the call
6. ETA: the estimated time of arrival is displayed as soon as a taxi is been assigned to the ride. This field is periodically updated.

If the passenger has no active request the page displays the message "no active request available".
The Reservation page is similar to the Call Taxi page, but the passenger has to specify other fields such as the destination, the pick-up time and date, and how he/she will like to be notified as soon as a taxi has been allocated. For the reservation the option to automatically locate the pick-up point from the GPS is not provided.



(a) website (b) app

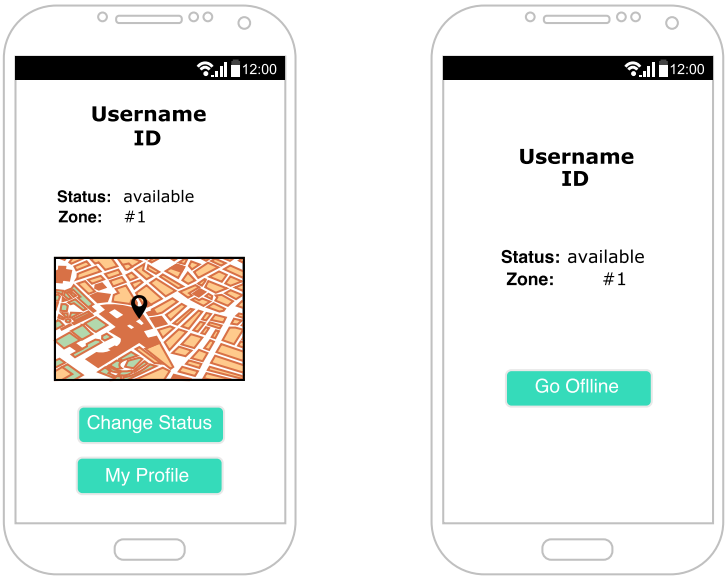
Figure 24: Call taxi pages

Figure 25: active request info

4.2.6 Driver Pages

In the driver's home page drivers are able to see their current status, position and zone. From the same page drivers can access the *change status* page and *my profile* functionalities.

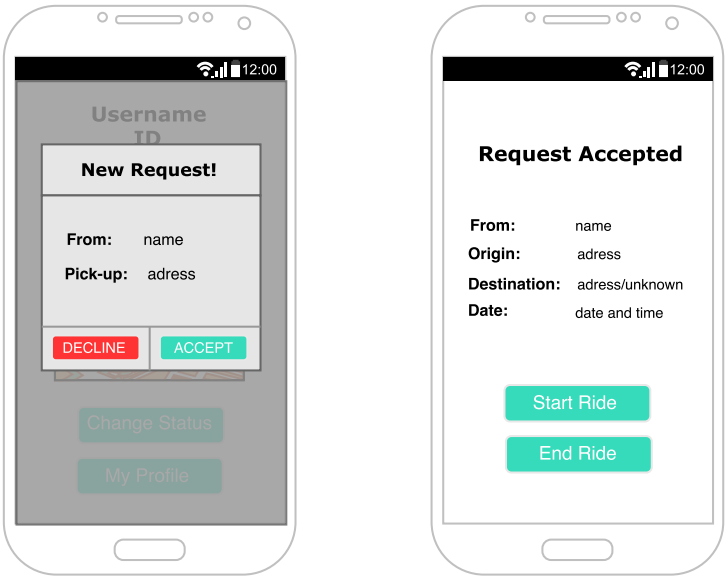
The *Change Status* page allows the driver to update his/her status: if the driver is "available" a "go offline" button is displayed, whereas if the driver's status is set to "offline" a "start shift" button is displayed. When the system selects a driver for a ride request, the driver is asked to respond with a pop-up notification, that shows the request's details. The request accepted page is shown only after a driver has accepted a ride. This page shows the request info and allows the driver to notify the system when he/she has picked up the customer and when he/she has finished the ride.



(a) Driver home page

(b) Change status page

Figure 26



(a) new request notification (b) ride

Figure 27: Ride request

5 Requirements Traceability

All design decisions presented in the previous sections were made taking into account the requirements identified in the RASD. Figure 28 shows the specific section dedicated to each functionality of the system. For every functionality that requires the user or guest to input data, the system always validates it before accepting it, and notifies the user or guest if any problem is found and the validation process didn't succeed.

Function	**RASD**	**DD**
sign up	3.3.2	1.5.1
log in	3.3.3	1.5.2
account managing	3.3.4	1.5.3, 1.5.4
call taxi	3.3.5	1.5.5, 3.2
reserve taxi	3.3.6	1.5.6, 3.2
ride notification	3.3.7	3.2
update status	3.3.8	1.5.8
active request	3.3.9	1.5.7

Figure 28: mapping of system's function

The queues have been designed to optimally manage the ride requests and the decision to include neighboring zones for the search has been made in order to guarantee the successful allocation of a taxi. This is particularly important in the case of a reservation, since if the allocation process fails passengers may perceive it as a critical problem of the system, that can't even provide a taxi even if it was booked hours before the pick-up time, and may choose to stop using the *myTaxiApp* or website. Moreover the algorithm specified in section 3.2 does not send the same ride request to a driver more than once, thus reducing the allocation time and avoiding the worst case scenario were a small set of drivers that have already declined the request keep receiving it. Nevertheless, the allocation process can fail, and if that happens the passenger is promptly notified of the problem, within the maximum waiting time of five minutes.

User interfaces have been designed to be as intuitive and simple as possible. Both passengers and drivers should be able to quickly understand how to access the functionalities of the system without any training. From every page of the selected application users can navigate back to the home page by clicking on the home button displayed in the upper left corner of the screen.

A Appendix

A.1 Software and tools

1. *Lyx* to redact and format the document
2. *StarUML* for the diagrams in chapter 2
3. *draw.io* for all the mockups and algorithm flowchart

A.2 Hours of work

- Carolina Beretta ~ 25h
- Cecilia Brizzolari ~ 25h