Politecnico di Milano

Scuola di Ingegneria dell'Informazione

Corso di Laurea Magistrale di Computer Science and Engineering

Software Engineering 2 Project
myTaxiService
Part II : DD
(Design Document)

Principal Adviser:
Prof. Raffaela Mirandola

Authors:

Turconi Andrea ID n. 853589

Raimondi Lorenzo ID n. 859001

Accademic year 2015-2016

Contents

1 Introduction

1.1 Purpose

This document represents the Design Document (DD). The aim of a Design Document is to describe the software design and architecture and to motivate the relative choices that have been made. This goal is accomplished by means of a description of the components which act in the system, with their relationships and interactions, using both verbal presentation and graphical modeling. This document is intended to application developers, who can so have a guidance over the system implementation.

1.2 Scope

The software to develop is myTaxiService, an application aimed to satisfy citizens' taxi requests. The system will help users to call a taxi on a desired time and location, and will help taxi driver to get a more sharp and suited service. The application can be accessed either via a web browser or with a mobile application compatible with most common mobile OSs, and requires a registration in order to use the service. Users can book a taxi in a specific location, either for an immediate request or for a reservation booking. Taxi drivers instead are disposed in queues, relative to a certain city zone; the queue allows to gain an order of drivers so that the system can forward a request with a fair and smart management. All the communications about useful informations like requests, response, time, are forwarded from and to the users by a notification system.

1.3 Definitions, acronyms and abbreviations

1.3.1 Definitions

- Request a taxi: send a request to the system of any type of taxi service, both reservations and demand
- Demand a taxi: send a request to the system of an immediate taxi service
- Reservation of a taxi: send a request to the system of a taxi service for a certain time, specified in the form
- Future reservation: a reservation that has not yet been executed
- Pending request: a request that has been sent by the use but not managed yet by the system.

- Accepted request: a request received and managed by the system, which has already accepted it, but has not yet selected a driver for fulfilling it.
- Ongoing request: a request that is being accomplished right at this moment; this time span goes from the driver selection instant to the user's arrival in the desired destination.
- Completed request: a past request, that has been completely performed.

1.3.2 Acronyms

- RASD: Requirement Analysis and Specification Document
- DD: Design Document
- GPS: Global Positioning System
- API: Application Programming Interface
- SOA: Service Oriented Architecture

1.3.3 Abbreviations

- [Gn] n-goal
- [Rn] n-functional requirement

1.4 References

- Specification document: assignment 1 and 2.pdf
- Requirement Analysis and Specifications Document v2
- Design Document Template.pdf

1.5 Document structure

This Design Document is composed of five sections. The first section introduces the document by giving basic information. In Section 2 is described the system architecture and design, as well as components and pattern used. In Section 3 the focus is on the algorithm design, with the implementation of some important algorithmic system feature. In Section 4 is provided an overview of the system-to-be Graphical User Interface. In the last section is described the requirements traceability, so the relationships between system requirements and components which fulfill them.

2 Architectural Design

2.1 Overview

In the Architectural Design section is presented how the system is composed, by means of a high-level description of all its components and focusing also on their interaction. This aim is achieved using a verbal description and meaningful UML models. Alongside this topic, this section includes also the presentation of the architectural style chosen for the system infrastructure, centering on patterns, styles, and the motivation which brought to this choices.

2.2 High-level Components and their interaction

2.2.1 User Interface Component

Regarding Graphical User Interface, there are three different subcomponents: UserAppGUI, TaxiDriverAppGUI, and WebGUI. The first is in charge of manage the interaction via mobile application for the Users. The second one is in charge of manage the interaction via mobile application for the taxi drivers. The last one instead is in charge of manager the interaction of the users via web. Each subcomponent is associated to a different Communicator, located in the client. Those components communicate with the Message Dispatcher and the Message Receiver, located in the logic application.

2.2.2 Communication Manager

This component is composed of two subcomponents: Message Dispatcher and Message Receiver. The first component is in charge of send messages and notifications to the clients and to the taxi drivers. An example of a message to the user is a notification where it is specified that the request has been accepted by a taxi. The message also shows the number of the taxi and the waiting time. To a taxi driver, the Message Dispatcher could send a message where there is the request of a ride from a user, where it is specified the position of the user. The second component is in charge of receive messages from clients and taxi drivers. An example of message that this component could receive from a user is a message where the user requires a ride from a specific position and for a specific hour. From taxi drivers, this component could receive a response message of a request, where the taxi driver accepts or rejects that ride. Message Dispatcher and Message Receiver communicate with the Request Manager and, according on the type of message received, this component communicates with the properly view Component. The Message Receiver communicates also with

Register Manager and Login Manager, when it receives special messages as to create a new account or to login.

2.2.3 Account Manager

This component is composed of two subcomponents: Register Manager and Login Manager. The first one is in charge of manage the registration from the visitors. This component receives a request of creation of a new account. Before accepting this request, this component sends all the field to the Data Component to compare them with the database. If it is all correct, this component is in charge of send to Data Manager the request of creating a new account, with all the information provided in the registration form. The Login Manager instead is in charge of send to the Data Manager the information inserted in the login form, with the request of searching an existing account with this information. After the response message, this component allows or not allow to the visitor to reach his personal page. Those two components communicate with the Data Manager in the way before described and with the Message Receiver.

2.2.4 Data Manger

This component is in charge of communicate directly with the data stored in the database, through the DBMS. It receives request of query from the Account Manager, Queue Manager, and Request Manager. It is a sort of intermediate layer between the logic application and the database.

2.2.5 Request Manager

The Request Manager is the system core component because it fulfills all the incoming ride requests. This component is in charge of elaborate the message received by the Message Receiver and call the Message Dispatcher to send the response of the request. Once received a request message, this component elaborates the request: from the input request informations, it interacts with the Queue Manager in way to find a driver that is eligible for the current request. Once the driver has been found, by means of the Message Dispatcher, it sends him a notification: in case the driver accepts the request, the Request Manager receive the confirmation from the Message Receiver, and with the Data Manager stores the request with all its attributes in the database, or in case the driver refuses, re-schedule the request for the subsequent driver. A very important Request Manager feature is relative to Reservation requests: this component triggers automatically the research and notification of a taxi driver to accomplish the request, by means of the Queue and the Communication managers. This component communicates with the Message Receiver, Message Dispatcher, Data Manager and the Queue Manager, in the way before described.

2.2.6 Queue Manager

The Queue Manager is the component that ensures a fair management of taxi drivers with a queue-based system. This component in fact assigns each available taxi driver to a queue basing on his real location. So every time a taxi driver turns available, after the login or a ride, the Queue Manager has to insert it in the correct queue. Also when a new request is accepted by the system, this component finds a taxi driver to fulfill it, by searching the first driver in the correct queue (so the queue whose zone contains the request starting location). In order to avoid inconsistency between drivers position in the real word and their location in queues, before selecting a driver for an incoming request the queues are realigned by this component. Once a driver accepts a request, the Queue Manager remove him from the head; It's very important that when removed from the queue, the driver is not appended to another one: he will be added only once he finished the current ride and turns available again. This component communicates with the Location Manager and Request Manager. It also communicates to the Data Manager to recover the data about taxi and queues.

2.2.7 Location Manager

The Location Manager acquire, manages and offers all the data referred to positioning and location. This component interacts with Google Maps API in way to elaborate users position and to map GPS coordinates with house numbers. Using Google APIs the Location Manager is able to compute shortest paths and waiting time, also considering city traffic conditions. All this informations are elaborated and made accessible to the other system components. The Location Manager communicates with the Queue Manager and the Request Manager, with the scope to insert a taxi in the correct zone and to communicate to the client an approximate waiting time based on the location of taxi and client and the traffic in that moment.

2.2.8 Taxi Manager

The Taxi Manager component handles all the operations regarding driver management. This component has to manage driver's availability, receives updates on their status by means of the Dispatcher and interacting with the Data Manager updates the database. So this components get active every time a driver starts or ends his service, and in particular, when a driver signals any unforeseen; in this case the update is not trivial as before, in fact this component have to update the Request Manager, that will have to forward a new request for a taxi driver, in way to solve this problem.

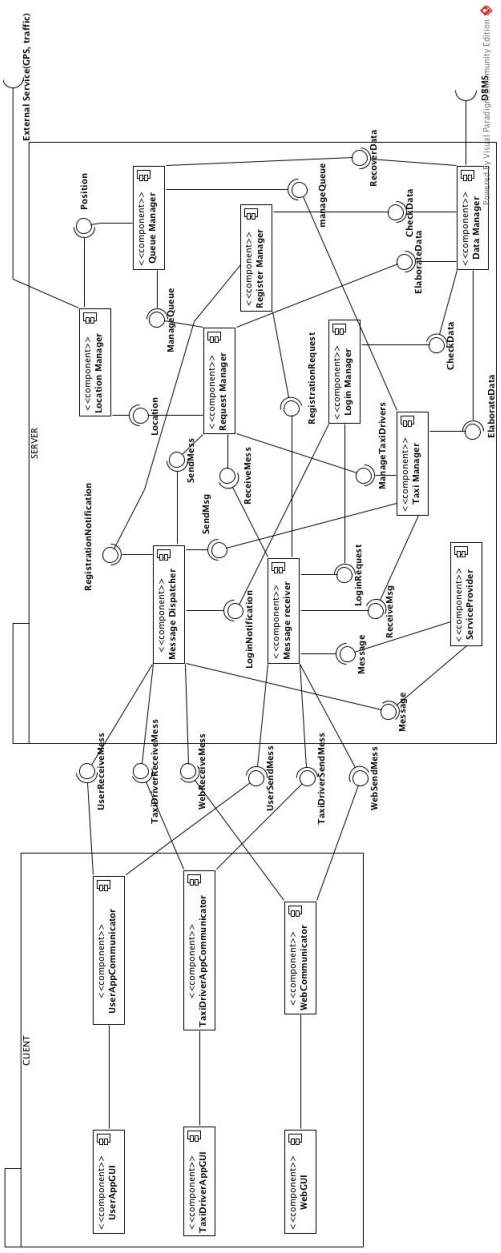
2.2.9 Service Provider

The Service Provider is the component that allow external applications to have access to the system services. It provides the chance to use system core logic and

data, in way to implement some functionalities and extend system features. As we imagined it, this component could be as a sort of Java interface, by which can be obtained methods which interact with the system. This middle component is placed between external applications and internal services, so that a filtering action on external services requests can be implemented by it.

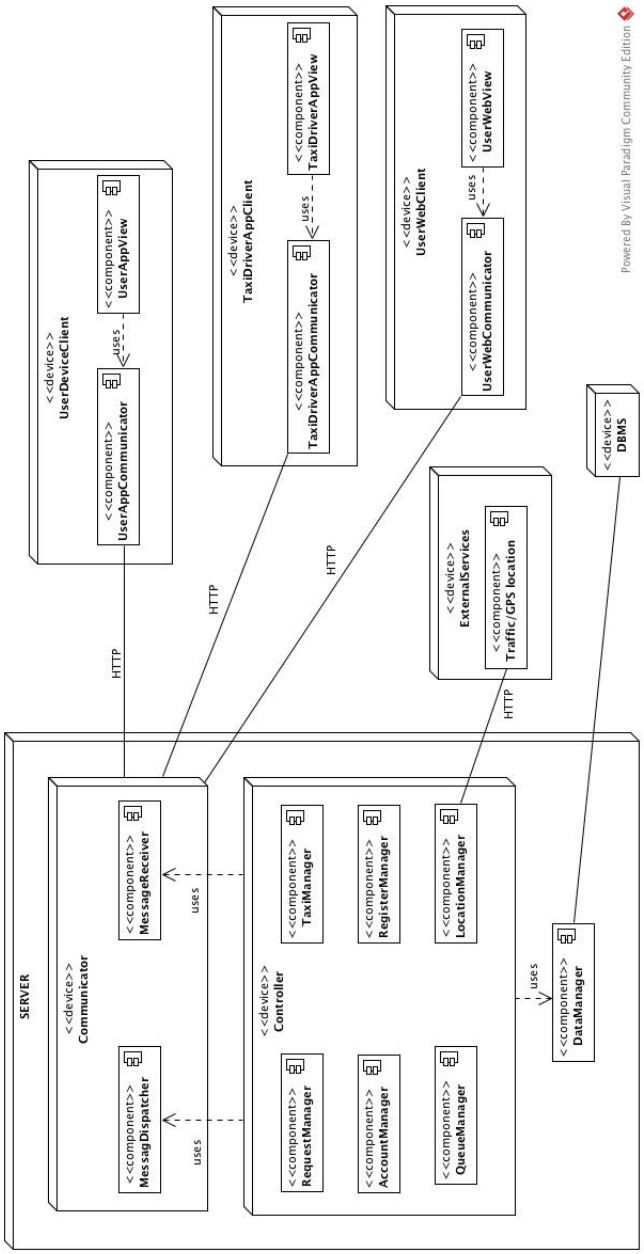
2.3 Component View

Here is presented an UML Component Diagram to graphically describe the components presented before and their interactions.



2.4 Deployment View

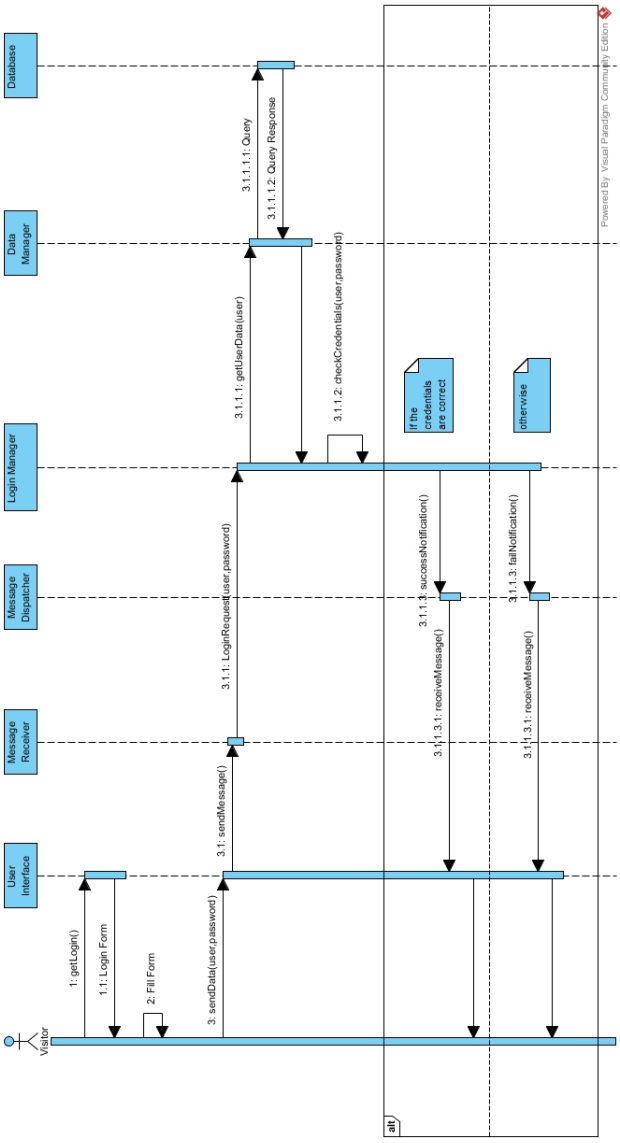
In this section an UML Deployment Diagram shows statically how software and hardware components are related.



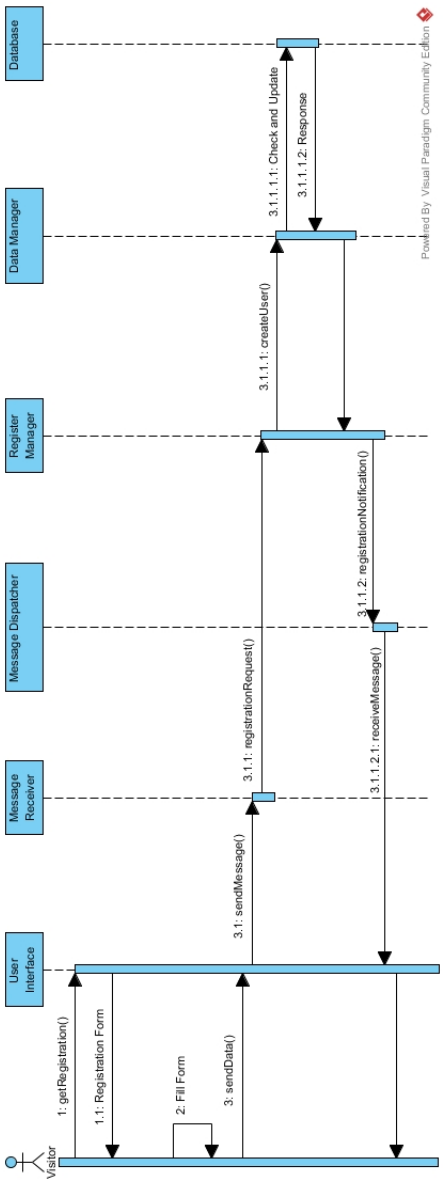
2.5 Runtime View

Regarding runtime processes and interaction, here are presented several different UML Sequence Diagrams in way to show how and when components interact. We chose to extend and detail the processes already presented in RASD.

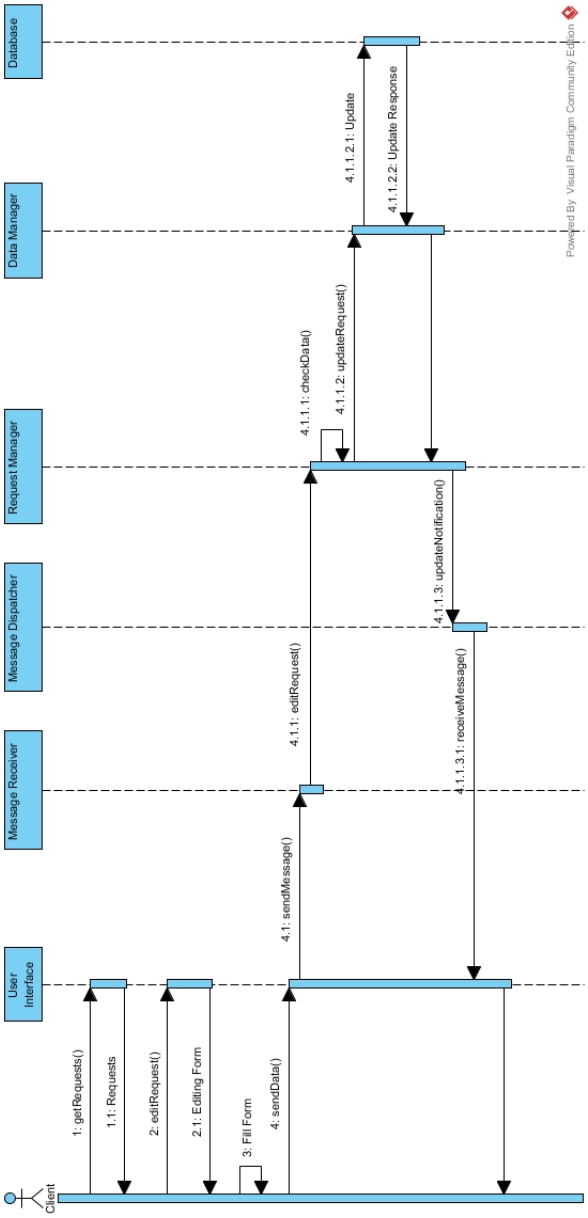
Login



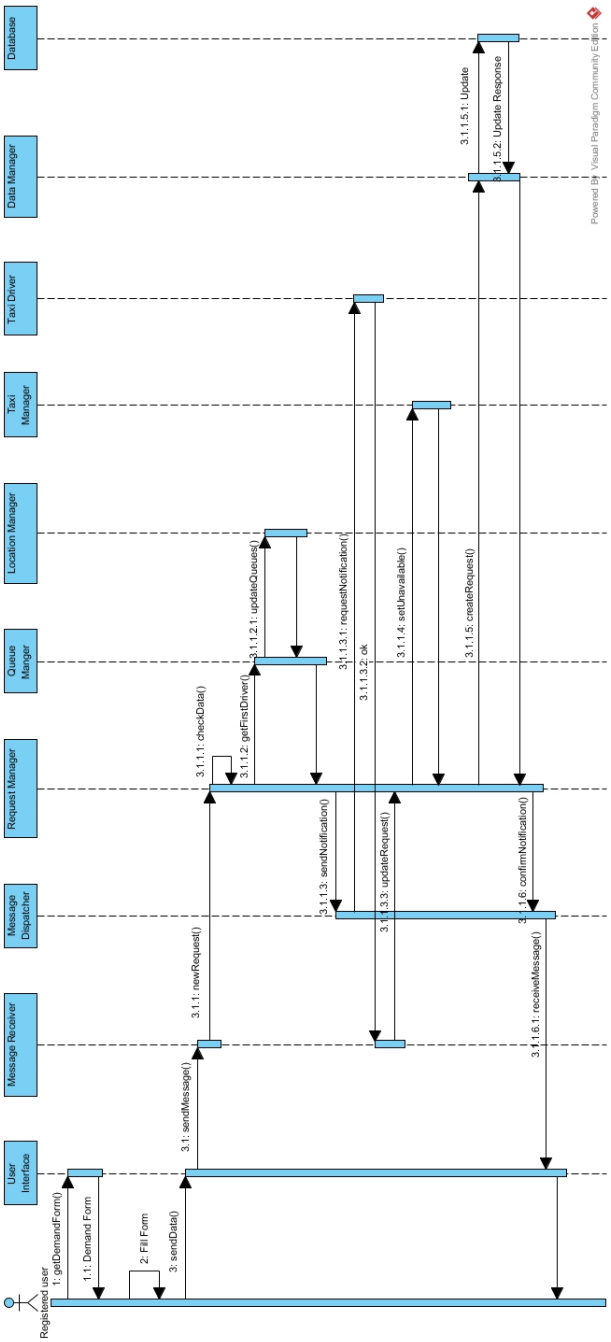
Registration



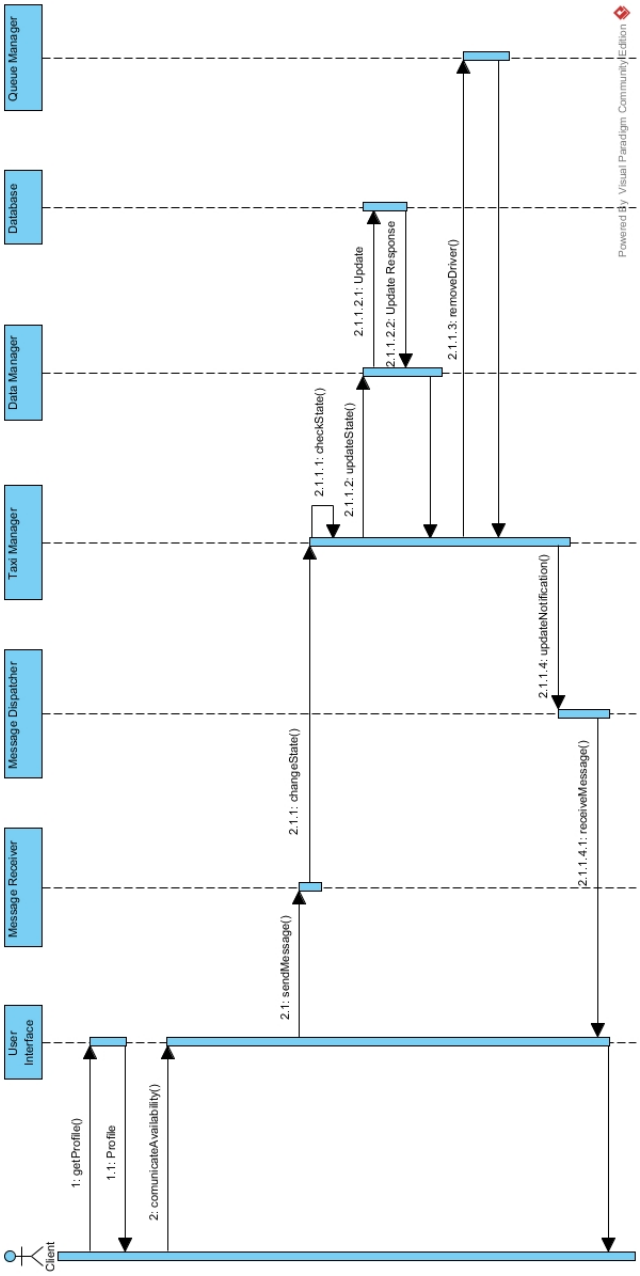
Edit request



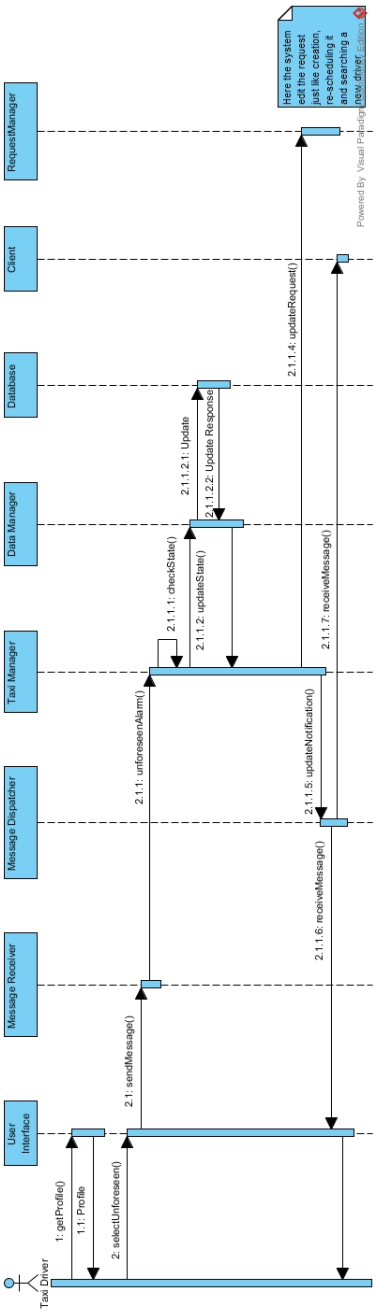
Request forwarding



Taxi Driver availability management



Unforeseen management



2.6 Component Interfaces

In this section are described, for each component, the most important interfaces and functions which allows information handling and data managing between the various system parts.

User Interface components

`receiveMessage()`

allows to receive messages from the system, about requested services. In this way the view components can be updated by the controller ones, according to MVC pattern.

`sendData()`

allows to input informations to the user interface, so that it can forward them to the system.

Communication Manager

`sendMessage()`

allows to contact the system and request any of its services, so that will be forwarded to the specific component.

`notificationMessage()`

allows the system to contact the user by sending him a notification message.

Account Manager

`loginRequest(username, password)`

allows the request of login by sending to the component the access informations given by the user.

`registrationRequest(data)`

allows to register a new user with informations entered in the registration form.

Data Manager

`createUser(data)`

allows the storing of registration data of a new user in the database.

`getUserData(user)`

allows to access user information stored in the database by means of a key value.

`createRequest(data)`

allows to store a new user request with all its related informations.

`updateRequest(data)`

allows to edit database entries after a user request editing.

`updateState(id)`

allows to update and save drivers availability lapse on the database.

Request Manager

`newRequest(data)`

allows to forward, create and elaborate a new user request

`editRequest(id, data)`

allows the editing of a certain Reservation Request with new data introduced by the user.

`updateRequest()`

allows to inform the manager about external-triggered events, like drivers accept/reject or unforeseens.

Queue Manager

`getFirstDriver()`

allows to get informations about the first driver of a specific queue.

`addDriver()`

allows to add a driver at the end of a certain queue, the one relative to his location.

`removeDriver()`

allows to remove the driver from the head of the queue of his queue.

Location Manager

`updateQueues()`

allows the other components to request a refreshing on driver location, in way to update queues.

`travelTime()`

allows the system to get a precise timing about taxi movement, in way to provide waiting time estimation.

Taxi Manager

changeState(id)

allows to change availability driver state, from unavailable to available and viceversa.

unforeseenAlarm(id)

allows to inform the system of an unforeseen that hit a specific taxi driver, so that the relative request can be reassigned.

Service Provider

For this component is not provided a specific interface description. This component in fact can offer all the system services to external application, so we can consider that own all the interfaces described above. From the internal point of view instead this component interact with the system by means of Communication Manager, so as just as the User Interface components.

2.7 Selected architectural styles and patterns

As regards to the system architecture, the chosen style is a 3-tiers Client- Server Application, pointed out here:

- A Database Management System hosted on a dedicated server, where is stored all the useful data about taxi drivers, clients, and requests.
- An Application Server, running the software logic components.
- A Web Server hosting the application web interface

This architectural design choice derives from different considerations on the system behave and requirements. In terms of performance, a 3-tiers architecture ensures a more customizable and scalable system: it allows to use data caching on different layers, from the web server to the database; in case of changes in application requirements, for example the expansion to hinterland covering of the service, the system can be easily scaled using parallel application servers, also useful to grant high service availability. For what concerns security, a 3-tiered architecture allows to define a more secure system, in which alongside with firewalls, the applicative layer act as a further protection, not allowing a direct access to data and user's personal informations in case of web attacks.

MVC Pattern The basic design choice is to use the MVC pattern to manage software components. This pattern aims to divide application into three parts: the Model, which stores and manage the application data; the View, that provides interface elements for user interaction; the Controller, which receives inputs, manage them, and provide Model information in a suitable form for the View. According to this pattern, the Model maps with DBMS and Data Manager components, used to store, elaborate, and access data; the View

matches with UserAppGUI, TaxiDriverAppGUI and WebGUI components, responsible for user interface; the Controller maps with all the other components, which receive inputs, elaborate them and control the application querying data from the model. MVC pattern is really helpful for decoupling system services in a smart and organized way, supporting cohesion and function reusing.

Publish/Subscribe Another architectural choice is to provide the Communication Manager of a Publish/Subscribe implementation, for what concerns the handling of asynchronous messages from the server. This feature allow the system to contact the clients in case of notifications and warnings, while a standard client-server architecture would be unable to achieve this result. In this way the clients in fact are subscribed to a publisher, and every time it publishes something new, the new message will be downloaded by the client. The publish/subscribe system is topic-based, so the subscribers would be connected only to interesting publishers; this allow to create topic for a request, connecting the relative driver and client, but would also allow for example to create customized zone-based messages, increasing the communication system scalability and flexibility.

Service Oriented Architecture Another important decision is to adopt a Service Oriented Architecture by means of system components. All service requests reach the system passing from the Communication component, which is responsible for their forwarding to the proper component. This permits a better system organization and allows an easier and standard communication between clients and the system. In particular, we chose to create an ad-hoc component for providing system services to external applications, because it allows a more sharp control of external service requests, like the chance to filter them, or setting a customizable limit on requests, independently from the regular service access.

3 Algorithm Design

3.1 Request management

In the RequestManager class arrive a request from the MessageReceiver. The algorithm extracts the identification number of the user and calls the function 'getFirst' of the QueueManager. The calling of this function has, as parameters, the identification number of user and the result is a TaxiDriver's instance, that is the first element of the queue related to the user's zone. For each request received, it creates a thread that manage this request with the following algorithm. Each thread creates a boolean variable set as 'false' and until this variable is false, it create a message and sends it to taxi driver returned by 'getFirst', waits for the response of this taxi driver and if the response is false, the thread call the function 'getNext' in QueueManager, that returns the next taxi driver in the queue, else exits from this loop. So when the response is 'true', the thread creates a mapping in the database, through the DataManager's function, where the user is associated to a taxi driver and all the information od the user's request. Then calls a LocationManager's function that from user's position and taxi driver's position, calculates the waiting time based on the traffic. User's position and taxi driver's position are obtained by a QueueManager's function. So the thread creates a message to send to the user with the waiting time and taxi's code. Finally, call a TaxiManager's function, to set this taxi driver as 'unavailable'. 'getFirst' and 'getNext' are two different function, that will be specified in the Queue Algorithm.

```
1
2  public class RequestManager{
3      //....
4
5      Message in=MessageReceiver.receive();              //receive a generic message of a request
6      int id=in.id();
7      TaxiDriver taxi=QueueManager.getFirst(id);
8      int taxiID=taxi.getId();
9
10     //For each request it create a new thread that manages it with the following code
11
12     Boolean msg=false;                                 //create the response message as 'false'
13      while(msg!=true){
14         Message mess=Message();                        //create a message to send to the taxi driver
15         MessageDispatcher.send(mess,taxiID);           //send the message to the taxi driver
16         msg=MessageReceiver.response(taxiID);          //waiting taxi driver's response
17         if(msg!=true)
18             taxiID=QueueManager.getNext(id).getId();   //save the new taxi's code
19     }
20
21     DataManager.createRequest(id,taxi,in.startingTime,StartingPosition,ArrivePosition);
22                                                        //create a map between user and taxi assigned
23
24     Time waitingTime=LocationManager.traffic(QueueManager.getPosition(id),QueueManager.getPosition(taxi));
25                                                        //calcolate waiting time with functions of QueueManager
26
27     Message note=Message(taxiID,waitingTime);          //create a message with taxi's code and waiting time
28     MessageDispatcher.send(id,note);                   //send the notification to the user
29
30     TaxiManager.setUnavailable(taxiID);                //set taxi as unavailable
31
32     //....
33 }
```

3.2 Queue management

In the QueueManager we have two important functions: 'getFirst' and 'getNext'. Those two functions have only one parameter: the zone. In each function, the first two statements are the same. Those statements create an empty list of taxi drivers, where to save the queue, and, through the LocationManager, obtain the user's zone. Each function returns a taxi driver.
In the 'getFirst', the function takes the queue related to the zone before obtained from the DataManager. It also takes the list of all taxi drivers stored in the database. Then, for all the queue saved, checks if each elements' zone is equal to user's zone. If this check is negative, it eliminates this element from the queue, through a private function of the QueueManager. After these operations, the algorithm checks for all the other taxi drivers, if their zone is equal to the user's zone and if they are available. If this check is positive, inserts those taxi drivers in the queue. So save the queue in the database and returns the first element of this new queue.
In the 'getNext', the function takes the queue related to the zone before obtained. So it moves the first element at the end of the queue, moving all the other element in one position forward. Then saves the new queue in the database and returns the first element of this queue.

```
1
2  public class QueueManager{
3      //....
4
5      public int getFirst(int i){
6          List queue=ArrayList(TaxiDriver);                    //empty list of taxi drivers
7          Zone zona=LocationManager.getZone(i);                //get the zone of the client
8
9          queue=DataManager.getQueue(z);                       //get the queue associated to zone z
10         List taxies=DataManager.getTaxies();                 //get the list of all taxi drivers
11         for(TaxiDriver c1:queue){
12             if(LocationManager.getZone(c1)!=z)               //check if taxi drivers in the queue don't
13              eliminateTaxi(c1,queue);                        //belong to zone z and eliminate them
14         }
15         for(TaxiDriver c2:taxies){
16             if(LocationManager.getZone(c2)==z && DataManager.getAvailability(c2)==true)
17                                                              //check if taxi drivers not in the queue
18                 insertTaxi(c2,queue);                        //belong to zone z and insert them
19         }
20     DataManager.saveQueue(queue,z);                          //save the new queue in the database
21     return queue.get(0);                                     //return the first element
22     }
23
24     public int getNext(int i){
25         List queue=ArrayList(TaxiDriver);                    //empty list of taxi drivers
26         Zone zona=LocationManager.getZone(i);                //get the zone of the client
27
28         queue=DataManager.getQueue(z);                       //get the queue associated to zone z
29         TaxiDriver t=queue.get(0);
30         eliminateTaxi(t,queue);                              //eliminate the first element and
31         insertTaxi(t,queue);                                 //insert at the end of the queue
32         DataManager.saveQueue(queue,z);                      //save the new queue
33         return queue.get(0);                                 //return the actual first element
34     }
35     //....
36 }
```

3.3 Taxi Driver availability

In the TaxiManager class, when arrives a request to set a taxi driver as unavailable, the class call the 'setUnavailable' function. This function takes as parameter the taxi's code as integer. Then checks if the availability of the taxi driver is set as available. So calls the specific function of the DataManager to set the variable availability, of the taxi driver, as 'false' in the database. Then takes the queue from the database that contains the taxi driver and eliminate him. This function doesn't return a value.

```
1
2  public class TaxiManager{
3      //....
4      public void setUnavailable(int taxi){
5          if(DataManger.getAvailability(taxi)==true){          //if the taxi driver is available
6              DataManager.getTaxi(taxi).setUnavailable();      //set him as unavailable
7              List queue=DataManager.getQueue(taxi);           //obtain the queue contains this taxi driver
8              QueueManager.eliminateTaxi(taxi,queue);          //and eliminate him from the queue
9              DataManager.saveQueue(queue)                     //save the new queue
10         }
11     }
12     //....
13 }
```

3.4 Unforeseen management

In the TaxiManager class, when it receives an unforeseen message from a taxi driver, the algorithm calls the 'manageUnforeseen' function, that it's private. The function checks if the taxi driver is unavailable. In case of positive result, the function eliminates the map between taxi driver and user from the database and calls the RequestManager to allocate another taxi driver for that user. In case of negative result, the function sends to the taxi driver an error message, through the MessageDispatcher.

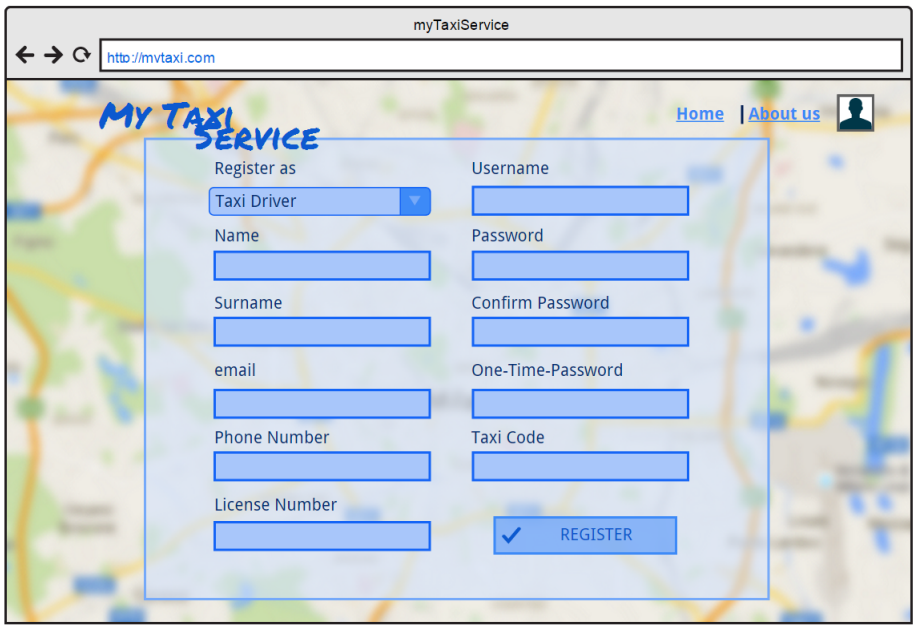
```
1
2  public class TaxiManager{
3      //....
4      Message in=MessageReceiver.getMessage();
5      if(in instanceof UnforeseenMessage)                    //if is a unforeseen message
6      //....
7      int taxi=in.id();
8      manageUnforeseen(taxi);                                //the call of a private method
9          //....
10     //....
11
12     private void manageUnforeseen(int taxi){
13         if(DataManager.getAvailability(taxi)==false){      //if the taxi driver is unavailable
14             DataManager.eliminateRequest(taxi);            //eliminate the association between taxi and user
15             RequestManager(id);                            //the call of RequestManager to allocate another
16                                                            //taxi to the user id
17         }
18         else
19             MessageDispatcher.send(taxi);                  //if the taxi driver is available, send to him an
20                                                            //error message
21     }
22 }
```

4 User Interface Design

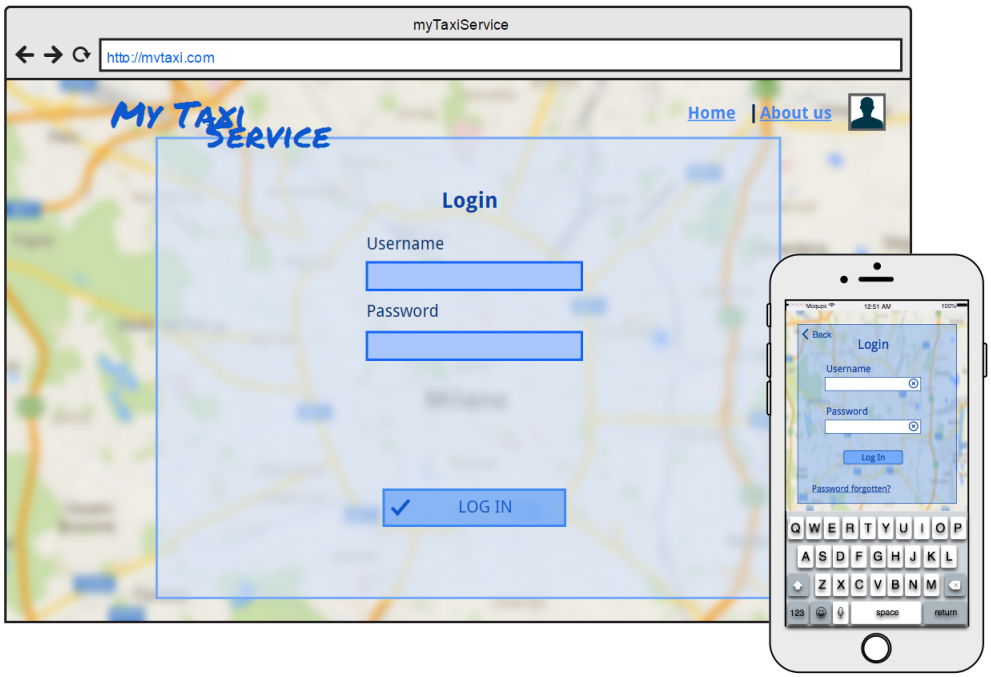
4.1 User interfaces

Here are presented some mockups which represent as we projected the Graphic User Interface. As already stated in the requirement document, we focused on create an intuitive and coherent interface among different compatible devices. To integrate RASD user interface description we introduced new mockups and sorted them as they occurs during application use.

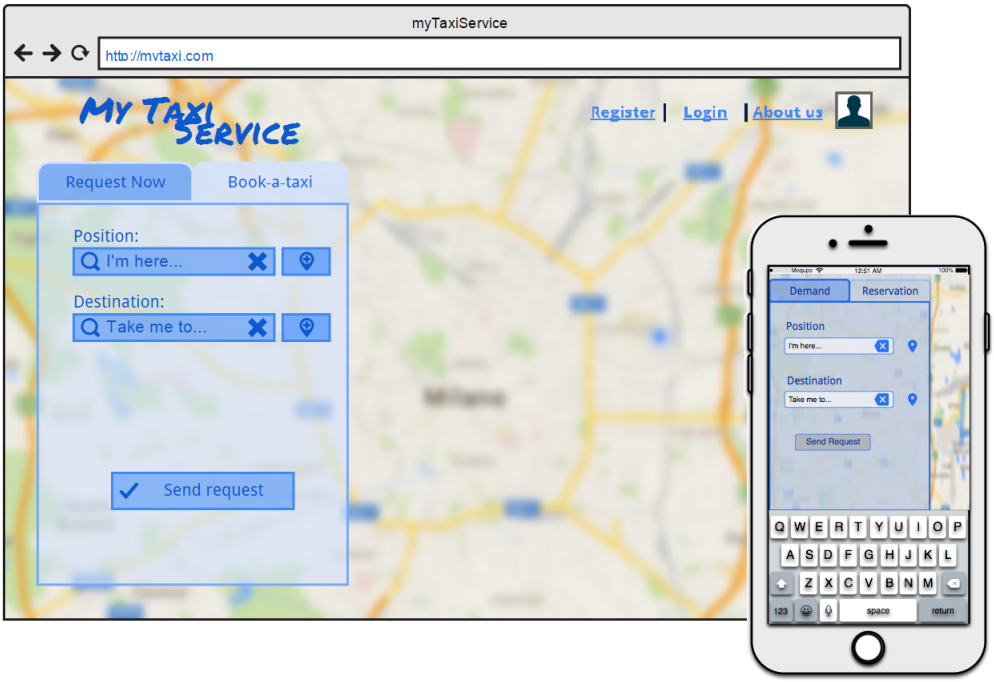
Registration Registration form allows both clients and taxi drivers to sign up to the service, either using the web application or the mobile one. After the registration process the user is redirected to the site home page were he can submit his first request.



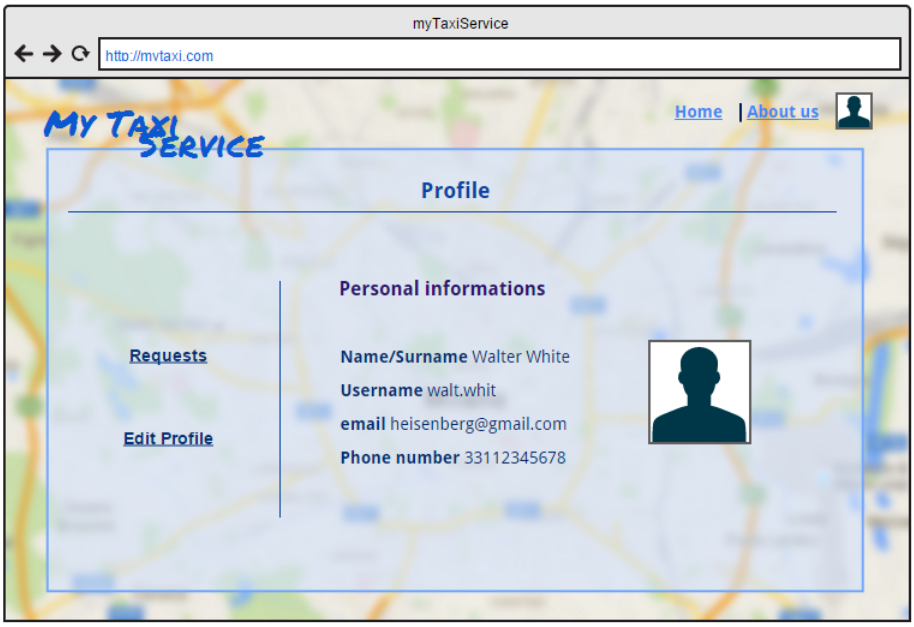
Login Whether a visitor has already signed up, he can access the service just using the login interface. Even after login process the user is redirected to home page.

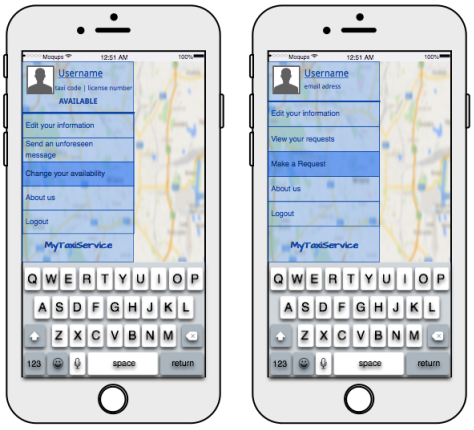


Home page Once logged, user from here can forward requests to the system, either demands or a reservations, by entering all required informations.

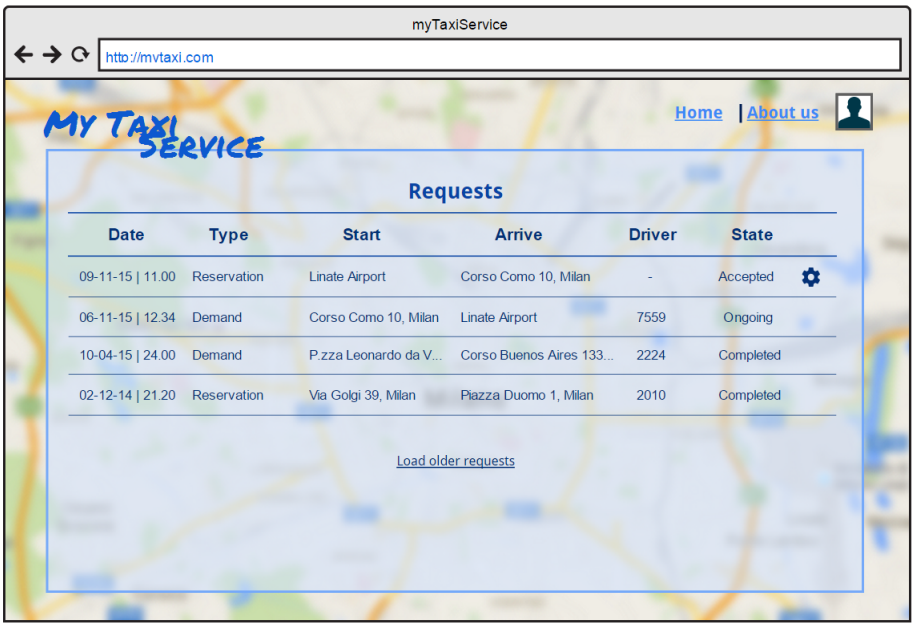


Personal Profile By clicking on its avatar in the website or accessing the left application bar in the mobile app, user access a personal profile interface, where he can find a request history and the chance to edit personal informations

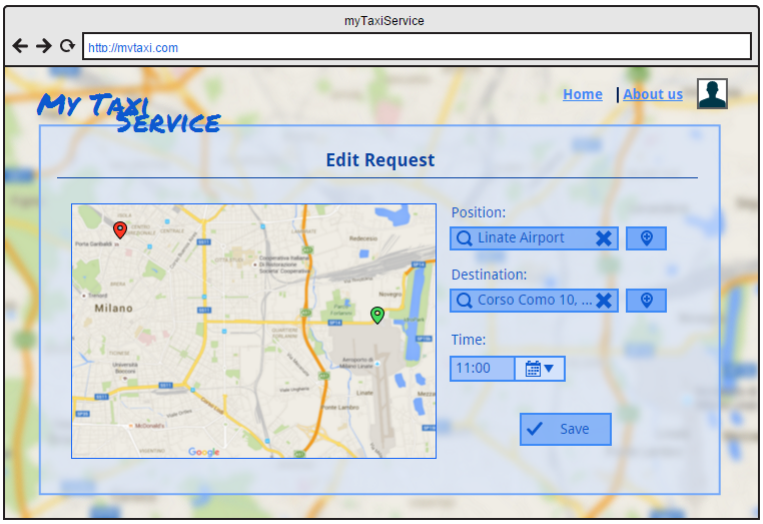




Requests In this page a user can access the history of his requests, with all their informations. For the accepted reservations he also can access the editing form.



Request editing As said before, reservations are allowed to be edited by the user, which is able to freely change the locations and the time gived during the first forwarding.



Notifications Here we present as a notification arrives to the user using the system from mobile. In the first picture we see a notification with all the informations sent to a client that has recently requested a taxi ride. In the second picture instead there are the ones sent to the taxi driver, which can interact with the system by accepting or refusing the request.

5 Requirements Traceability

In this section is presented the requirement traceability table, in which for every requirement pointed out in RASD document is mapped the system component which fulfill the desired functionality. In order to make more comprehensible the subsequent requirements, here are reported also the goals previously reported on RASD.

- [G1] allow a visitor to become a registered user
- [G2] allow a visitor to log in to the application
- [G3] allow user to make a request of taxi service, both reservations and for immediate service
- [G4] allow taxi drivers to communicate their availability
- [G5] allow taxi drivers to accept or reject a request
- [G6] allow registered members to view their reservations and change them.

Requirement	Component
[G1.R1] visitors can just see the login page	User Interface Components
[G1.R2] visitors can access only the registration form	User Interface Components
[G1.R3] if, during registration, the phone number or mail address provided are already present in the database of the application, the registration is denied	Account Manager, Data Manager
[G1.R4] visitor must complete all the mandatory fields of the registration form	User Interface Components
[G1.R5] the visitor who wants to be registered as taxi drivers must provide their number of taxi license and the one-time password received	User Interface Components
[G2.R1] misinformation does not allow access to the account	Account Manager, Data Manager
[G2.R2] visitors cannot request a taxi before the login	User Interface Components
[G2.R3] the application implements the retrieve password mechanism, using mail address provided in the registration	Account Manager, Data Manager
[G3.R1] user must complete all the mandatory elds of the taxi request form	User Interface Components
[G3.R2] a demand is rejected if there is already another not replied demand, performed by the same user	Request Manager
[G4.R1] taxi drivers must be in only one status between available and unavailable	Taxi Manager
[G4.R2] an available taxi driver that has an unforeseen, must declare himself unavailable without communicating to the system the unforeseen	Taxi Manager
[G4.R3] only taxi drivers unavailable but in service must communicate the unforeseen to the system	User Interface Components, Taxi Manager
[G5.R1] taxi drivers must be available to accept or reject a request	Taxi Manager
[G5.R2] taxi drivers must not both accept and reject a request	Request Manager
[G5.R3] taxi drivers must not receive more than one requests simultaneously	Request Manager
[G5.R4] the taxi driver who accepts a request must be set as unavailable	Request Manager, Taxi Manager
[G6.R1] registered user view only his own reservations	Data Manager
[G6.R2] only the future reservations, i.e. those who have set the date that is later than the current one, can be edited	Request Manager
[G6.R3] all changes made on reservations must comply the requirements previously presented for the reservations of a taxi	Request Manager

6 Appendix

6.1 Used tools

- *LyX:* to reduct and format this document
- *Visual Paradigm:* to create UML diagrams
- *moqups.com:* to create application's GUI mockups

6.2 Hours of work

Time spent for redacting this specification document:

- Andrea Turconi: ~23 hours
- Lorenzo Raimondi: ~23 hours