POLITECNICO
MILANO 1863

Politecnico di Milano
A.A. 2015/2016
Software Engineering 2: "*myTaxyService*"

Design **D**ocument

Alessandro Baldassari (mat. 841561)
Alberto Bendin (mat. 841734)
Francesco Giarola (mat. 840554)

December 3, 2015

Contents

1 Introduction

1.1 Purpose

This document presents the architecture on which *myTaxiService* will be developed; it describes the decisions taken during the design process and justifies them. The whole design process is described including also the improvements and modifications to provide additional valuable informations in case of future changes of the architecture structure.

1.2 Scope

Accordingly to the definition of the architecture design this document will focus on the non functional requirements of *myTaxiService*. Since the system architecture defines constraints on the implementation this document will be used to provide fundamental guidelines in the development phase of *myTaxiService*.

1.3 Definitions, Acronyms, Abbreviations

The following acronyms are used in this document:

- JEE: Java Enterprise Edition
- RASD: Requirements Analysis and Specification Document
- UI: User Interface
- EIS: Enterprise Information System
- API: Application Programming Interface

The following definitions are used in this document:

- Thin client: is a computer that depends heavily on another computer (its server) to fulfill its computational roles.
- Fat server: is a type of server that provides most of the functionality to a client's machine within a client/server computing architecture.
- Event-driven architecture: is a software architecture pattern promoting the production, detection, consumption of, and reaction to events, that is a change of state of an object.
- Availability: is the "status" of a taxi driver, s/he can be *waiting* (available) to take care of new jobs, thus the system should forward compatible requests, or can be *offline* (unavailable), meaning that the taxi is temporarily off line with respect to the system, thus the system should not consider that taxi for requests assignment. The driver can also be *working* (busy), as a third "status", that is when s/he is carrying passengers.

1.4 Reference Documents

- Specification document: myTaxiService project
- Template for the Design Document
- IEEE Std 1016-2009 - IEEE Standard on Design Descriptions
- Requirements Analysis and Specification Document for *myTaxiService*

1.5 Document Structure

This document specifies the architecture of the system using different levels of detail. It also describes the architectural decisions and justifies them. The design is developed in a top-down way, then the document reflects this approach. The document is organized in the following sections:

1. Introduction
   Provides a synopsis of the architectural descriptions.

2. Architectural design
   It is the core of the design document, here are presented all the components of the system and the interaction between them.

3. Algorithm design
   Flowchart and pseudocode description of the fundamental algorithms of *myTaxiService*.

4. User interface design
   Some mock-ups of the UI to better understand how the functionalities have been implemented from a graphical viewpoint.

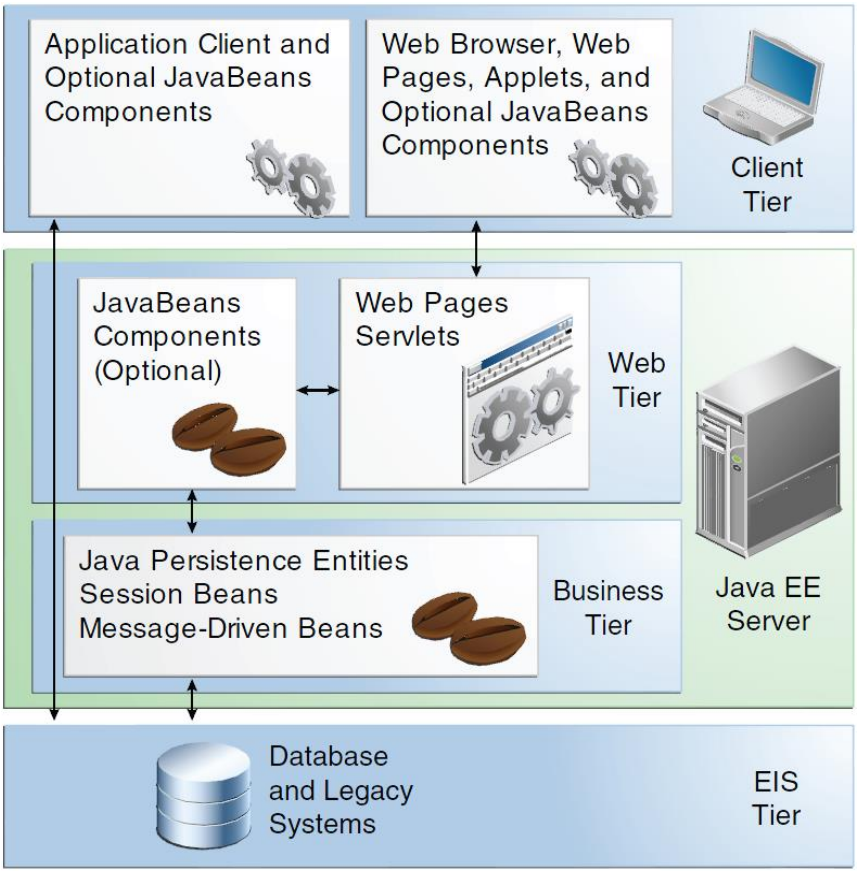
5. Requirements traceability
   Presents the mapping between the requirements and the components used to satisfy them.

6. References
   List of sources for material used in this document: internet links, books.

2 Architectural Design

2.1 Overview

This section of the design document provides a general description of the design of the system and its processes; it includes the general design context, the general approach and describes the overall design.
The architectural style adopted by *myTaxiService* is the well-known "client-server", where the client is a "thin client" and the server is a "fat server". The architecture is "event-driven", thus the interaction between the components takes place through events, in other words asynchronous triggers.
JEE has a "four-tier" architecture divided as shown in the picture below:



1. **Client Tier**: contains Application Clients and Web Browsers and it is the layer designed to interact directly with the actors. *myTaxiService* is a web and mobile application, then the client will use a web browser or a smartphone to access the pages.

2. **Web Tier**: contains the Servlets and Dynamic Web Pages that need to be elaborated. This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier, eventually formatted.

3. **Business Tier**: contains the Java Beans, which contain the business logic of the application, and Java Persistence Entities.

4. **EIS Tier**: contains the data source. In our case, it is the database allowed to store all the relevant data and to retrieve them.

When the second and third tier are considered together the architecture becomes a "three-tier" with *client tier*, *business logic tier* and *persistence tier*.
To design the system a top-down approach is used. After the identification of the main three layers, the system is decomposed in components that capture subsets of related functionalities. For each component is specified the role in the architecture and its interactions with the rest of the system.

2.1.1 Identifying sub-systems

The functionalities of *myTaxiService* are divided into these functional areas:

- myTaxiService

  This component describes the system that is going to be developed.

- Users

  This component represents all the users that will use the service.

- Notification service

  This component represents the notification mechanism that the system will use (text-messages, emails and push notifications).

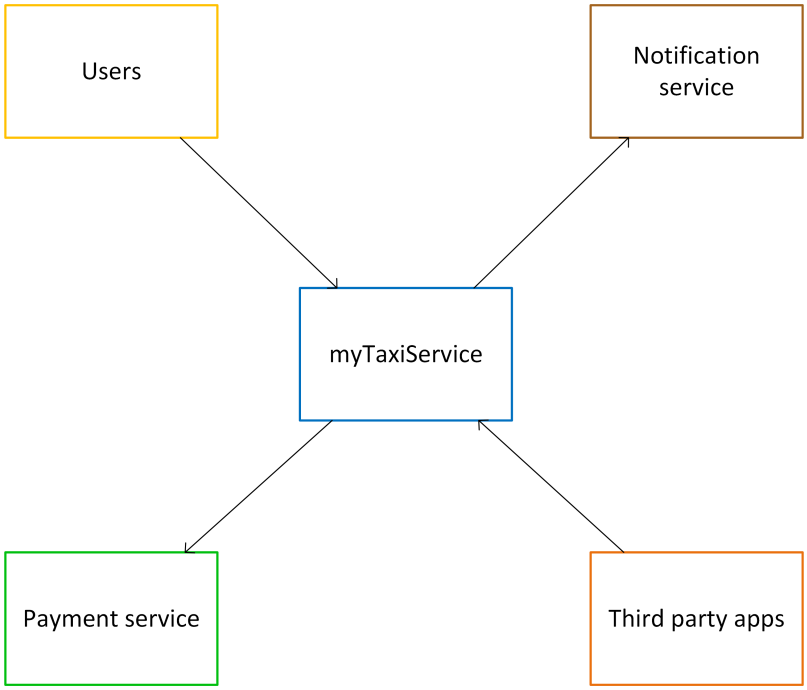
- Payment service

  This component represents the external service in charge of managing the payments.

- Third party apps

  This component represents generally the external apps that can possibly connect to the system via our public web APIs.

The following schema shows the above-mentioned components and their interaction.



The "notification service", "payment service" and "third party apps" are external components that are not part of the *myTaxiService* system.

2.2 High level components and their interaction

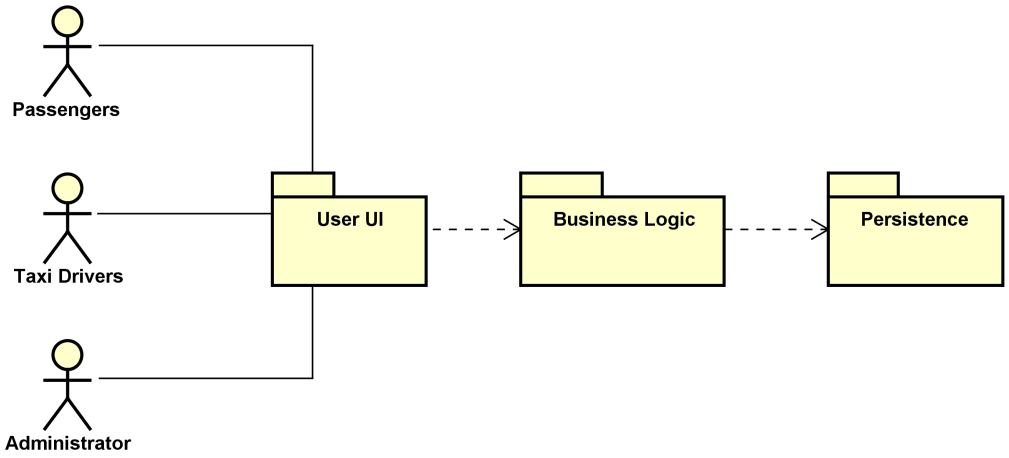
2.2.1 General Package design

Considering the three-tier view of the architecture three packages are identified:

- User UI: this package is in charge of interacting with the user; it receives the user requests, sends these to the business logic package, obtains the information needed from the latter and displays them to the user accordingly. In general the package contains the user interfaces.

- Business logic: this package is in charge of receiving and processing the User UI's package requests, accessing the Persistence package when needed and sending a response accordingly.
- Persistence: this package is in charge of managing the data requests from the Business logic package.

The main users: Administrator, passengers and taxi drivers directly access the User UI package but cannot see the other packages, as shown in the picture below.

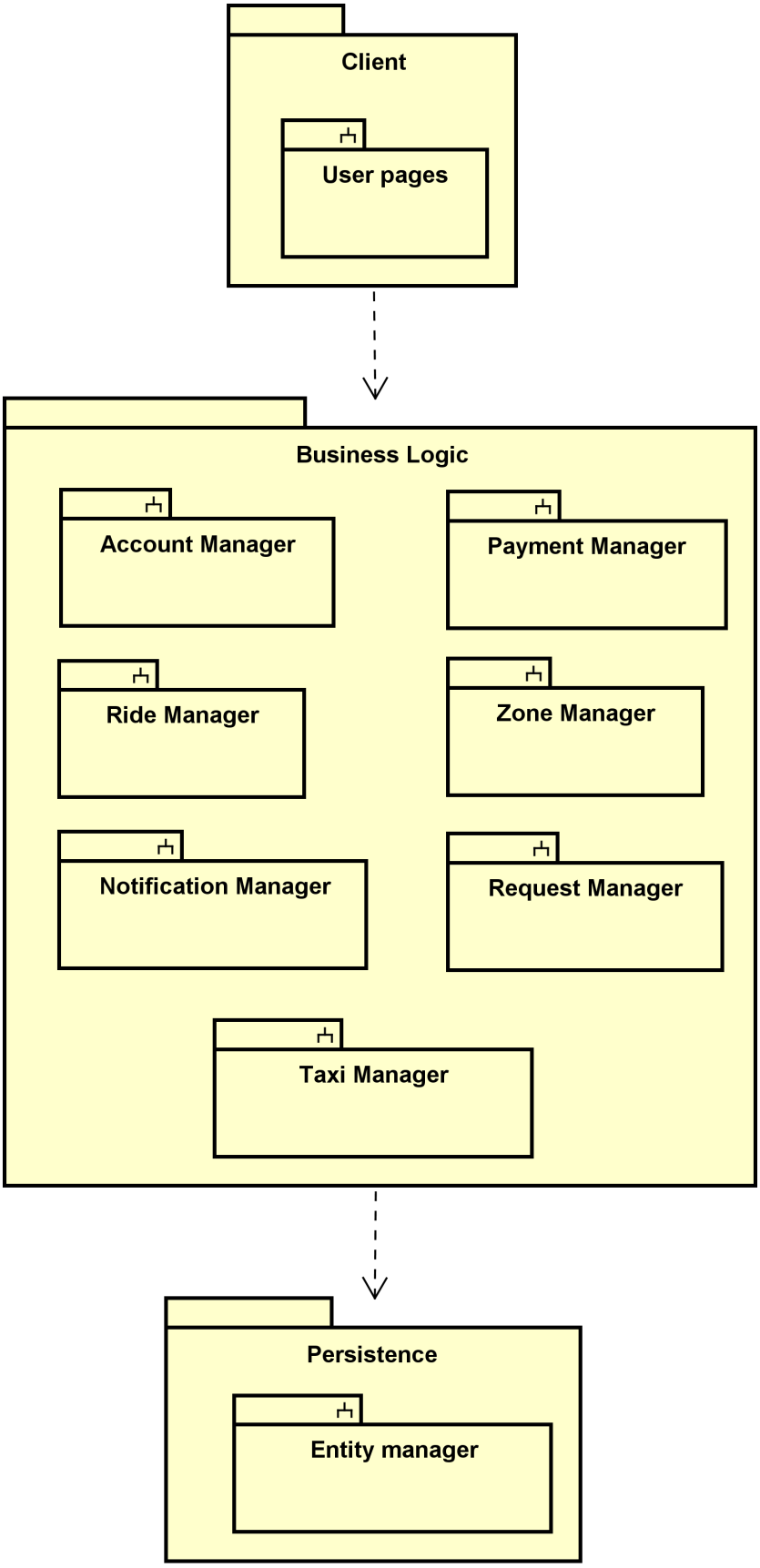


2.2.2 Detailed Package Design

The inner packages are described as follows:

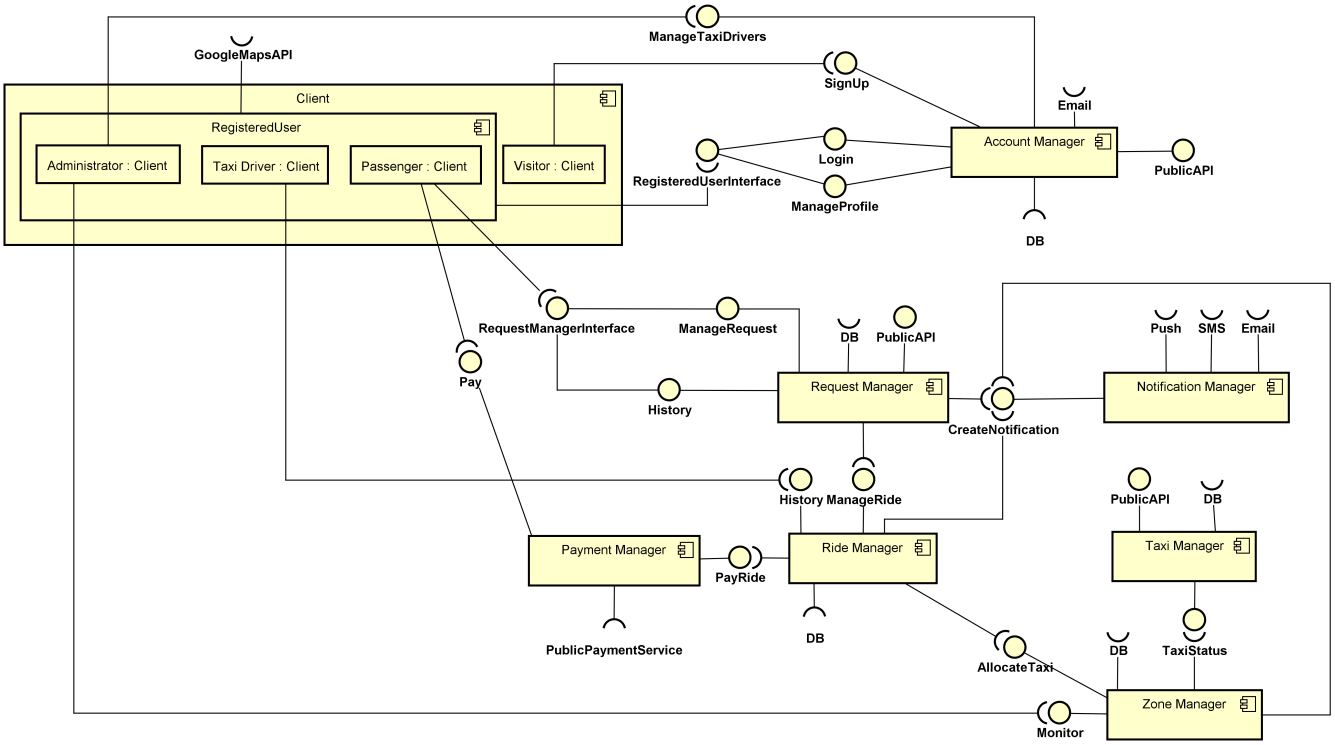
- User UI: this set of sub-packages is responsible for encapsulating the user's actions and forwarding information requests to the Business Logic sub-packages.
- Business logic: this set of sub-packages is responsible for handling requests from the User UI package, processing them and sending back a response. These packages may access the Persistence package.
- Persistence: this set of sub-packages contains the data model for the system. It accepts requests from the Business Logic package.

A more detailed view of the system:



2.2.3 High Level Component View

The picture below represents the main components and interfaces of *myTaxiService*.



From the diagram above it is evident that:

⇒ The "*UserManager*" interface contains all the information related the every user and his/her profile; it also offers the various "login", "sign-up", "changePassword", "changeNotificationType" methods. It also enables the administrator to manage the drivers list.

⇒ The "*PaymentManager*" is the interface in charge of exchanging the information about the transaction with the payment service provider.

⇒ The passenger is able to make a request or reservation through the interface "*RequestManager*" that is in charge of receiving a complete and customized ride request.

⇒ The "*RideManager*" interface communicates with the "*RequestManager*" to prepare the physical ride; it communicates with the "*QueueManager*" interface to be able to select to which taxi the jobs are proposed and with the "*NotificationManager*" to notify all the users involved in ride.

⇒ The "*QueueManager*" interface is accompanied by a "*Zone*" class to control the sections of the city and their queue of available taxis.

⇒ The three types of users are represented by three respective classes with the same name: "*Passenger*", "*TaxiDriver*" and "*Administrator*".

⇒ The "*RequestManagerImpl*" and "*RideManagerImpl*" classes accompany their respective interfaces and keep track of the history of rides and request for every users.

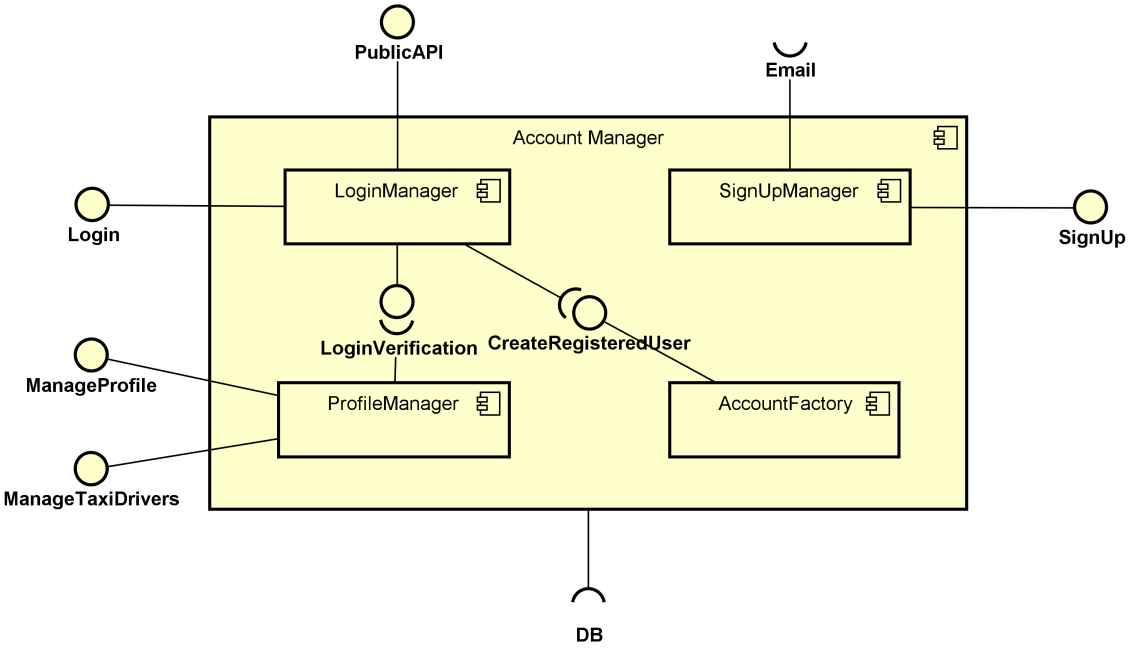
⇒ The connection between the "*Administrator*" and the "*Zone*" class enables the administrator to supervise the situation of all the queues in real-time.

⇒ The connection between the "*TaxiDriver*" and the "*Zone*"and "*RideManagerImpl*" classes enables the system to keep track of the availability of the taxi drivers and allows the drivers to accept or reject job proposals.

2.3 Component view and interfaces

Here is a more detailed view of every component with its interfaces.

2.3.1 Account Manager



SignUpManager:

Definition	Component controlling the visitors' sign up.
Responsibilities	This component allows visitors to sign up into *myTaxiService* and become registered users. It connects to the DB to store the credentials and requires and Email interface to verify the sign up procedure through an confirmation link sent via email.
Interaction	With the visitors, the DB and email service.
Interfaces offered	• Sign up for Visitors
Interfaces required	• DB • External Email service
Pattern	

LoginManager:

Definition	Component controlling the users' login.
Responsibilities	This component allows users to login into *myTaxiService*. It connects to the DB to verify the credentials and grants access to the ProfileManager. It offers the possibility to login to the service to third party apps.
Interaction	With all the users of the system, with the DB and external services (APIs).
Interfaces offered	• Login for User • LoginVerification for ProfileManager • PublicAPI for external services
Interfaces required	• DB
Pattern	

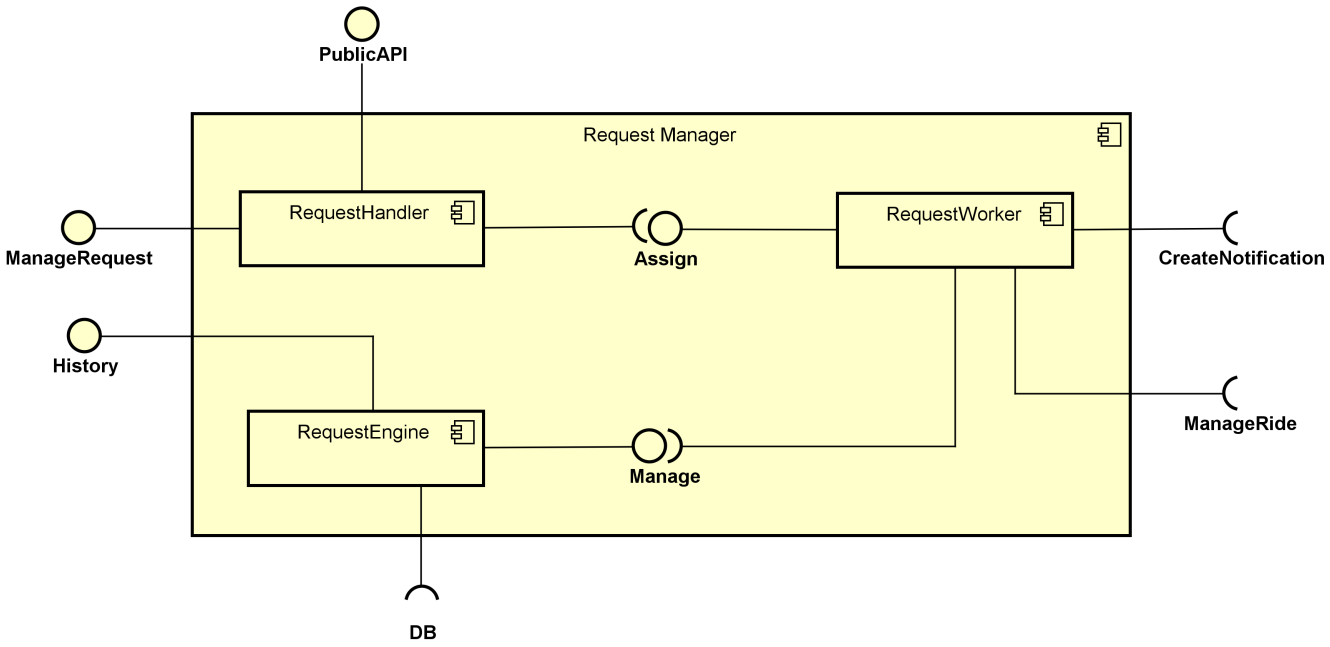
ProfileManager:

Definition	Component controlling the users' profile.
Responsibilities	This component allows users to edit their profile, for instance to change the password, credit card data, phone number, notifications type. It permits the Administrator to add or edit taxi drivers.
Interaction	With all the RegisteredUser of the system, with the DB and with the LoginManager component.
Interfaces offered	• ManageProfile for RegisteredUser • ManageTaxiDrivers for Administrator
Interfaces required	• DB • LoginVerification for LoginManager
Pattern	

AccountFactory:

Definition	Component which instantiate the users.
Responsibilities	This component is in charge of the creation of an instance for every logged in user. It is an example of "Factory method" pattern.
Interaction	With the LoginManager and the DB.
Interfaces offered	• CreateRegisteredUser for LoginManager
Interfaces required	• DB
Pattern	

2.3.2 Request Manager



RequestHandler:

Definition	Component receiving the passengers' requests.
Responsibilities	This component receives the requests or reservations of the clients and for each of them starts a "worker" that will handle them singularly. It also offers functions to enable users accessing from external services to send in requests.
Interaction	With all the Passengers from inside the system or external services and with the RequestWorker component to start a thread for every request.
Interfaces offered	• ManageRequest for Passengers • PublicAPI for external services
Interfaces required	• Assign for RequestWorker
Pattern	

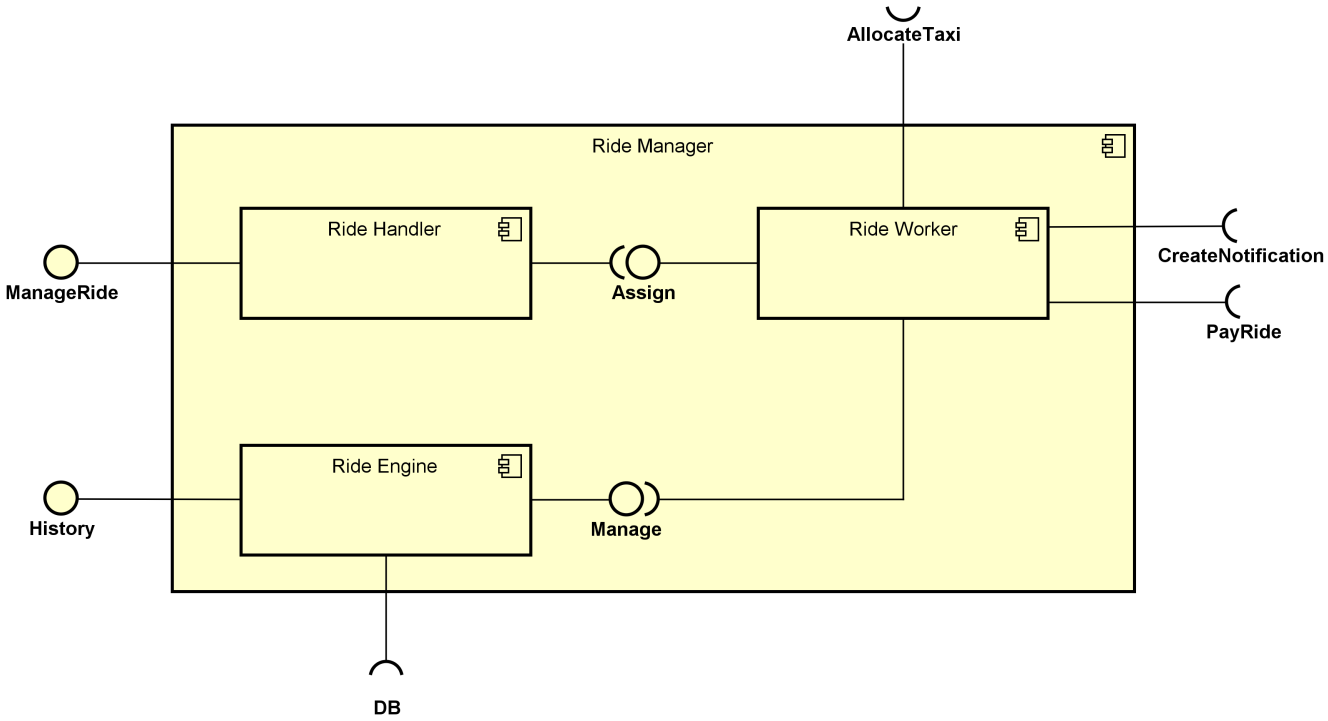
RequestEngine:

Definition	Component maintaining the history of the passengers' requests.
Responsibilities	This component keeps stored in the DB all the information of the past and active requests for every user who can browse his/her history or modify current requests. Of course it accesses the DB.
Interaction	With all the Passengers, with the RequestWorker components to receive updates from every request and with the DB to store persistently the data.
Interfaces offered	• History for Passengers • Manage for RequestWorker
Interfaces required	• DB
Pattern	

RequestWorker:

Definition	Component managing individually the requests of the users.
Responsibilities	There is an instance of this component for every request generated by the users. It communicates updates of the request to the Notification manager. It is also connected to the RideManager to start the process of creation of the ride at the decided time.
Interaction	With the NotificationManager, with the RideManager, and with the RequestEngine to store the data about the requests in the DB.
Interfaces offered	• Assign for RequestHandler
Interfaces required	• CreateNotification for NotificationManager • ManageRide for RideManager • Manage for RequestEngine
Pattern	

2.3.3 Ride Manager



RideHandler:

Definition	Component that creates the rides.
Responsibilities	This component creates the rides from the corresponding requests at the decided time (10 minutes before the meeting time for the reservations), for this reason it communicates with the RequestManager. It later assigns every generated ride to an instance of RideWorker which will handle it singularly.
Interaction	With the RequestManager and with the RideWorker component to start a thread for every ride.
Interfaces offered	• ManageRide for RequestManager
Interfaces required	• Assign for RideWorker
Pattern	

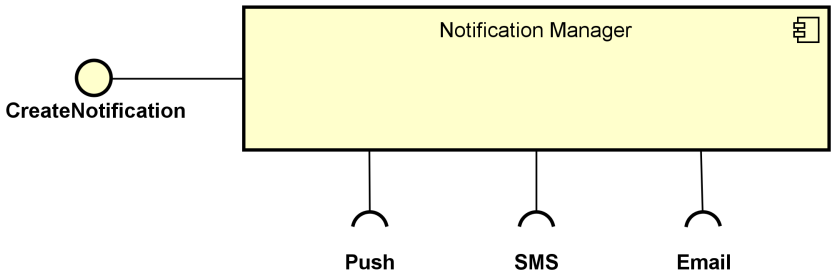
RideEngine:

Definition	Component maintaining the history of the drivers' rides.
Responsibilities	This component keeps stored in the DB all the information of the past and active rides for every user who can browse his/her history or modify current rides. Of course it accesses the DB.
Interaction	With all the drivers, with the RideWorker components to receive updates from every ride and with the DB to store persistently the data.
Interfaces offered	• History for TaxiDrivers • Manage for RideWorker
Interfaces required	• DB
Pattern	

RideWorker:

Definition	Component managing individually the rides of the users.
Responsibilities	There is an instance of this component for every ride generated by the system. It communicates updates of the ride to the NotificationManager to warn involved passengers and the driver. It is also connected to the ZoneManager to allocate a taxi once the ride is created and with the PaymentManager to collect money when the passenger needs to pay a fine or the taxi fare. It is connected with the RideEngine to store the data of the ride in the DB.
Interaction	With the NotificationManager, with the ZoneManager, with the RideEngine and with the PaymentManager.
Interfaces offered	• Assign for RideHandler
Interfaces required	• CreateNotification for NotificationManager • AllocateTaxi for ZoneManager • PayRide for PaymentManager
Pattern	

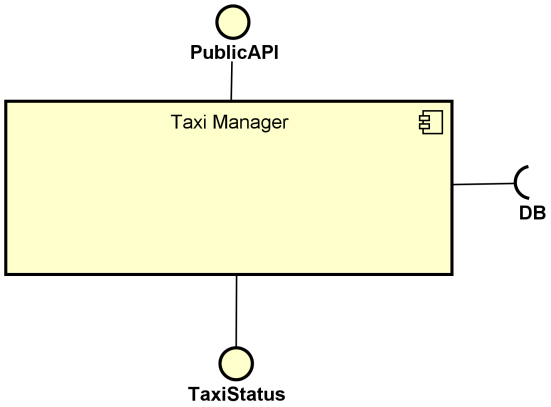
2.3.4 Notification Manager



NotificationManager:

Definition	Controller of the current situation of every taxi.
Responsibilities	This components keeps track of all the information about the whole taxi fleet. It can retrieve the position of taxis as well as their availability status and send these to the ZoneManager which uses them to control the queues and assign jobs. Thanks to the information about the location of every taxi, an external service can retrieve the current situation of the taxis spread across the city and possibly show them on a map.
Interaction	With the ZoneManager to make available information about taxis, with the DB to store the data and with external services to make these same information available outside the system.
Interfaces offered	• TaxiStatus for ZoneManager • PublicAPI for external services
Interfaces required	• DB
Pattern	

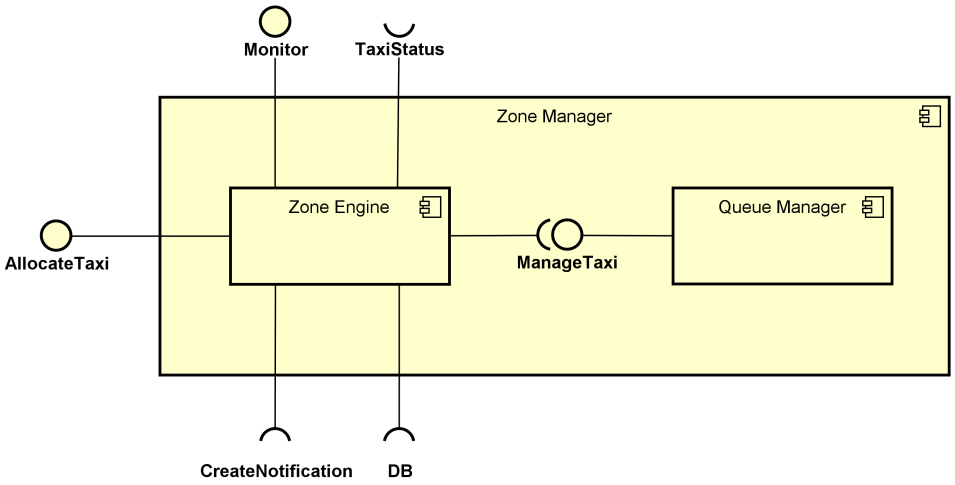
2.3.5 Taxi Manager



TaxiManager:

Definition	Component controlling the notifications sent to the users.
Responsibilities	This component is able to receive a message to be sent to some users with the specification of the mean of communication to be used. For this reason it is connected to the external services managing the dispatching of SMS and Emails.
Interaction	With the RequestManager and RideManager to send updates about the request/rides to the passengers involved, with the ZoneManager to send updates to taxi drivers, and with the external services of SMS and Email.
Interfaces offered	• CreateNotification for RideManager, RequestManager and ZoneManager
Interfaces required	• Push • SMS for external services • Email for external services
Pattern	

2.3.6 Zone Manager



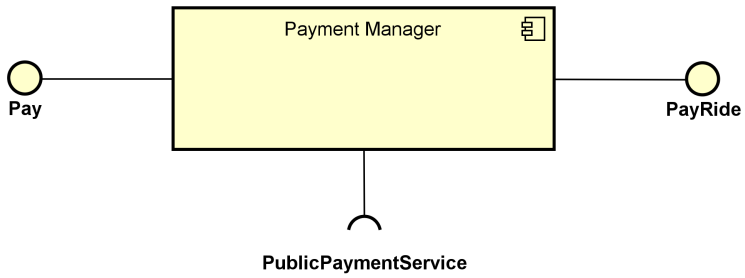
ZoneEngine:

Definition	Component controlling the overall management of the city zones.
Responsibilities	This component is the manager of all the single queues, for this reason it is connected to the n QueueManagers, where n is the number of zones in the city. It is in charge of forwarding the requests of jobs/taxi assignation to the correct QueueManager. Being an overall summary of the current situation of the system it is connected with the Administrator who can monitor the real-time status of the service. All the data about the zones are stored in the DB. ZoneManager is called when RideManager needs to assign a driver to a specific ride, and the assignation is carried out with the cooperation of the TaxiManager which provides the availability status and position of every taxi. Once the assignation has been successfully completed ZoneManager notifies the driver through the NotificationManager.
Interaction	With the Administrator, TaxiManager, RideManager, NotificationManager and QueueManagers.
Interfaces offered	• Monitor for Administrator • AllocateTaxi for RideManager
Interfaces required	• TaxiStatus for TaxiManager • CreateNotification for NotificationManager • DB
Pattern	

QueueManager:

Definition	Component controlling the queue of taxis in a single zone of the city.
Responsibilities	This component runs the algorithm to assign the jobs to its taxis and keeps the management of its queue fair. It receives the tasks to carry out from the ZoneEngine which only selects the correct zone.
Interaction	With the ZoneEngine.
Interfaces offered	• ManageTaxi for ZoneEngine
Pattern	

2.3.7 Payment Manager



PaymentManager:

Definition	Component controlling the payment process
Responsibilities	This component is in charge of computing the amount of money (fine or fare) owed by each passenger (if the ride is shared). It interacts with the RideManager to receive information about the ride and with the Passenger to collect the money. It uses an interface to process the payment with external services.
Interaction	With all the Passengers of the system, with the RideManager and with external services.
Interfaces offered	• Pay for Passenger • PayRide for RideManager
Interfaces required	• PublicPaymentService for communication with external payment services.
Pattern	

2.4 Deployment view

2.5 Runtime view

2.6 Selected architectural styles and patterns

The decision of using the classical client-server architecture was straightforward. In a system like *myTaxiService* it is important to have one single point of failure (the server) and control, with respect to distributed architectures. This decision is also compatible with the fact of having only one centralized database which sustains the whole service.
For the type of interaction with clients required by the system, the "thin client" approach is employed; only a very simple layer of logic is implemented on the client side, for instance the acquisition of coordinated with the GPS sensor and the rendering of the graphical user interface.
myTaxiService relies on an "event based" or "event-driven" architecture. The system reacts to the change of state of some objects, for instance the switch of "availability" for the taxi driver is detected by the zone manager that operates accordingly in the respective queue. Other examples could be the change of state of a request to a modified request that must be taken

into account (eg. cancellation of the request), or the consequence of the approval/refusal of a job by a driver. All of these message-events are carried out by the notification manager as notifications.
Riepilogo veloce a parole :

- patterns
- multithreading (componenti fissi vs componenti ad istanze...)

2.7 Other design decisions

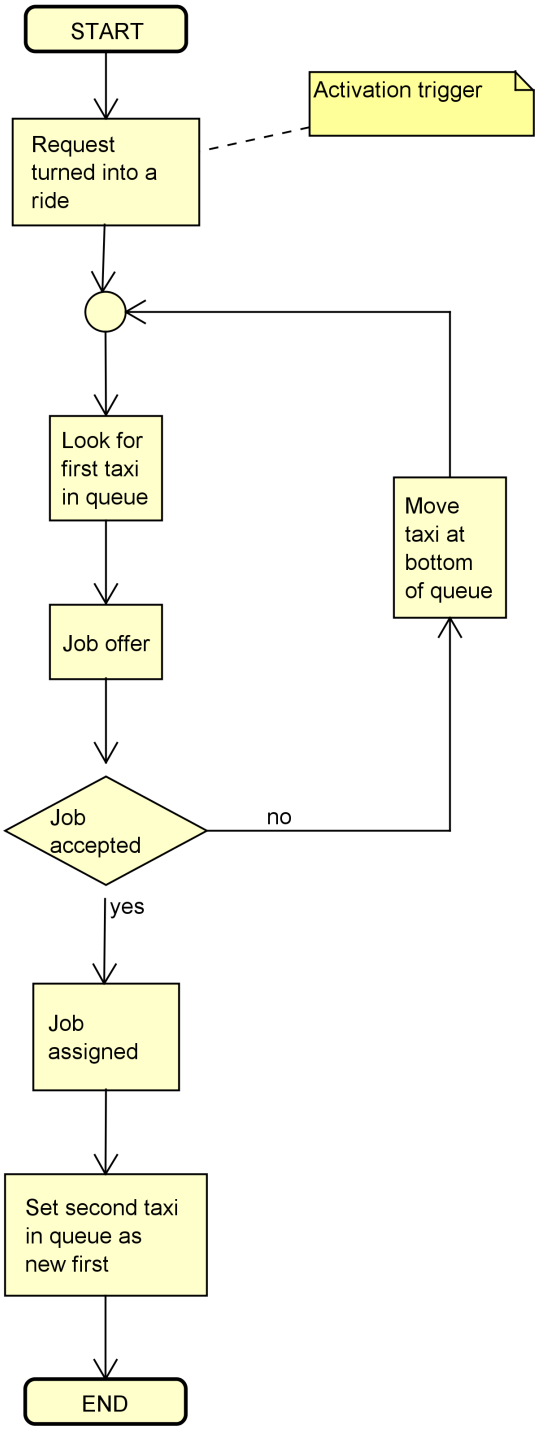
3 Algorithm design

Here are some distinctive algorithms of *myTaxiService* presented with both pseudocode and flowchart diagrams.

3.1 Queue managing and taxi assignment

This algorithm depicts what happens when the system assigns a ride to a driver. It sends the proposal to the first driver of the queue in the zone where the client is located; if the driver accepts the job s/he is removed from the queue and the assignation is completed, otherwise the system forwards the request to the second driver in the queue and places the first one at the bottom of the queue. This is repeated until a driver accepts the job.
This algorithm runs on the QueueManager (selected by the ZoneEngine).

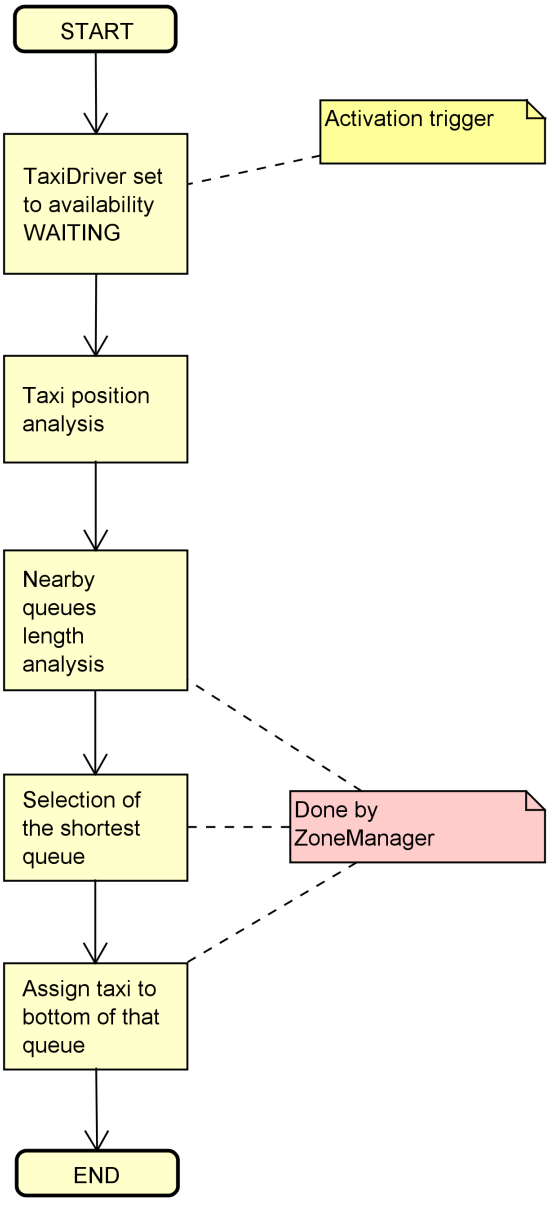
```
for (every request turned into a ride)
        search first taxi in queue;
        send proposal to taxi;
        while (job is not accepted) do
                move first taxi in bottom of queue;
                search first taxi in queue;
                send proposal to taxi;
        confirm job assignation;
        set second taxi in queue as new first;
```


START
Activation trigger
Request turned into a ride
Look for first taxi in queue
Move taxi at bottom of queue
Job offer
Job accepted
no
yes
Job assigned
Set second taxi in queue as new first
END

3.2 Zone assignment

This algorithm depicts what happens when the system assigns the taxi to a specific zone according to the taxi's current position obtained through the GPS sensor of the driver's phone. In particular all the zones nearby the taxi are analyzed and the zone with the shortest queue is selected. When the taxi is assigned it enters the queue at the bottom.
This algorithm runs on the ZoneManager and exploits both the ZoneEngine (to select the shortest queue) and the QueueManager (to assign the taxi to the bottom of the queue).

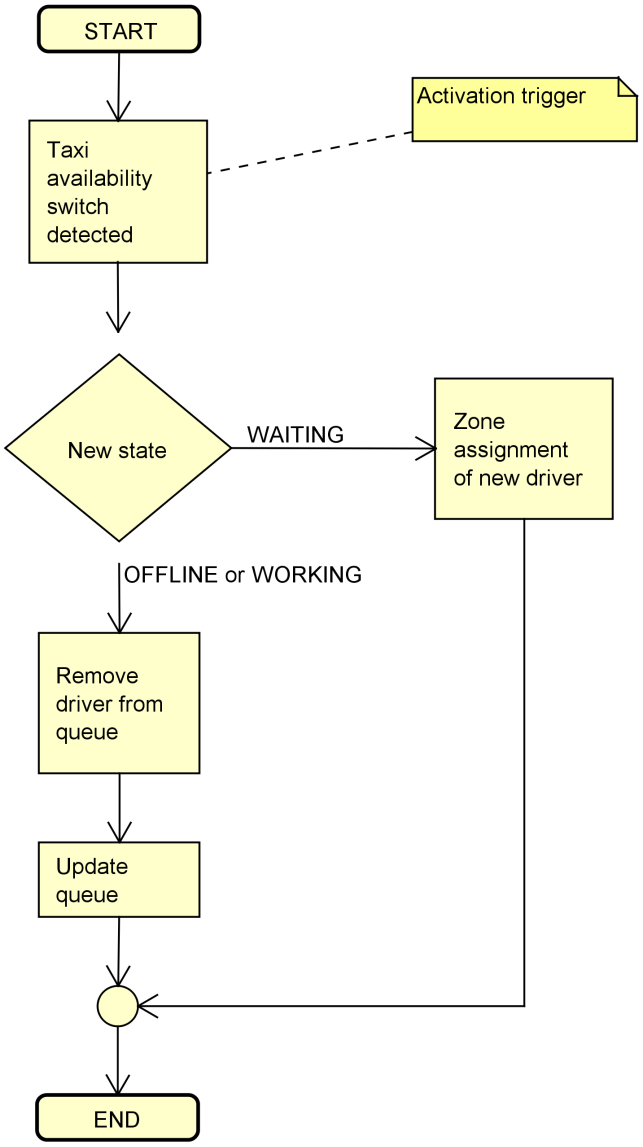
```
for (every taxi with availability == WAITING)
        detect taxi position;
        nearby queues length analysis;
        select the shortest queue;
        insert taxi at bottom of the queue;
```



3.3 Taxi availability

This algorithm depicts what happens when the driver switches his/her own availability. If s/he turns it *on* (*waiting* to be assigned) then the system will proceed with the zone assignment according to the position, while if the driver switches the availability *offline* the system will simply remove him/her from the current queue.
This algorithm runs on the TaxiManager which detects the availability trigger and relies on the ZoneManager for the update of the queue according to the availability status.

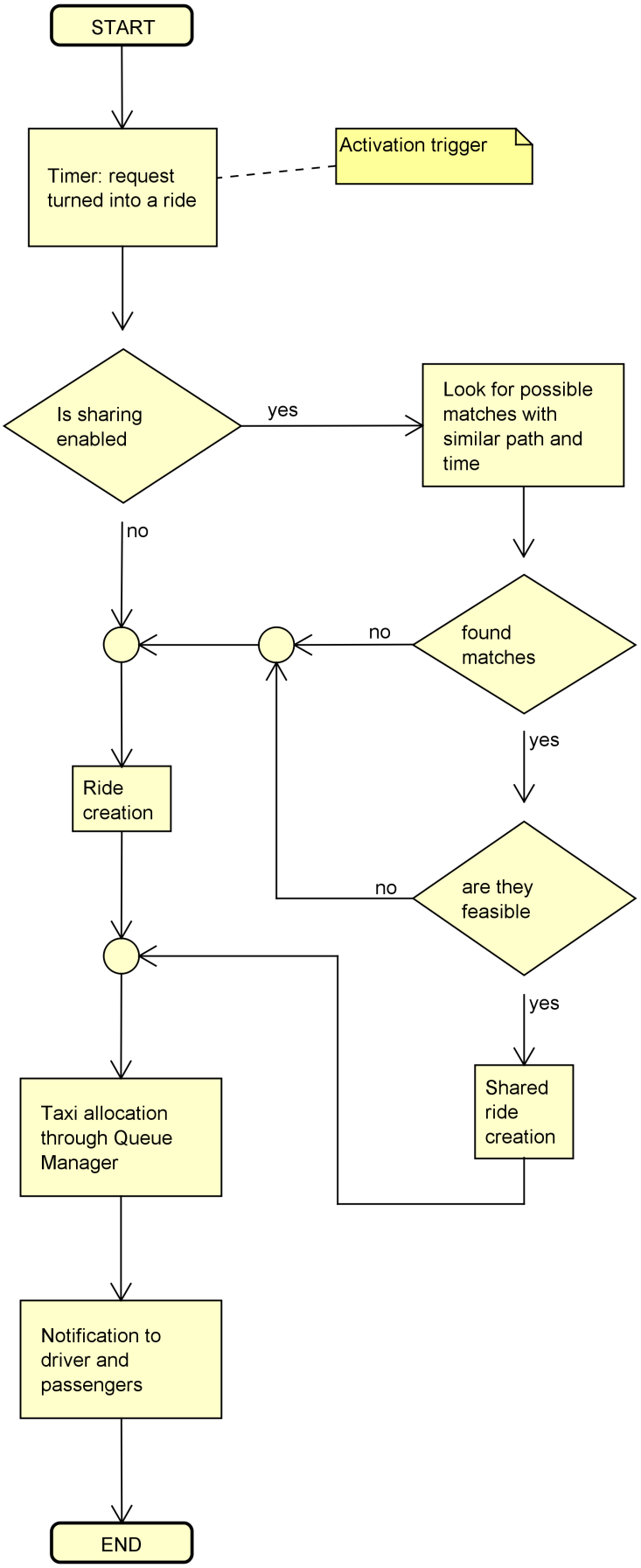
```
for (every taxi availability change detected)
        if (Availability:WAITING—>Availability:OFFLINE)
                remove driver from current queue;
                update queue;
        else
                proceed with the assignment of driver to zone;
                //see "Zone assignment" algorithm
```



3.4 Ride creation

This algorithm depicts what happens when the system creates a ride based on a request or a reservation; this happens 30 seconds after the arrival of the request and 10 minutes before the meeting time for the reservation. For the request the process is straightforward and it involves the taxi allocation and the notification to all the users involved; the same is for the reservation without the sharing option. On the contrary when a ride is reserved with the possibility of sharing the trip the system must look for possible matches with same pick up and drop off zone and same date and time; if any compatible match is found then the system computes the total number of passengers. If there is enough room on the taxi for all the passengers then a shared ride is created, otherwise the system proceeds with the standard ride.
This algorithm runs on the RideManager which communicates with the RequestManager to analyze matching requests, with the ZoneManager to allocate the taxi and with the NotificationManager to eventually inform all the users (passenger(s) and driver).

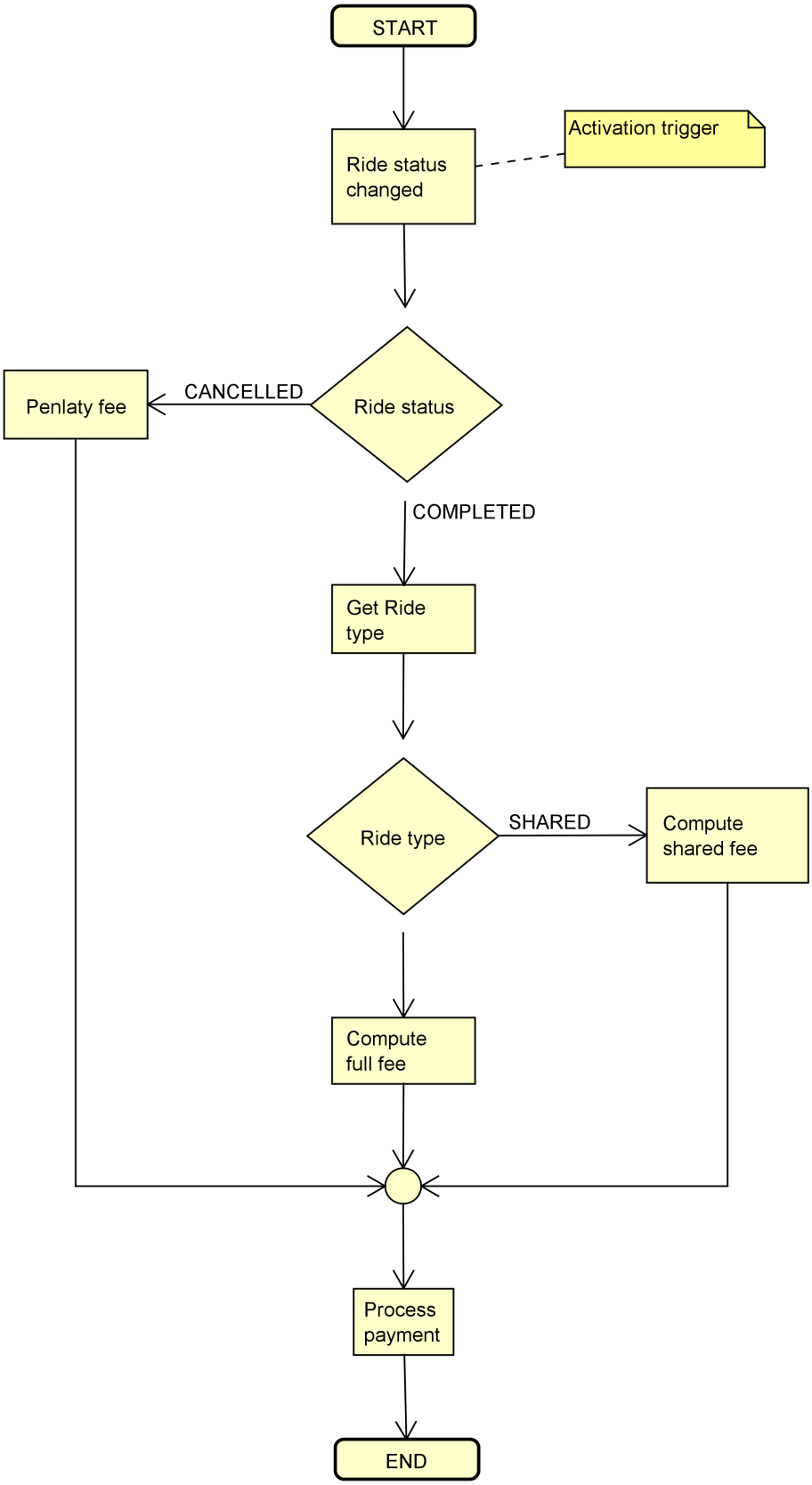
```
for (every ride creation)
        if (!sharing enabled)
                ride creation; //normal ride
        else
                search matches with same zone and time;
                if (!matches found)
                        ride creation; //normal ride
                else if (!compatible match)
                                ride creation; //no room for all passengers
                        else
                                shared ride creation;
        taxi allocation; //See "Queue manager" algorithm
        send notifications to driver and passengers involved;
```


START
Timer: request turned into a ride
Activation trigger
Is sharing enabled
yes
Look for possible matches with similar path and time
no
found matches
no
yes
Ride creation
are they feasible
no
yes
Shared ride creation
Taxi allocation through Queue Manager
Notification to driver and passengers
END

3.5 Payment

This algorithm represents the way the system deducts money from the passenger's credit card. Basically it can collect the standard fare computed on the distance covered on the taxi or a penalty fee. The latter is only applied when the user deletes (or modifies) a request/reservation beyond the allowed time frame, that is within 30 seconds after the application for the request and until 10 minutes before the meeting time for the reservation. A cancellation within the allowed time window does not involve any penalty fine.
When the ride is carried out successfully the fare is computed on the basis of the type of ride *shared* or regular.
This algorithm runs on the PaymentManager which communicates with the RideManager.

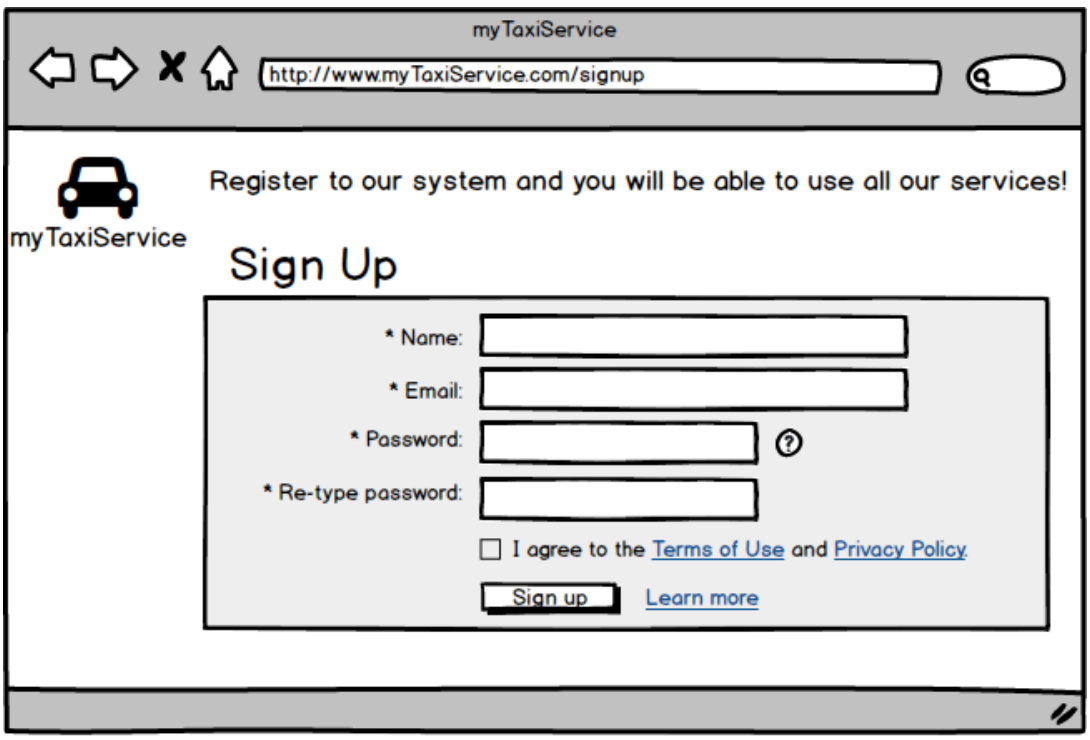
```
for (every ride status changed)
        if (ride status == CANCELLED)
                collect(penalty fee);
        else
                get Ride type;
                if (Ride type == SHARED)
                        compute shared fare;
                else
                        compute regular fare;
                collect(fare);
        process payment;
```


START
Ride status changed
Activation trigger
Ride status
CANCELLED
Penlaty fee
COMPLETED
Get Ride type
Ride type
SHARED
Compute shared fee
Compute full fee
Process payment
END

4 User Interface Design

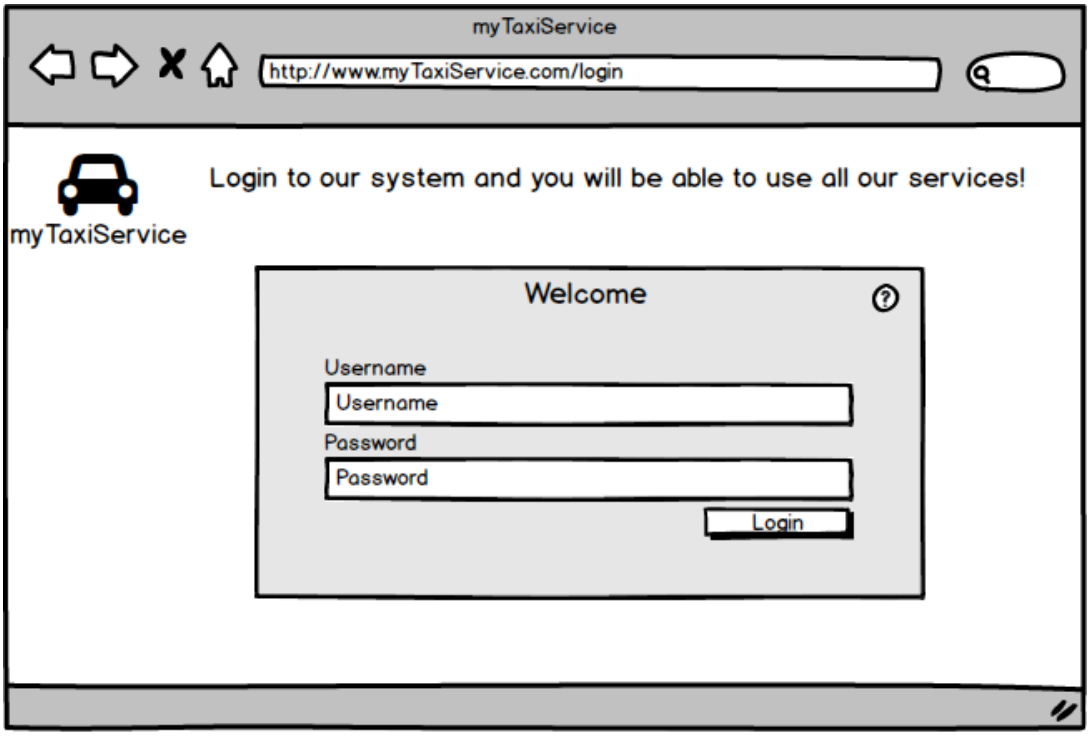
This section is the same featured in the RASD.
Shown below are some mock-ups that preview the user interface of the main features the system shall provide.

4.0.0.1 Sign-up This page presents the sign up form for the clients. The user will need to provide his/her personal data, home and email addresses, phone number, favorite payment option, favorite notification type.



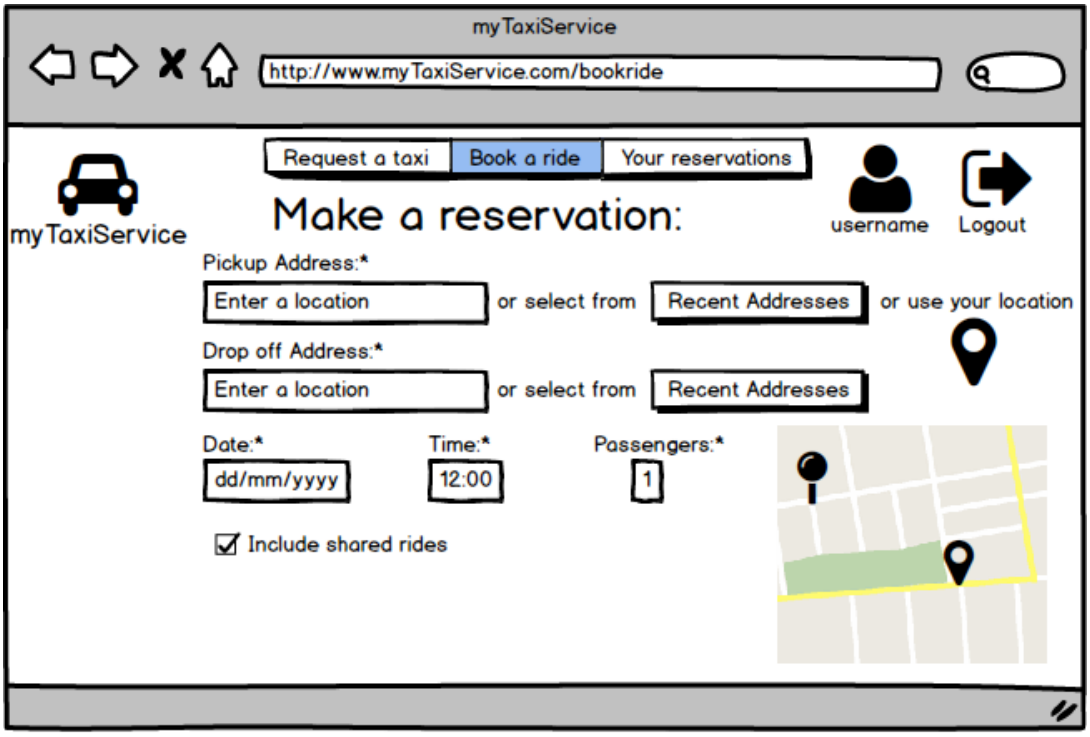
4.0.0.2 Login

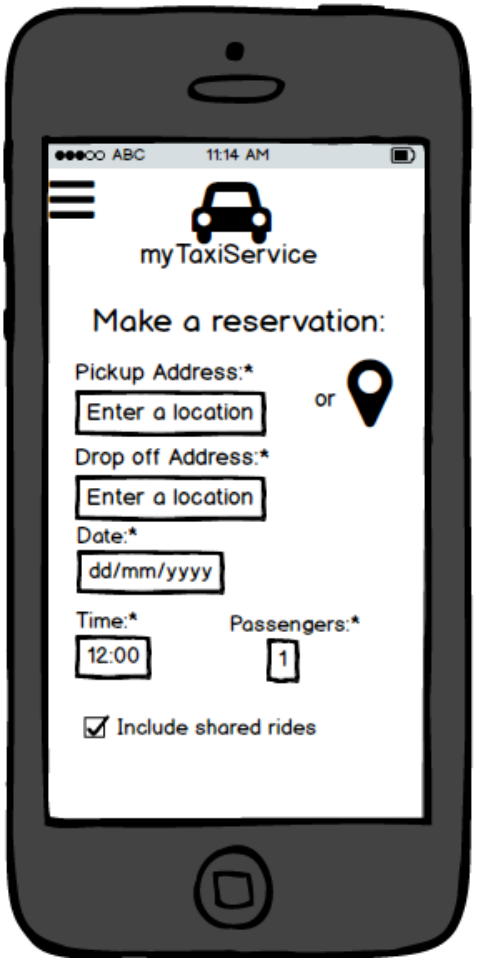
This page shows the login form for the final users.



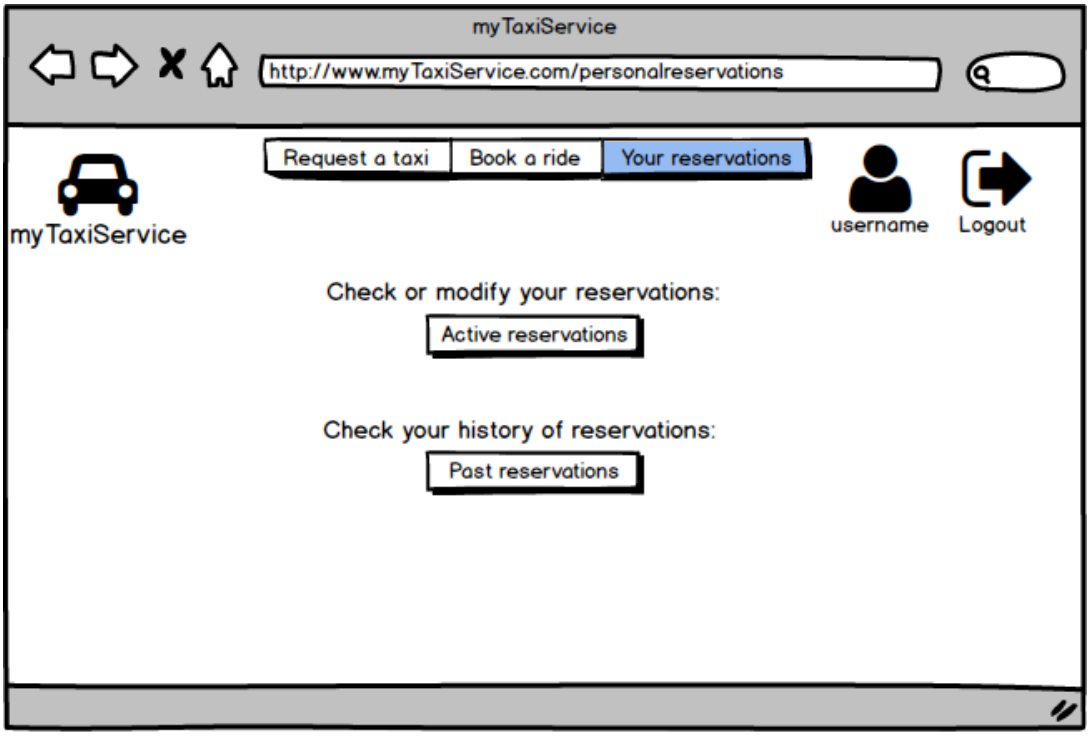
4.0.0.3 Call a taxi The user can ask for a taxi providing the pickup location through a complete address or through the automatic detection of his/her location thanks to the browser or the GPS. The user can also choose from an history of "recent addresses".

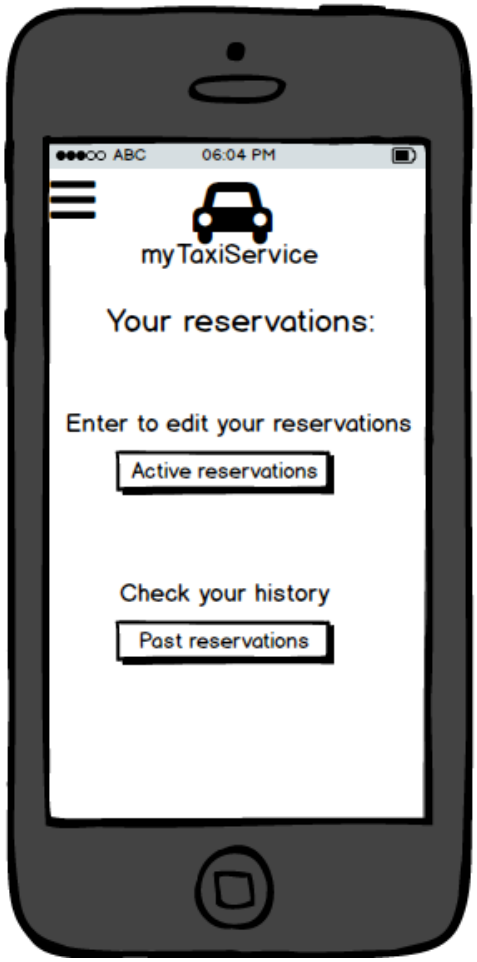
4.0.0.4 Plan a ride The user can plan a ride in advance providing the pickup and drop off locations, the date and time, the willingness to share the ride and the number of passengers.



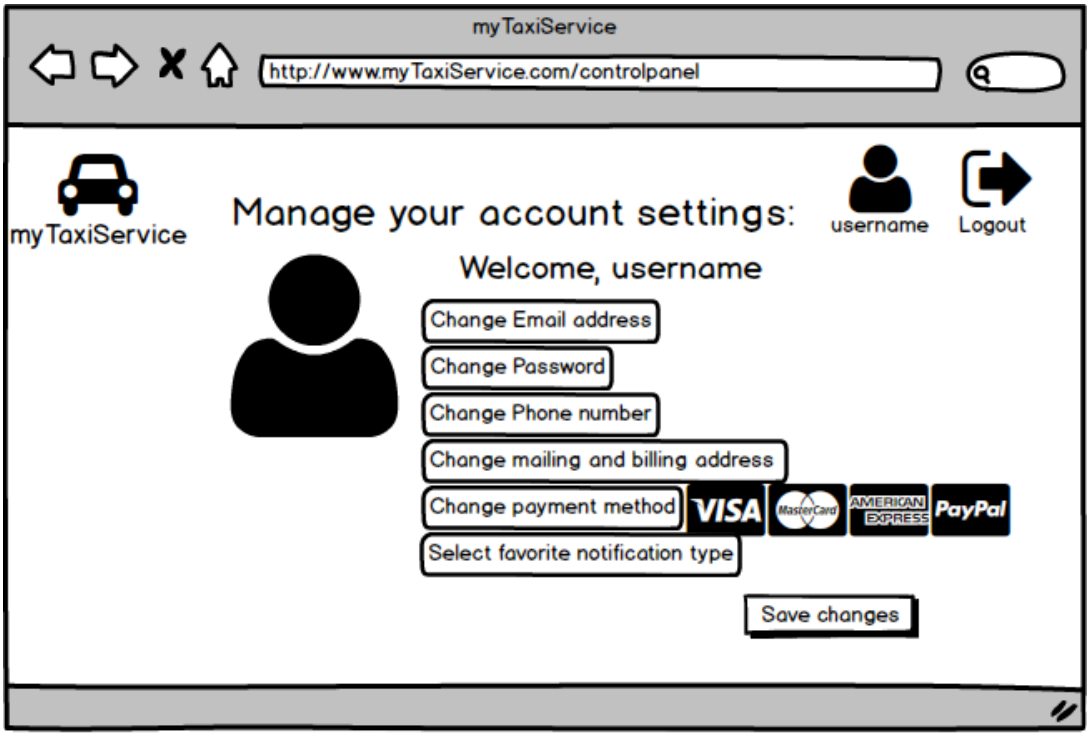


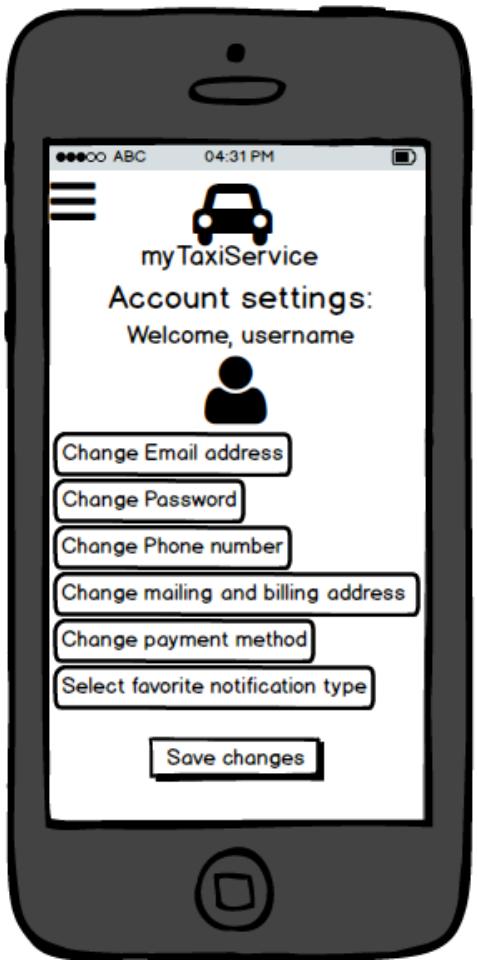
4.0.0.5 Your reservations The user can see both the active reservations and the past ones. S/he can edit the active reservations within the established time frame before the meeting time.



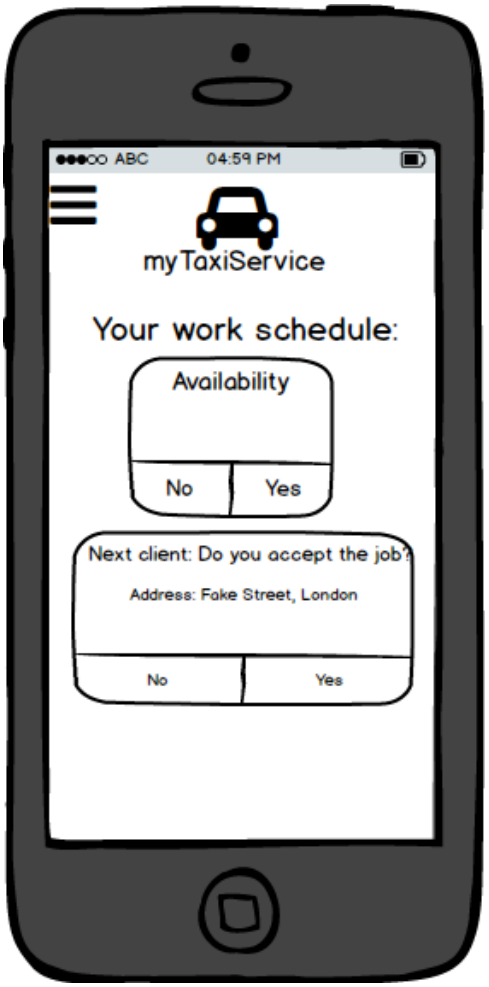


4.0.0.6 Your profile The user can edit the personal profile modifying the password, phone number, email address, permanent address, payment method and notification type.

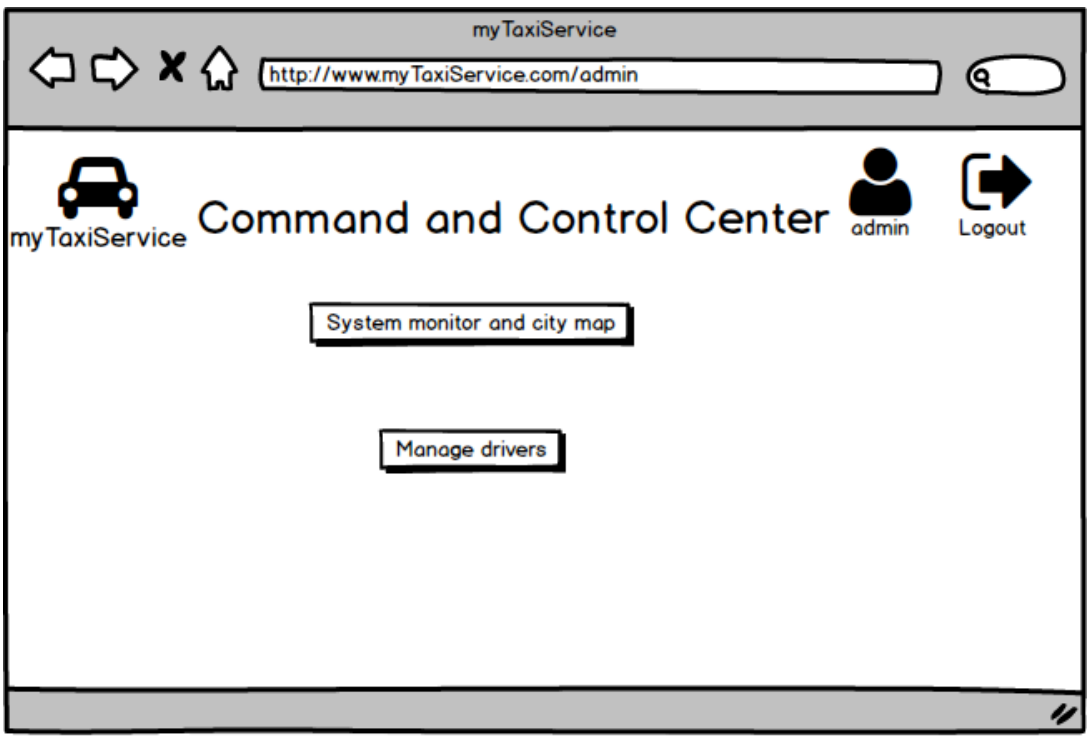




4.0.0.7 Manage jobs The taxi driver's home page: s/he can accept or reject the requests forwarded by the system and set her/his own "availability".



4.0.0.8 Administrator panel The administrator's home page: s/he can monitor the whole system or manage taxi drivers inserting new employees in the system DB or modifying/deleting existing ones.



5 Requirements Traceability

6 References

- Testo: *Principi di Ingegneria del Software* 5ed, Roger S. Pressman, McGraw-Hill
- Material from Wikipedia
  - Thin client: https://en.wikipedia.org/wiki/Thin_client
  - Event-driven architecture: https://en.wikipedia.org/wiki/Event-driven_architecture
  - Factory method pattern: https://en.wikipedia.org/wiki/Factory_method_pattern
  - Singleton pattern: https://en.wikipedia.org/wiki/Singleton_pattern
  - Observer pattern: https://en.wikipedia.org/wiki/Observer_pattern
  - Facade pattern: https://en.wikipedia.org/wiki/Facade_pattern

7 Appendix

7.1 Software and tools used

- TeXstudio 2.10.4 (http://www.texstudio.org/) to redact and format this document.
- Astah Professional 7.0 (http://astah.net/editions/professional): to create Use Cases Diagrams, Sequence Diagrams, Class Diagrams and State Machine Diagrams.
- Microsoft Office Visio Professional 2016

7.2 Hours of work

The time spent to redact this document:

- Baldassari Alessandro: 30 hours.
- Bendin Alberto: 30 hours.
- Giarola Francesco: 30 hours.