POLITECNICO MILANO 1863

myTaxiService

DESIGN DOCUMENT

Paolo Antonini 858242
Andrea Corneo 849793

version 1.1 – 21st January 2016

Contents

1 Introduction

1.1 Purpose

After the presentation of our *Requirement analysis and specification document*[1] on 6th November 2015, we are requested to provide a functional description of myTaxiService project. The result is this document, which tries to give a comprehensive look on the architecture of the system.

As a consequence, this document addresses professionals, and primarily developers, who will use it as a solid basis for their implementation work. A section is also dedicated to designers, who will have to work on the user interfaces of the applications and website.

[1] Briefly, RASD.

1.2 Scope

The behaviour of myTaxiService system was completely outlined in the RASD, so it is appropriate to refer to that document for a general description of it. In the following, we are taking for granted a reasonable familiarity with its contents.

Here we will detail the overall design of the system and its architecture, and we will also show how the components of the system interact to accomplish the specific tasks. An analysis of the main algorithms that govern the operation of myTaxiService is provided, as well, to aid developers in their work. Moreover, to complete the specification, we give an overview on the user interface design.

This document being the natural follow-up of the analysis started in the RASD, we will try to provide a solid cross-referencing to that document, for the sake of coherence and consistency.

1.3 Definitions, acronyms, and abbreviations

Please refer to the corresponding section in the RASD for the definitions of words used in the document. Some technical expressions and abbreviations are used in the following, but definitions are given when necessary.

1.4 References

As it was already mentioned, throughout this writing, we keep the consistency and traceability with the the *Requirement analysis and specification document*. The RASD can be retrieved on the following link: `https://github.com/Cordaz/SE2_AntoniniCorneo/raw/master/Deliveries/1_RASD.pdf`.

1.5 Overview of the document

This document develops as follows. Chapter 2 analyses the architectural design of the system, at different levels of detail and from different points of view. Some UML diagrams will be used as a support. Chapter 3 defines the most relevant algorithms on which the system relies. In chapter 4 we offer an overview on how the user interfaces of your system will look like, through some mock-ups.

Section 5.2 is to be considered external to the document.

2
Architectural design

2.1 Overview

This chapter, arguably the longest of this document, analyses all the different aspects of the architecture of myTaxiService system, at different levels of detail and from different points of view.

In section 2.2 we give a high level description of the structure of the system. Inevitably this section will be pretty abstract.

Then, section 2.3 focuses on the components of the system. With the word *component* we refer to an independent software element that encapsulates a set of related functions. This is to say that a component is defined by its interfaces. The interfaces between the components are fully analysed in section 2.4. In opposition to the focus on the interactions within the system, the deployment view in section 2.5 provides further details on the physical structure of its various parts.

In the runtime view (section 2.6) we show the behaviour of the system in case of a request and a reservation.

Finally, section 2.7 is meant to give further details and explanations about our design and architectural choices.

UML being an agreed and relatively easy to understand language, in the whole chapter we will make extensive use of it.

2.2 High level components and their interaction

For our myTaxiService ecosystem, we will adopt the Java Enterprise Edition[1] application model for enterprise applications. This will allow the design, building and production of a solid, fast and reliable system, with a special focus on money and resources. We believe that such an architecture can be easy to understand and to set up; also, it is highly scalable, which is a huge bonus since we cannot foresee the evolution of the use of the system in the city.

[1] In the following, simply JavaEE.

The JavaEE platform uses a distributed multitiered application model. Application logic is divided into components according to function, and the application components that make up a JavaEE application are installed on various machines depending on the tier to which the application component belongs. Figure 2.1 shows the model we will adopt, which consists of four tiers:

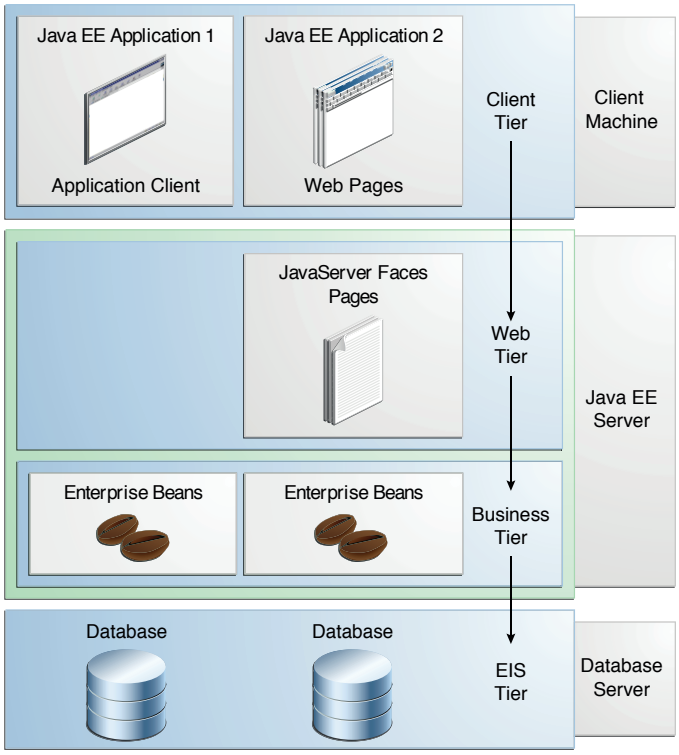

Figure 2.1: High level architecture. Source: *Java Platform, Enterprise Edition. The Java EE Tutorial, Release 7*. Oracle, September 2014.

Client tier it contains all the components which run on the client machine (applications, web pages); in our case myTaxiWeb web pages, and myTaxiApp, myTaxiAssist mobile applications are the clients. These all are *thin clients*, which means that they do not directly query the database, nor execute complex operations.

Web tier the components of this tier run on the JavaEE server; this tier is intended to manage the data flow between clients and JavaEE.

Business tier this tier, which runs on the JavaEE server as well, contains the so called *enterprise beans*. Enterprise beans handle business code, which is logic that govern myTaxiService system; to do so, they also retrieve data from storage, processes it (if necessary), and sends it back to the client program, through the web tier.

EIS tier this tier, typically, handles EIS[2] software and includes enterprise infrastructure systems, such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. In our specific context, JavaEE application components might need access to enterprise information systems for database connectivity.

[2] EIS stands for Enterprise information system.

2.3 *Component view*

Now let us refine what was given in section 2.2. In the following component diagram (figure 2.2) the architecture of the system has been expanded.

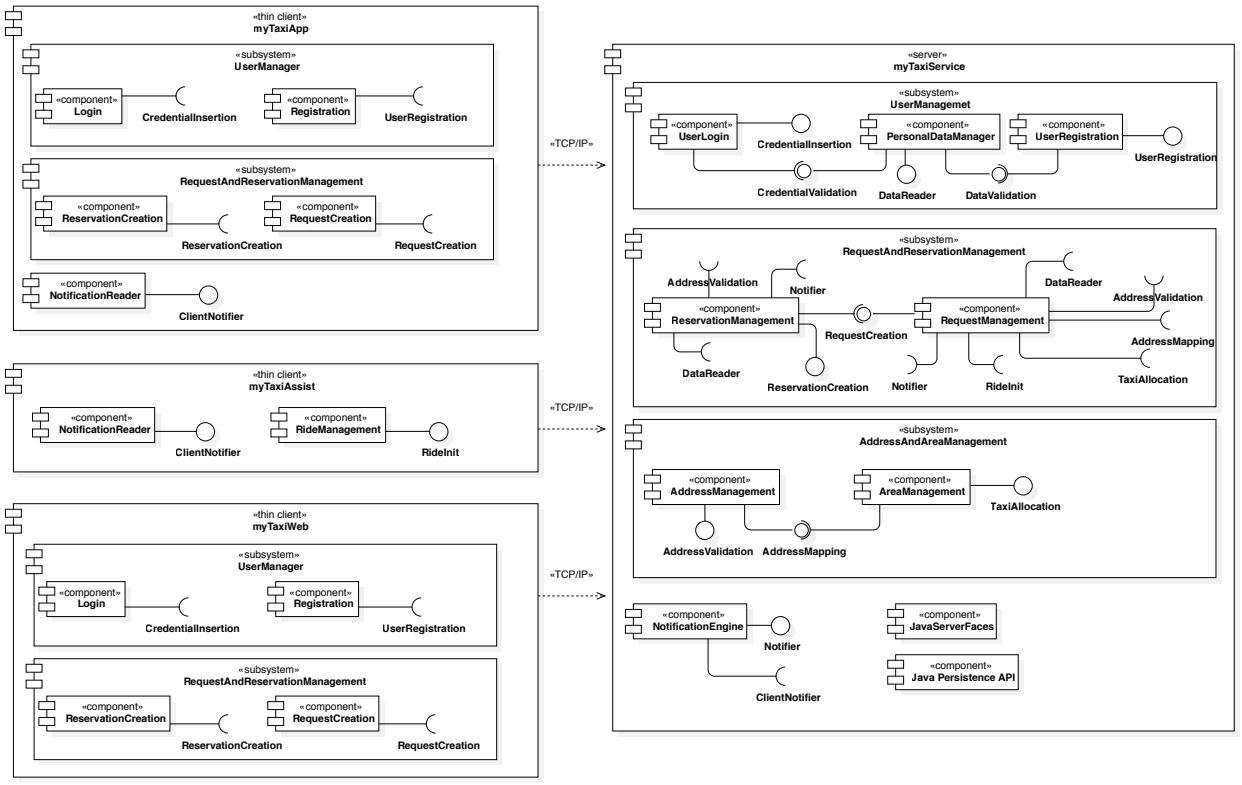

Figure 2.2: Component diagram.

In the diagram, each application of the system (myTaxiApp, myTaxiWeb, myTaxiAssist, myTaxiService) is represented, together with its own components. Moreover, for the sake of clarity, some subsystems have been introduced, to gather logically similar components. Each component may offer or require some interfaces. These interfaces reflect the services that the component provides (which are indicated in UML by a circle at the end of a line from the component icon) and the services that the component requires to operate correctly (the symbol for this kind of interface is a semicircle at the end of a line from the component icon).

For example, let us consider the `RequestAndReservationManagement` subsystem in myTaxiService server. It has two components, `ReservationManagement` and `RequestManagement`. The former, as the name suggests, handles the reservations, from the reception until their conversion to requests[3]. In order to do the conversion, the component needs a service, namely `RequestCreation`, which is conveniently offered by `RequestManagement` component.

For the sake of clarity, we preferred to draw only the connections between the applications, and to exclude all the links between interfaces that would lie outside the subsystems. By doing so, we avoid the otherwise inevitable tangle of connections. Correspondence is given by giving the same name to offered and required services; however, further details about the interfaces are provided in section 2.4.

Before we proceed with our analysis, we would like to focus our attention on two components in myTaxiService server, `Java-`

[3] Remember that our system confirms the reservation and allocates a taxi 10 minutes before the meeting time with the customer, as though it was a request.

ServerFaces and JavaPersistenceAPI. They both provide all the JavaBeans components[4] which are essential for the operation of the system. The former handles the server side for the clients, the latter instead makes the usage of MySQL database management system possible.

[4] *JavaBeans* denotes a set of standards and conventions useful to build components in Java language.

2.4 Component interfaces

In this section we are going to analyse the most significant interfaces between components which were shown in figure 2.2. In the following table, the components are presented along with the interface they offer and a brief description.

Component	Interfaces	Description
AddressManagement	AddressMapping	Provides the methods to map addresses into the system, returning the corresponding area.
	AddressValidation	Offers the methods to validate addresses, checking their existence in the database. Also converts addresses in GPS coordinates and vice versa.
AreaManagement	TaxiAllocation	Provides the methods for managing the taxi allocation (e.g., enqueue and dequeue, and the change of a taxi availability).
NotificationEngine	Notifier	Provides the methods for the notification of client systems.
NotificationReader	ClientNotifier	Provides the methods to notify the customers.
PersonalDataManager	CredentialValidation	Provides the methods that check the personal credentials in the database.
	DataValidation	Provides all the methods to validate personal data, for instance the correctness of the name (it cannot contain numbers) or of a birthdate (it shall not be in the future).
	DataReader	Offers the methods to access customer data, stored in the database.
RequestManagement	RequestCreation	Offers the methods to make a request. It validates data, interacts with the required components to allocate a taxi and stores the request in the database.
ReservationManagement	ReservationCreation	Provides the methods to make a reservation. It interacts with other components for data validation and storing.
RideManagement	RideInit	Provides the methods necessary to instantiate a ride for the driver. This component interacts also with the on board navigator.

COMPONENT	INTERFACES	DESCRIPTION
`UserLogin`	`CredentialInsertion`	Offers the methods useful to log in a User (both a registered customer and a taxi driver). It interacts with `PersonalDataManager` component to complete the login and start the user session.
`UserRegistration`	`UserRegistration`	Provides the methods for the insertion of a new customer in the database, validating his data.

Notice that PersonalDataManager `PersonalDataManager` component allows access to the personal data, which are stored in the database. Moreover, it provides validation methods on all the personal data.

2.5 *Deployment view*

After focusing on the logical structure of the system and on the interactions within, it is appropriate to provide a lower level view on the architecture of the system. In particular the deployment diagram in figure 2.3 shows the distribution of the concrete software artefacts[5] over the various computational nodes of our system.

[5] That is, files and libraries.

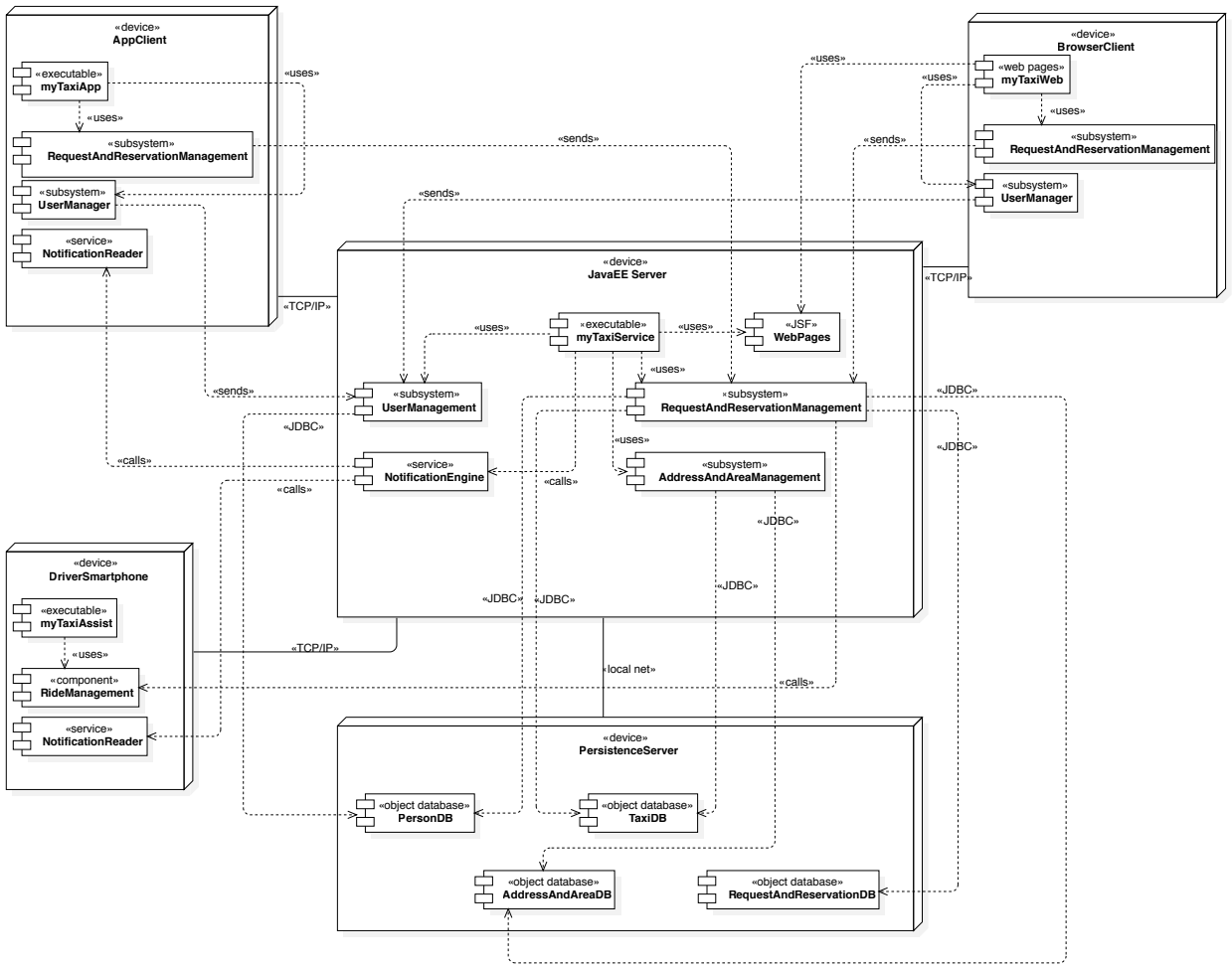

Figure 2.3: Deployment diagram.

To reflect the architecture outlined in section 2.2, the diagram develops over three types of hardware systems: `AppClient`, `BrowserClient` and `DriverSmartphone` are the three possible client machines, then we have the `JavaEE Server`, intended for the business logic, and the `PersistenceServer`, dedicated to the database.

The main interactions between the components have been drawn. Moreover, the nature of the various components (executable, service) has been specified, so as to give a more complete view on the architecture of the system.

2.6 *Runtime view*

Before introducing a runtime view of myTaxiService (figure 2.6), we would like to expand the two sequence diagrams which were presented in the RASD regarding the request and the reservation of a taxi ride (in that document they are, respectively, figure 3.2 and figure 3.10) in order to show the actual behaviour of the system.

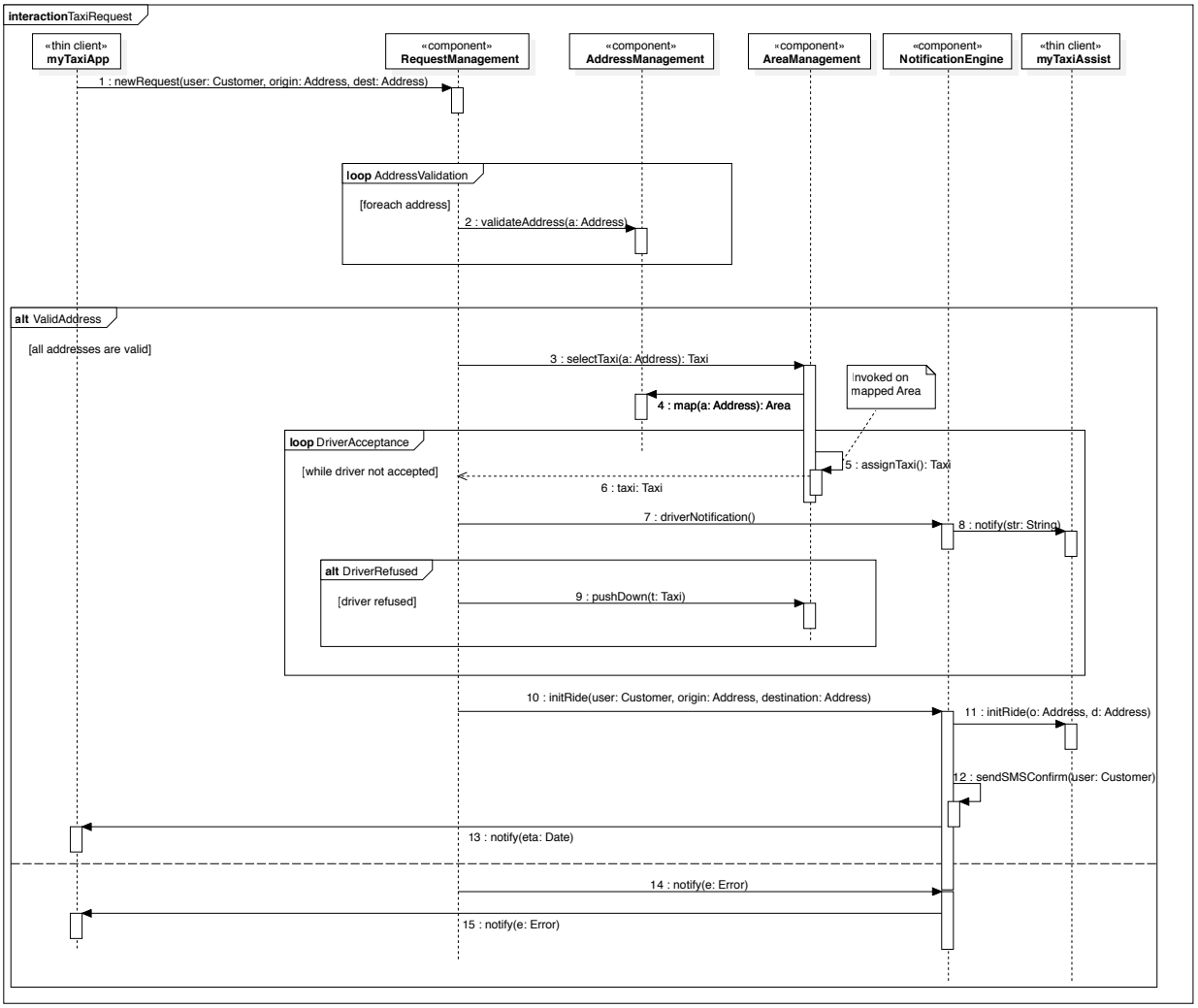

Figure 2.4: Taxi request sequence diagram.

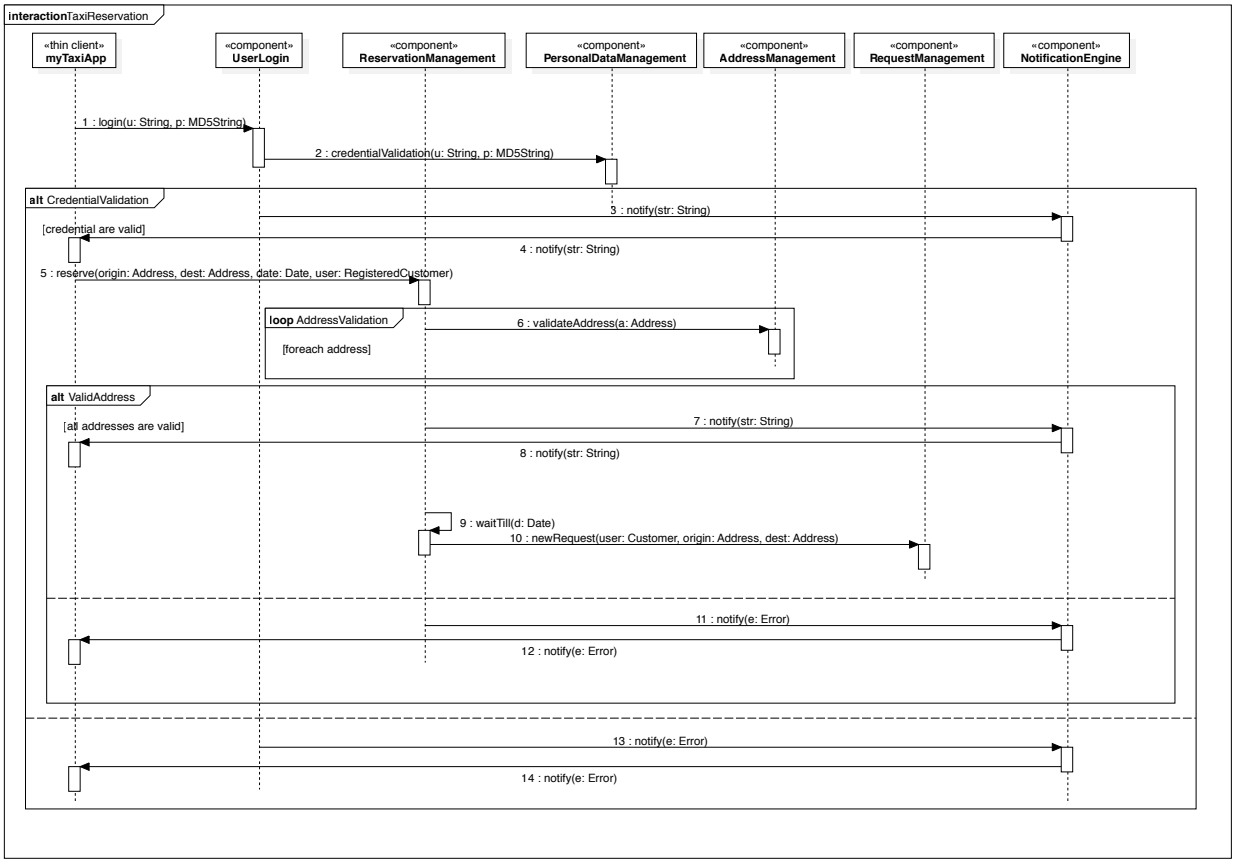

Figure 2.5: Taxi reservation sequence diagram.

Notice that the taxi reservation sequence diagram exploits the request procedure to complete, on the basis of what is stated in sidenote 3 on page 9.

Now, to complete our extensive analysis on myTaxiService ecosystem, we introduce here a runtime view of the system, by means of a UML-like diagram[6]. In this diagram we represent the instances active in the system during its operation.

In particular, we are supposing the following flow of events:

1. `customer1` makes a reservation for a taxi ride (`reservation1`) through myTaxiApp;
2. `reservation1` is converted into a request (`request1`), and `taxi1` is allocated;
3. `customer2` requests a taxi ride (`request2`) via myTaxiWeb, and `taxi1` is again allocated[7].

This is obviously an ideal and simplified world, which nevertheless should make clear how the system actually works.

Some of the links between instances have been omitted, to preserve readability (for example, `reservation1` and `request1` should be both linked to `customer1`, but the association was omitted, since trivial, as the request is automatically created as a result of the reservation).

[6] Notice that no UML standard is defined to represent a runtime view. Nevertheless, we will use UML elements anyway, to maintain graphical consistency.

[7] Time constraints compliance is intended: `request2` occurs after the completion of `reservation1`/`request1` task.

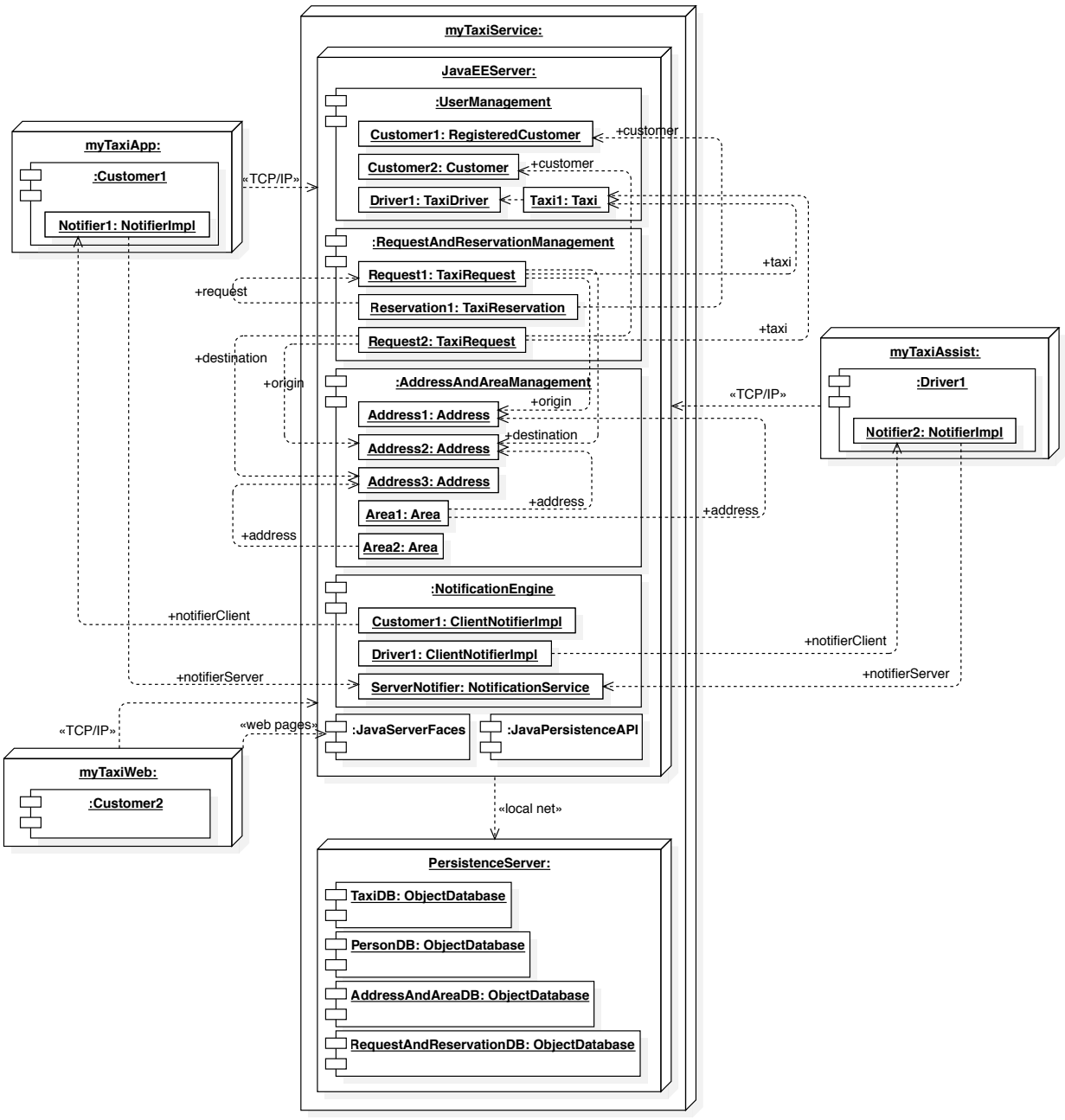

Figure 2.6: Runtime view.

2.7 *Selected architectural styles and patterns*

The approach we used to design the whole system traces back to the well known Model-View-Controller (MVC) architectural pattern.

Basically, this pattern separates presentation and interaction from the system data by structuring the system into three logical components[8] that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction and passes them to the View and the Model (see figure 2.7).

[8] Notice that here we are not using the word *component* with the same technical meaning as in section 2.3, but in a broader sense.

This pattern allows the data to change independently of its representation and vice versa, which is particularly useful when there are multiple ways to view and interact with data and the evolution of the system is unknown. Also, it supports presentation of the same data in different ways.

The main drawback is that it may involve additional code and code complexity, but we believe that the advantages largely offset this downside.

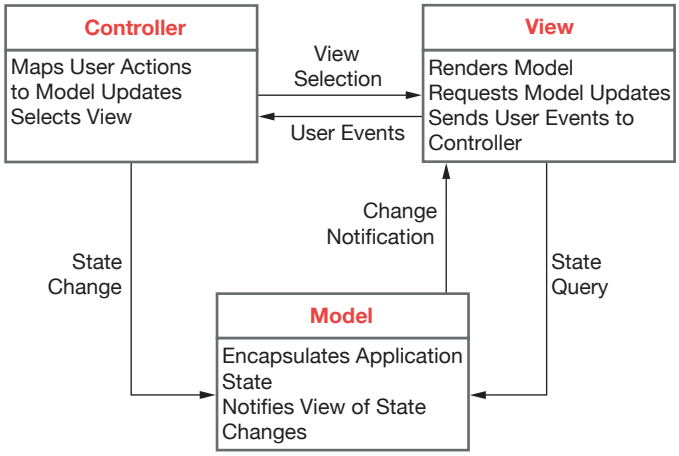

Figure 2.7: Model-View-Controller general organisation.
Source: Ian Sommerville. *Software Engineering*. 10th edition. Pearson, 2015.

Referring in particular to the main components of our system (see sections 2.3 and 2.4) we can draw the following correspondences:

Model for example, the components `PersonalDataManager` and `AddressManagement` are part of the Model of the system, as they deal with the system data;

View the three clients (myTaxi App, myTaxiWeb and myTaxiAssist) constitute, trivially, the View;

Controller the `RequestAndReservationManagement` subsystem, for example, receives and manages requests and reservations from customers, so it belongs to the Controller.

3
Algorithm design

In this chapter we are going to outline the main algorithms that will govern our ecosystem. The application logic will reside in myTaxiService core system, whereas the applications myTaxiApp, myTaxiAssist and the website myTaxiWeb will offer only the presentation layer (that is, the layer concerned with presenting the information to the user and managing user interactions).

To specify the algorithms in this chapter we will take advantage of the great expressivity and conciseness of pseudocode. We will present them as high-level procedures, in order not to make premature implementation decisions, which instead should be made at the time of coding. That is why many of the procedures mentioned in the following listings will not be further specified.

As one may remember, the city is divided into small areas, each of which has an associated taxi queue. Basically, the whole system has to manage the taxi queue of each area. While a taxi driver is waiting for a call in an area, his taxi is enlisted in the queue of that area: upon the reception of a request, the system forwards it to the first taxi in the queue of the area, according to the algorithms that will be specified soon.

3.1 Theoretical recall

First, let us make a brief theoretical introduction, for the sake of clarity. Notice that the word *queue* has a technical meaning, which nevertheless corresponds to the common idea of a queue. This means that a queue is a data structure which models, for example, a line of customers waiting to pay a cashier. The queue has a head and a tail. When an element is inserted in the queue, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element to be removed by the queue is always the one at the head of it, like the customer at the head of the line (who, by the way, has waited the longest).

This is to say that the management policy of a queue is FIFO[1]. We call the insertion operation on a queue *enqueue*, and we call the deletion operation *dequeue*.

We prefer not to detail the specific procedures to manage a queue for a number of reasons: they are well documented in every

[1] FIFO stands for "First In, First Out", which is the mnemonic for the queue management policy.

valid book about algorithms[2]; they are (or rather, they should be) well known to every good developer; they strongly depend on the implementation choices. We believe that other algorithms deserve greater attention.

[2] e.g. Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

3.2 *Implementation in myTaxiService*

We can now refer to the specific context of our project, in order to provide further details. Every area has an associated queue Q. Its elements will be taxis, and their number is represented by the attribute $Q.length$. Since there is a one to one relationship between areas and queues, we may use the two words as synonyms.

Suppose that a request is made in the area which has Q_0 as associated queue. The system then looks for an available taxi in that area; if none is available, then it chooses one from the nearest area which has the longest queue, iteratively, up to a distance of two areas, then it stops[3].

[3] Remember that to cross an area a taxi may need up to 10 minutes, and that in section 2.5 of the RASD we assumed that the number and distribution of taxi cabs in service are always sufficient to serve 100 % requests within 30 minutes.

In the following algorithm, *validQueues* is the set of queues whose length is greater than 0; $final$ is, among those, the longest one, thus the one to consider.

Algorithm 1: Procedure to select a taxi.

```
1:  procedure SelectTaxi(Q_0)
2:      i ← 0                                  ▷ i: counter for the distance.
3:      repeat
4:          validQueues ← null
5:          for all Q : Distance(Q, Q_0) = i do
6:              if Q.length > 0 then
7:                  validQueues ← validQueues + Q
8:          for all Q ∈ validQueues do
9:              if Q.length > final.length then
10:                 final ← Q
11:         i ← i + 1
12:     until (final ≠ null) ∨ (i ≥ 3)
13:     t ← Dequeue(final)                     ▷ t: taxi to return.
14:     return t
```

The cycle in lines 3 to 12 is obviously crucial in the algorithmic definition; its body is run until either one or no valid taxi cab is found. If no taxi is found at distance 0 ($i = 0$), which means in area Q_0, then the algorithm performs the same set of operations on the areas at distance 1, and if needed also at distance 2. The first **for** loop (lines 5 to 7) scans all the queues at distance i to find and store in *validQueues* those whose length is not null, that is, those which enlist available taxis. Once the *validQueues* set has been (possibly) populated, then the second loop (lines 8 to 10) selects the longest one within the set.

The **repeat** cycle ends either when a queue from which to extract a taxi is selected, or when no valid area at a suitable distance (namely, at more than two areas from Q_0, which means at more

than 30 minutes by car) is found.

Algorithm 1 presented above is used in a greater procedure (algorithm 2), which aims at effectively assigning a taxi to a request r_0. The procedure is shown next.

Algorithm 2: Procedure to assign a taxi.

```
1:  procedure AssignTaxi(r_0)
2:      Q_0 ← GetQueue(r_0)
3:      repeat
4:          t ← SelectTaxi(Q_0)
5:          NotifyDriver(t)
6:          StartTimer(timer, 2)            ▷ Set timer to two minutes.
7:          while (Status(t) ≠ "accepted") ∧ ¬(timer.out) do
8:              Wait()
9:          if Status(t) ≠ "accepted" then
10:             Enqueue(Q_0, t)
11:     until Status(t) = "accepted"
12:     Assign(t, r_0)
```

After getting the queue Q_0, corresponding to the area from which the request r_0 arrived (line 2), a cycle begins (lines 3 to 11). The cycle is designed to select a taxi with the procedure shown in algorithm 1 (line 4), and then to wait for the selected taxi driver to confirm or for a timer to expire (lines 6 to 8). Moreover, if the taxi driver does not confirm (that is, the Status function called on taxi t returns either `"rejected"` or null), then the taxi t shall be enqueued (hence put at the tail of the queue). On the contrary, if taxi t accepts the request (Status(t) returns `"accepted"`), the cycle ends and the taxi t is assigned to the request r_0 (line 12).

Given that in the city there are only taxi cabs for four passengers, procedure in algorithm 2 is performed as many times as needed.

3.3 *Final considerations*

As it was already mentioned, the algorithms in this chapter are presented as high-level procedures, as some implementation decisions should be made only at the time of coding. As a result we are not specifying what data structure should be used to actually implement a queue, or, for instance, how the condition $Q : \text{Distance}(Q, Q_0) = i$ (line 5 in algorithm 1) should be checked.

As a consequence, the algorithms presented above cannot be directly translated in actual code. However, they should not be regarded as the one and only source of information for developers: the whole chapter, which this section closes, contains useful information for development phase.

4
User interface design

As the product we are designing is intended to be distributed to general public, its user interfaces shall be friendly and appealing. Nevertheless, the purpose of this document is to guide the many professionals who will work on the project, without excessively constraining their abilities and creativity. For this reason, we are going to provide some mock-ups of the two mobile applications (myTaxiWeb is omitted, for the sake of compactness, as it does not offer to users anything more than myTaxiApp). The following mock-ups should be intended as a reference for the structure of each screen, and by no means shall be used as an "artistic" model.

While working on the user interfaces for this project, the requirements expressed in the RASD (paragraph 2.1.2 and section 2.3) must be taken into consideration, together with this chapter.

4.1 myTaxiApp

Upon opening the application, the user is presented with the following options: log in his account, sign up, and request a taxi ride (as a guest).

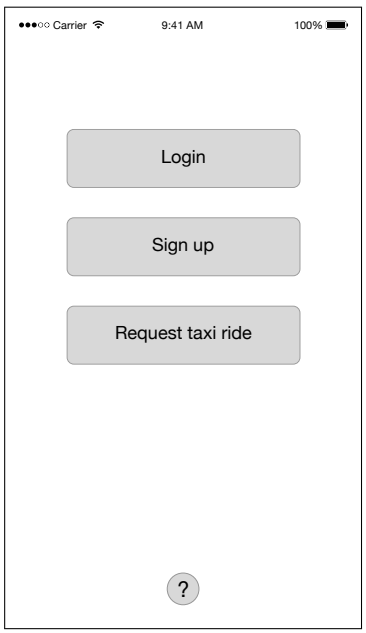

Figure 4.1: Home page, presented to the user after his opening the application.

If he chooses to log in, then he will need to fill in the form with his email and password details. Obviously he needs to be registered on the system, otherwise the login procedure will fail. Should he not be registered, he can sign up, by providing some personal details (name, phone number, email).

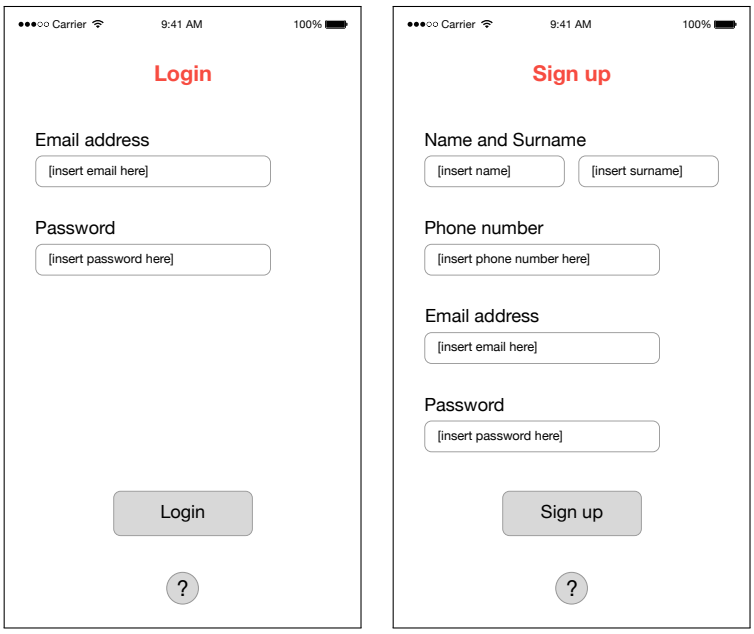

Figure 4.2: Login and Sign up pages.

After the login, the user (now he is identified in the system as a registered customer) has the possibility to request or reserve a taxi, manage his favourite addresses (a convenient feature which lets users to save some addresses, for a faster retrieval) and also his own account.

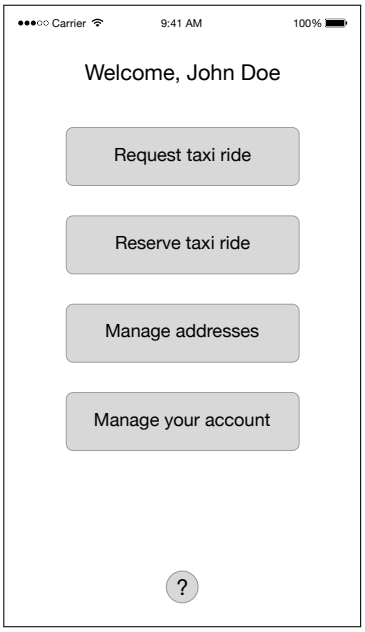

Figure 4.3: Registered customer's home page.

Actually, the screen which allows to request a taxi is the same for registered and guest users, as origin, destination and number of passengers shall be specified in both cases. However, we have to make some distinctions: only registered customers can select favourite addresses as origin and destination; name and number details are requested only to guest users.

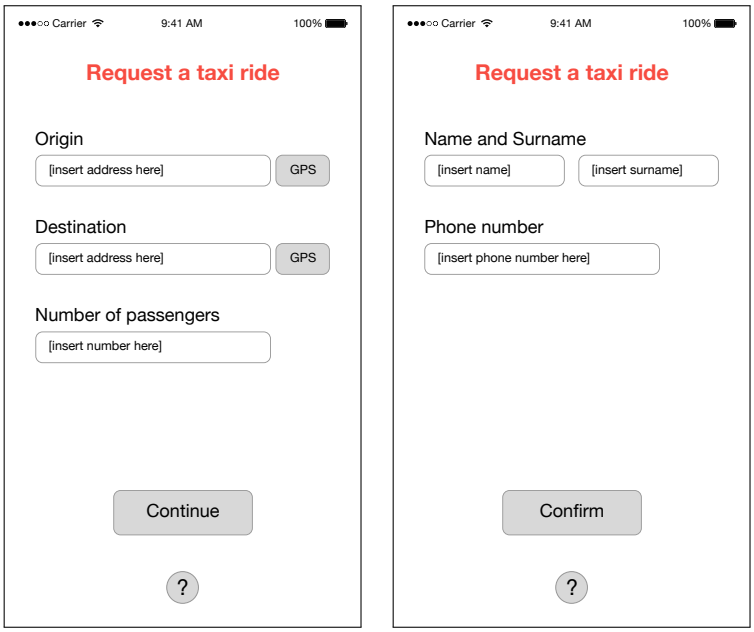

Figure 4.4: Request a taxi ride (mind the distinctions for registered and guest users).

Registered users are given the possibility to reserve a taxi ride. This means that they have to select date and time for the ride.

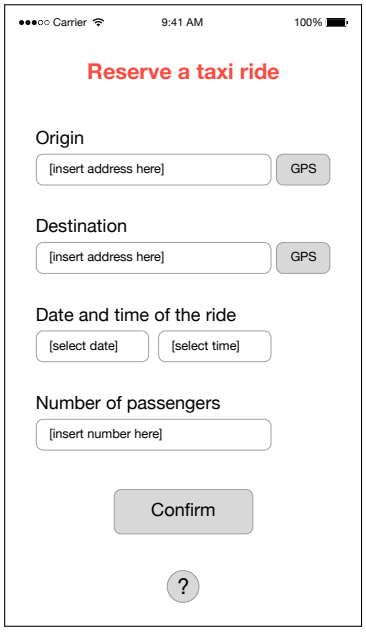

Figure 4.5: Reserve a taxi ride.

Also, registered users can have a list of favourite addresses, which they can manage by adding, deleting and editing addresses.

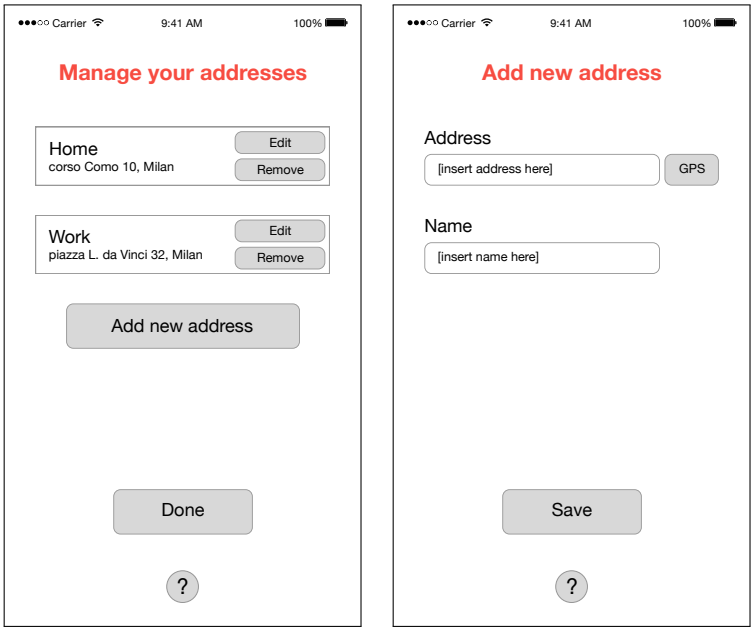

Figure 4.6: Manage the list of favourite addresses.

Finally, registered users can obviously manage their own account by changing the phone number, email address or password, and even delete the account (these functions are not expanded since trivial).

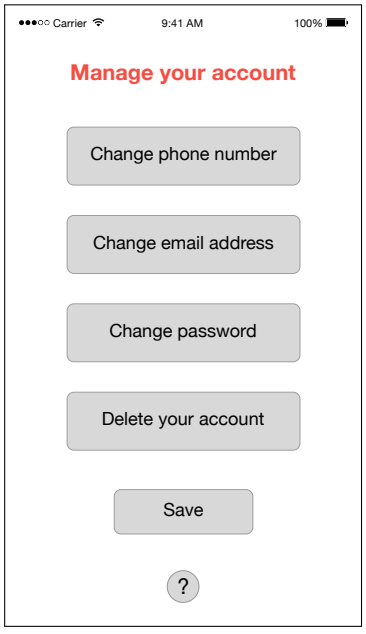

Figure 4.7: Manage account.

4.2 *myTaxiAssist*

Taxi drivers receive their credentials from the municipality office in charge of the taxi service. As a consequence, upon opening myTaxiAssist application, they are presented with the login form. After the login process, they gain access to the functionalities of the application, which are the report of their availability and of a taxi accident.

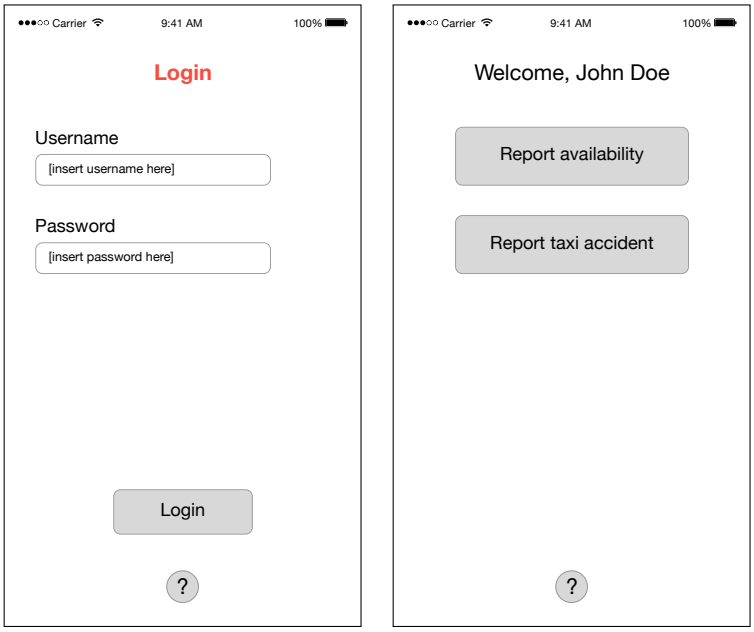

Figure 4.8: Login and homepage for taxi drivers.

The taxi driver has to report his availability: he has to announce to the system the begin and the end of his shift, by selecting respectively "On duty" and "Off duty", and also the successful end of the ride, by pressing on "Available" (this also enlists him in the queue of the current area).

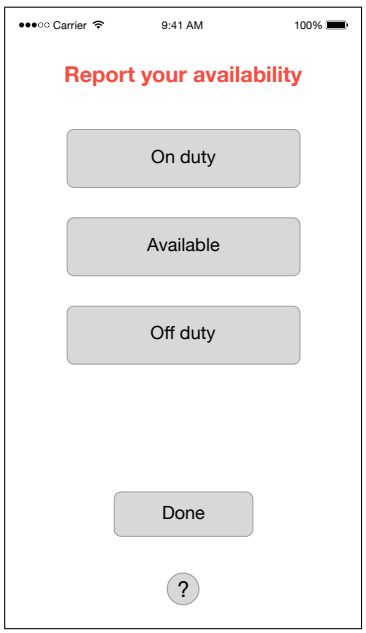

Figure 4.9: Availability report.

Should an accident happen, the taxi driver shall report the situation to the system. In particular, it is important to know whether some passengers are in the taxi or not (see paragraph 3.1.3 in RASD for further details on this particular situation).

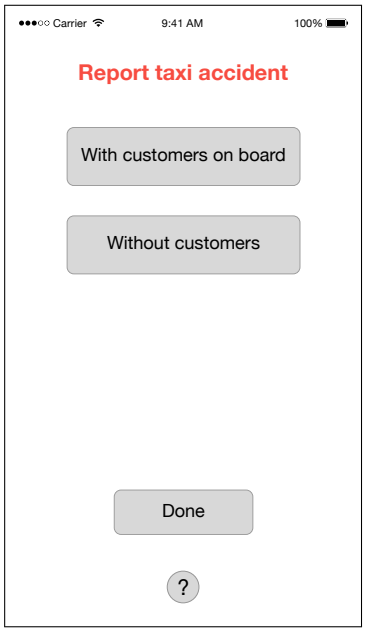

Figure 4.10: Accident report.

At any moment while available (that is, while he is neither driving somewhere a customer nor off duty), the taxi driver can receive the notification of a new request for a ride. The notification informs the driver of origin, destination and number of passengers of the requested ride.

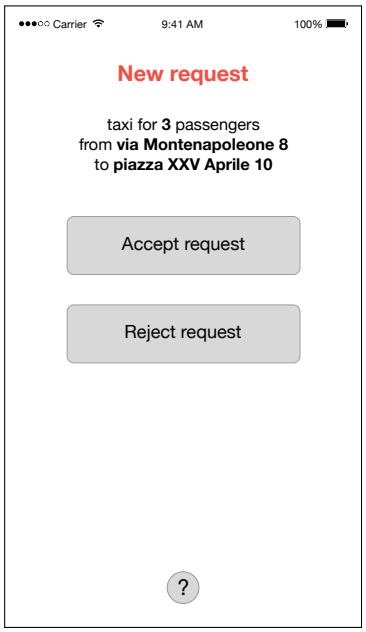

Figure 4.11: New request notification.

4.3 Final considerations

Before concluding, we find it appropriate to point out a few things about the chapter.

As it was requested in RASD (paragraph 2.1.2 and section 2.3), every screen contains a link to help pages. Also, every function can be exploited in no more than two screens, to improve usability.

With these mock-ups we presented only the functional screens of the two applications[1], which means that other informative pages can be inserted. A tenet to keep always in mind, however, is the ease of use.

[1] As we already mentioned, we are not presenting mock-ups for myTaxiWeb, since they would be similar to those of myTaxiApp, from the point of view of offered functionalities.

5
Requirements traceability

To conclude this document, we would like to highlight some significant correspondences between this document and the previous *Requirement analysis and specification document*. Indeed, functional and non-functional requirements expressed there were kept as a guide while writing this document.

5.1 Functional requirements

Functional requirements are presented in a compact way in the RASD in section 3.2.

The component view (sections 2.3 and 2.4) was designed bearing in mind what was stated there. Every component contributes to comply with those requirements (most of the correlations are trivial). Requirement number 6 was particularly useful to design the main algorithms of the system (chapter 3).

5.2 Non-functional requirements

The architecture presented in sections 2.2 and 2.7 wants to be the counterpart of sections 3.3 to 3.6 in the RASD, where non-functional requirements were outlined. It was designed to comply as much as possible to what was stated in those sections.

Nevertheless, there are some non-functional requirements which cannot be defined by any architectural decision (for instance, maintainability of the code, defined in paragraph 3.6.4 in the RASD).

Appendix

Hours of work

The writing of this document took the following amount of time:

Paolo Antonini 38 hours.

Andrea Corneo 30 hours.

Version control

- **1.0**, 6th November 2015: first release;
- **1.1**, 21st January 2016: final release, with graphical fixes.