Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

---

myTaxiService
Design Document

Supervisor:
Prof. Elisabetta Di Nitto

Students:
Alberto Gasparin
Vito Matarazzo

Year 2015-2016

Contents

Revision History

Date	Reason for changes	Version
December 4, 2015		v1.0

1 Introduction

1.1 Purpose

This document describes the system architecture and design of the myTaxiService project. The architecture represents a high level view of the components of the system, while the design describes interactions between the different modules and how they will be mapped into programming elements in the future development process. This document will explain the architectural and design decisions and tradeoffs chosen in the design process and their justifications. However, it is intended to be constantly updated and revised, as the architecture decisions can change during the development process, so this document will always reflect the current state of the system architecture.
This document is intended to be used by all team members during the system development. It will also be used by the project supervisor to get insight into the project work. This is a technical document and it is not initially intended for the customers. However, technically educated customers may also get a general overview of the project using this document. Finally, this document is intended for the developers who will be able to get the detailed description of the system architecture in order to continue working on this project in future.

1.2 Scope

The myTaxiService system is divided into three main components: Taxi Driver (Mobile) Application, Passenger (Mobile/Web) Application and Server. This document will describe each component and all the interfaces between them, as well as the interfaces to external components and users. The architectural descriptions provided concern component view, deployment view, runtime view, component interfaces, selected styles and patterns. These descriptions will consider all the main functionalities already explained in the RASD, that are:

- manage the profiles of the different types of registered users, which can be passengers, taxi drivers, administrators and developers.
- allow users to make taxi requests or reservations.
- manage the taxi drivers' status and the taxi queues of the city.
- notify taxi drivers of incoming requests.
- provide API to developers that want to extend the current application.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definition

Functional requirement: a requirement that specifies a function that a system or a system component must be able to perform.

Queue: a list in which items are appended to the last position of the list and retrieved from the first position.

Requirement: (1) a condition or capability needed by user to solve a problem or achieve an object. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation or capability as in (1) or (2).

User: the person, or persons, who operate or interact directly with the product. The user(s) and the customer(s) are often not the same person(s).

Client: a piece of computer hardware or software that accesses a service made available by a server.

Server: a computer program or a machine that waits for requests from other machines or software (clients) and responds to them. The purpose of a server is to share data or hardware and software resources among clients.

1.3.2 Acronyms

API : Application Programming Interface.

DB : Database.

ETA : Estimated Time of Arrival.

GPS : Global Positioning System.

HTML : HyperText Markup Language.

HTTP : HyperText Transfer Protocol.

MVC : Model-View-Controller design pattern.

UML : Unified Modelling Language.

XML : extensible Markup Language.

1.3.3 Abbreviation

1.4 Reference Document

Specification Document: myTaxiService Project AA 2015-2016.pdf.

RASD document: RASD.pdf on GitHub.

IEEE Std 1016-2009, IEEE Standard For Information Technology - System Design - Software Design Descriptions.

IEEE Std 42010, 2011 Edition, IEEE System and Software Engineering - Architecture Description.

1.5 Document Structure

The rest of the document is organized in :

- Section 2: Architectural Design. This section will describe all the architectural and design decisions taken, from the general overview to the different specific views (component, deployment, runtime) and the specific styles and patterns adopted.
- Section 3: Algorithm Design. This seciton will focus on the definition of the most relevant algorithms that will be implemented in the project, described in pseudocode.
- Section 4: User Interface Design. As all the relevant mockups have already been described in the RASD, this section will show how the users will navigate through those screens during some of the main interactions with the system.
- Section 5: Requirement Traceability. This section will explain how the requirements defined in the RASD map into the design elements defined in this document.

2 Architectural Design

2.1 Overview

The system to be created is a web application that will allow a simple communication between passengers and taxi drivers, while the system acts as a dispatcher. So it will need two types of client applications, one for the passengers and one for taxi drivers, and a main server which will answer to the requests coming from them and perform all the necessary operations. Besides the system will need also a database to store all the data necessary for the system, like users' accounts and rides' information. Therefore, we have decided to adopt a 3-tier architecture for our system, composed of these parts:

- client (both the passenger's and the taxi driver's one);
- application server;
- database server.

This type of architecture is very common especially for web and mobile applications, as it allows to easily separate the user interface from the fundamental logic of the system, and from the data processing.

2.2 High level components and their interaction

The high level components of the system are:

Client: this component represents the presentation tier of the architecture, so it provides the GUI that allows users to communicate with the application. In particular:

Passenger Client: provides the interface between the passengers and the system, in order to allow them to make requests (find or reserve a taxi, show rides history, cancel reservation). Requests are sent to the application server, which elaborates them and returns responses, like taxi proposals and notifications. The passenger client can be the myTaxiService app, that users can install on their smartphone, or a web browser, from which passengers can access the web application provided by the server from any PC device. The web app and the mobile app for passengers provide exactly the same features.

Taxi driver Client: is used by taxi drivers while they are on duty, to receive from the server taxi requests made by passengers, and to answer to them. This client allows also taxi drivers to change their status and signal unexpected events to the system. Taxi driver client needs to be installed on a mobile device provided with a GPS chip, since it constantly sends the current location of that taxi to the application server, in order to manage its position in the map and in the different queues.

Besides the 2 main clients, there are also these two other types:

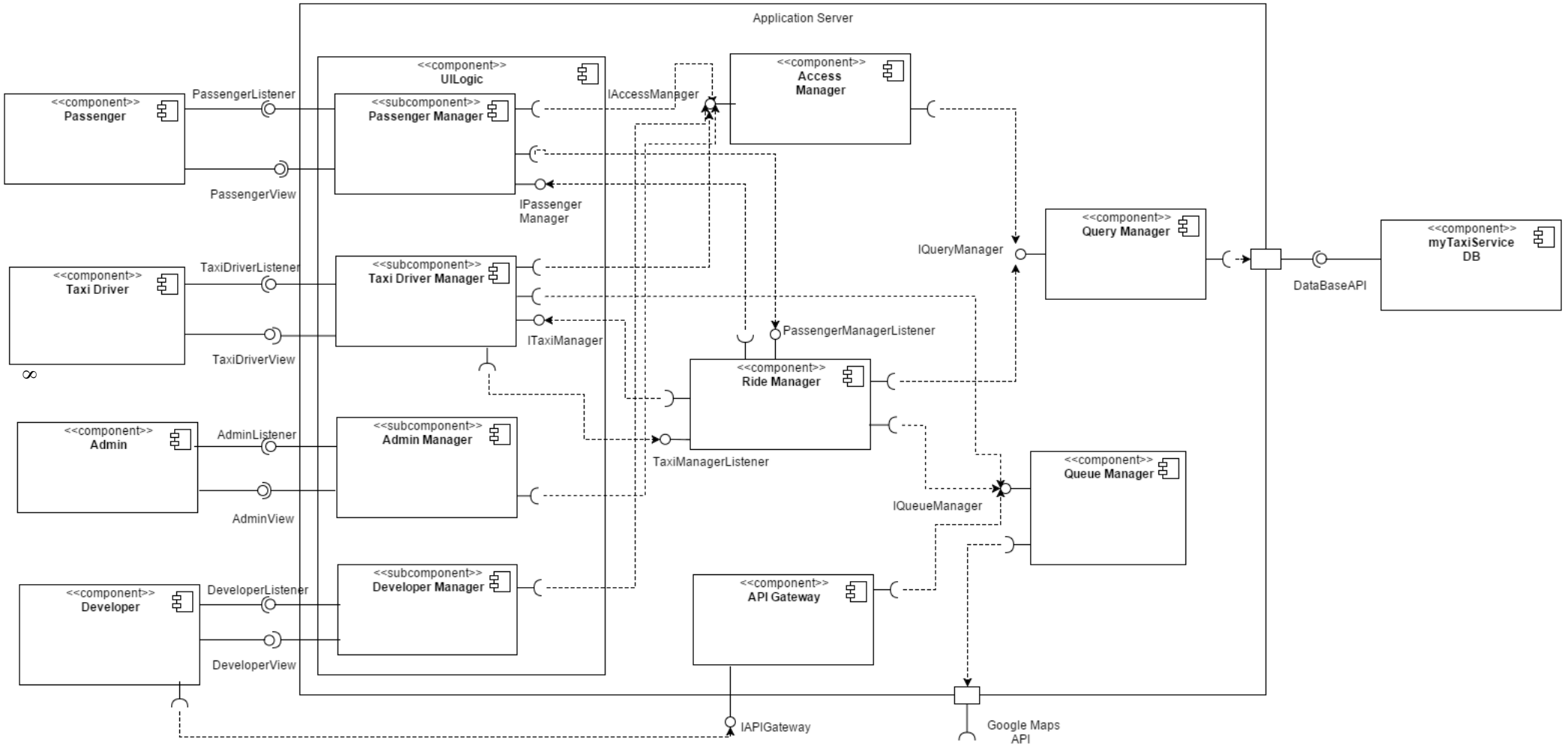
Admin Client: is used by the administrators of the system to register new taxi drivers. It is implemented only by web browsers.

Developer Client: is used by developers to access the system and request API keys and documentation, in order to extend the application. Also developers can access only through the web application, so their client will be a web browser.

Application Server: this component represents the middle tier of the system, which acts as a server for the different type of clients. This means that it receives all the requests coming from the clients, performs the requested operations using the business logic contained in it, and returns responses to the users.

Database Server: is the data tier where the information are stored. The DBMS is a relational one and is accessed by the Application Server making SQL queries.

2.3 Component view



The components displayed in the previous figure are described here:

UI Logic this component receives all the requests coming from the different clients and interacts with the other server components to perform the actions requested. The UI Logic manages visitors before they are identified by the system, by simply showing them the initial page from which they can login or sign up. After being identified, each type of user, and so each type of client, communicates with one of these three subcomponents:

Passenger Manager this subcomponent communicates with the Passenger client both via web browser and mobile application. The Passenger can request several operations (Login/Logout, Sign Up, Find a taxi, Reserve a taxi, Show history and update profile) and interacts with the Access Manager and the Ride Manager depending on the operation requested. The Passenger Manager is also listening to the Ride Manager in order to receive messages that should be forwarded to the client, in particular the ride proposals generated by the Ride Manager itself.

Taxi Driver Manager this subcomponent communicates with the Taxi Driver client via a mobile interface. It receives requests sent by Taxi Drivers, in particular:

- Login/Logout and View profile that are managed by the Access Manager,
- Change Status, which is sent to the Queue Manager,
- Signal Unexpected Event, that is sent to the Ride Manager.

The Taxi Driver Manager is also listening to the Ride Manager to receive taxi requests that must be sent to a specific taxi driver client, waiting for its response. Besides, this component constantly receives GPS coordinates of taxi driver devices (every 30 seconds) and sends them to the Queue Manager that keeps they're position saved.

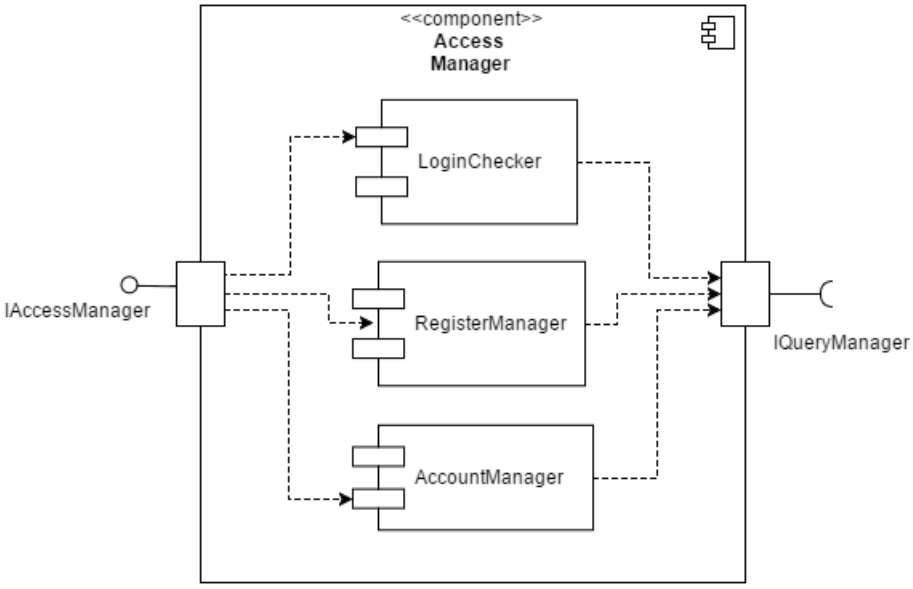
Admin Manager this subcomponent is in charge of supporting Admin client, that can communicate only through web browser. In particular it allows them to login/logout and create a new tuple in the database referred to a new taxi driver profile. It interacts only with the Access Manager to perform these tasks.

Developer Manager this subcomponent is in charge of supporting Developer client, that can communicate only through web browser. It allows the developer to login/logout, request an APIkey and retrieve API documentation. The developer should be asked for an APIKey in order to make API calls.

Access manager this component is in charge of providing authentication and account operations to the different subcomponents that ask for them. It is composed of these modules:

- LoginChecker, which receives login data (username, password, and type of user) from each of the user managers, retrieves the corresponding data from the database, and verifies if the inserted data are correct.
- RegisterManager, which receives sign up requests and if they are valid, creates a new account for each of them. This new entity is sent to the Query Manager, which will store a corresponding tuple in the database.
- AccountManager, which manages the requests involving existing accounts, in particular: view profile or specific profile data(like the ride's history of a passenger), update profile, delete profile. All of them are then translated into a corresponding query by the Query Manager.

The following figure shows a visual representation of the described internal structure:



Queue Manager this component is in charge of queue managing, meaning that it has to move taxi drivers from one queue to another tracking their GPS position and mapping them to a specific zone of the city through Google Maps API. The Queue Manager keeps a representation of some information about each taxi driver, in particular their position, which is received every 30 seconds from the Taxi Driver Manager, and their status, which is updated when they are moved in or out of queues. Moreover, using the algorithms presented in Section 5, it performs the extraction and insertion of taxis in each queue. Those operations are activated by other server component, specifically Taxi Driver Manager and/or Ride Manager.

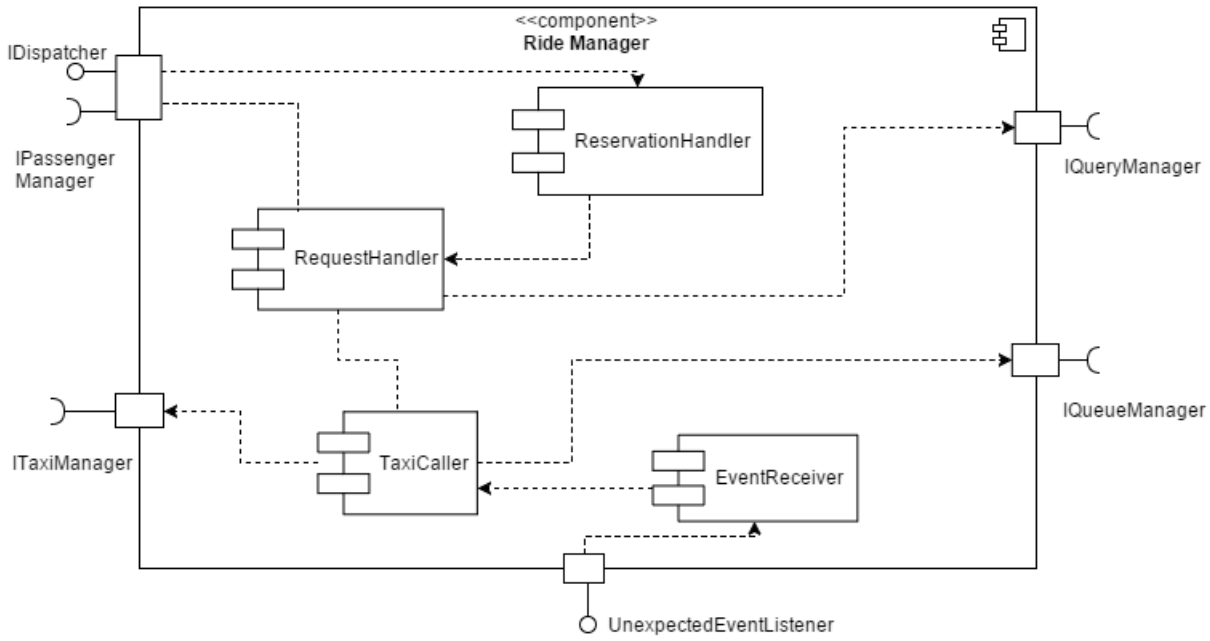
Query manager this component has been introduced to receive requests coming from other server components and addressed to the database. The requests are managed through a queue and sent to the database after translating them in SQL queries. The component contains a cache module which is used for keeping an

internal copy of the most recent data retrieved from the database. This is useful both to speed up the reading process and also to avoid an overload of requests to the DBMS.

Ride Manager this component is in charge of organizing the rides, by supporting the communication between the Passenger Manager and the Taxi Driver Manager, so it represents the dispatcher of our system. In particular it has the following modules:

- RequestHandler, which receives information about all the current taxi requests coming from the Passenger Manager, and keeps them saved until they have been completely executed. When receving a request, it sends the pickup location to the TaxiCaller and waits for a proposal, which will be forwarded to the PassengerManager. From this it will receive the passenger's response, and, if the ride has been accepted, will send a confirm to the TaxiCaller and the corresponding ride's information to the Query Manager in order to store it in the database.
- ReservationHandler, which receives information about the reservations coming from the Passenger Manager. The reservations of the current day are saved in a local storage, and each one of them has a timer associated to it, so that when the time arrives (15 minutes to the starting time indicated in the request) it is tranlated into an actual request managed by the RequestHandler. At the beginning of each day this module pulls all the reservations of that day from the database and saves them in the local storage.
- TaxiCaller, that receives from the RequestHandler the pickupLocation of a ride request, asks to the Queue Manager for an available taxi in the corresponding zone, and after receiving one, it contacts the Taxi Driver Manager that will send the ride request to the corresponding Taxi client. After receiving a response, if positive, the TaxiCaller sends the proposal to the RequestHandler, which will complete the request.
- EventReceiver, which receives possible notifications of unexpected events, signaled by taxi drivers, sending them to the TaxiCaller which will look for a new taxi driver for the corresponding request.

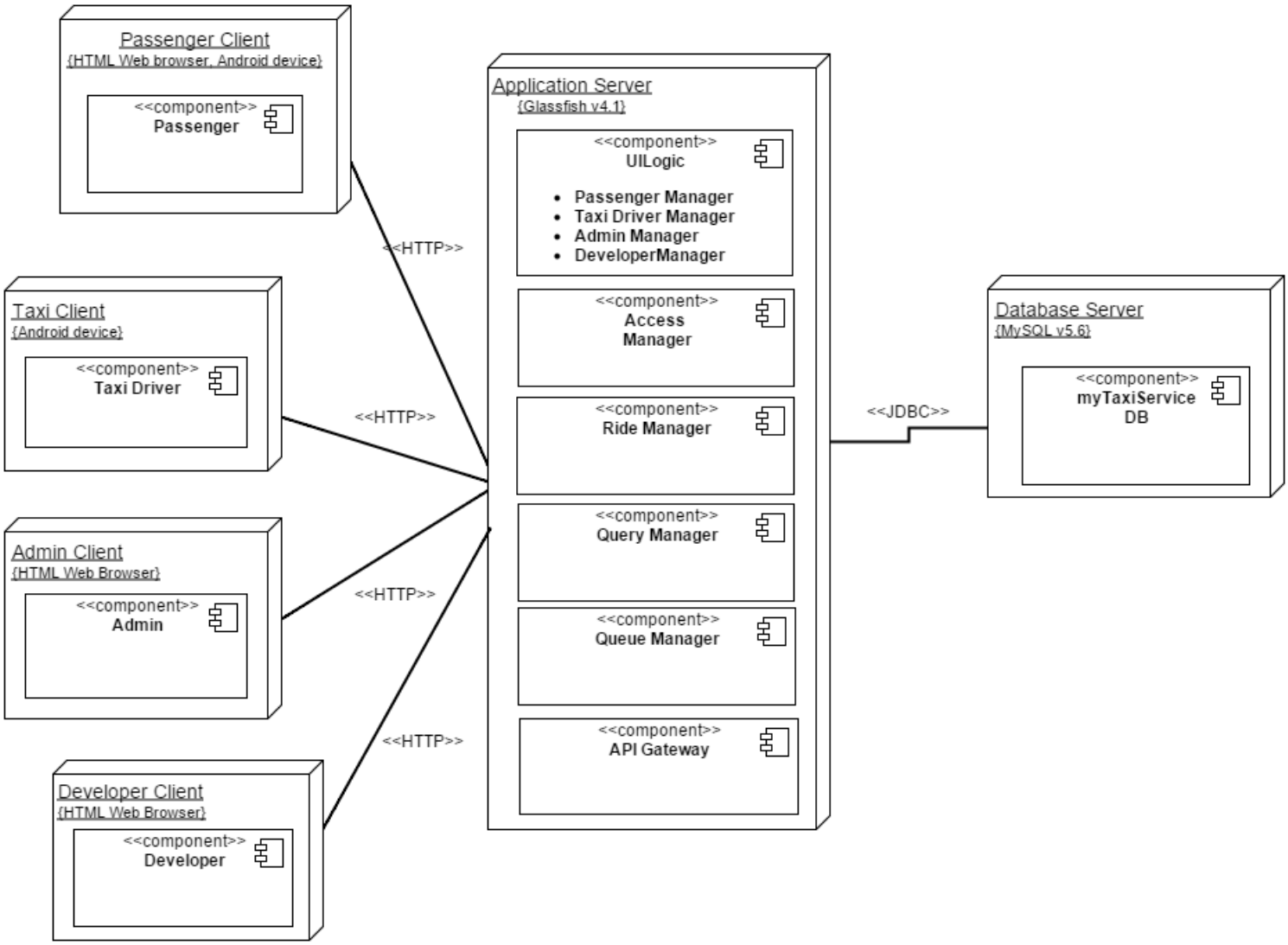
The following figure shows a visual representation of the described internal structure:



Client the components named Passenger, Taxi Driver, Admin and Developer represent the clients in our system, one for each type of user. They are thin clients, so they are only able to render the GUI of each type of user and to exchange HTTP requests and responses with the external interfaces of the application server.

API Gateway The web API Gateway is just the façade or exposure point of the web API, it represents a central point where all the abstracted APIs functionality are located and managed. The API Gateway receives the request from the client and then proxies them to the service's backend, if no failures occur (not valid APIKey for example), then the service response is sent back to the consumer of the service. Before communicating resources to a client over an HTTP connection, they need to be serialized to a textual representation. This representation can then be included as an entity in an HTTP message body.

2.4 Deployment view



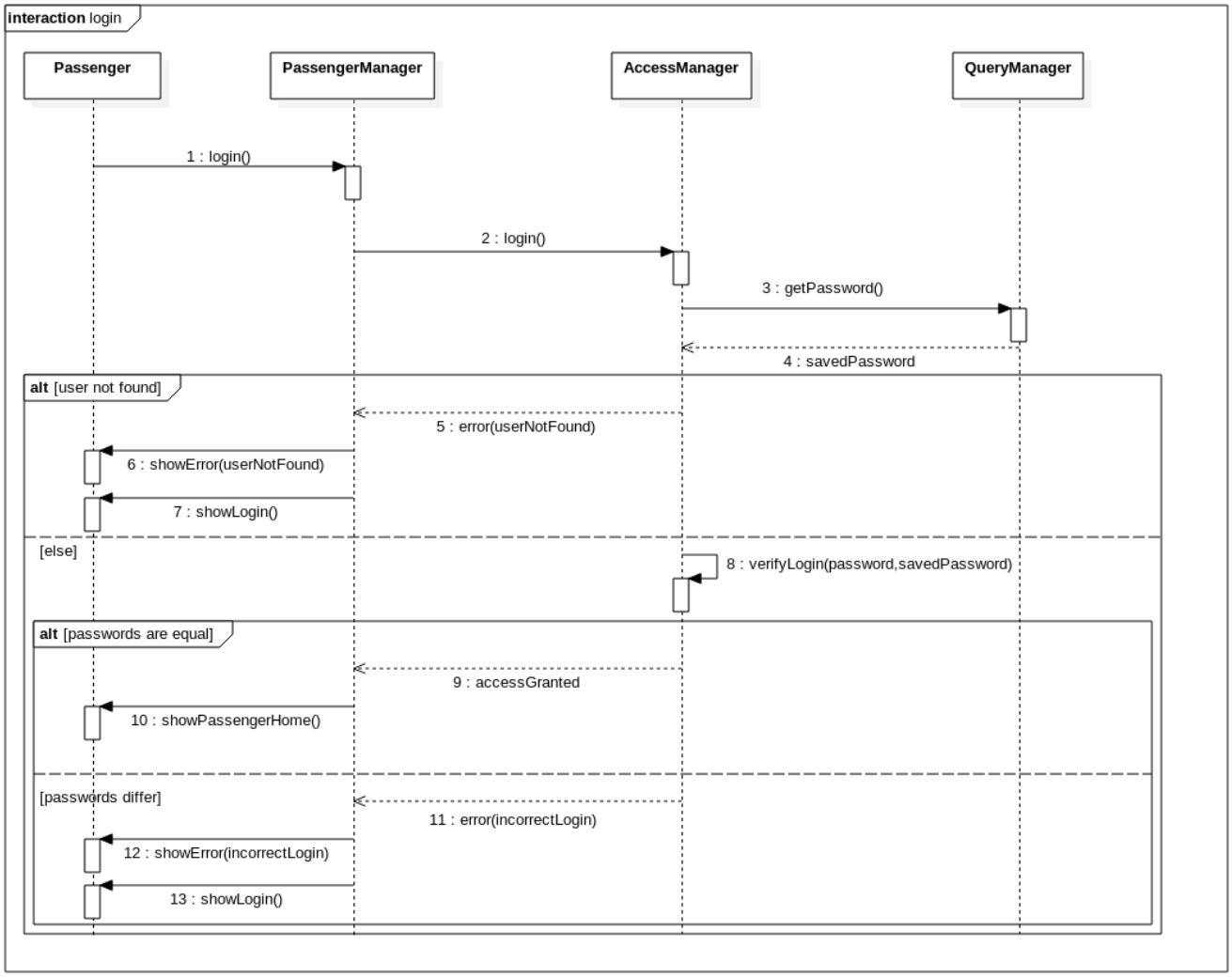
The previous diagram depicts a static view of the run-time configuration of processing nodes and the components that run on those nodes. In other words, it shows the physical machines on which our software will be installed, how the components described in the previous section map onto those nodes, and the type of connection between the different machines.

2.5 Runtime View

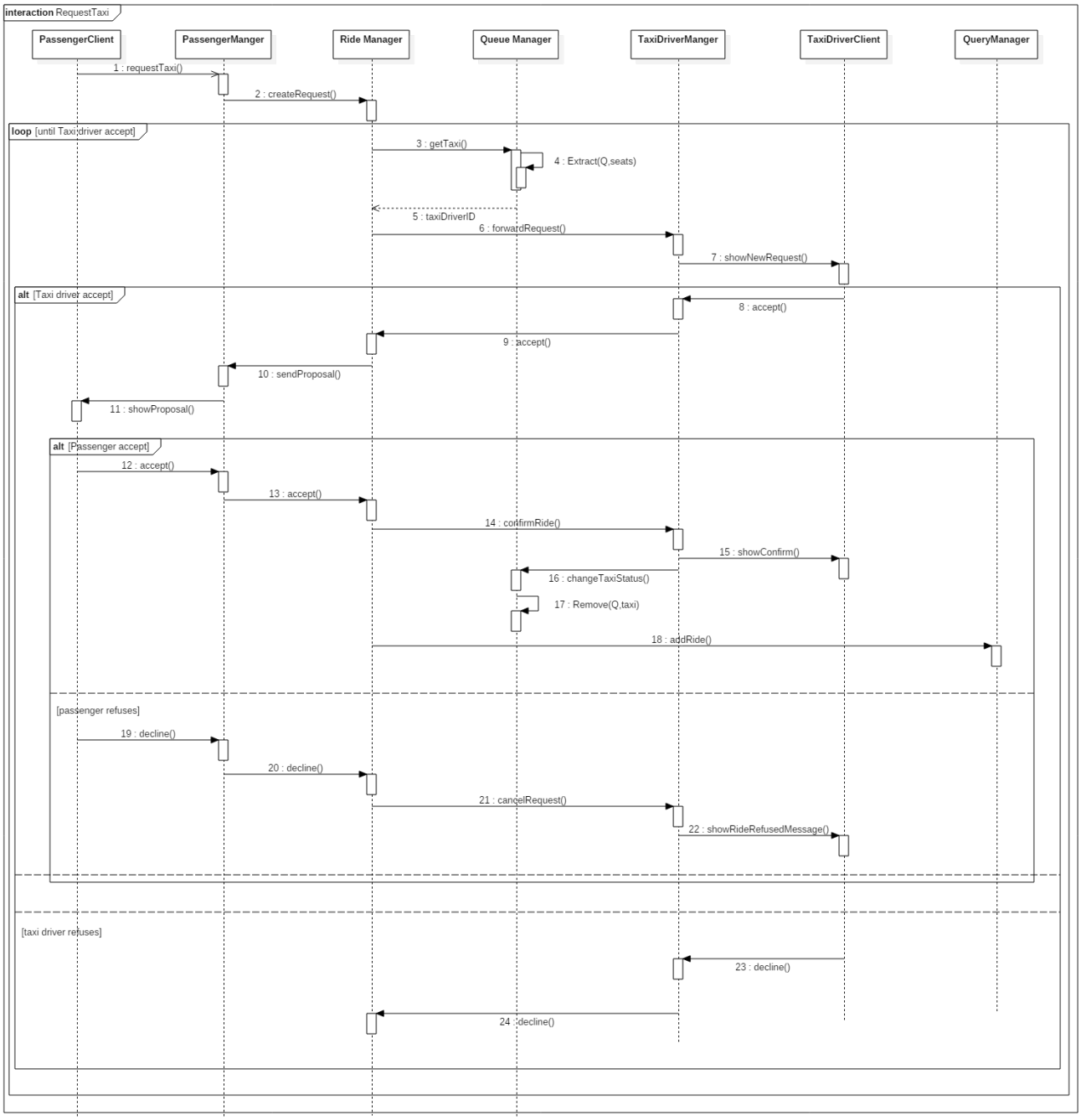
In this section we will present the main interaction between the previously described components at run time. All the methods called from one component to another correspond to methods offered by the different interfaces, which are

described in the next section; so they are showed here without parameters, except for the operations which are executed internally in a component.

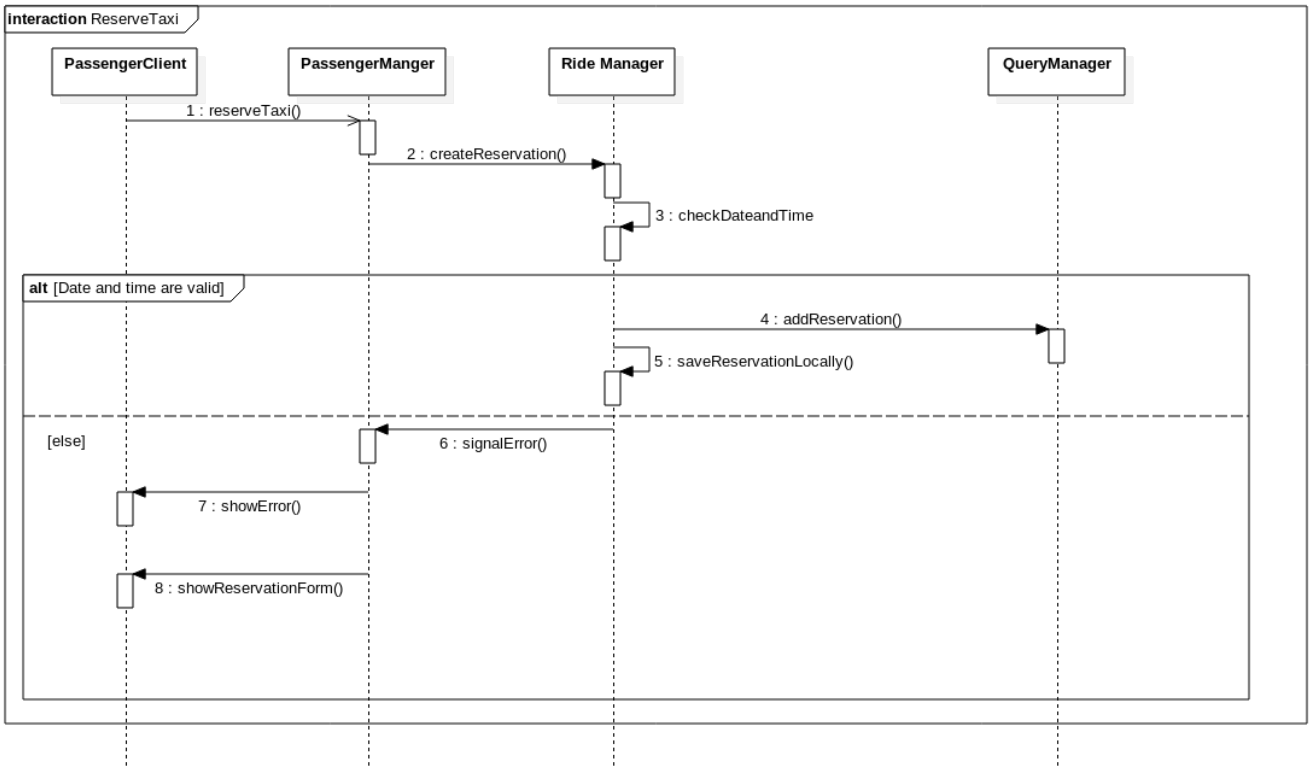
2.5.1 Passenger login



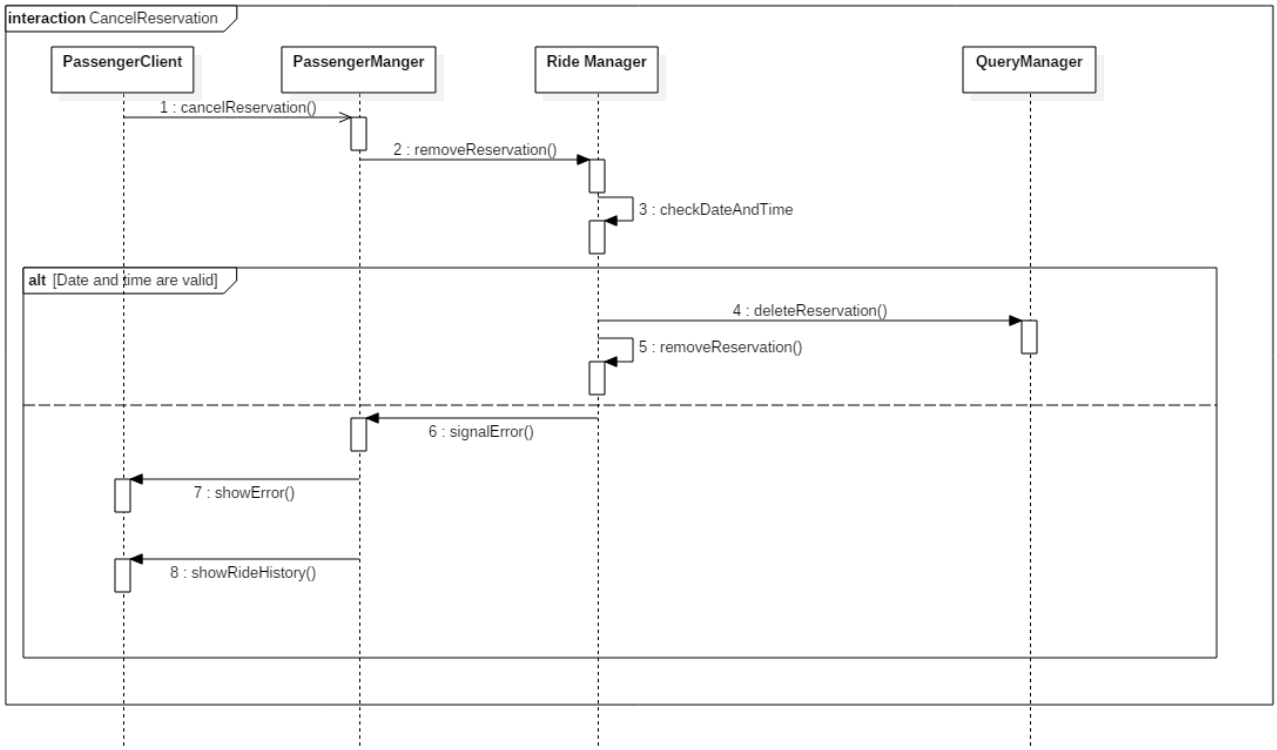
2.5.2 Taxi request



2.5.3 Taxi reservation

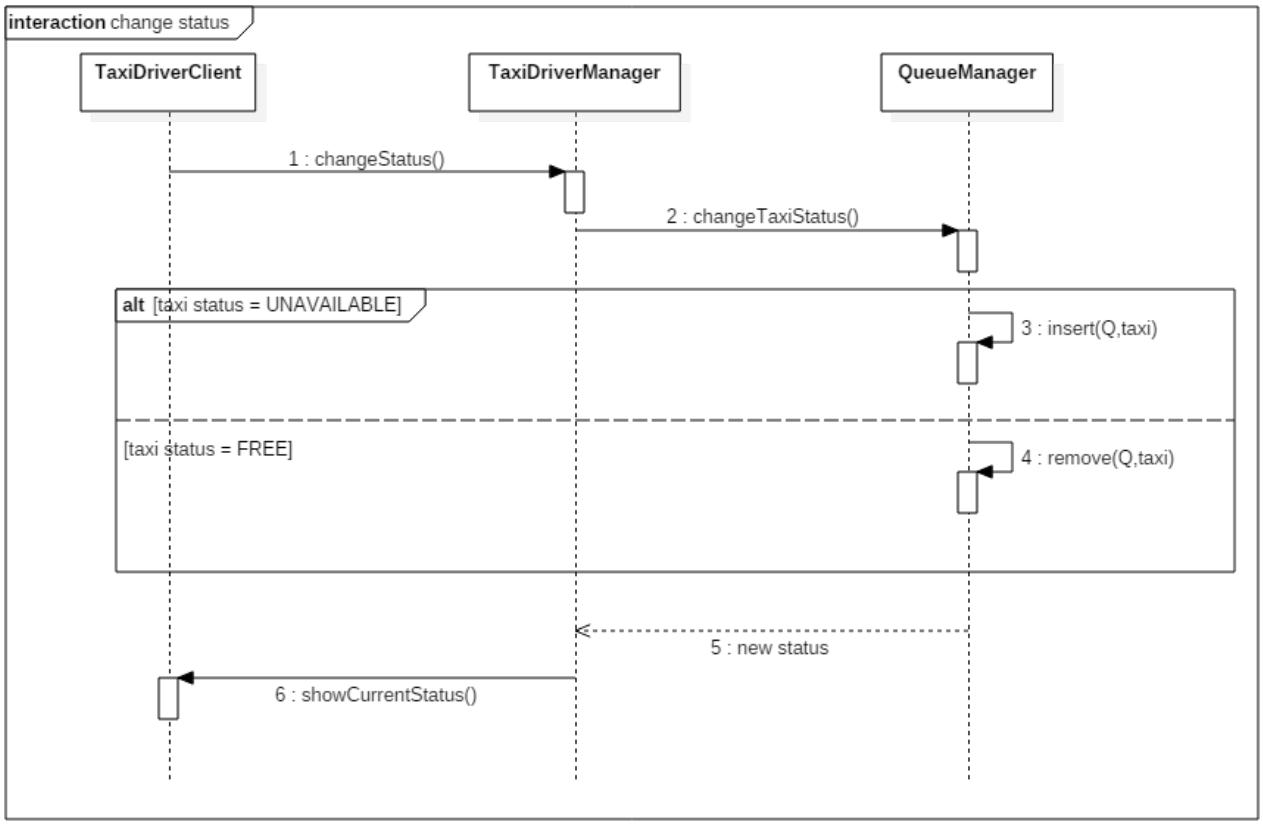


2.5.4 Cancel reservation



2.5.5 Change status

When a taxi driver changes his status and so is moved inside a queue, the correct queue Q is identified by his current position, which is given by the GPS coordinates currently sent to the Queue Manager. This is done in parallel by a different task every 30 seconds, so it's not represented in the following diagram.



2.6 Component interfaces

2.6.1 PassengerListener

The passenger client sends request to the server using this interface provided by the UI Logic component of the server.

- login(username,password);
- logout();
- register(username,password,email,firstname,lastname,address,number);
- updateProfile(username,password,email,firstname,lastname,address,number);
- requestTaxi(username, pickupLocation, destination, nSeats);
- reserveTaxi(username, date, time, pickupLocation, destination, nSeats);

- cancelReservation(username, reservationID);
- rideHistory();
- accept(requestID);
- decline(requestID);

2.6.2 PassengerView

This interface(and the other "view" interfaces) is used to display information on the passenger's device (or on the other devices).

- showLogin();
- showPassengerHome();
- showRegistrationForm();
- showRequestForm();
- showReservationForm();
- showRideHistory();
- showError(message);
- showProposal(username, ETA, taxiCode);

2.6.3 PassengerManagerListener

This interface is provided by the Ride Manager to communicate with the Passenger Manager which sends taxi requests using these methods:

- createRequest(username, pickupLocation, destination, nSeats)
- createReservation(username, date, time, pickupLocation, destination, nSeats)
- removeReservation(username, reservationID);
- accept(requestID);
- decline(requestID);

2.6.4 IPassengerManager

- sendProposal(username, ETA, taxiCode);
- signalError(message);

2.6.5 TaxiDriverListener

The taxi driver client sends request to the server using this interface provided by the UI Logic component of the server.

- login(username, password);
- logout();
- changeStatus(taxiCode);
- accept(requestID);
- decline(requestID);

2.6.6 TaxiDriverView

- showLogin();
- showTaxiDriverHome();
- showCurrentStatus();
- showNewRequest(pickupLocation, destination, nSeats);
- showConfirm();
- showRideRefusedMessage();

2.6.7 AdminListener

- login(username,password);
- logout();
- registerTaxiDriver(userID,password,email,firstname,lastname,address,number,drivingLicense);

2.6.8 AdminView

- showLogin();
- showAdminHome();
- showRegistrationForm();
- showError(message);

2.6.9 IAccessManager

- login(username,password,usertype); usertype can be an enum used to identify the type of user that request to login. usertype can be "Passenger","Taxi driver","Administrator","Developer".
- register(username,password,email,firstname,lastname,address,number);
- registerTaxiDriver(userID,password,email,firstname,lastname,address,number,drivingLicense);

- viewProfile(username);
- updateProfile(parameters);
- assignAPIKey(username)

2.6.10 TaxiManagerListener

This interface is provided by the Ride Manager to receive responses or events coming from the Taxi Driver Manager.

- accept(requestID);
- decline(requestID);
- signalUnexpectedEvent(taxiCode);

2.6.11 ITaxiManager

- forwardRequest(taxiCode, requestID, pickupLocation, destination);
- confirmRide(requestID): once the user has accepted the proposal a confirmation is sent to the taxi driver.
- cancelRequest(requestID): this method is called if passenger refuses the ride proposal to signal this to the taxi driver.

2.6.12 IQueueManager

- getTaxi(pickupLocation, seats);
- changeTaxiStatus(taxiCode);
- getTaxisInZone(pickupLocation);
- getETA(pickupLocation);

2.6.13 IQueryManager

- getUserBy(parameters): returns all the user that match the parameters of the research (firstname, lastanme, e-mail..). Is used by the access manager to determine if a user has already registered or if it has selected a username already in use.
- getPassword(username);
- addPassenger(username,password,email,firstname,lastname,address,number)
- addTaxiDriver(userID,password,email,firstname,lastname,address,number,drivingLicense));
- addAdiministrator(username,password,email,firstname,lastname,address,number)
- addDeveloper(username,password,email,firstname,lastname,address,number, APIkey);
- addRide(username, taxiCode, pickupLocation, destination, nSeats, date, time);

- addReservation(username, pickupLocation, destination, nSeats, date, time);
- deleteReservation(username, pickupLocation, destination, nSeats, date, time);
- updateProfile(username,password,email,firstname,lastname,address,number);
- pullReservation(date): this method is called by the ride manager that will save locally all the reservations of the day.

2.6.14 DeveloperListener

- login(username,password);
- logout();
- register(username,password,email,firstname,lastname,address,number);
- updateProfile(username,password,email,firstname,lastname,address,number);
- requestAPIKey();

2.6.15 DeveloperView

- showLogin();
- showDeveloperHome();
- showAPIKey();
- showDocumentation();
- showError(message);

2.6.16 IAPIGateway

API calls are made via the HTTP methods GET, POST, PUT, DELETE, and HEAD. The responses return status codes indicating success or failure, along with any applicable headers, and JSON representing the affected fields in the message-body.

- GET(resource)
- HEAD(resource)
- POST(resource)
- DELETE(resource)

2.7 Selected architectural styles and patterns

3-Tier-Architectural style: A three-tiered architecture partitions the system in distinct functional units:

1. The client tier is everything which runs in the client machine, in our specific case the web browser and the custom UI in the mobile application.
2. The business-logic tier represents the core of the application. It receives requests from the client tier, processes the business logic based on the requests, and mediates access to the data tier's resources. It is divided in two layers:
   - The web layer is where the pages that the client will display are generated. It interacts directly with the client tier receiving it's request and generating the response. To do that this layer has to interact with the underlying business layer.
   - The business layer is where the main logic of the application is located. Here we will find the main code devoted to queue and ride managing. To do that this layer interacts with the database.
3. Finally the data tier contains the relational database of the application. Here persistent data are collected. In the database we save both sensitive and non sensitive information.

This ensures a clean division of responsibility and makes the system more maintainable and extensible.
Design Pattern: MVC and Publisher-Subscriber MVC is the architectural design pattern for the presentation layer. It divides a given software application into three interconnected parts (model-view-controller), so as to separate internal representations of information from the ways that information is presented to the user. This helps creating better-organized applications, thanks to the modular division and the collaboration among the different components. It also makes code more flexible and maintainable, because it is possible to apply changes to individual components without affecting the remaining ones. Since our clients are thin clients, almost the entire model, view and controller logic are placed on the server. In this approach, the client sends HTTP requests to the controller, which is performed by the UI Logic on the application server, and then receives a complete and updated web page (or a screen in the mobile application) from the view. So the view of each type of client is actually managed by the respective subcomponent in the UI Logic, while the clients make only the rendering of the responses. Finally, the model exists entirely on the application server, and is managed by the different business components. Passenger's client and Taxi driver's client are both subscribers of the respective UI logic component (Passenger Manager and Taxi driver Manager), that act as publishers. The publisher notifies the subscribers as soon as a certain event occurs, using the "at least once" delivery policy so that the publishers are sure that the message has arrived to the proper client.

2.8 Other design decisions

Given the described architectural choices, we decided to use the Java EE platform to develop our application, since it provides a runtime environment for developing and running multi-tiered, scalable, reliable, and secure web applications. This platform is particularly suited for creating an application that offers services to multiple clients simultaneously, just like our case, while handling low-level concerns like transactions, scalability and concurrency. Therefore, using Java EE will greatly simpify the development of our project, as developers will be able to concentrate more on the business logic of the components rather than on infrastructure and low-level problems.
For the presentation tier we decided to use JavaServer Faces (JSF) technology, a user interface framework for Java EE applications. It is particularly suited, by design, for use with applications based on the MVC (Model-View-Controller) architecture that is one of the main design pattern of our application. Moreover with the JSF framework the appearance of a UI-component can be easily varied for the type of display device available (for example, a mobile phone). For these reasons we thought that JSF can perfectly suit are needs.
To implement business logic, Enterprise JavaBean (EJB) is the dedicated technology in Java EE. It is a server-side software component that encapsulates the business logic of an application, so it can be used to deploy the various server components that perform operations in our system (like the Ride Manager, Access Manager and Queue Manager). An EJB web container provides the runtime environment for these components.
For the actual persistence Java Persistence API (JPA) can be used to map between objects in code and database tables. This can "hide" the SQL code from the developers so that all they deal with are java classes. Any JPA provider accomplish this mapping by automatically writing SQL and using JDBC to read and write to the DB, so the actual low-level technology for interaction with the DB is JDBC.

3 Algorithm Design

The Queue's queueing discipline decides which element should be removed, so it has to be implemented in such a way that select the first taxi driver available if and only if he/she can handle the client request in terms of required seats. So, the queuing discipline adopted would be quite similar to FIFO, except for the fact that if the first taxi driver in the queue doesn't match the parameters of the request, the algorithm look at the second taxi driver and so on. The queue should also be able to remove a precise taxi driver if he/she becomes unavailable. A queue managed according to the previously describes queueing disciplines has to support this main operations:

- INSERT(Q,x) : insert taxi driver x at the bottom of queue Q.
- EXTRACT(Q,n) : extract the first taxi from queue Q that matches the passenger's request in terms of available seats (n).
- REMOVE(Q,x) : remove a desired taxi driver x from the queue Q.

Pseudocode is used for describing the algorithms. The queue will be implemented as an array of unbounded length, because it gives a general idea of what the algorithm should do without being bidden to some particular programming language; the following notation is used:

- Q is the queue.
- $Q.head$ is the index of the oldest element in the queue. (The one inserted before the others).
- $Q.tail$ is the index where the next element would be store, meaning that $Q.tail - 1$ is the index of the last added element.
- n is the number of seats required by the passenger.

Algorithm 1 Insert

```
1: procedure INSERT(Q, x)
2:     Q[Q.tail] ← x
3:     Q.tail ← Q.tail + 1
4: end procedure
```

The Queue Manager calls Extract when an incoming request arrives. If a suitable taxi driver is found the EXTRACT function returns it, otherwise it returns null. In the last case the Queue manager will forward the search on the adjacent queues by calling SEARCHINADJACENT.

Algorithm 2 Extract

```
1: procedure EXTRACT(Q, n)
2:     i ← Q.head
3:     while Q[i] has not enough available seats ∧ i ≠ Q.tail do
4:         i ← i + 1
5:     end while
6:     if i ≠ Q.tail then          ▷ There is a taxi driver that can take care of the request
7:         x ← Q[i]
8:         for j = i; j < Q.tail; j++ do          ▷ Move queue's elements position up
9:             Q[j] ← Q[j + 1]
10:        end for
11:        return x
12:    else                        ▷ There is not a taxi that can take care of the request
13:        return null
14:    end if
15: end procedure
```

Algorithm 3 Remove

```
1: procedure REMOVE(Q, x)
2:     i ← SEARCH(Q,x)                                   ▷ i gets the index of x
3:     if i ≠ null then
4:         for j = i; j < Q.tail; j++ do     ▷ Delete x and move queue's elements position up
5:             Q[j] ← Q[j + 1]
6:         end for
7:     end if
8: end procedure
```

Algorithm 4 Search

```
1: procedure SEARCH(Q, x)
2:     for i = Q.head; i < Q.tail; i++ do
3:         if Q[i] == x then
4:             return i                                  ▷ Element found in position i
5:         end if
6:     end for
7:     return null                                       ▷ Element not found
8: end procedure
```

The city map is divided into square zones of $2km^2$.
$Q.adjacents$ is an array that contains the queues adjacent to Q.
In the algorithm that follows given a queue Q, we look up to its adjacent one by one until we find an available taxi that fits the request. If a taxi is not found in the immediate neighbours of Q the research is forwarded to the neighbours of the neighbours. Stating that level of depth 1 is represented by immediate adjacent zones, the further reachable depth is 3. Going over that depth would be useless since the passenger would have to wait too much. What we perform is basically a BFS with Closed List (we remove the nodes corresponding to the same state).

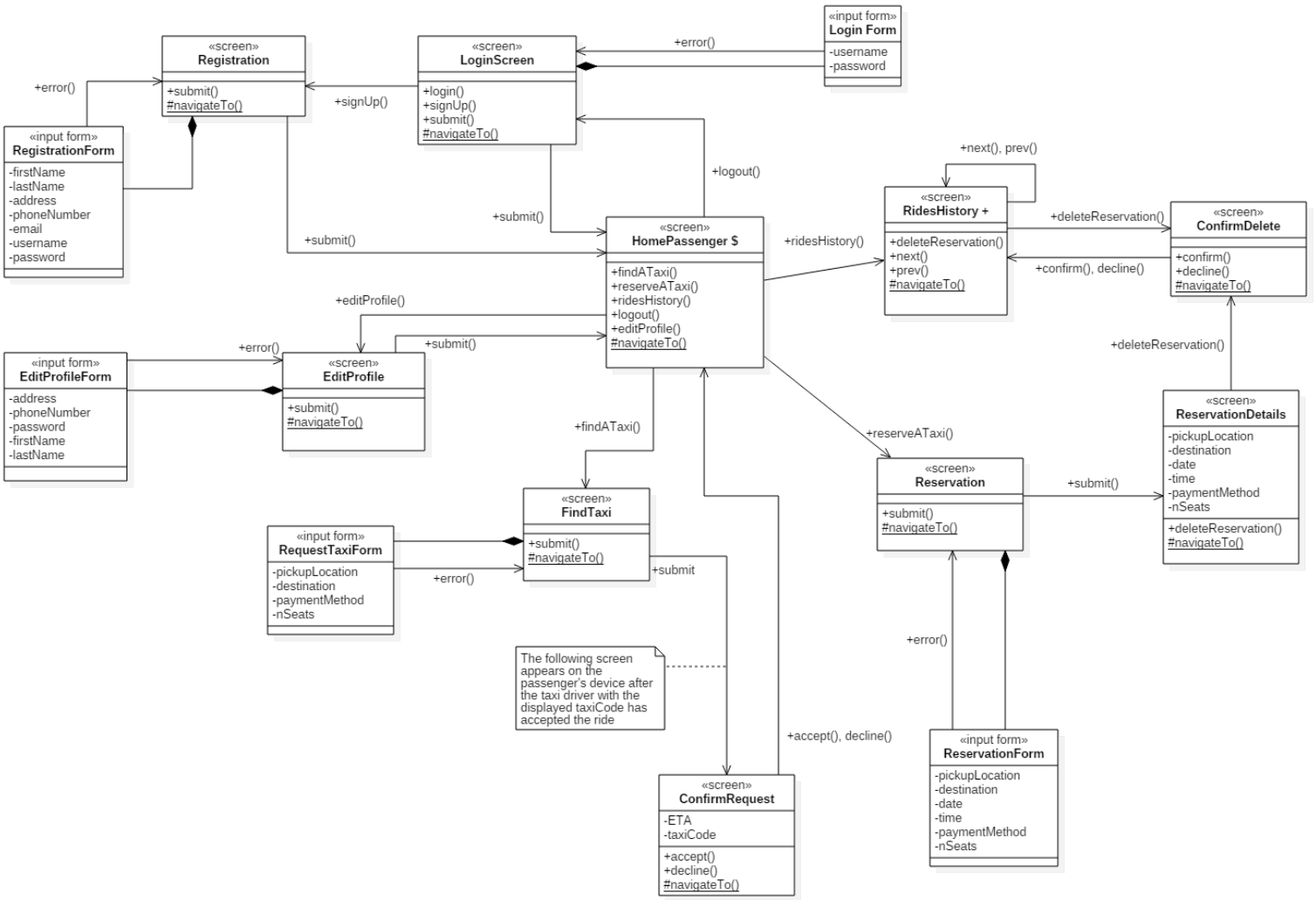
Algorithm 5 SearchInAdjacent

```
1: procedure SearchInAdjacent(Q, n)
2:     frontier ← Q                                  ▷ Create a queue of queues
3:     closedlist ← empty                            ▷ Create a list of visited queues
4:     while frontier is not empty do
5:         u ← frontier.dequeue
6:         if u is not in closedlist then            ▷ If u has not been visited yet
7:             closedlist.add(u)
8:             for all adjacent queues adj of queue u do
9:                 x ← Extract(adj,n)()
10:                if x ≠ null then                  ▷ There is an available taxi in adj
11:                    return x
12:                else
13:                    frontier.enqueue(adj)
14:                end if
15:            end for
16:        end if
17:    end while
18:    return null                                   ▷ Element not found
19: end procedure
```

Calculation of ETA and distance is achieved through Google Maps API.

4 UIDesign

The following UX diagram models the user experience of a passenger in the application.



Here are some examples of how the UI navigation will look like:

4.1 Taxi Request

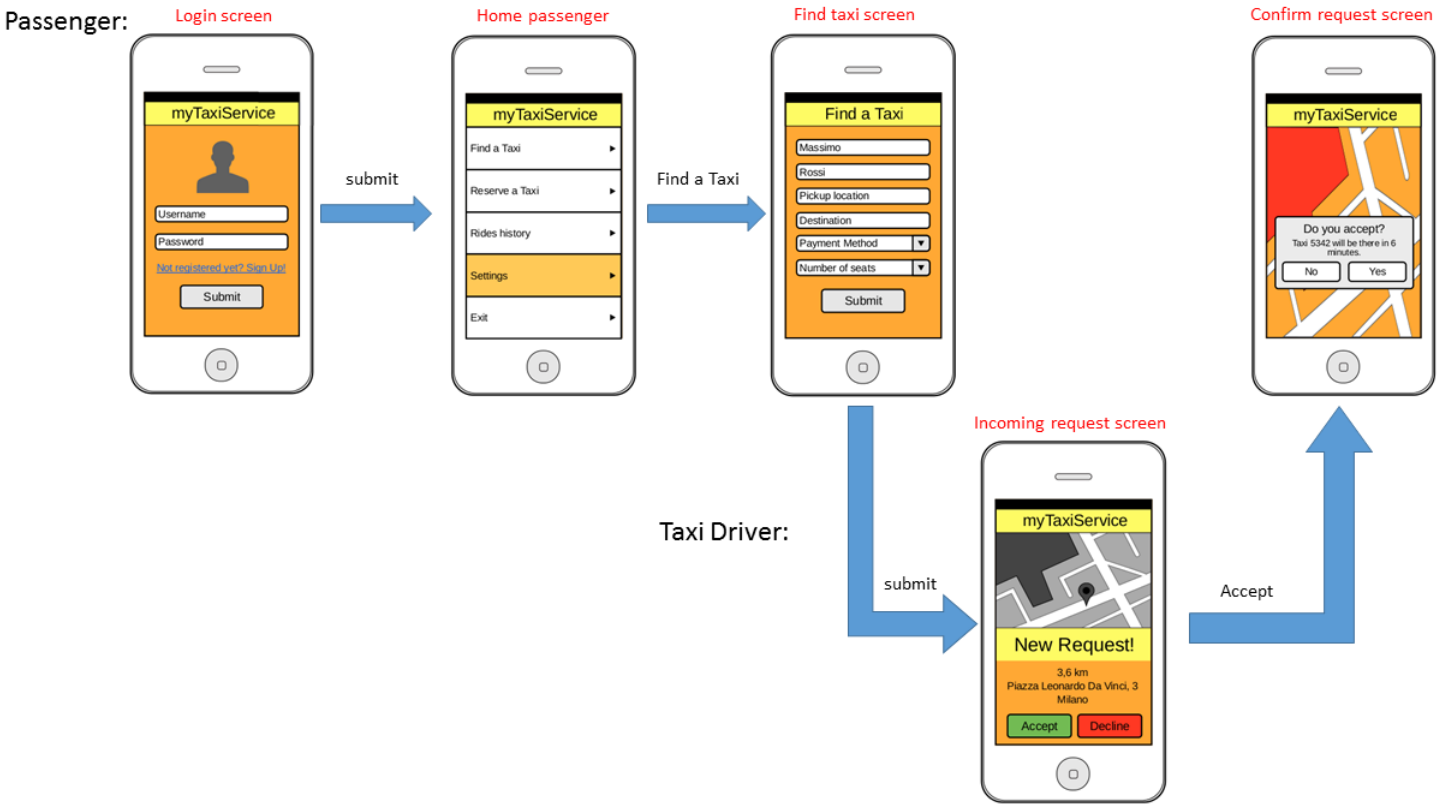


Figure 1

Figure 1 shows the process of requesting a taxi, both from the passenger's point of view, who compiles the request and submits it, and from the taxi driver's point of view, who receives a notification on his/her mobile app.

4.2 Taxi Reservation

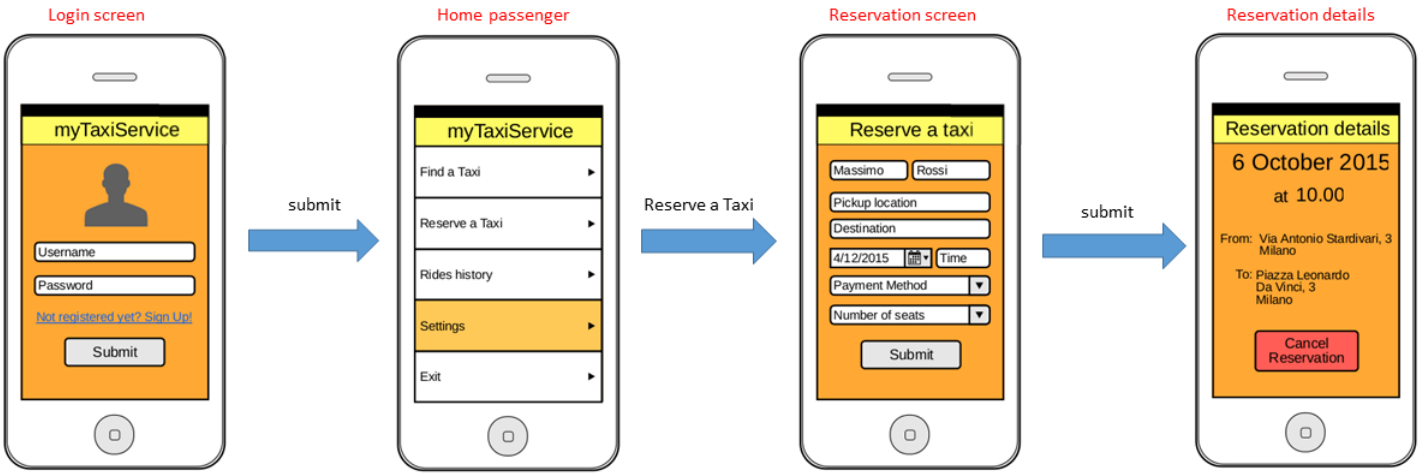


Figure 2

Figure 2 shows the process of reserving a taxi on the passenger's mobile application.

4.3 Cancel Reservation

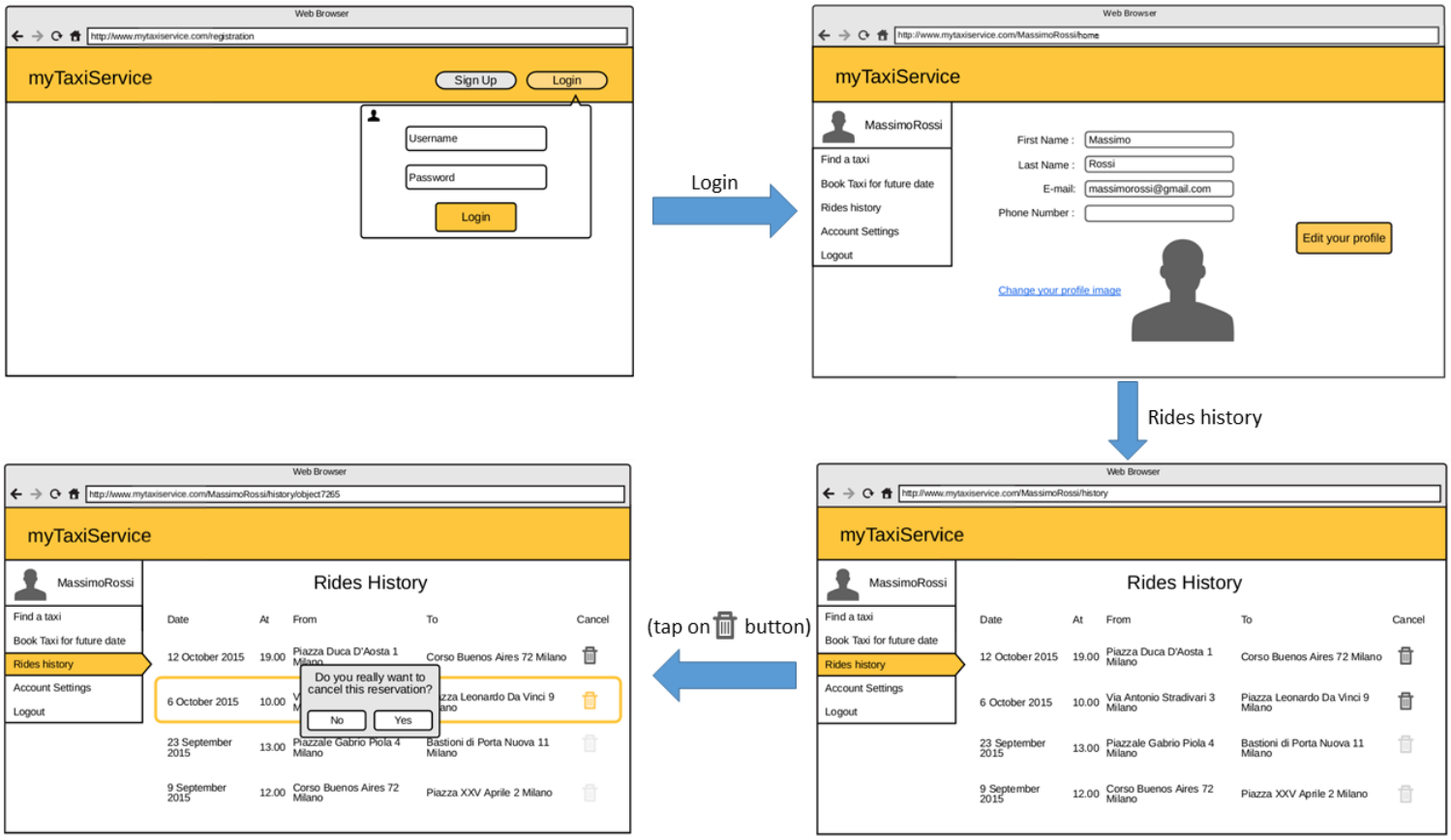


Figure 3

Figure 3 shows the process of cancelling a reservation in the web application. The passenger goes into the the "Rides history" section and can tap on the trash button of the rides which have not already taken place, in order to remove a reservation within the time constraints already defined.

4.4 Change Status

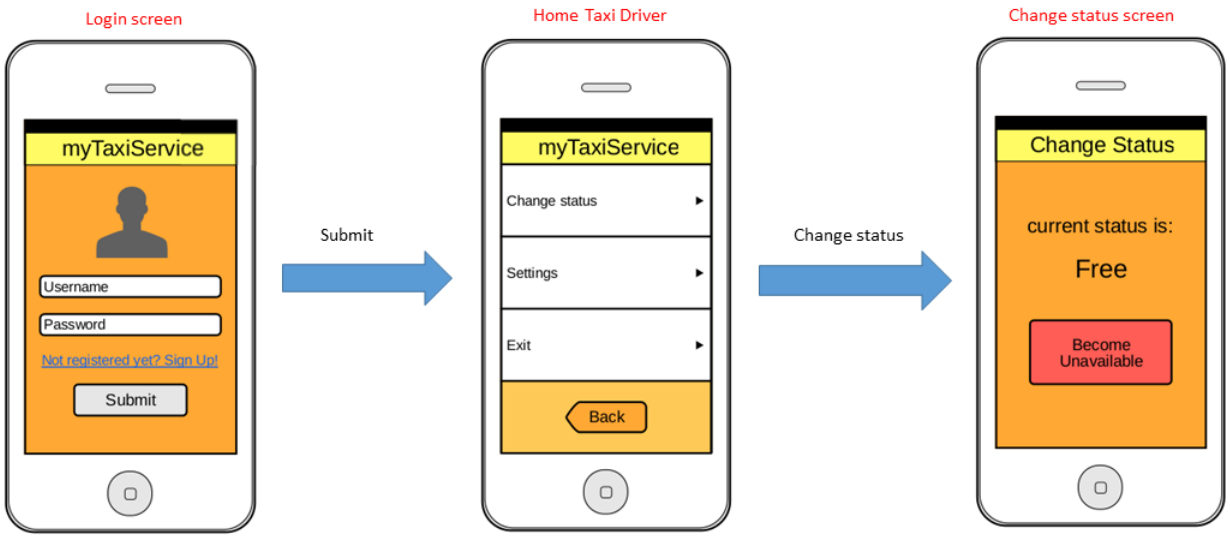


Figure 4

Figure 4 shows the taxi driver's menu and how they can change their status. In this example, the taxi driver was unavailable, so tapping on the "change status" button allows him/her to become free.

5 Requirement Traceability

Requirement	Component	Note
[FR1] The system should provide a "Sign Up" function so that every visitor can become a registered user.	• Passenger Client • Administrator Client • Access Manager	
[FR2] The system should provide a "Login" function that allows registered users to interact with the application.	• Passenger Client • Taxi Driver Client • Administrator Client • Access Manager	
[FR3] The system should not allow a visitor to input a username and/or an e-mail that is already in use.	• Access Manger	Once a user has submitted the registration form, the Access Manager performs a search on the Database to verify that the username and/or e-mail supplied by the client is unique in the database. If it is not an error is returned.
[FR4] The system has to check that a user does not sign up twice.	• Access Manager	The access manager performs a search on the database to verify if already exists a tuple with the same firstname, lastname, email supplied by the client. If such a tuple exist, an error is displayed on the passenger screen.
[FR5] The system has to keep trace of all the orders (immediate requests and reservations) done by passengers and show them when required.	• Ride Manager	The ride manager keeps a local storage where pending request and reservation done for the current day are stored.
[FR6] The system should allow passengers to request a taxi for the current date and time.	• Passenger Client • Passenger Manager • Ride Manager	
[FR7] The system should allow passengers to reserve a taxi for a desired date and time.	• Passenger Client • Passenger Manager • Ride Manager	
[FR8] The system should allow passengers to accept or decline the proposed solution.	• Passenger Client • Passenger Manager	
[FR9] The system should display to passengers the ETA and code of the selected taxi.	• Passenger Client • Passenger Manager	

[FR10] The system should allow passengers to cancel a reservation within 15 minutes before the pickup time.	• Passenger Client • Passenger Manager • Ride Manager • Query Manager	
[FR11] The system should not allow reservation where the date is previous to the current one and/or the pickup time occurs within 2 hours from the current time.	• Ride Manager	
[FR12] The system should divide the city into zones of $2km^2$ and keep a queue for each of them.	• Queue Manager	
[FR13] The system should constantly receive taxi's position (via GPS) and status to keep trace of the zone they are into.	• Taxi driver Client • Queue Manager	The taxi driver device signal every 30 seconds its current position. The Queue manager is able to move the taxi driver from one queue to another according to the zone (that corresponds to a given location) he/she is into.
[FR14] The system should maintain the queue up to date, adding free taxi drivers and removing the unavailable ones.	• Queue Manager	
[FR15] When a request from a passenger arrives, the system should forward it to the first taxi driver in the queue.	• Ride Manager • Queue Manager	
[FR16] The system should notify taxi drivers of all incoming request.	• Taxi driver Client • Taxi driver Manager • Ride Manager	
[FR17] The system should allow taxi drivers to change status from free to unavailable and vice versa.	• Taxi driver Client • Taxi driver Manager • Queue Manager	
[FR18] The system automatically changes the status of a taxi driver when he/she has accepted a request and the associated passenger has also accepted the proposal.	• Ride Manager • Queue Manager	
[FR19] If an unexpected event occurs to the taxi driver while going to the pickup place, the system should notify the passenger and contact another taxi driver.	• Taxi driver Client • Taxi driver Manager • Ride Manager • Queue Manager	

[FR20] The system should provide an API that allow users to make a call to the API Server which will return all taxi that are available in the zone containing the specified location.	• API Gateway • Queue Manager	
[FR21] The system should provide an API that allow users to make a call to the API Server, which will return the ETA of the first available taxi.	• API Gateway • Queue Manager	
[FR22] The system should provide an API key when the developer request it.	• Developer Manager • Access Manager • Query Manager	
[FR23] The system should provide the API documentation to the developers.	• Developer Manager	

6 Appendix

6.1 Software Tool used

- TexMaker (http://www.xm1math.net/texmaker/): LaTeXeditor, used to redact this document.
- StarUML (http://staruml.io/) : used to build Sequance diagrams.
- Draw.io (http://www.draw.io): to create Deployment View, Component View.

6.2 Hours of work

- Alberto Gasparin $\sim$ 30 h
- Vito Matarazzo $\sim$ 30 h