Software Design Document

February 26, 2016

POLITECNICO
MILANO 1863

Daniele Grattarola (Mat. 853101)

Ilyas Inajjar (Mat. 790009)

Andrea Lui (Mat. 850680)

Contents

1 Introduction

1.1 Purpose

The presented document is the Software Design document (SDD) for the MyTaxiService platform project. The main purpose of this document is to be a guideline for the concrete implementation of the platform, provide developers with high level descriptions of the main algorithms, describe the architectural styles and patterns, and generally establish the design standards for the development phase. This document is intended for stakeholders, software engineers, and programmers and must be used as reference throughout the whole development of the system.

The secondary audience for this document includes system maintainers and developers who wish to integrate the platform's services with their own software. The design choices listed here must be respected throughout the initial development of the platform and any other further modification to the codebase.

1.2 Scope

The descriptions listed in this document define the design specifications that must be used by design stakeholders, in order to implement the "design leads to code" philosophy. These design standards represent constraints to the development of the platform, with all unspecified design decisions left to the developers.

Key reasons for the design language described in this document are:

1. To facilitate the development, integration, expansion and maintenance of the platform.
2. To define a business identity (consistency of UX and marketing).
3. To implement the requirements listed in the RASD of the project in a consistent way.

1.3 Definitions, acronyms, and abbreviations

Throughout this document, the following definitions will be applied without further explanations:

- **Platform**: the set of software applications and hardware infrastructure that make up the MyTaxiService service; these include the back-end server software, the web application and the mobile application used by the customers, the mobile application used by the taxi drivers, and all the necessary hardware needed to run the mentioned software and any needed support software.
- **System**: any subset of software components of the platform.
- **Back-end**: the software run on the back-end server of the platform which is used to handle the communication between the user applications. The term also addresses all the necessary software components that are needed to store data, perform calculations and manage the hardware (e.g. an operating system).
- **Customer-side application**: software run on a personal device which is used to send taxi requests to the system and to handle the system's replies. It is designed to be used by customers (see below) and can either be a mobile application (run on a smartphone or tablet) or a web application (run on any personal device through an Internet browser).
- **Taxi-side application**: software run on a personal device which is used to manage taxi requests forwarded by the system and to reply to the system with information on how to handle the requests. It is designed to be used by taxi drivers (see below) and is a mobile application (run on a smartphone or tablet).
- **User-side application**: the set of customer-side and taxi-side applications
- **Taxi driver**: the owner of a taxi license in Pallet Town, who uses the taxi-side application to interact with the platform.
- **Customer**: a person which intends to exploit the taxi service of the town, and who uses the user-side applications to interact with the platform.
- **User**: any customer or taxi driver who uses a user-side application.

The following acronyms will also be used in place of the extended form:

- **SDD:** Software Design Document
- **RASD**: Requirement Analysis and Specification Document

- **CGI**: Common Gateway Interface
- **DBMS**: Data Base Management System
- **DMZ**: De-Militarized Zone
- **SOA**: Service Oriented Architecture
- **UX**: user experience

The following convention will be used to refer to different items in the document:

- **sec. / secs.**: section / sections
- **req.**: requirement.

A typical use of the aforementioned abbreviation would be in the form "element Xx, sec. x.x.x" (e.g. *req. 1, sec. 1.3* - if this section contained a numbered requirement with index 1).

One last observation is to be done regarding the use of the singular *they*, which will be used to refer to single persons throughout the whole document.

1.4 Document structure

The presented SDD is divided in sections and structured as follows:

- **Section 1 - Introduction**: contains support information to better understand the presented document.
- **Section 2 - Architectural design**: contains a description of the architectural styles and patterns selected for the platform, which serve as an implementation guideline for developers.
- **Section 3 - Algorithms design**: contains a high-level description of the core algorithms of the back-end.
- **Section 4 - User interface design**: contains a description and a conceptual preview of the user interface and UX.
- **Section 5 - Requirements traceability**: links the decisions described in this document to the requirements specified in the RASD.
- **Section 6 - Additional comments**: contains a summary of the hours spent in producing the document.

2 Architectural Design

2.1 Overview

The platform must be designed with performance, scalability, and user experience in mind, and the architectural decisions must therefore reflect these core principles.

Because the platform is web-based, a communication medium must be implemented between the back-end and the user-side applications in order to coordinate the relevant events as required by sec. 3 of the RASD.

The architecture in the back-end must allow interaction between the different core components (sec. 2.3) and must facilitate upscaling and redundancy, in order to guarantee a high fault-tolerance and performance.

2.2 High level components and their interaction

From a top-level perspective, the platform is composed by a high number of clients interacting with a single server (or at least a single query endpoint or API).

The back-end is composed by different core components that implements the following high-level functions (related to the functions in sec. 2.2 of the RASD):

[HLF1] Manage and react to user requests (functions F1, F4 and F9)

[HLF2] Manage and allow management of user data (functions F2 and F5)

[HLF3] Manage user authentication (function F8)

[HLF4] Manage secondary platform functions (functions F3, F6 and F7)

The client side applications are merely an access medium to the back-end functions and mostly implement the presentation layer of the platform, which makes them of little interest in the high-level perspective that we are giving.

HLF1 is achieved with the interaction of three main software components, two dedicated to the management of requests and the remaining one dedicated to the management of the taxi queues.

HLF2 and HLF3 are achieved with a dedicated software component with direct access to the DBMS component of the platform.

HLF4 is achieved with a subset of the functions of the Request manager, which is part of the components dedicated to HLF1.

Other core components of the platform are dedicated to client-server communication and of little interest in this description (they will be described later).

2.3 Component view

The platform is composed by some main core software element that are needed to realize the functional and non functional requirements presented in the RASD. Any external dependency to make the platform operational is listed in sec. 2.1.1 of the RASD and is not explicitly treated in the present document.

2.3.1 Request manager

This module is responsible for managing the taxi requests forwarded by the RESTful API to the back-end main server.

Its main function is to forward the ride requests to the Ride manager at the right moment, based on the type of requests it receives: in the case of simple requests, it forwards them to the Ride manger immediately, whereas it stores reservation requests locally and forwards them at the right moment.

The module also handles the modification of reservations and the merging of *taxi sharing* requests when compatibles requests are found.

2.3.2 Queue manager

This software module is responsible for managing the taxi queues associated with each zone of the town.

It receives requests directly from the Ride manager and executes the selection algorithms based on the needs of the requester. It also has a communication channel through the RESTful API with the taxi-side applications, which is used to update the availability status of taxi drivers.

2.3.3 Ride manager

This module is responsible for creating the *Ride* objects when prompted by the Request manager.

It communicates with the Queue manager to identify available taxis, with the Notification manager to handle replies to the customers and with the DBMS

component to keep track of rides and reports.

2.3.4 User manager

This module is responsible for managing user data, registration and authentication.

It has direct access to the DBMS component ad receives read and write requests directly from the RESTful API.

2.3.5 Notification manager

This module is responsible for implementing the push notification service towards the user-side applications.

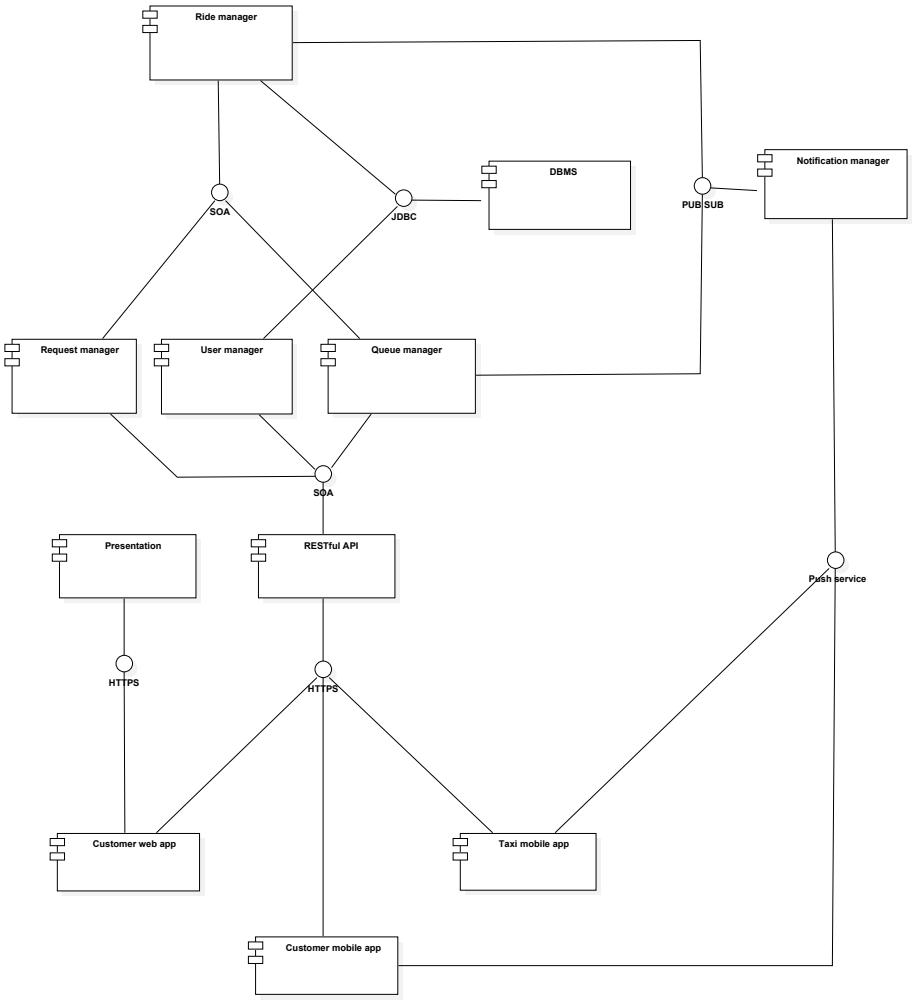
It is necessary to have this type of module running in the back-end because there are cases in which a communication must happen between the clients and the back-end, but no direct request is made to the RESTful API by the clients: these, for example, include the notifications of incoming requests to taxi drivers, or the notification of incoming taxis to users.

2.3.6 RESTful API

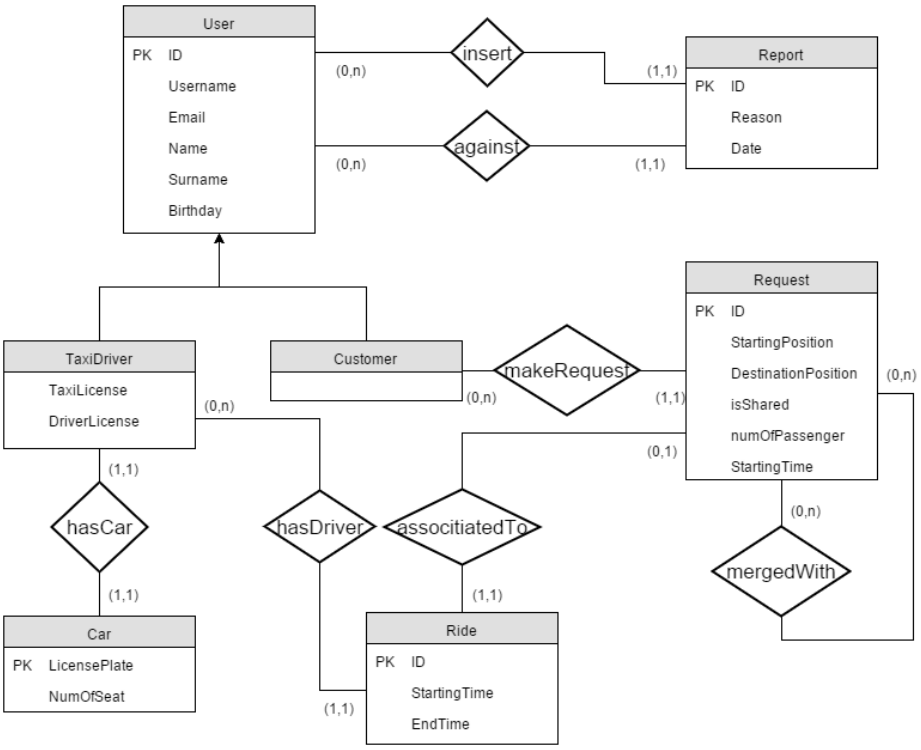
This module is the main gateway of communication that the clients have towards the back-end.

It implements a stateless service in which http calls are made by the clients when they need to submit requests o data into the system, and replies are immediately returned as a result for the requested computations. The decision of having a RESTful API is due to the fact that the system must be able to ideally support a vast number of users on a different number of platforms, and using a REST API as a single endpoint is an optimal solution for providing the same, integrated user experience throughout all the platforms.

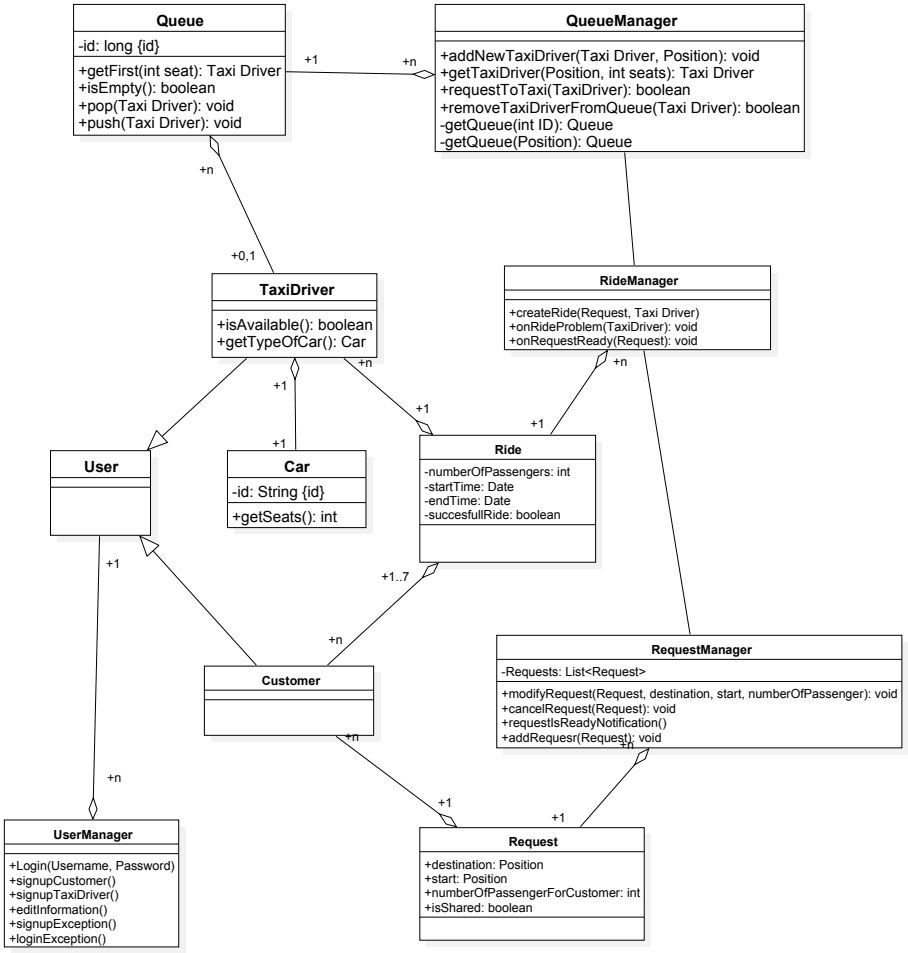
2.3.7 Component diagram



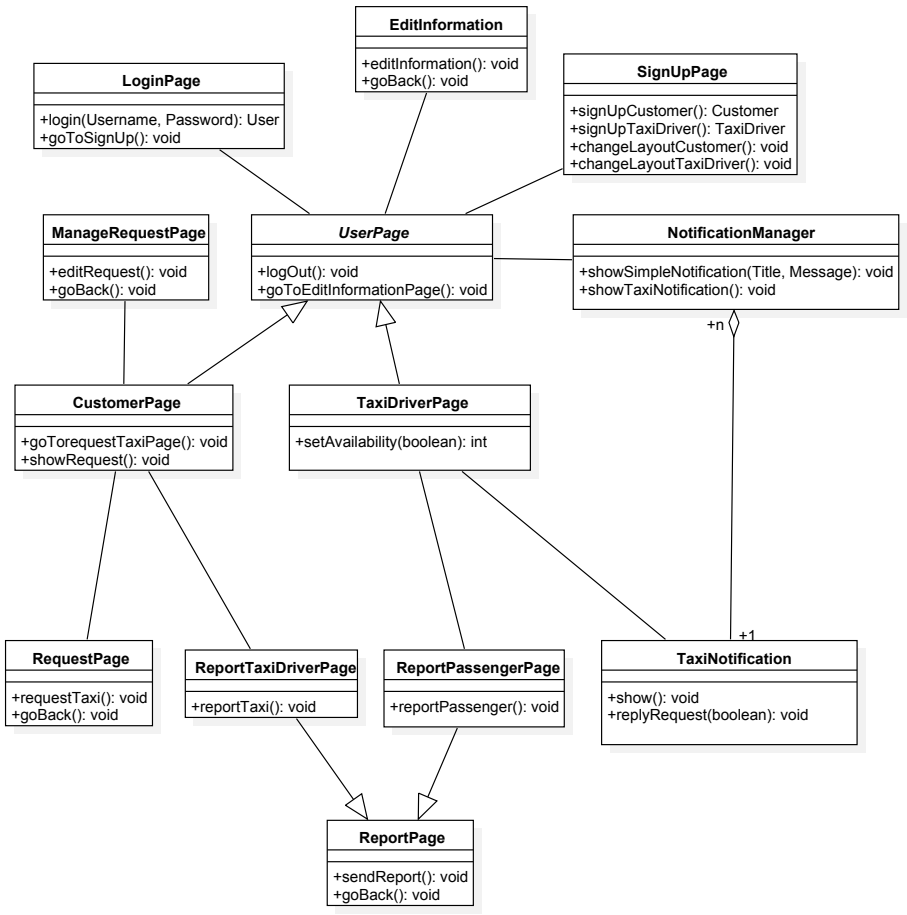
2.3.8 Entity-relationship diagram



2.3.9 Class diagram: back-end



2.3.10 Class diagram: client-side



2.4 Deployment view

The following section contains a schema of the hardware infrastructure, which represents the main operational components of the back-end and their relations. The high-level software components run on each piece of hardware (or possibly the generic hardware's function) are indicated below each node.

For clarity's sake, it was decided to omit the representation of the purely network-related hardware (routers, switches, etc.).

The chosen architecture is a typical web-based client-server architecture, with the following components:

1. **Main server array**: typical rack of servers dedicated to run the main back-

end logic. The rack is composed by independent nodes managed by usual load balancing technology, where each node is equipped with the full stack of operational software (sec. 2.3) and its dependencies (sec. 2.1.1 of the RASD) and is able to be fully functional under average load of the system.

The redundancy of the rack has the dual function of improving fault tolerance and performance under above-average load conditions.

This node is connected to the web server, through which the communication to and from the users happens, and to the DBMS server (either primary or secondary - the connection is transparent to the node and is hot-swappable in the event of a failure).

2. **Main DBMS server**: server dedicated to run the DBMS software on account of the main server array and to manage the communication with the secondary storage node.

   This node is connected to the main server array, from which it receives requests, and the storage nodes (either primary or secondary - the connection is transparent to the node and is hot-swappale in the event of a failure).

3. **Secondary DBMS server**: server dedicated to run the DBMS software and manage the backup requests from the primary DBMS server.

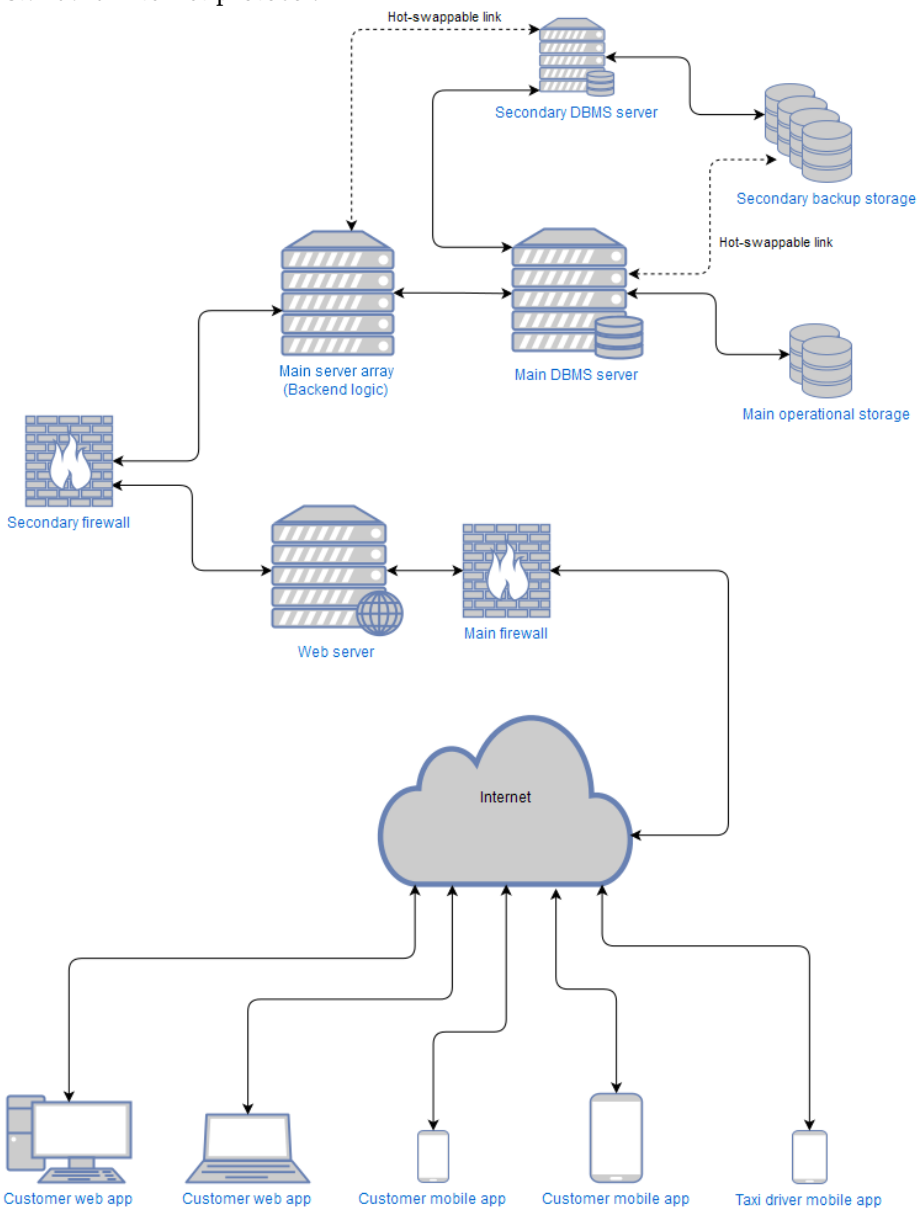
   The node is connected to the secondary storage and to the primary DBMS sever, for the reasons described above.

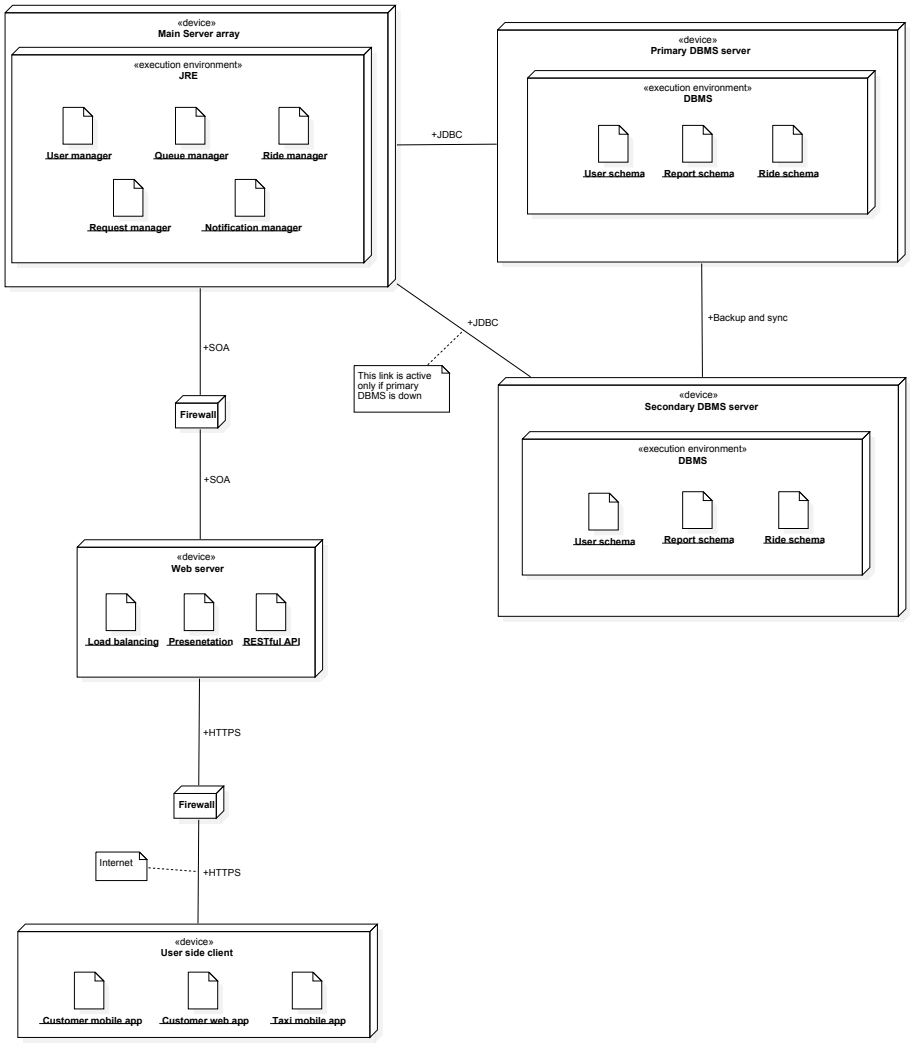
4. **Primary and secondary storage**: typical long term-storage technology on which the operational database is saved. The primary storage is the node with the highest R/W speeds and is used under normal operational regime, whereas the secondary is the most reliable one (in terms of long-term storage) and it is used to store data backups (it becomes fully operational only in the even of a failure on the primary storage or DBMS nodes).

5. **Web server**: server dedicated to run the web server software and manage the load balancing towards the main server array. It is isolated by firewalls to implement a DMZ and minimize the risk of unauthorized access to the main back-end components (especially to the storage nodes).

   This is the main gateway node of the back-end and is the one exposed to the outside world.

Any communication between the user-side apps and the back-end happens through the standard Internet protocol.

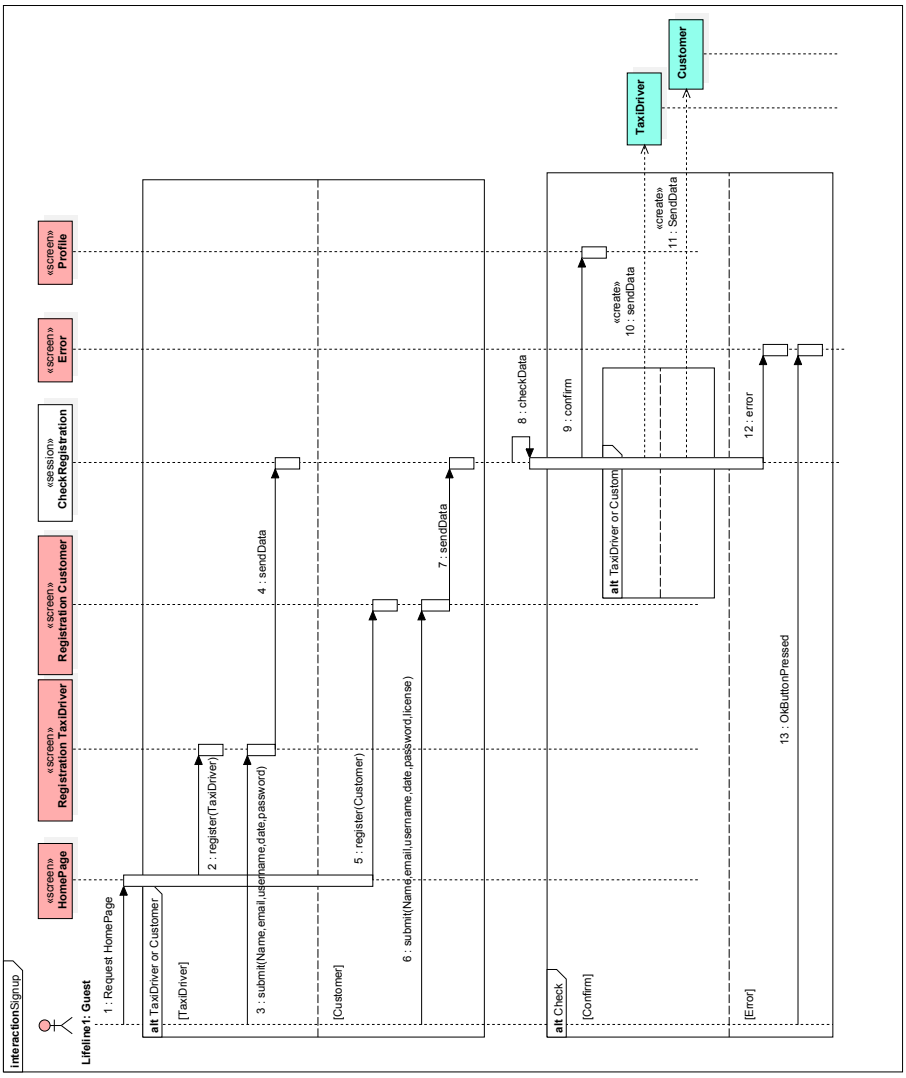


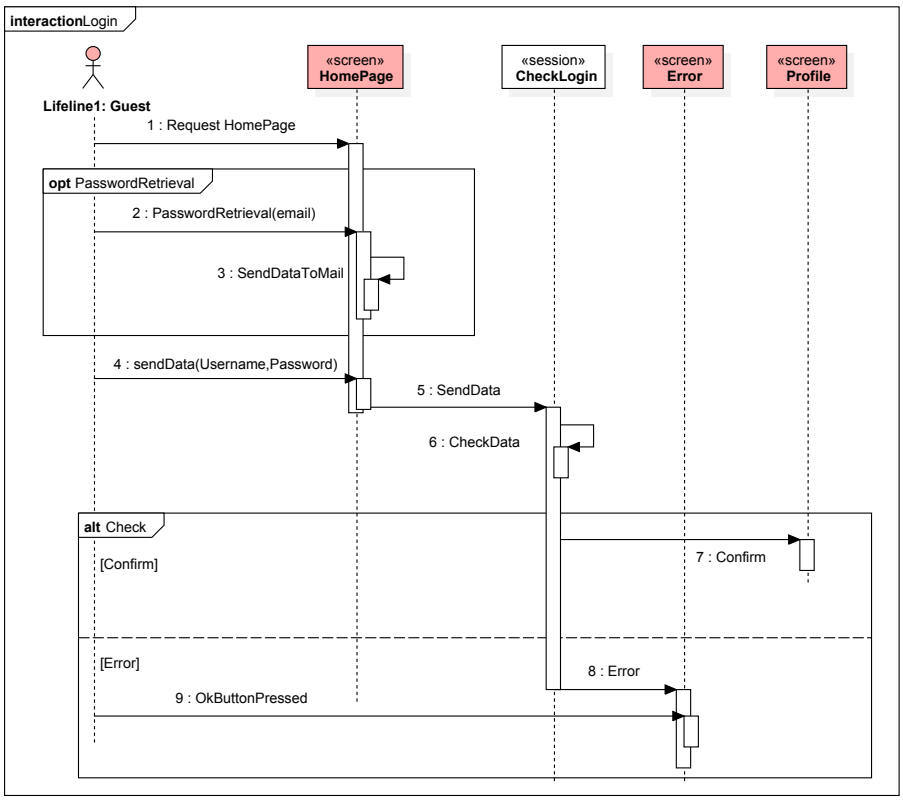


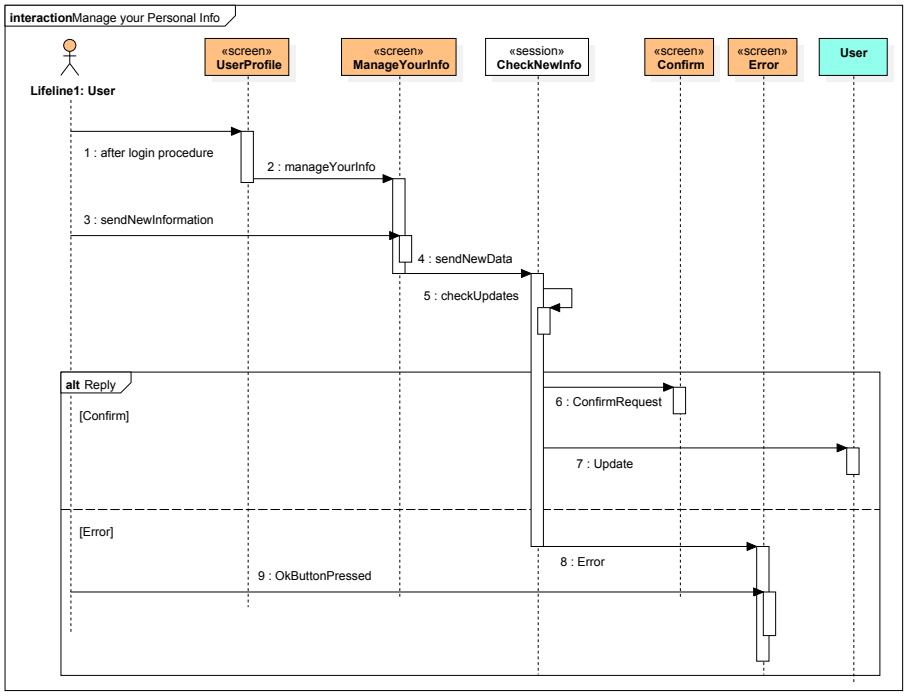
2.5 Runtime view

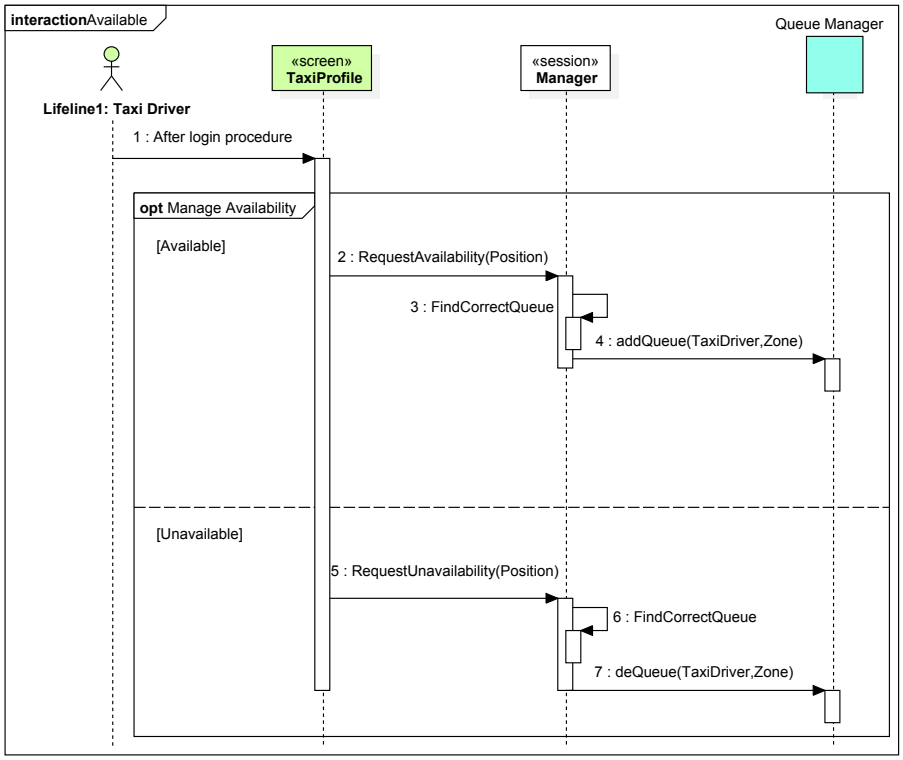
This section contains a detailed descriptions of how the software deployed to the platform's nodes must behave in response to the events of interest.

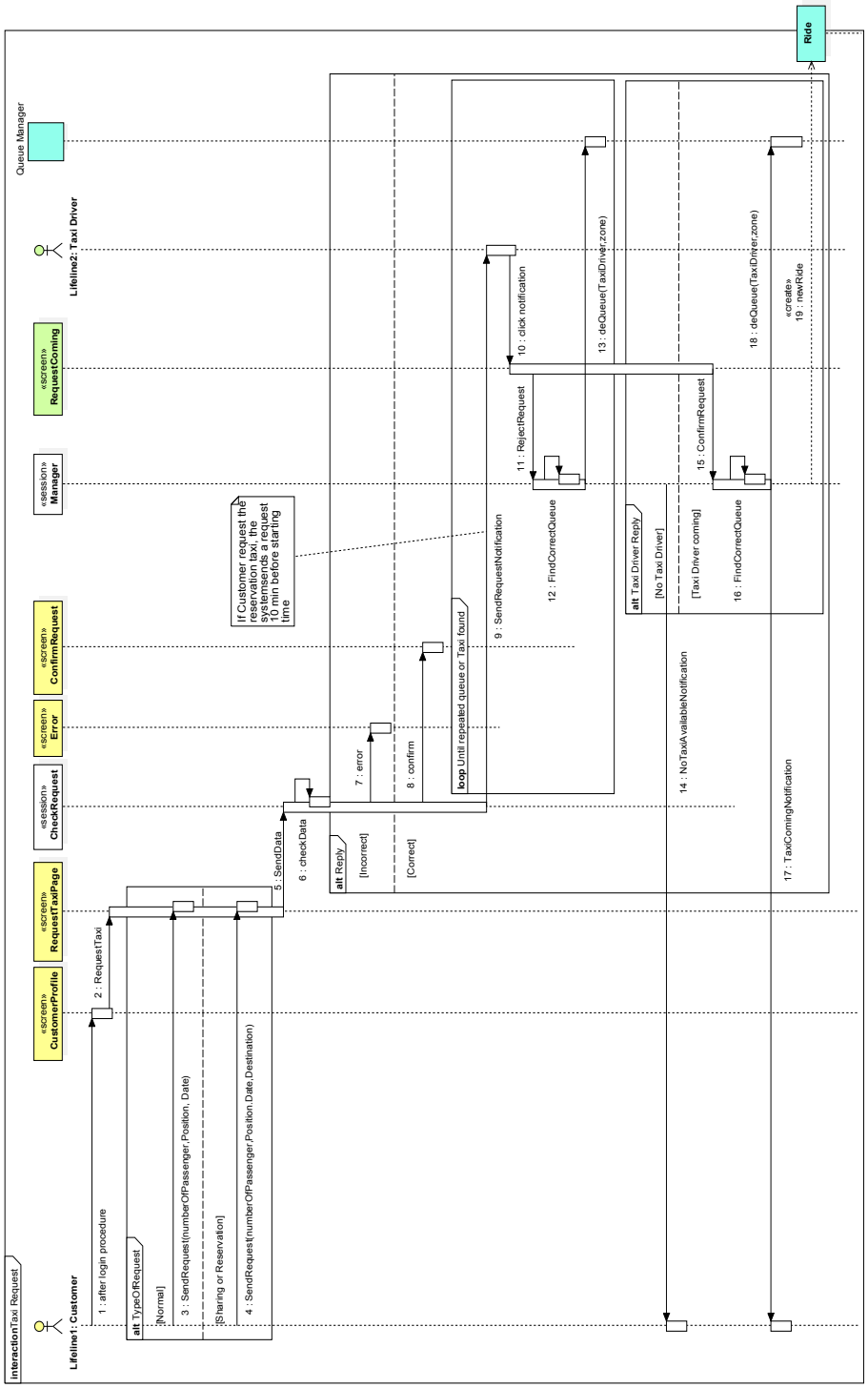
2.5.1 Sequence diagrams

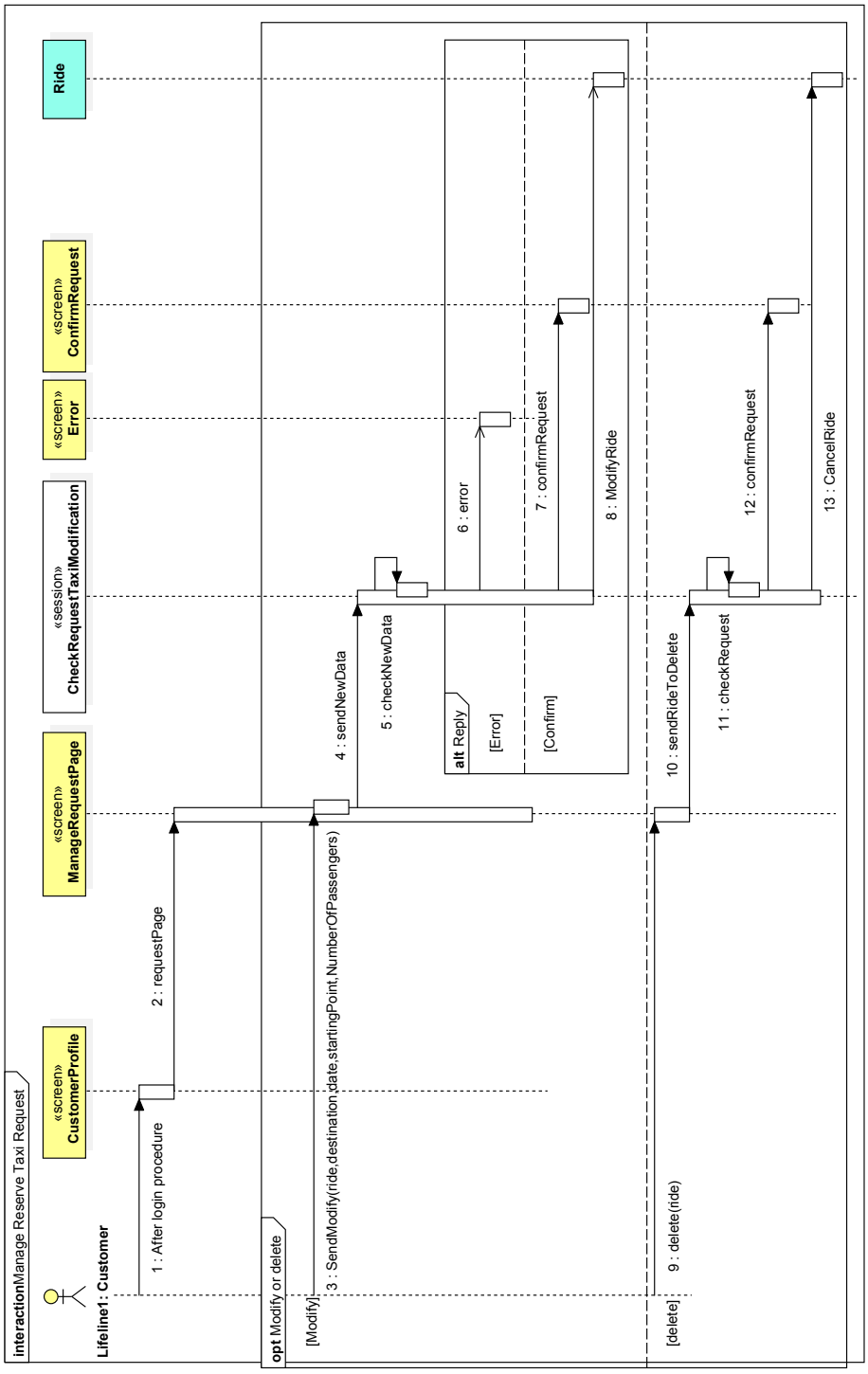


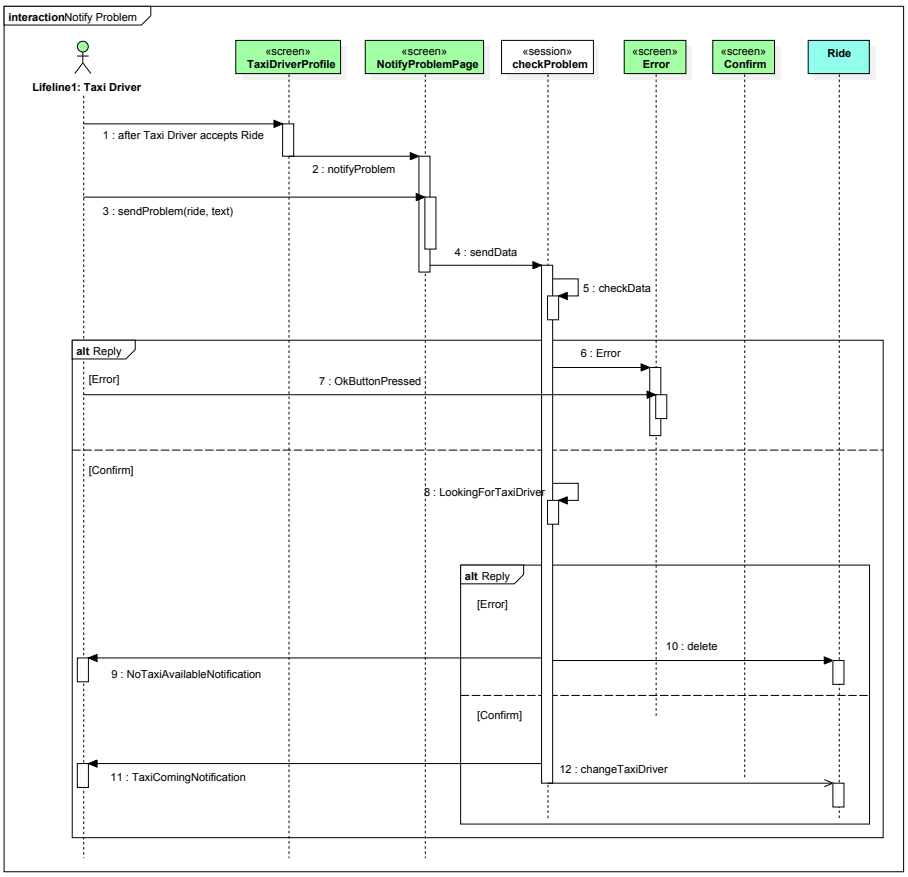

interactionLogin
«screen»
HomePage
«session»
CheckLogin
«screen»
Error
«screen»
Profile
Lifeline1: Guest
1 : Request HomePage
opt PasswordRetrieval
2 : PasswordRetrieval(email)
3 : SendDataToMail
4 : sendData(Username,Password)
5 : SendData
6 : CheckData
alt Check
[Confirm]
7 : Confirm
[Error]
8 : Error
9 : OkButtonPressed

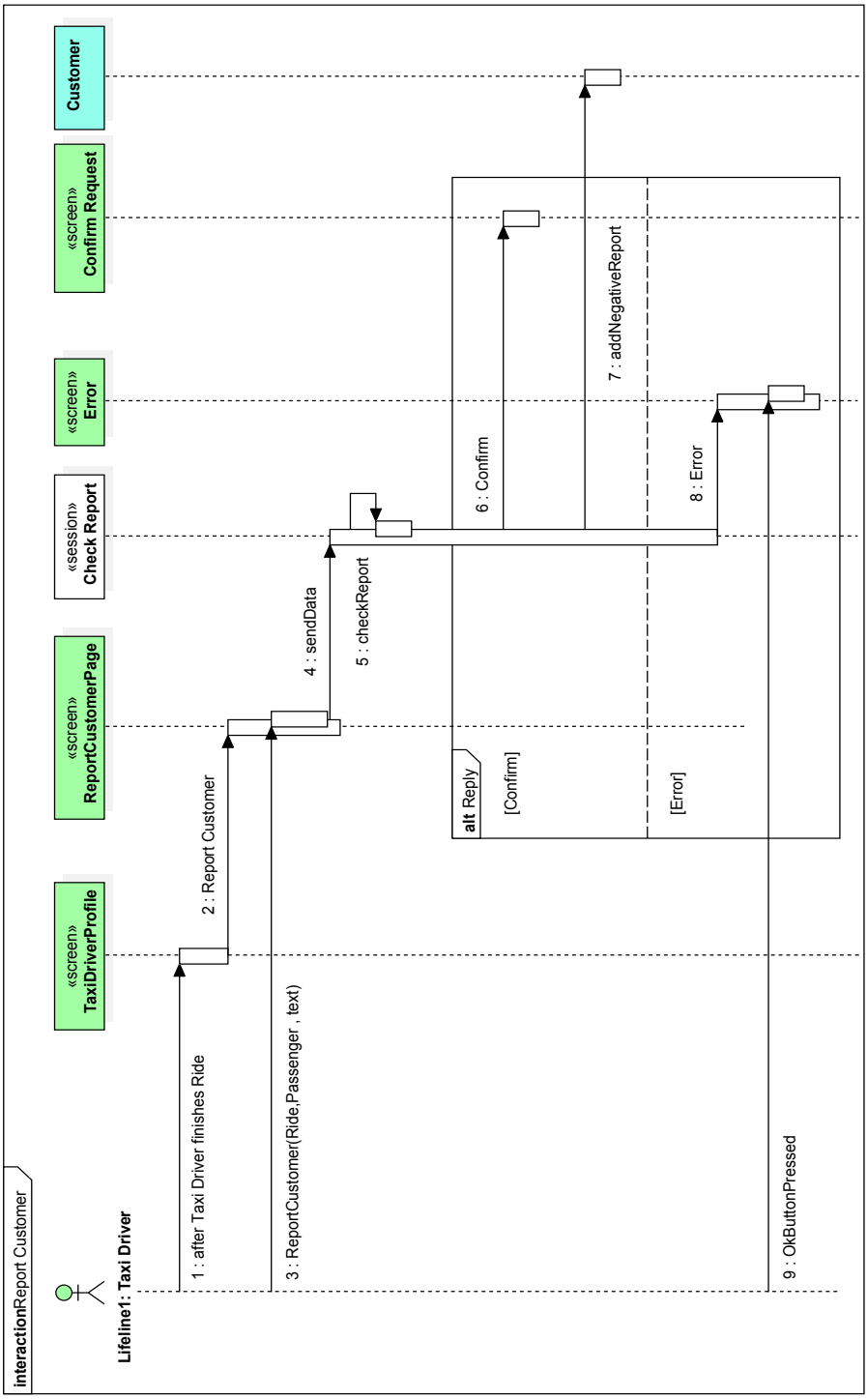

interactionManage your Personal Info
«screen» UserProfile
«screen» ManageYourInfo
«session» CheckNewInfo
«screen» Confirm
«screen» Error
User
Lifeline1: User
1 : after login procedure
2 : manageYourInfo
3 : sendNewInformation
4 : sendNewData
5 : checkUpdates
alt Reply
[Confirm]
6 : ConfirmRequest
7 : Update
[Error]
8 : Error
9 : OkButtonPressed

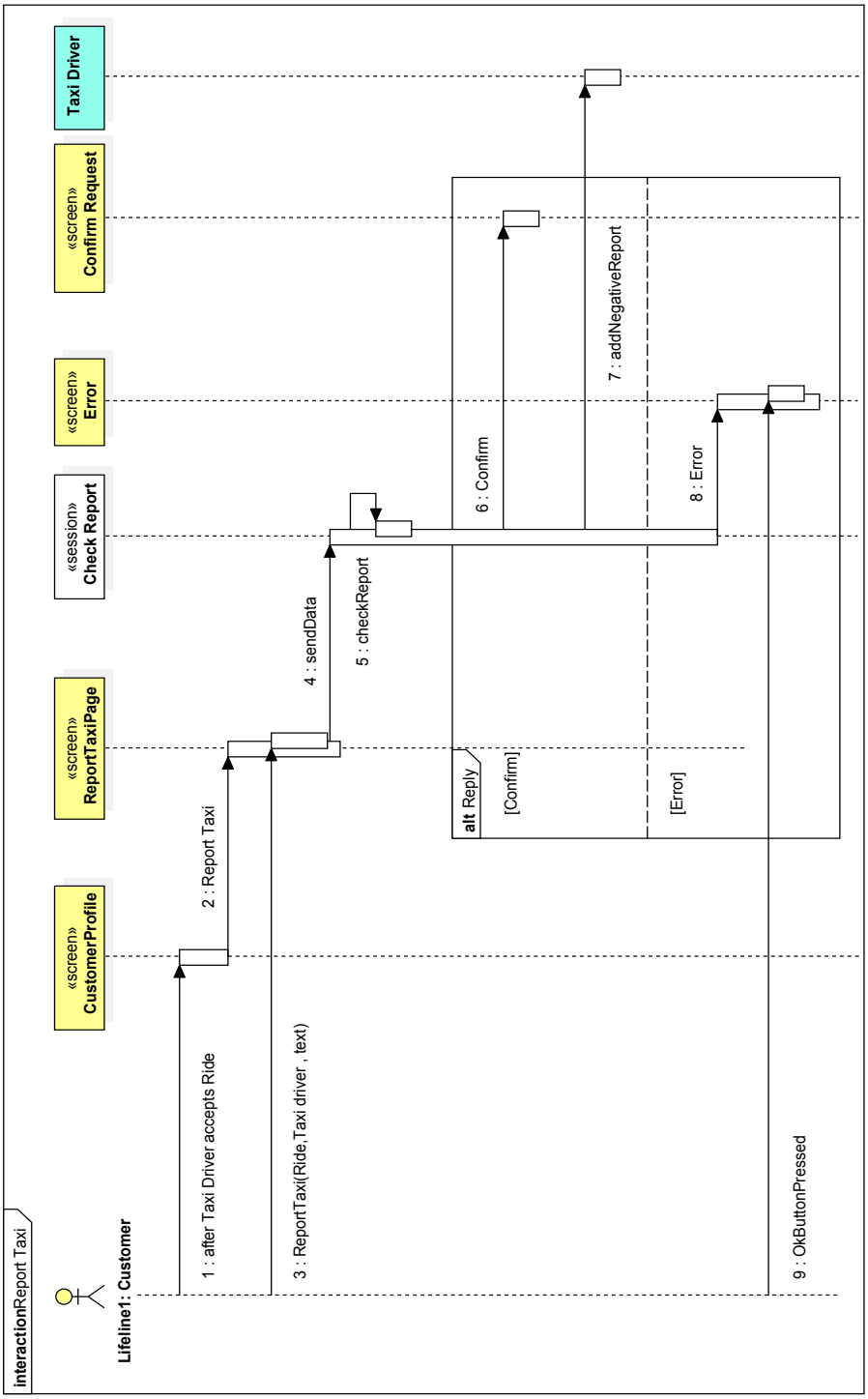

interactionAvailable
Queue Manager
«screen»
TaxiProfile
«session»
Manager
Lifeline1: Taxi Driver
1 : After login procedure
opt Manage Availability
[Available]
2 : RequestAvailability(Position)
3 : FindCorrectQueue
4 : addQueue(TaxiDriver,Zone)
[Unavailable]
5 : RequestUnavailability(Position)
6 : FindCorrectQueue
7 : deQueue(TaxiDriver,Zone)


interaction Taxi Request
Lifeline1 : Customer
«screen» CustomerProfile
«screen» RequestTaxiPage
«session» CheckRequest
«screen» Error
«screen» ConfirmRequest
«session» Manager
«screen» RequestComing
Lifeline2 : Taxi Driver
Queue Manager
Ride
1 : after login procedure
2 : RequestTaxi
alt TypeOfRequest
[Normal]
3 : SendRequest(numberOfPassenger,Position, Date)
[Sharing or Reservation]
4 : SendRequest(numberOfPassenger,Position,Date,Destination)
5 : SendData
6 : checkData
alt Reply
[Incorrect]
7 : error
[Correct]
8 : confirm
loop Until repeated queue or Taxi found
9 : SendRequestNotification
10 : click notification
11 : RejectRequest
12 : FindCorrectQueue
13 : deQueue(TaxiDriver.zone)
If Customer request the reservation taxi, the systemsends a request 10 min before starting time
alt Taxi Driver Reply
[No Taxi Driver]
14 : NoTaxiAvailableNotification
[Taxi Driver coming]
15 : ConfirmRequest
16 : FindCorrectQueue
18 : deQueue(TaxiDriver.zone)
«create» 19 : newRide
17 : TaxiComingNotification


interaction Manage Reserve Taxi Request
Lifeline1 : Customer
«screen» CustomerProfile
«screen» ManageRequestPage
«session» CheckRequestTaxiModification
«screen» Error
«screen» ConfirmRequest
Ride
1 : After login procedure
2 : requestPage
opt Modify or delete
[Modify]
3 : SendModify(ride,destination,date,startingPoint,NumberOfPassengers)
4 : sendNewData
5 : checkNewData
alt Reply
[Error]
6 : error
[Confirm]
7 : confirmRequest
8 : ModifyRide
[delete]
9 : delete(ride)
10 : sendRideToDelete
11 : checkRequest
12 : confirmRequest
13 : CancelRide

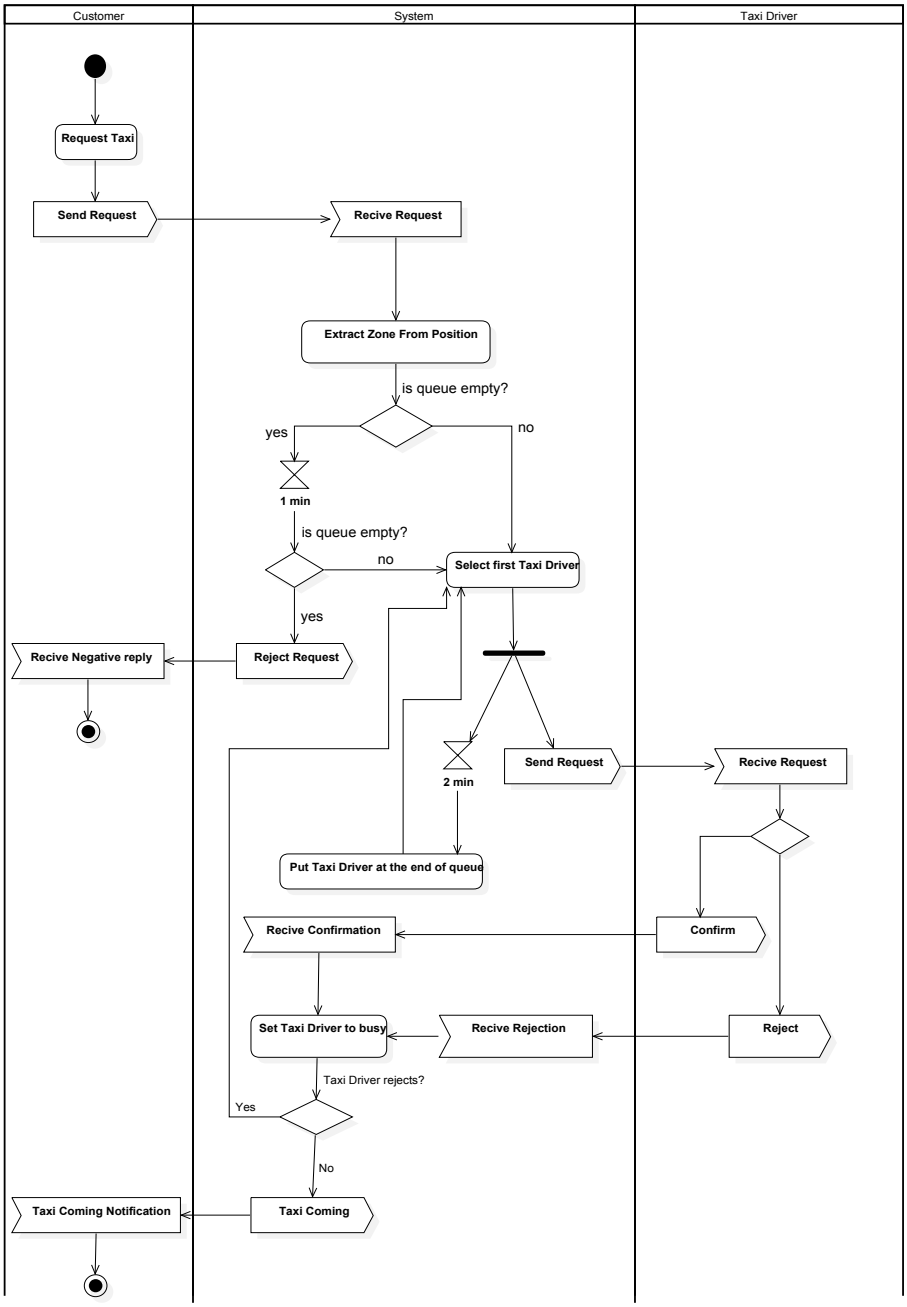

interactionNotify Problem
«screen» TaxiDriverProfile
«screen» NotifyProblemPage
«session» checkProblem
«screen» Error
«screen» Confirm
Ride
Lifeline1: Taxi Driver
1 : after Taxi Driver accepts Ride
2 : notifyProblem
3 : sendProblem(ride, text)
4 : sendData
5 : checkData
alt Reply
[Error]
6 : Error
7 : OkButtonPressed
[Confirm]
8 : LookingForTaxiDriver
alt Reply
[Error]
10 : delete
9 : NoTaxiAvailableNotification
[Confirm]
12 : changeTaxiDriver
11 : TaxiComingNotification


interactionReport Customer
Lifeline1: Taxi Driver
«screen» TaxiDriverProfile
«screen» ReportCustomerPage
«session» Check Report
«screen» Error
«screen» Confirm Request
Customer
1 : after Taxi Driver finishes Ride
2 : Report Customer
3 : ReportCustomer(Ride,Passenger , text)
4 : sendData
5 : checkReport
6 : Confirm
7 : addNegativeReport
8 : Error
9 : OkButtonPressed
alt Reply
[Confirm]
[Error]


interactionReport Taxi
Lifeline1: Customer
«screen» CustomerProfile
«screen» ReportTaxiPage
«session» Check Report
«screen» Error
«screen» Confirm Request
Taxi Driver
1 : after Taxi Driver accepts Ride
2 : Report Taxi
3 : ReportTaxi(Ride,Taxi driver , text)
4 : sendData
5 : checkReport
6 : Confirm
7 : addNegativeReport
8 : Error
9 : OkButtonPressed
alt Reply
[Confirm]
[Error]

2.5.2 Taxi request: activity diagram

It was decided to analyze the taxi request process more in detail as it is the core function of the platform.



2.6 Component interfaces

2.6.1 SOA

The software modules in the back-end are are designed to be as independent as possible from each other, meaning that there must be single points of contact between the different core components.

Therefore, between the RESTful API and the Queue, User and Request managers, and between the latter and the Ride manager, communications happen by implementing different exposed interfaces that the relevant modules can exploit to start processes on the other components.

2.6.2 Publish/subscribe

The publish/subscribe design pattern is used to interface the Ride and Queue manager components with the Notification manager, where the latter is the subscriber and must react to events suitably generated by the former, which are the publishers.

2.6.3 JDBC

JDBC is a Java API that manages the access of a generic client to a database. In our platform, it acts as interface between the Ride and User managers and the DBMS component (on both the primary and secondary nodes).

2.6.4 Push technology

Push technology is an implementation of the publish/subscribe design pattern, which is widely used in modern mobile applications to manage notification services.

In our platform, requests are generated by the Notification manager and picked up by the user-side mobile applications, which react to them by displaying the suitable notifications on the users' devices.

2.7 Selected architectural styles and patterns

From a generic point of view, the platform is obviously designed as a client-server architecture, in which the back-end nodes are the server and the various users' devices are the clients.

The main point of communication between clients and server is a RESTful API, which implements a stateless communication protocol: data exchanges between the clients and the servers only happens in the case of a request from the clients (with the exception of the notification service implemented by the Notification manager component, which uses push technology to "send requests" to the clients - sec. 2.6.4). For this reason, the state of the user sessions must be stored in the back-end components in order to implement coherent user sessions and the selected design pattern to achieve this is the standard MVC pattern.

Of the back-end modules:

- The RESTful API is part of the *view*, as it offers access from the "outside" to the other components.
- The Request and Queue manager are part of the *model* from the platform point of view, but are internally implemented with a MVC pattern themselves as they need to react to events and change their model accordingly.
- The Ride manager is part of the *controller*, as its purpose is purely to bridge the model and the view, and create ride objects using the information stored in the model when needed.
- The User manager can be interpreted as part of the *controller*, although the "model" it refers to is the DBMS itself and has a different connotation with respect to the traditional MVC pattern.
- The Notification manger is part of the *view*, but implements a publish/subscribe-like pattern (see. 2.6.4) to interact with the clients.

Moreover, as specified in sec. 2.6.2, it was decided to use a publish/subscribe pattern to handle communication between the Ride and Queue managers and the Notification manager.

Finally, the client components (user-side applications) are also designed with a MVC pattern, but can be loosely considered (from a high-level perspective) as part of the back-end *view*. The pattern is implemented with:

- The user interface being the *view;*
- The local data structures needed to mirror the relevant parts of the back-end *model* being the *model;*
- The logic associated to actions on the UI being the *controller*.

3 Algorithm Design

3.1 Request manager

The Request manger provides all the methods to manage and dispatch the incoming requests.

- **addRequest(Request)**: inserts the given Request in the correct position w.r.t. the starting hour, analyzes the *shared ride* requests into the list and, using the GoogleMaps API, checks for compatible paths and tries to merge them.
- **modifyRequest(Request, Position destination, Position start, int passengers)**: if the request was merged in a single *shared ride* request, then splits the two and deletes the relevant request; after that inserts the edited request in the queue, with **addRequest(Request)**.
- **requestIsReadyNotification()**: checks every 10 seconds whether the first request in the list must be dispatched, and possibly notifies to the Ride manager to create a Ride.

3.2 Ride manager

The Ride manager manages the pending Rides and Taxi problem reports.

- **onReadyRequest(Request)**: asks to the Queue manager an available TaxiDriver with a suitable Car (based on the relevant request); if an available Taxi exists, creates a Ride with **createRide(Request, TaxiDriver)**, otherwise:
  - if the request is a *shared ride* request composed by different requests, tries to split the ride into as few smaller rides as possible.
  - if the request is a normal request, start an error procedure and notify the delay to the user via the Notification manager
- **onProblem(TaxiDriver)**: terminates the TaxiDriver's active ride, generate a new Request to match the customer's original request and then uses the **onReadyRequest(Request)** to find a new taxi.

3.3 Queue manager

The QueueManager manage the zones of the city managing them like a Queue.

- **getTaxiDriver(Position, int seats)**:
  - Looks for the first available taxi driver, with a car with a capacity greater or equal to the given *seats* parameter, in the queue associated to the given position:
    - If a taxi is found, the driver is notified: if they accept the ride, the TaxiDriver object is returned to the caller; else, the TaxiDriver is put at the end of the queue.
  - If no taxi driver is found, tries to look for a suitable taxi driver in the neighboring queues with the same procedure as above. If the search fails, an error is generated and the Ride manager reacts as described above.
- **requestToTaxi(TaxiDriver)**: prompt the Notification manager to notify the selected taxi driver with the data contained in the request.
- **addNewTaxiDriver(TaxiDriver, Position)**: called in reaction to a change in the availability status of a taxi driver, inserts the driver at the end of the queue.

3.4 Low level algorithm description

3.4.1 RequestManager

- **addRequest(Request request)**

```
for(i=0;i<requests.size();i++)
        if(request.startTime.isPrevious(requests.get(i).startTime)
                requests.add(request, i)
Path possiblePath := GoogleMaps.getPath(request.start, request.destination)
forall(Request r : requests)
        Path rPath := GoogleMaps.getPath(r.start, r.destination)
        if((possiblePath.contains(r.start) && possiblePath.contains(r.destin
                || (possiblePath.contains(r.start) && rPath.contains(request
                || (rPath.contains(request.start) && rPath.contains(request.
                || (rPath.contains(request.start) && possiblePath.contains(r
```

```
                request.merge(r)
```

- **modifyRequest(Request request, Request newRequest)**

```
if(requests.contains(request))
        requests.remove(request)
else
        forall(Request r : requests)
                if(r.containsRequest(request))
                r.extractRequest(request)
addRequest(newRequest)
```

- **requestIsReadyNotification()**

```
while(!requests.isEmpty())
        if(requests.get(0).startTime > getTime() −10)
                RideManager.onReadyRequest(requests.remove(0))
```

3.4.2 RideManager

- **onReadyRequest(Request request)**

```
TaxiDriver taxiDriver:= QueueManager.getTaxiDriver(request.start, request.nu
if(taxiDriver is not null)
        createRide(request, taxiDriver)
else
        onReadyRequest(request.splitLeft())
        onReadyRequest(request.splitRight())
```

- **onProblem(TaxiDriver taxiDriver)**

```
forall(Ride r : rides)
        if(r.getDriver = taxiDriver)
                rides.remove(r)
                onReadyRequest(r.getRequest())
                break
```

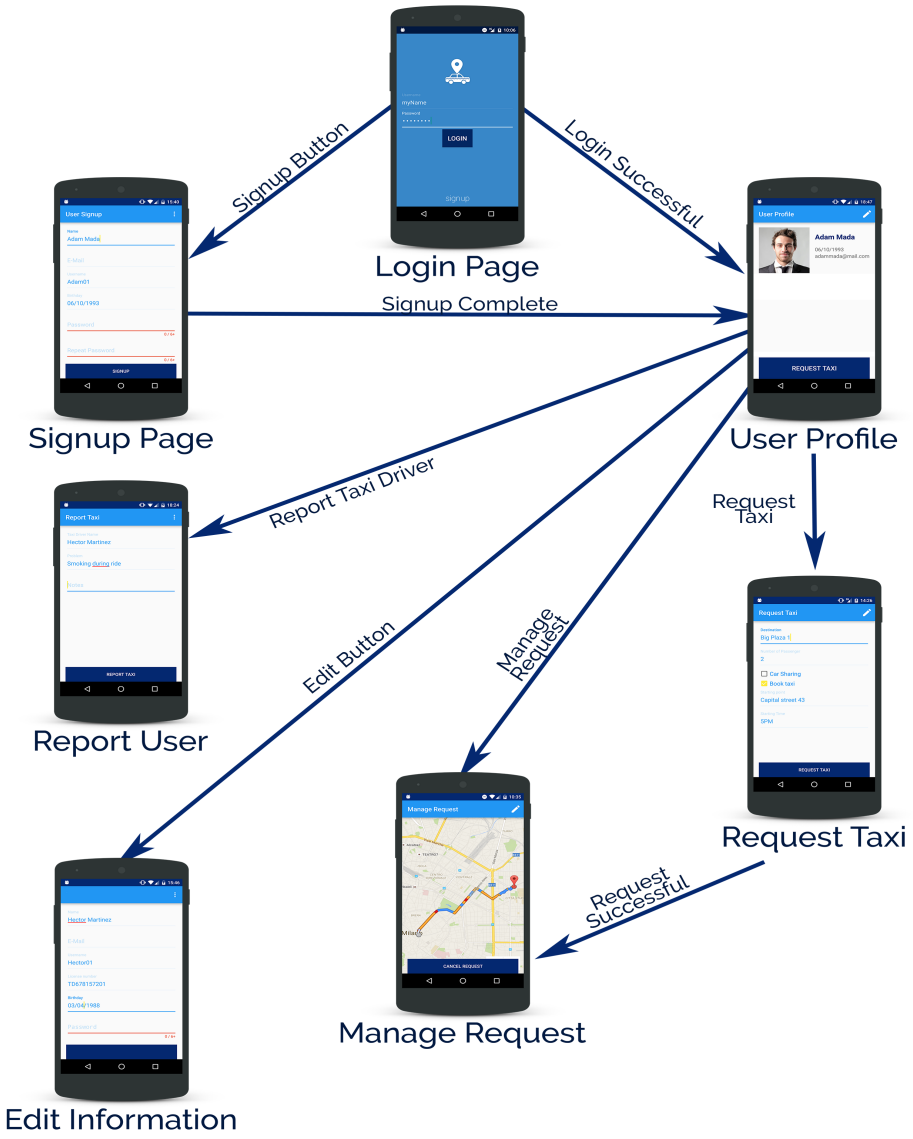
3.4.3 QueueManager

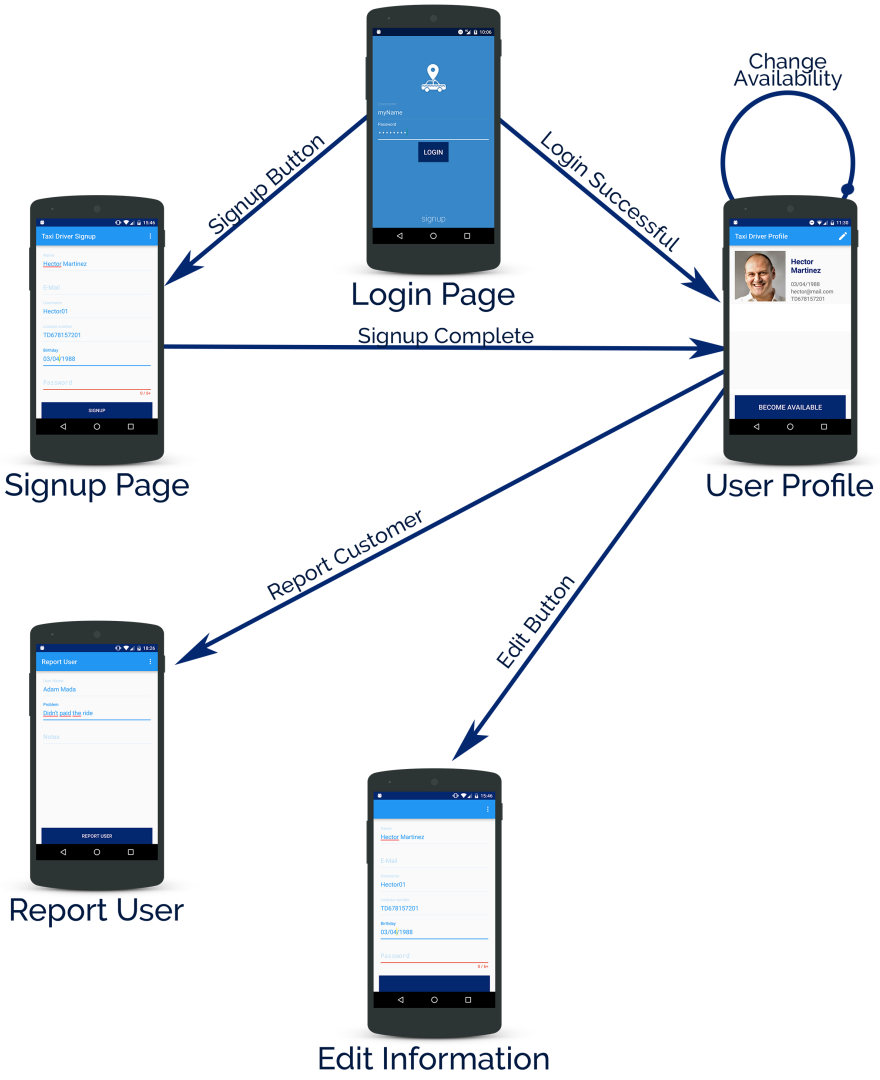
- **getTaxiDriver(Position position, int seats)**

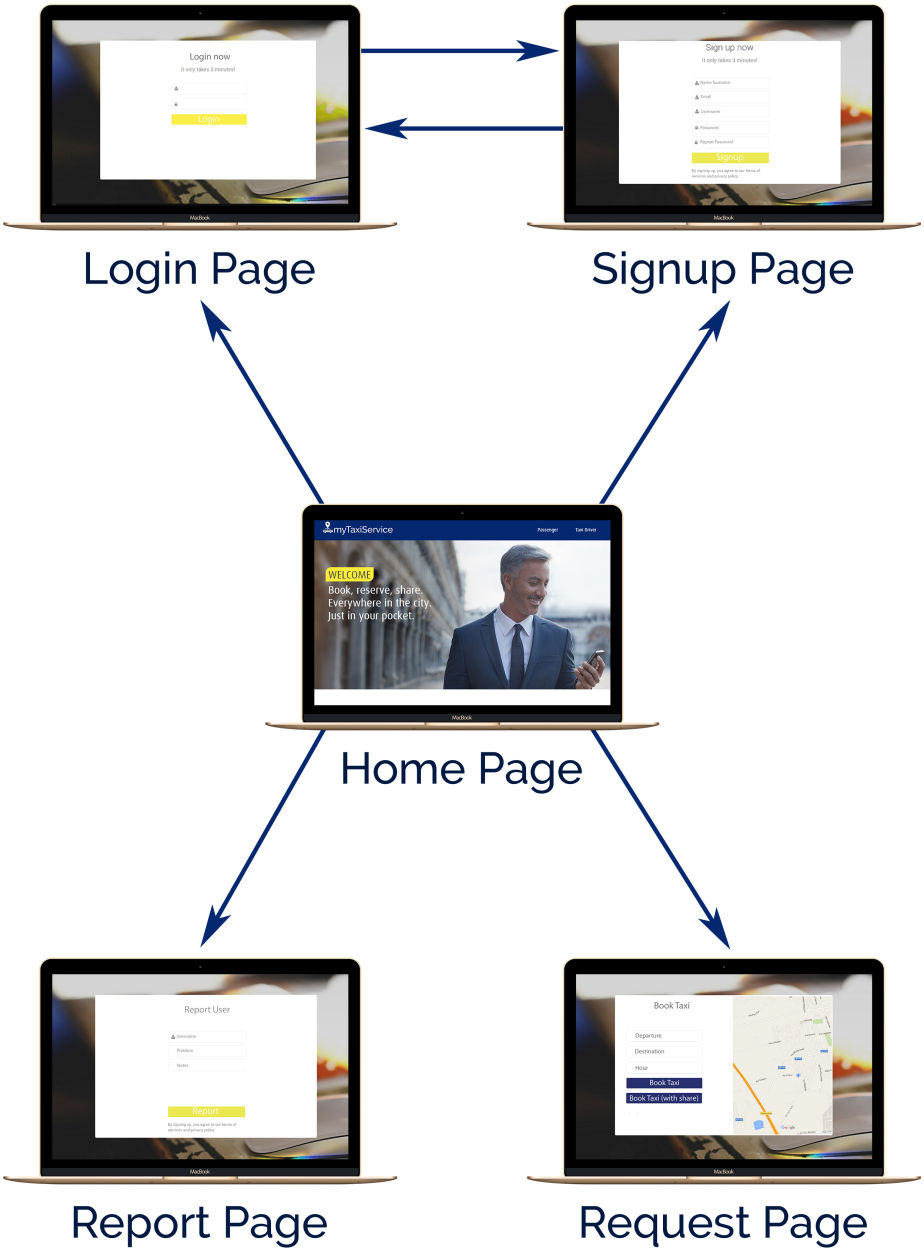
```
Queue queue := getQueue(position)
TaxiDriver taxiDriver := queue.getFirst(seats)
if(taxiDriver is not null)
        return taxiDriver
else
        forall(Queue q : MapUtils.getNeighbor(position))
                taxiDriver := q.getFirst(seats)
                if(taxiDriver is not null)
                        return taxiDriver
return NoTaxiDriverAvailableException
```

4 User Interface Design

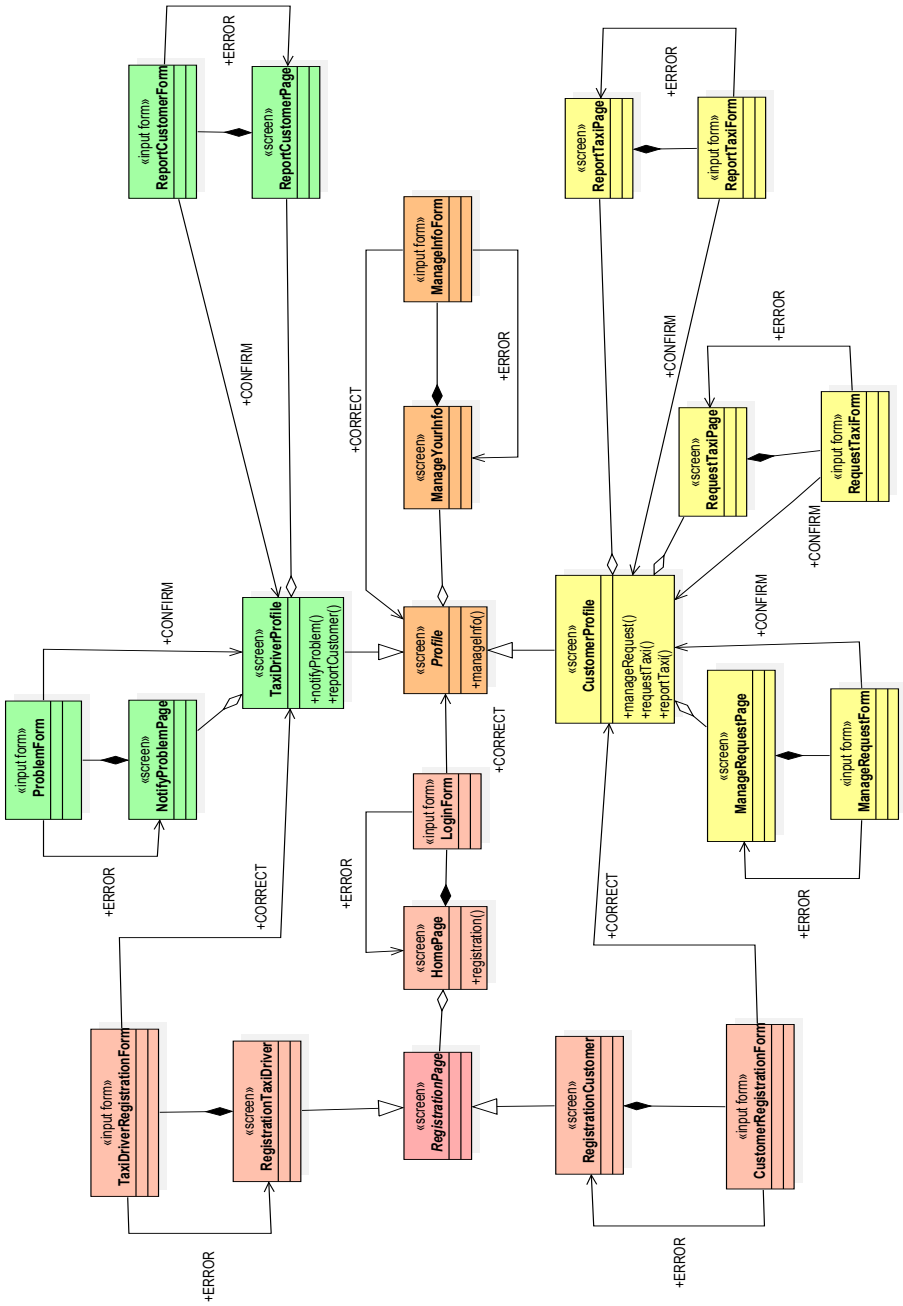
4.1 Page flow




Change Availability
Signup Button
Login Successful
Signup Complete
Report Customer
Edit Button
Login Page
Signup Page
User Profile
Report User
Edit Information


Login now
Login
Sign up now
Signup
Login Page
Signup Page
myTaxiService
WELCOME
Book, reserve, share.
Everywhere in the city.
Just in your pocket.
Home Page
Report User
Report
Book Taxi
Departure
Destination
Hour
Book Taxi
Book Taxi (with share)
Report Page
Request Page

4.2 UX diagram



5 Additional comments

The production of this document has been a joint effort of all the authors, with a fair distribution of the mansions which caused each member of the group to work on all the parts of the document.

The production has been carried out between 12/11/2015 and 4/12/2015 for a total time expense of:

- **Group work**: 21 hours
- **Individual work**:

Daniele Grattarola (Mat. 853101)	9 hours
Ilyas Inajjar (Mat. 790009)	11 hours
Andrea Lui (Mat. 850680)	10 hours