POLITECNICO DI MILANO

Computer Science and Engineering

Project of Software Engineering 2: "*myTaxiService*"

Design Document

Author: Andrea Maioli (mat. 852429)
Reference Professor: Mirandola Raffaela

Summary

1. Introduction

1.1 Purpose

The purpose of this document is to provide a guideline to the design of the software and the architecture of the system for MyTaxiService web and mobile application.
This document also provides a description of the interaction between the system and the actors (drivers, passengers, call center operator, non-registered users) and the actions they can perform.

1.2 Scope

The scope of this document is not focused on the implementation details, which will be defined during the implementation phase of the project.
All the details defined in this document have the scope to meet the requirements specified in the RASD documentation and are focused on a high-level definition of the architecture and software design.

1.3 Definitions, Acronyms, Abbreviations

- **API**: stands for *Application Programming Interface* and consist in a set of routines, protocols and tools for building software application. They can facilitate the integration of new features into existing applications, even external from the considered one.
- **Call Center Operator**: is a person who works in the call center and answers the phone calls.
- **Cloud Computing**: is a kind of Internet-based computing, where shared resources, data and information are provided to computers and other devices on-demand. It is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources.
- **DBMS**: stands for *Data Base Management System* and consist in a system software for creating and managing databases. It provides users and programmers a systematic way to create, retrieve, update and manage data.
- **DMZ**: stands for *demilitarized zone* and is a physical or logical subnetwork that contains and exposes an organization's external-facing services to a larger and untrusted network, usually the Internet.
- **Drop-off point**: location in which the passenger will be drop off by the taxi. It is specified by the passenger when entering the taxi vehicle.
- **Firewall**: is a network security system that monitors and controls the incoming and outgoing network traffic based on predetermined security rules. A firewall typically establishes a barrier between a trusted, secure internal network and another outside network, such as the Internet, that is assumed to not be secure or trusted.
- **GPS**: stands for *Global Positioning System* and is a system that uses satellites to determine the position of a vehicle.
- **HTTP**: stands for *HyperText Transfer Protocol* and is the underlying protocol user by the World Wide Web. It defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands.
- **HTTPS**: stands for *HyperText Transfer Protocol Secure* and basically consists in a HTTP communication over an SSL tunnel, that ensures the security of the communication.
- **Method**: is a function provided inside a class.
- **Pick-up point**: location in which the passenger will be picked up from the taxi. It is automatically specified by the application or by the passenger (if the request comes from a phone call).
- **POST**: is one of many request methods supported by the HTTP protocol.
- **Salt**: salt is random data that is used as an additional input to a one-way function that hashes a password or passphrase. The primary function of salts is to defend against dictionary attacks versus a list of password hashes and against pre-computed rainbow table attacks.

- **Storage Engine**: is a library for MySQL that implements the methods for the physical management of the data.
- **Tier**: refers to the physical level of an architecture. Can be represented by a machine or multiple one that in the complex acts like one.

1.4 Reference Documents

- Requirements Analysis and Specification Document
- DD Table Of Content
- IEEE Standard for Information Technology—Systems Design— Software Design Descriptions
- IEEE Standard Systems and software engineering — Architecture description
- Web References:
    - Secure hashing in java: https://www.owasp.org/index.php/Hashing_Java
    - JEE 7 documentation: https://docs.oracle.com/javaee/7/

1.5 Document Structure

This document is structured in five parts:

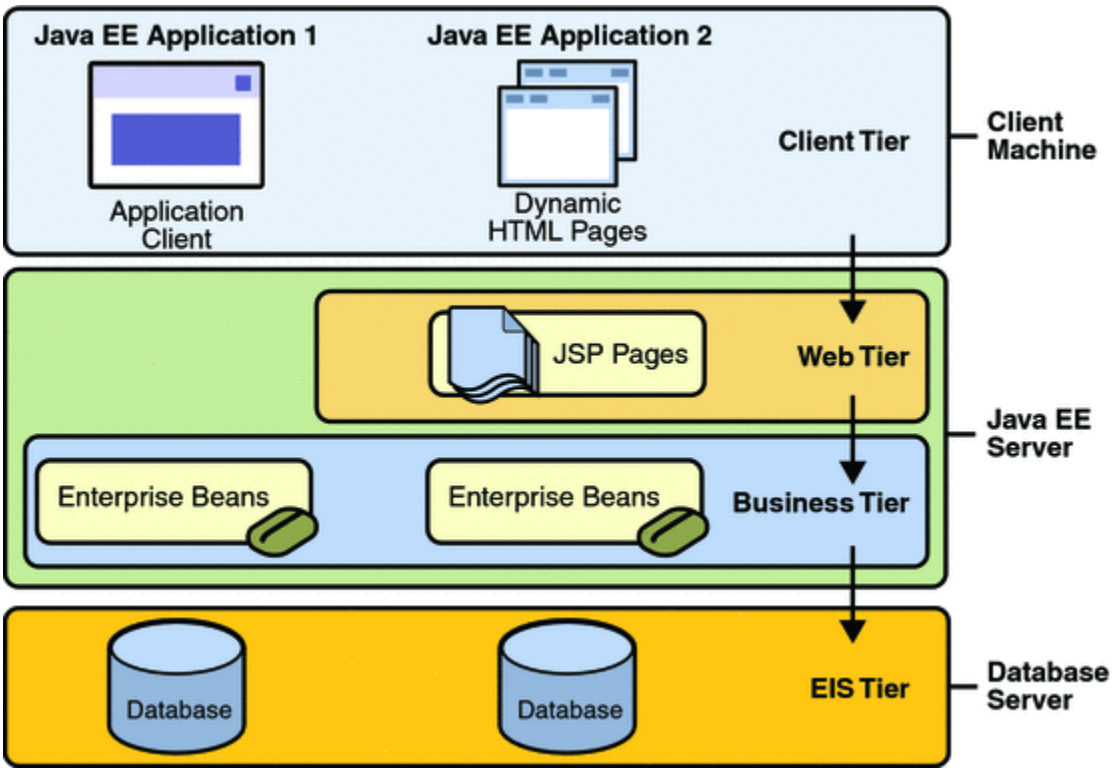
1) Overview: provides an overview of the entire document.
2) Architectural Design: focus on the definition of the architecture of this project.
3) Algorithm Design: focus on the definition of the most relevant algorithmic part of this project.
4) User Interface Design: provides an overview on how the user interfaces of the system will look like.
5) Requirements Traceability: provides a description of how the requirements defined in the RASD map into the design elements that have been defined.

2. Architectural Design

2.1 Overview

This chapter will provide a view of the system components, both at a physical and logical level.
Before analyzing the design choice made in this document, is important to know that the infrastructure of this application is based on the model provided by Java Enterprise Edition, which is composed by four tiers:

1) Client Tier: it represents the presentation layer of the application and manages the user interface of both web and mobile applications.
2) Web Tier: manages the communication between the Client tier and the Business Tier. It contains the Servlets and Dynamic Web Pages that needs to be elaborated.
3) Business Tier: manages all the business logic of the application and contains the Enterprise Java Beans, which contain the business logic, and Java Persistence Entities.
4) Enterprise Information System Tier: it contains the database that stores all the data used by the application and permits access to it.



The main benefit for using Java EE is that while the presentation logic and business logic must be implemented by the developers, a various amount of other services can be imported and used without developing them (such as server and connection handling or session handling).

2.2 High Level Components and their Interaction

Client

Is represented both by the Browser, used to access the web application, and the Mobile Application installed on the user smartphone or tablet.
According to the RASD, it will be developed for the most relevant Mobile Operative Systems (iOS, Android and Windows Phone) with their respective native programming language. The view of each mobile application will be identical, independently with the platform used and will also have the same type of interaction with the system.
Both the web browser and the mobile application will interact with the Web Server.

Web Server

The web server provides a web interface to access the system via web browsers and doesn't contains any application logic.
The Web Server will interact with the Application Server by directly accessing the Remote EJB of interest.

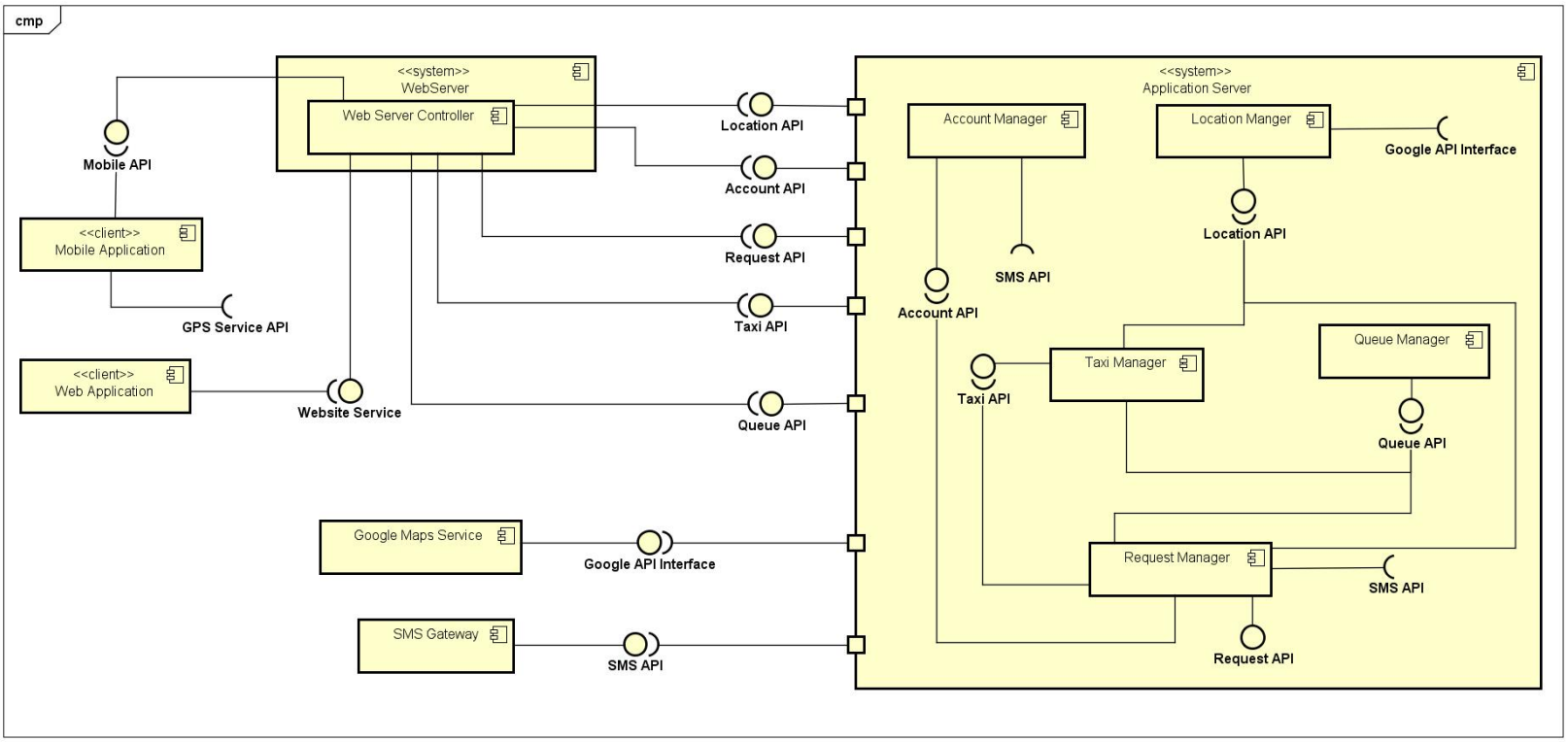
Application Server

The application server contains all the application logic of the system and manages all the actions inside it, and corresponds to the Business Tier of the Java EE Model.
This components will contains the Session Beans and the Message Driven Beans, and will provide a RESTful API used both by the web server and the mobile application in order to access the functionalities offered by myTaxiService.
The Application Server will interact with the Database Server through its JPA component.

Database Server

The database server contains the application data and information used by the entire system.
It can be accessible only from the Application Server, which will manage the interactions with all the data stored.
The database server will consists in MySQL Server as DBMS.

2.3 Component View



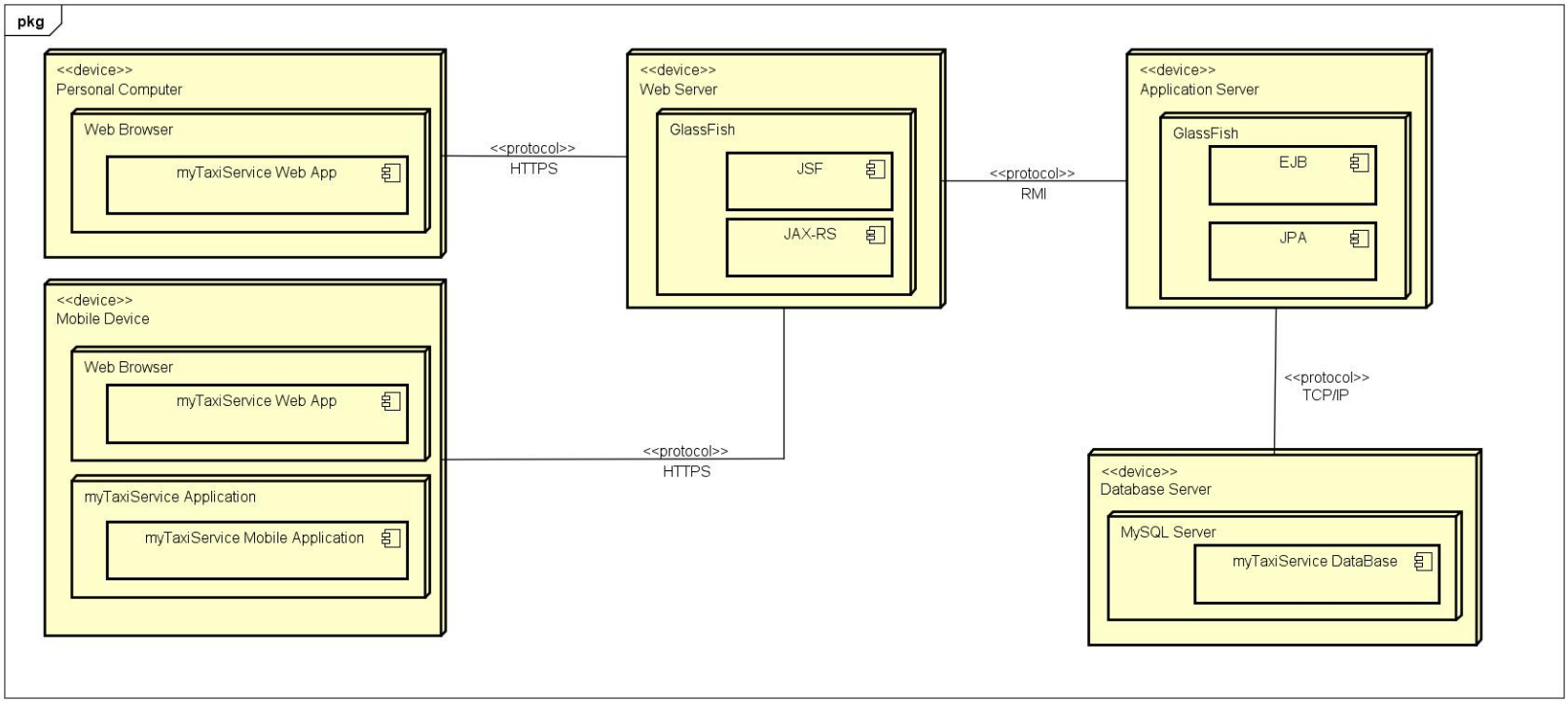
All the application logic implemented in the Application Server will be divided into different components, in base of their function:

- ***Request Manager***: this component provides all the functionalities required to create and manage taxi requests, such as create a request, update its status, get its assigned taxi and more.
- ***Queue Manager***: this component provides all the functionalities required to manage a queue, such as move a taxi up or down in the queue, remove a taxi from the queue, get the area of a queue and more.
- ***Account Manager***: this component provides all the functionalities required to manage an account, such as register, login, edit information, get user type and more.
- ***Location Manager***: this component provides all the functionalities required for handling geographic coordinates and taxi's areas, such as finding the associated queue of an area, get the area of a location, computing the time required to arrive to a place and more.
- ***Taxi Manager***: this component provides all the functionalities required to manage a taxi, such as getting its status, update its status and more.

The Web Server will contain only the **Web Server Controller** that is in charge of the invocation of the methods provided by the API exposed by the Application Server. This component will also manage both the request through the provided RESTful API (or Mobile API) and the Website.

Is important also to notice that the only difference between the *mobile application* and the *web client* is that the mobile application will access the RESTful API and the web browser will access the website in order to use the functions provided by the system.

2.4 Deployment View



The ***Web Server*** Tier will be run using *Glassfish Server* and will be implemented using:

- JavaServer Faces (JSF): a framework based on MVC that will combine HTML code with Java code, handling the presentation layer of the web application.
- Java API for RESTful Web Services (JAX-RS): is an API that will be implemented in order to allow the mobile application to access the provided services.

The ***Application Server*** Tier will run using *Glassfish Server* too, but will be implemented using:

- *Enterprise Java Beans* (EJB) for implementing the application logic, which will be:
    - *Session Beans*: A session bean encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server. A session bean is not persistent.
    - *Entity Beans*: represents persistent data maintained in a database. An entity bean can manage its own persistence (Bean managed persistence) or can delegate this function to its EJB Container (Container managed persistence). An entity bean is identified by a primary key. If the container in which an entity bean is hosted crashes, the entity bean, its primary key, and any remote references survive the crash.

      The Entity Beans, which will be created, are: Queue, Area, Request, Administrator, Driver, Passenger, Operator and User. All of these beans will be used from the components of the Application Server, in order to interact with the database, which won't be interrogated by other classes and/or components.
    - *Message Driven Beans*: is an enterprise bean that allows Java EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events. The messages can be sent by any Java EE component (an application client, another enterprise bean, or a web component) or by

a JMS application or system that does not use Java EE technology. Message-driven beans can process JMS messages or other kinds of messages.

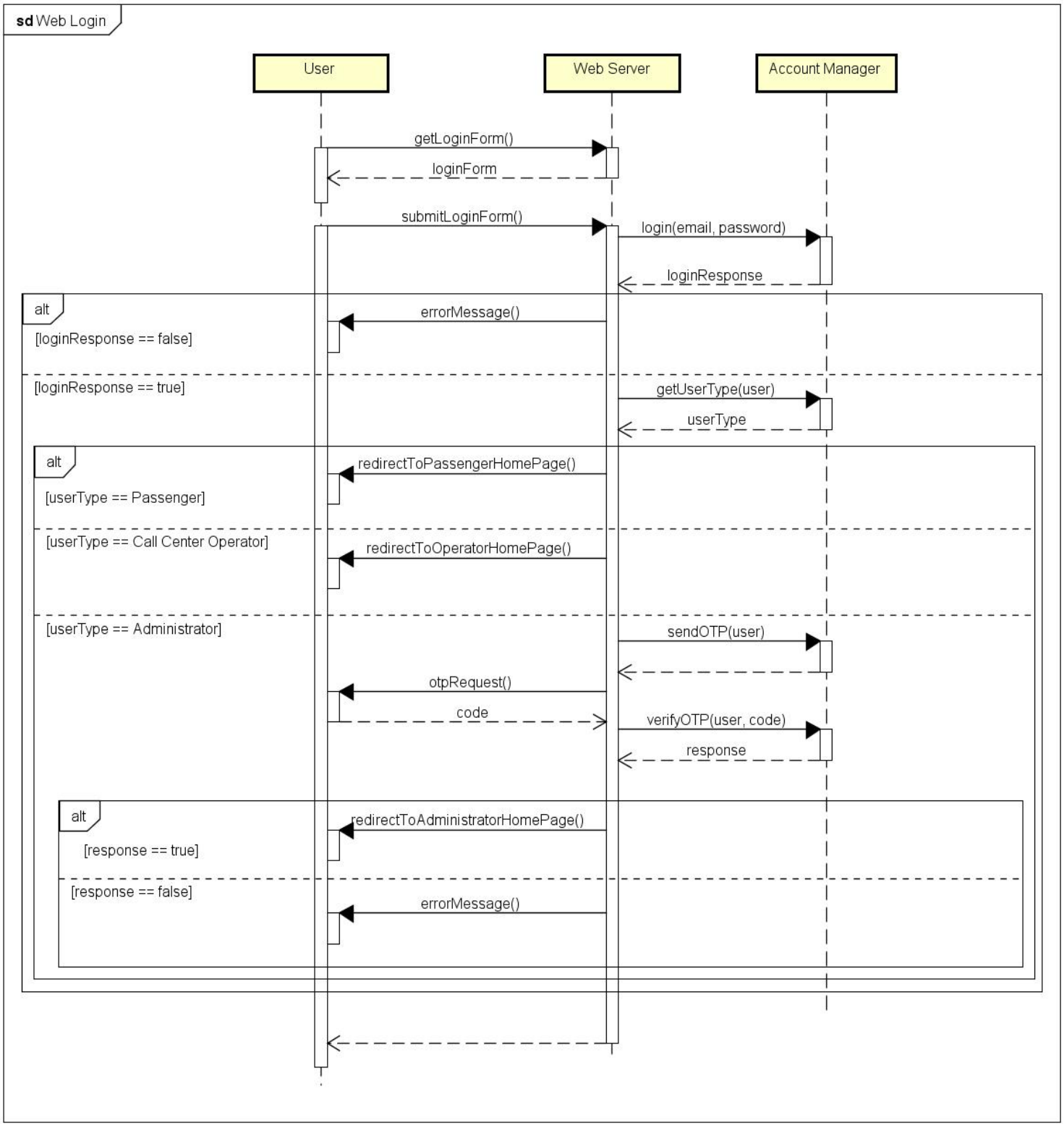
- *Java Persistence API* (JPA): is a collection of classes and methods to persistently store the vast amounts of data into a database.

The ***Database Server*** Tier will be run using *MySQL Server* with the InnoDB storage engine.

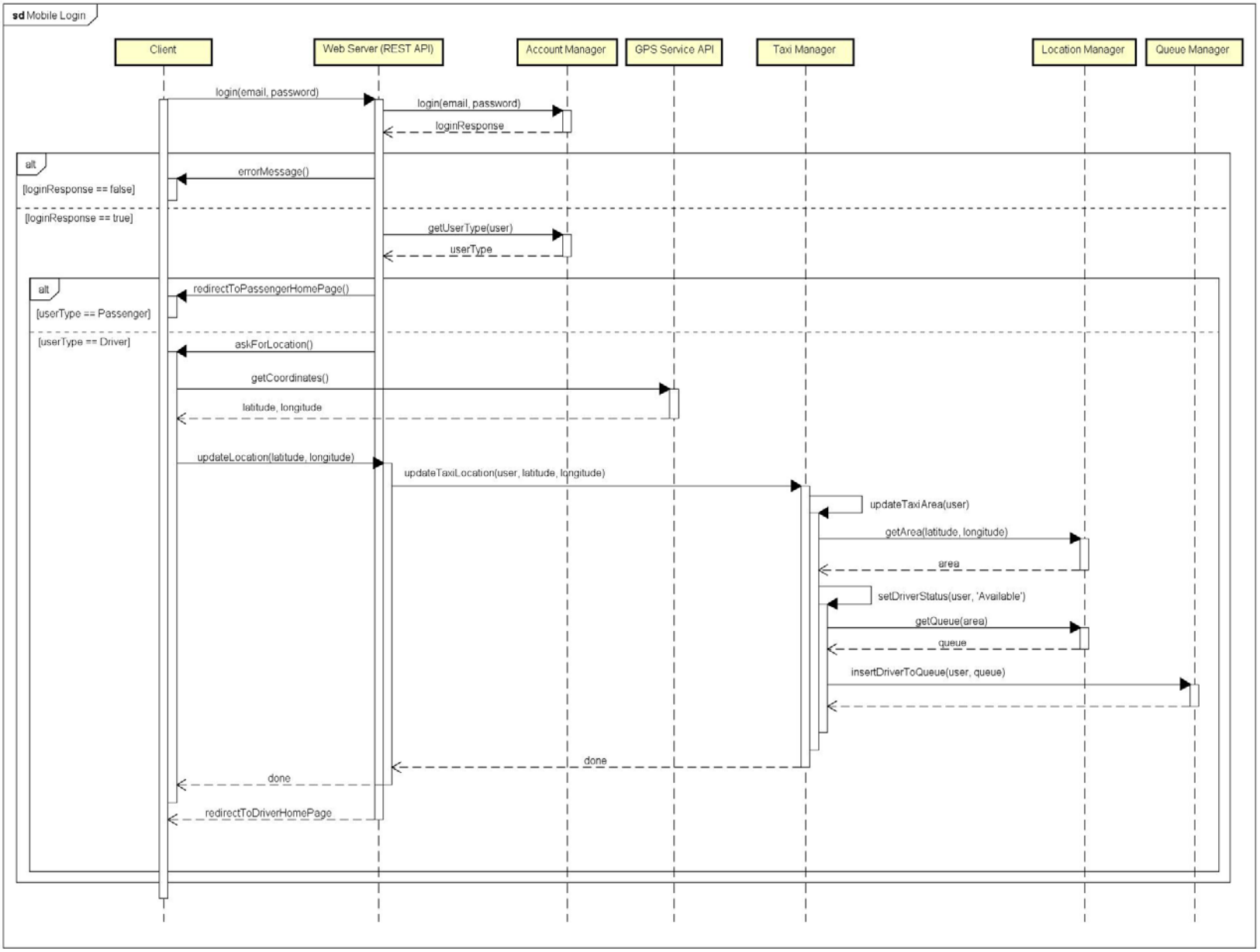
2.5 Runtime View

This section will provide a description of the dynamic behavior of the system, showing the interaction of the components running in the different tiers when an action is performed by a user.

2.5.1 Web Login



2.5.2 Mobile Login

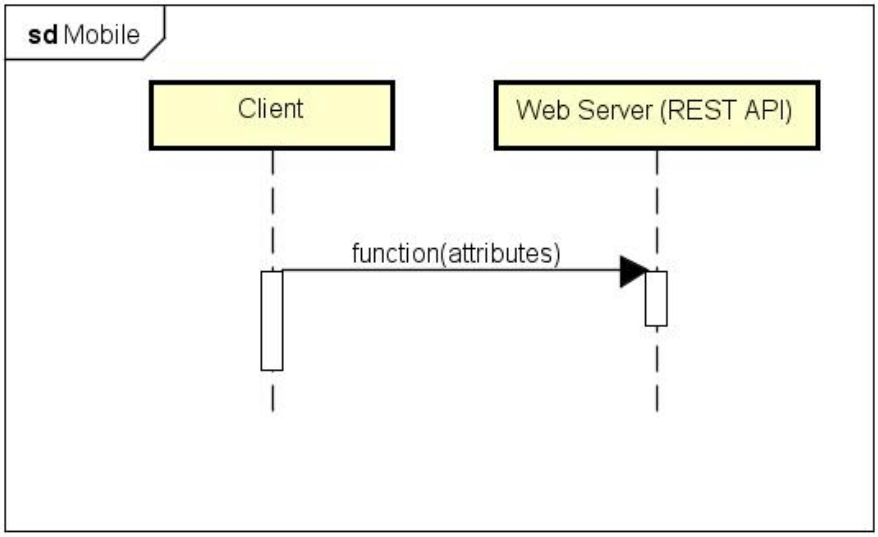


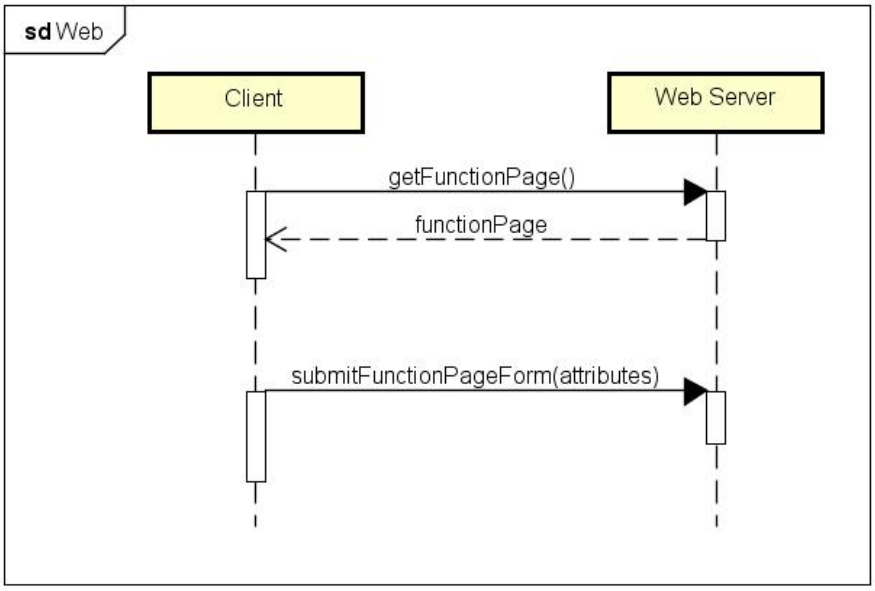
The only difference between the mobile application and the web application runtime view is the interaction with the webserver (it can be seen on the mobile and web login sequence diagram):

- The web application asks for a specific page and, once received it, send a POST request submitting a form.
- The mobile application skips asking for a page and directly send the request to the webserver, through the REST API provided.

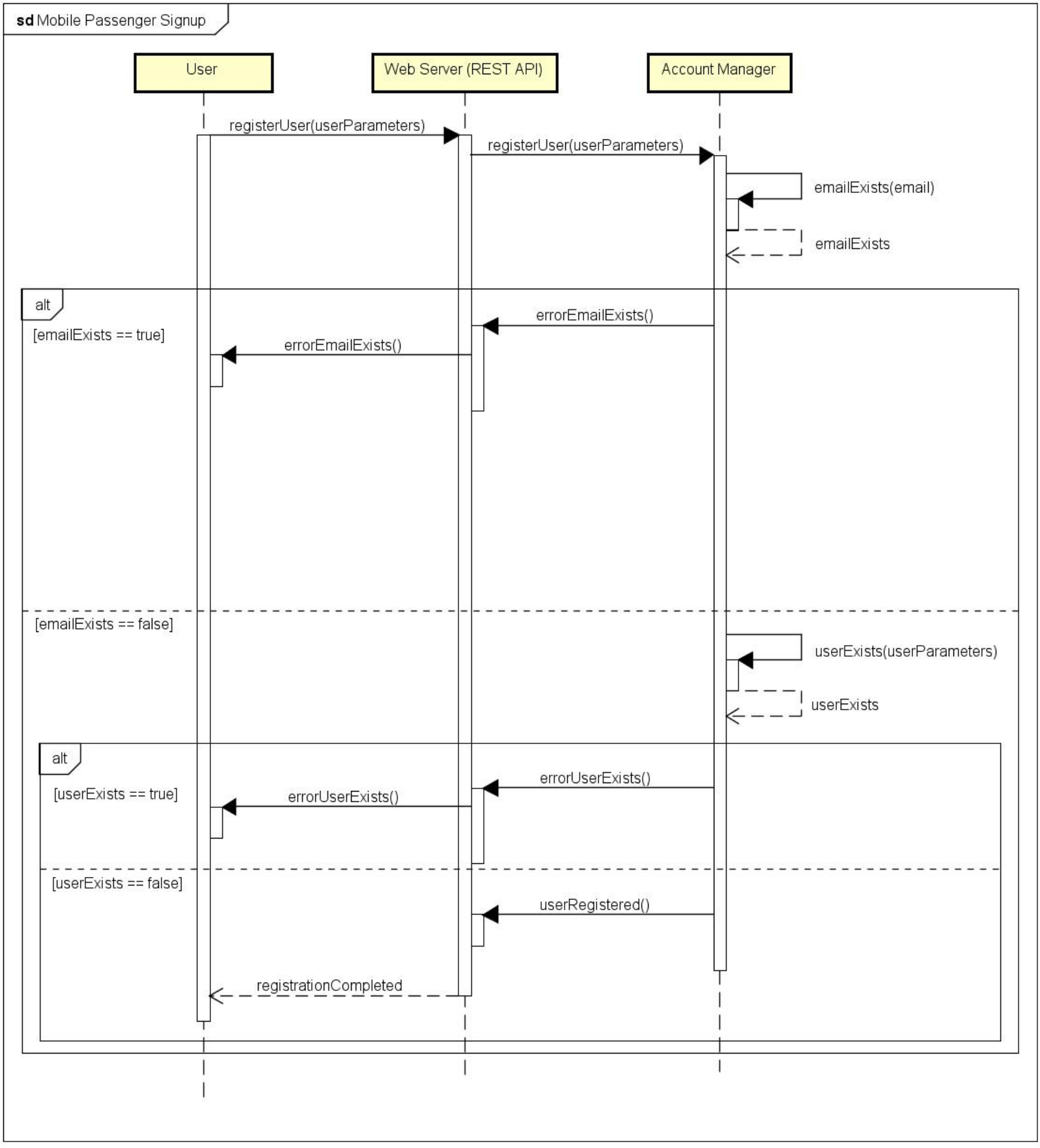
From now, for simplicity, the sequence diagrams will not consider the request for a specific page and each action will be considered coming from the mobile application.

In order to consider the web application case of a given sequence diagram, for each request sent to the web server from a client must be considered the relative page request and wait for the response, as shown below.

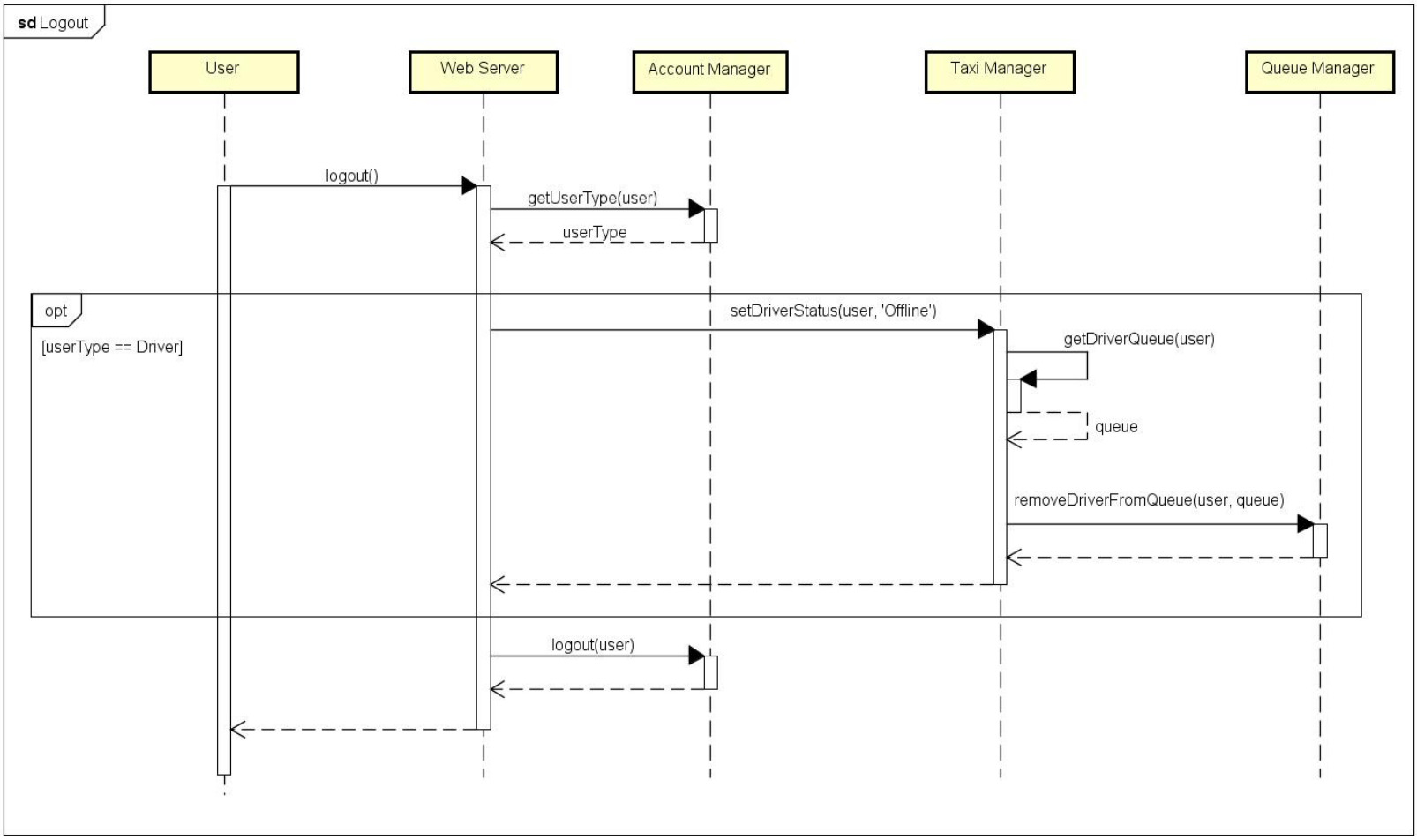




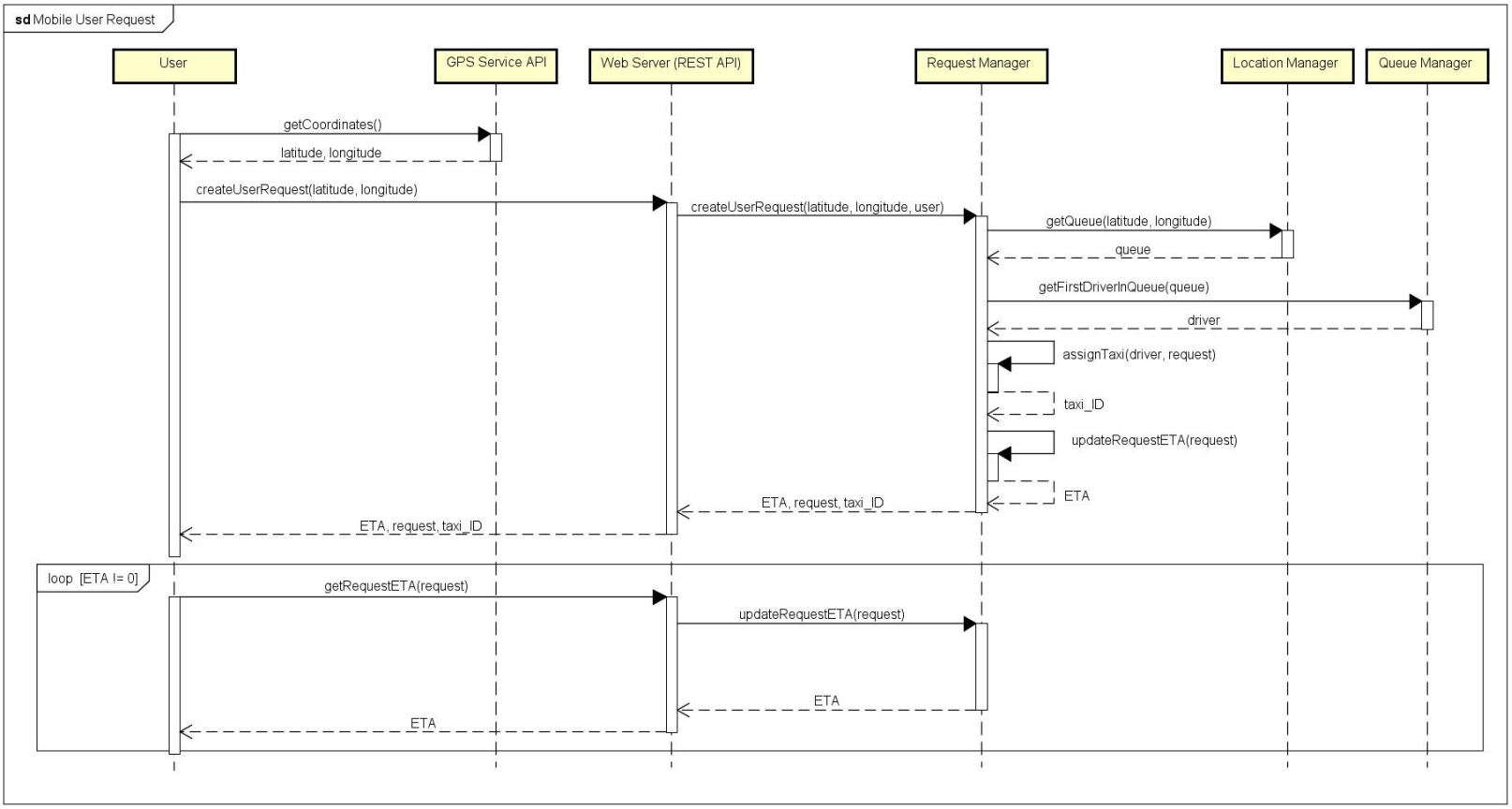
2.5.3 Mobile Passenger Signup



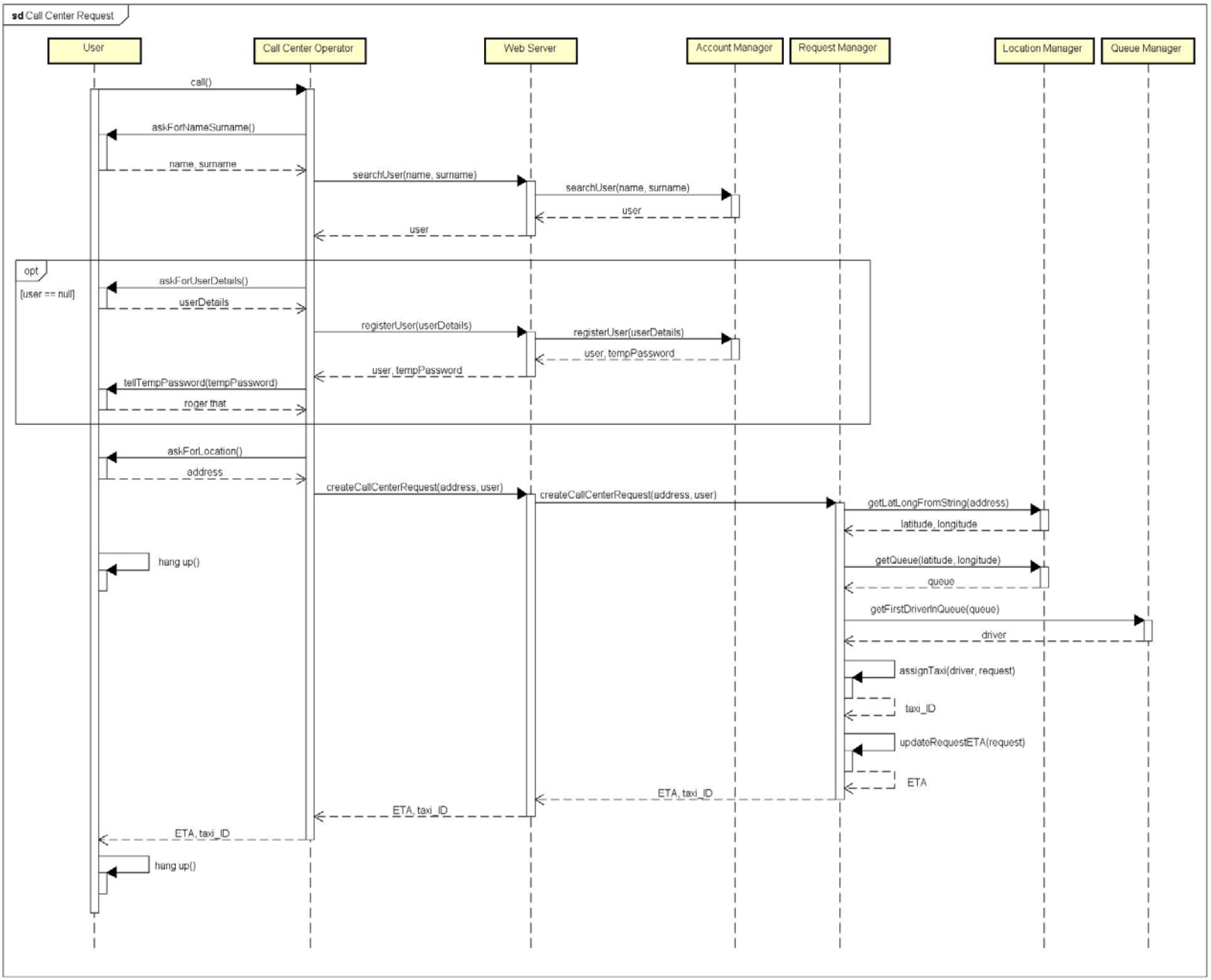
2.5.4 Logout


sd Logout
User
Web Server
Account Manager
Taxi Manager
Queue Manager
logout()
getUserType(user)
userType
opt
[userType == Driver]
setDriverStatus(user, 'Offline')
getDriverQueue(user)
queue
removeDriverFromQueue(user, queue)
logout(user)

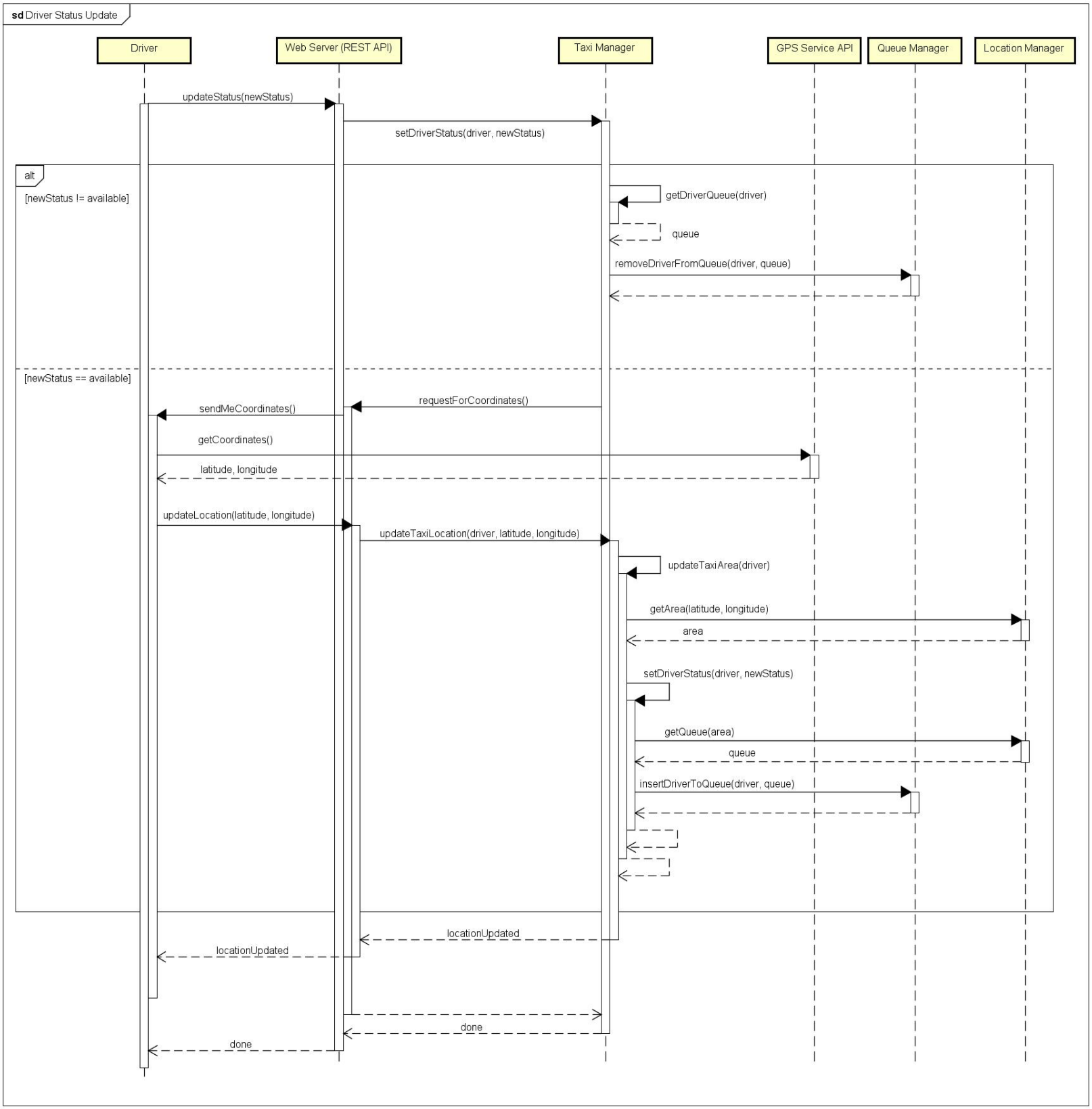
2.5.5 Mobile User Taxi Request



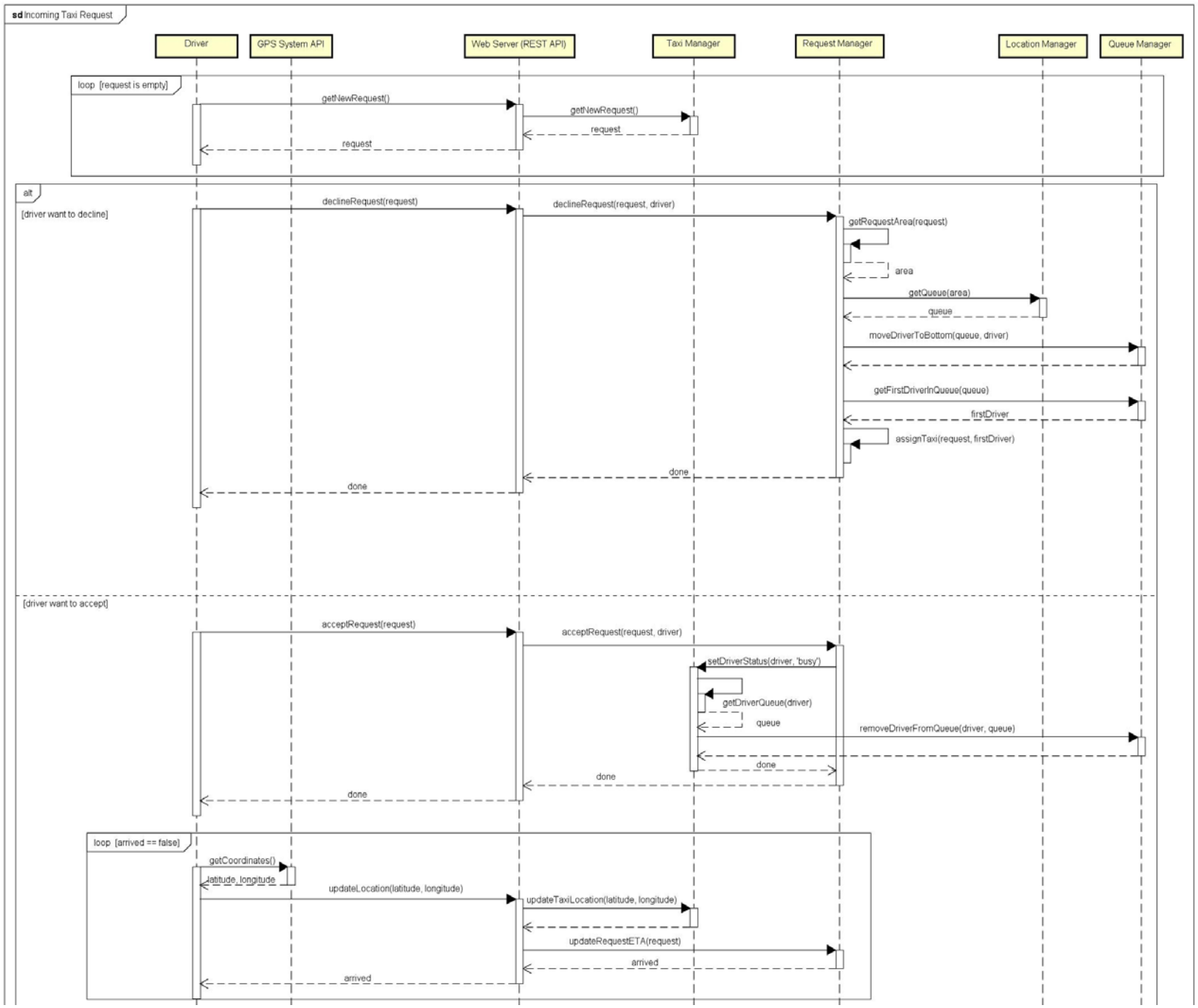
2.5.6 Call Center Request

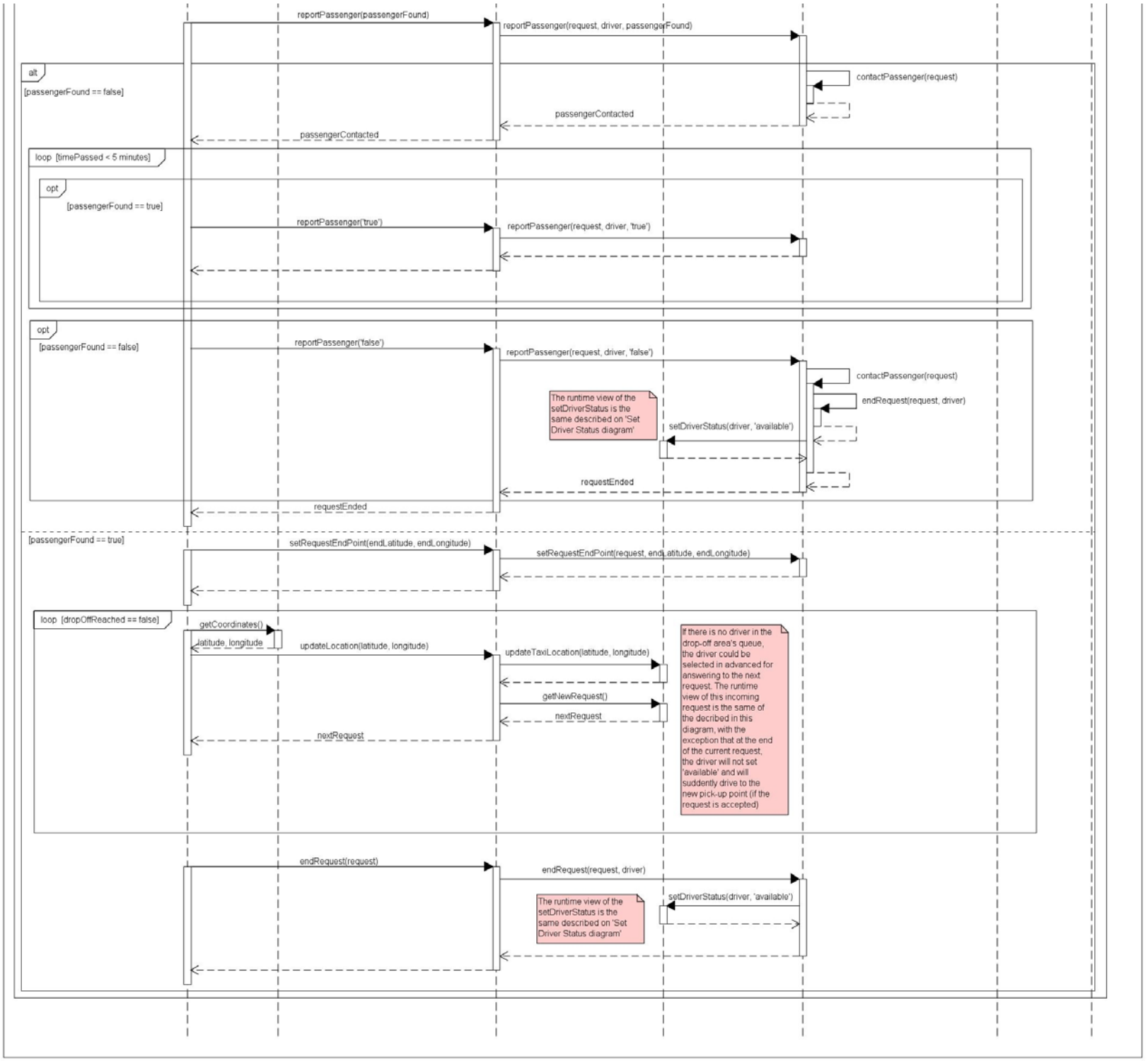


2.5.7 Driver Status Update

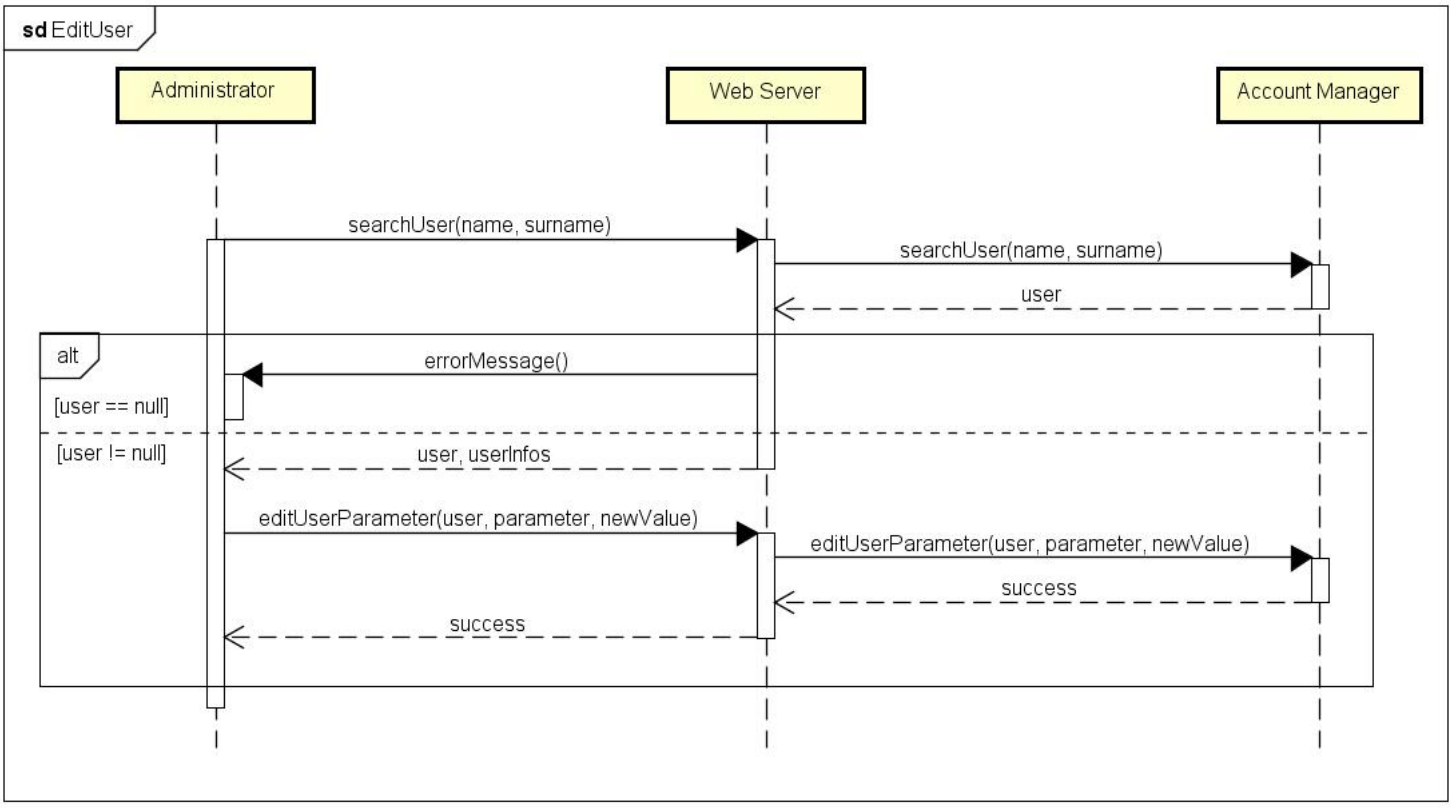


2.5.8 Incoming Taxi Request

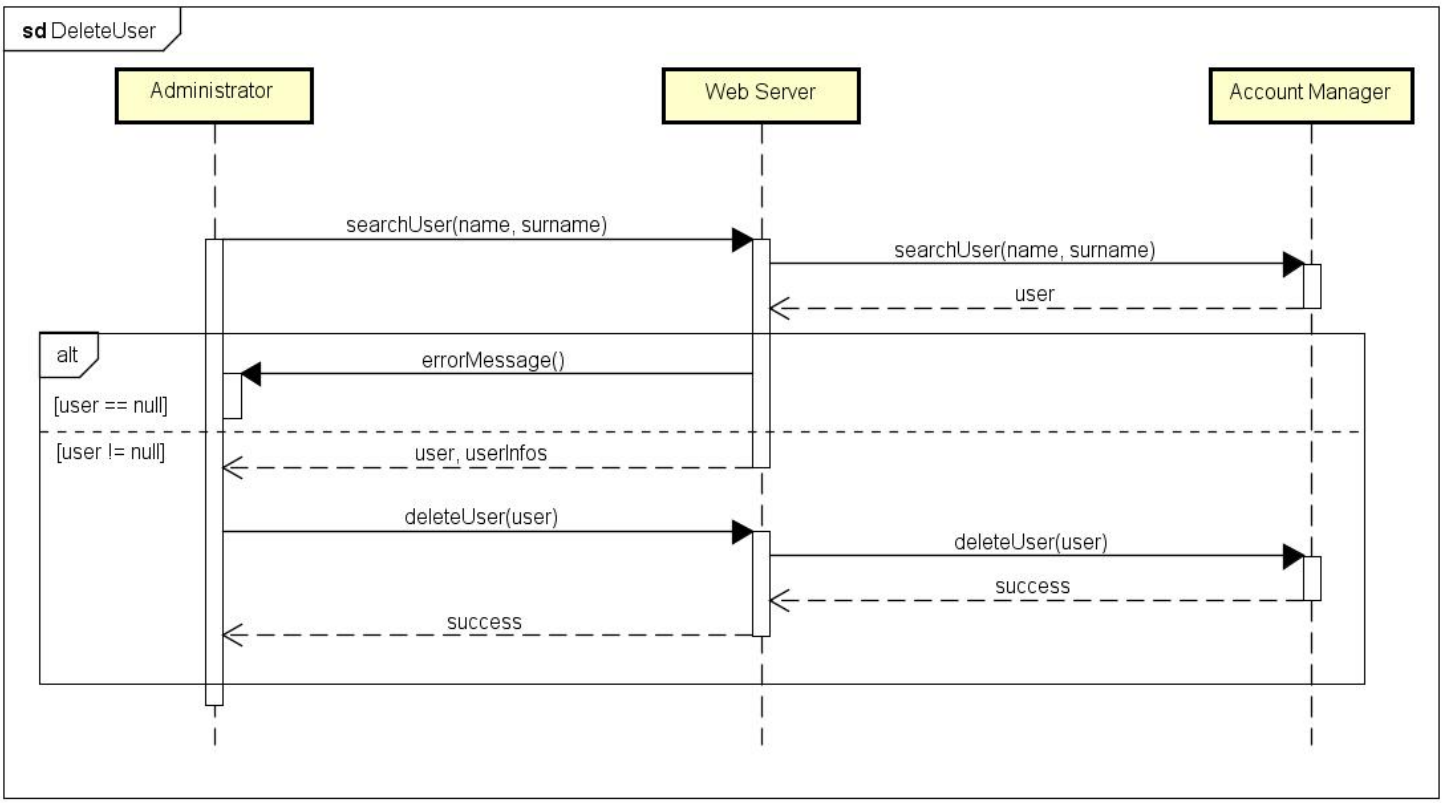



reportPassenger(passengerFound)
reportPassenger(request, driver, passengerFound)
alt
[passengerFound == false]
contactPassenger(request)
passengerContacted
passengerContacted
loop [timePassed < 5 minutes]
opt
[passengerFound == true]
reportPassenger('true')
reportPassenger(request, driver, 'true')
opt
[passengerFound == false]
reportPassenger('false')
reportPassenger(request, driver, 'false')
contactPassenger(request)
endRequest(request, driver)
The runtime view of the setDriverStatus is the same described on 'Set Driver Status diagram'
setDriverStatus(driver, 'available')
requestEnded
requestEnded
[passengerFound == true]
setRequestEndPoint(endLatitude, endLongitude)
setRequestEndPoint(request, endLatitude, endLongitude)
loop [dropOffReached == false]
getCoordinates()
latitude, longitude
updateLocation(latitude, longitude)
updateTaxiLocation(latitude, longitude)
If there is no driver in the drop-off area's queue, the driver could be selected in advanced for answering to the next request. The runtime view of this incoming request is the same of the decribed in this diagram, with the exception that at the end of the current request, the driver will not set 'available' and will suddently drive to the new pick-up point (if the request is accepted)
getNewRequest()
nextRequest
nextRequest
endRequest(request)
endRequest(request, driver)
The runtime view of the setDriverStatus is the same described on 'Set Driver Status diagram'
setDriverStatus(driver, 'available')

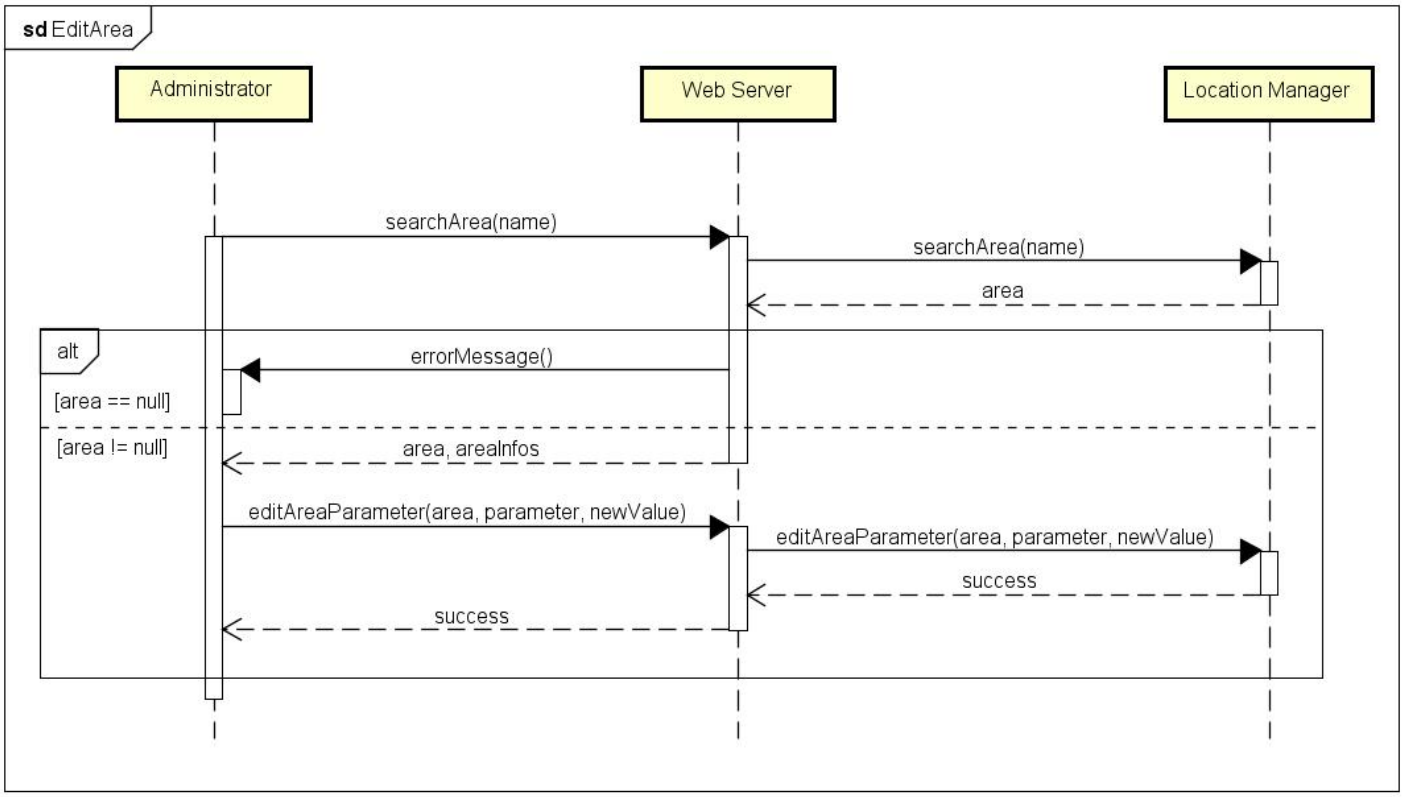
2.5.9 Edit User



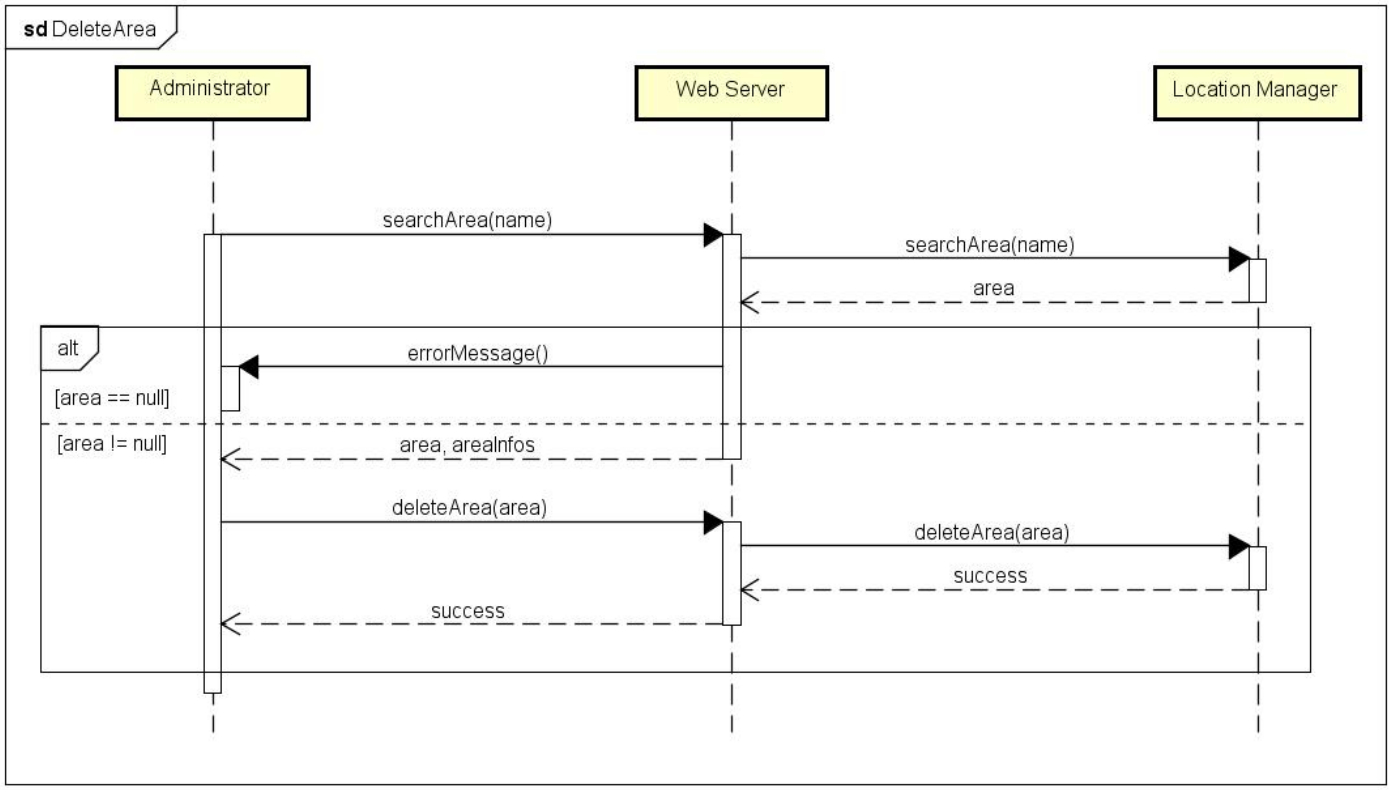
2.5.10 Delete User



2.5.11 Edit Area



2.5.12 Delete Area



2.6 Component Interfaces

2.6.1 Account Manager

```
interface AccountManager {
        public boolean login(String email, String password);
        public boolean logout();
        public boolean registerUser(ArrayList<String> parameters);
        public boolean editUserParameter(User user, String parameter, String
newValue);
        public String getUserType(User user);
        public boolean emailExists(String email);
        public boolean userExists(ArrayList<String> parameters);
        public User searchUser(String name, String surname);
        public boolean setUserLevel(User user, String level);
        public boolean deleteUser(User user);
        public boolean sendOTP(User user);
        public boolean verifyOTP(String code, User user);
}
```

This is a possible implementation of the Java Interface of the Account Manager component, which offers these functionalities:

- login: this method will try to login an user into the system, verifying the given combination of email and password. This function will return true if email and password match with the ones inside the databases, false otherwise.
- logout: this method will logout an user from the system.
- registerUser: this method will register an user inside the system, using the parameters given to the function (such as email, name, surname and others). The returned value corresponds to the success of the operation.
- editUserParameter: this method will edit a certain parameter of a given user. The returned value corresponds to the success of the operation. Only an administrator can invoke this method.
- getUserType: this method will return the type of a given user (Passenger, Administrator, Driver or Call Center Operator).
- emailExists: this method will return if an email is already registered in the system.
- userExists: this method will return if an user is already registered in the system.
- searchUser: this method will search if the user with the given name and surname exists, and return its associated data object (User).
- setUserLevel: this method will set the level of “User” to “level”. It is used by an administrator to modify the user type (Driver, Call Center Operator, Normal User). Only an administrator can invoke this method.
- deleteUser: this method will be used by an administrator and deletes the given user from the database. Only an administrator can invoke this method.
- sendOTP: this method will be invoked on the administrator login and will send to his/her mobile phone a code through SMS, that will be used to verify his/her identity.
- verifyOTP: this method will verify the second factor of authentication provided on the administration login of a given user.

2.6.2 Request Manager

```
interface RequestManager {
        public boolean createUserRequest(Double latitude, Double longitude, User
user);
        public boolean createCallCenterRequest(String address, User user);
        public String getRequestStatus(Request request);
        public User getRequestUser(Request request);
        public Driver getRequestDriver(Request request);
        public int assignTaxi(Request request, Driver driver);
        public void declineRequest(Request request, Driver driver);
        public void acceptRequest(Request request, Driver driver);
        public void reportPassenger(Request request, Driver driver, boolean
passengerFound);
        public void reportExceptionForRequest(Request request, Driver driver,
String reason);
        public int getRequestETA(Request request);
        public Date updateRequestETA(Request request);
        public void endRequest(Request request, Driver driver);
        public Area getRequestArea(Request request);
        public void setRequestArea(Area area, Request request);
        public void contactPassenger(Request request);
        public void setRequestEndPoint(Request request, Double latitude, Double
longitude);
}
```

This is a possible implementation of the Java Interface of the Request Manager component, which offers these functionalities:

- createUserRequest: this method will create a Taxi Request for a user. The pickup point is passed as latitude and longitude coordinates. The returned value corresponds to the success of the operation.
- createCallCenterRequest: this method will create a Taxi Request for a user from a Call Center Operator.
- getRequestStatus: this method will return the status of a given request.
- getRequestUser: this method will return the User data object associated to a request.
- getRequestDriver: this method will return the Driver data object associated to a request.
- assignTaxi: this method will assign a given taxi to a given request and return the taxi id.
- declineRequest: this method will make a given driver decline the given request, making the system assign the request to another driver.
- acceptRequest: this method will make a given driver accept the given request.
- reportPassenger: this method will report if the passenger of a given request is found or not by the driver.
- reportExceptionForRequest: this method will insert inside the system a report an exceptional event that make the taxi driver unable to complete the ride or getting to the pick-up point of a request. The invocation of this method will force the system to assign the request to another driver.
- getRequestETA: this method will return the estimated time in which the driver gets to the pick-up point of a given request.
- updateRequestETA: this method will update the estimated time in which the driver gets to the pick-up point of a given request and return it.
- getRequestArea: this method will return the area object assigned to the request.
- setRequestArea: this method will set the area associated to a given request.
- contactPassenger: this method will contact a passenger of a given request, that is not found by the driver, with an sms. It will be invoked when a passenger is reported as not found.
- setRequestEndPoint: this method will set the request drop-off location coordinates, used for the estimation of the end of the request.

2.6.3 Queue Manager

```
interface QueueManager {
        public void insertDriverToQueue(Queue queue, Driver driver);
        public void removeDriverFromQueue(Queue queue, Driver driver);
        public Driver getFirstDriverInQueue(Queue queue);
        public void moveDriverToBottom(Queue queue, Driver driver);
        public boolean isDriverInQueue(Queue queue, Drier driver);
        public int getPositionOfDriver(Driver driver);
        public Area getQueueArea(Queue queue);
}
```

This is a possible implementation of the Java Interface of the Queue Manager component, which offers these functionalities:

- insertDriverToQueue: this method will insert a given driver to a given queue.
- removeDriverFromQueue: this method will remove a given driver from a given queue.
- getFirstDriverInQueue: this method will return the first driver of a given queue.
- moveDriverToBottom: this method will move a given driver to the bottom of its queue.
- isDriverInQueue: this method will return if a given driver is inside a given queue.
- getPositionOfDriver: this method will return the queue position of a given driver.
- getQueueArea: this method will return the Area data object associated to a given queue.

2.6.4 Location Manager

```
interface LocationManager {
        public Area getArea(Double latitude, Double longitude);
        public Area searchArea(String name);
        public boolean createArea(String name, Double[] coordinates);
        public boolean deleteArea(Area area);
        public boolean editAreaParameter(Area area, String parameter, String
newValue);
        public Queue getQueue(Area area);
        public Queue getQueue(Double latitude, Double longitude);
        public ArrayList<Dobule> getLatLongFromString(String string);
        public Date getEstimatedTimeToLocation(Double fromLatitude, Double
fromLongitude, Double toLatitude, Double toLongitude);
}
```

This is a possible implementation of the Java Interface of the Location Manager component, which offers these functionalities:

- getArea: this method will return the area that corresponds to a given combination of coordinates.
- searchArea: this method will return the area with a given name.
- createArea: this method will be invoked by the administrator and insert a new area inside the database. Only an administrator can invoke this method.
- deleteArea: this method will be invoked by the administrator and removes an area from the database. Only an administrator can invoke this method.
- editAreaParamenter: this method will be invoked by the administrator and modify a given parameter of the given Area inside the database. Only an administrator can invoke this method.
- getQueue: this method will return the queue associated to a given area or to a given combination of coordinates.

- getLatLongFromString: this method will return the latitude and longitude of a given street address.
- getEstimatedTimeToLocation: this method will return the estimated time to get from a location to another. This time is calculated with the coordinates passed to the method and with the help of the Goolge Maps API.

2.6.5 Taxi Manager

```
interface TaxiManager {
        public Request getNewRequest();
        public void updateTaxiArea (Driver driver);
        public void getDriverQueue(Driver driver);
        public ArrayList<Double> getTaxiLocation(Driver driver);
        public void updateTaxiLocation(Driver driver, Double latitude, Double
longitude);
        public void setDriverStatus(Driver driver, Status status);
        public Date getEstimatedTimeToRequestLocation(Drier driver, Request
request);
        public String getDailyStatistics(Driver driver);
        public String getDriverStatus(Driver driver);
}
```

This is a possible implementation of the Java Interface of the Taxi Manager component, which offers these functionalities:

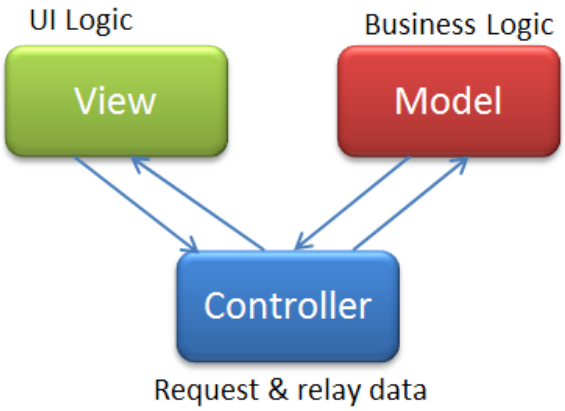
- getNewRequest: this method will return a new request for a driver, if a new one is present.
- updateTaxiArea: this method will update the taxi area based on the current location.
- getDriverQueue: this method will return the queue object associated to a driver.
- getTaxiLocation: this method will return the latitude and longitude of a given taxi.
- updateTaxiLocation: this method will update the coordinates representing the position of a given driver.
- setDriverStatus: this method will set the status of a given driver.
- getEstimatedTimeToRequestLocation: this method will return the estimated time for a given driver to get to a pick-up point of a given request.
- getDailyStatistics: this method will return the workday statistics of a given driver.
- getDriverStatus: this method will return the status of a given driver.

2.7 Selected architectural styles and patterns

MVC

Model View Controller (MVC) is a design pattern for successfully and efficiently relating the user interface to underlying data models.
The model-view-controller pattern proposes three main components or objects to be used in software development:

- Model: represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface.
- View: is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth).
- Controller: represents the classes connecting the model and the view, and is used to communicate between classes in the model and view.



Client-server

The client–server model of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.
A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function.
In this project an example of client-server style are:

- The Web Server that request an operation to the Application Server
- The Application Server that request some data operation to the Database Server
- The Browser that request a page to the Web Server

REST

REpresentational State Transfer (REST) is the software architectural style of the World Wide Web. REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher-performing and more maintainable architecture.
To the extent that systems conform to the constraints of REST they can be called RESTful. RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol (HTTP) with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers.
This type of style will be used in the Web Server for providing the Mobile API and will ensure:

- Better Scalability: will be possible to extend the system without redesigning it.
- Better Portability: will be possible to use the system on a various amount of environment, since the fact that all the functionalities are provided through HTTP with JSON or XML responses.

2.8 Other design decisions

Google Maps API

Creating a proprietary maps system is very time-consuming and expensive, so this project will use Google Maps API for handling geolocation data and routes.
Google Maps API will provide the best route to a specific location, considering the available traffic, providing also its estimated travel time.
This service will be available for the taxi drivers and will be also used from the system to handle the latitude and longitude data got from the mobile application users, taxi drivers and taxi zones.

Amazon EC2 with Elastic Load Balancing

Due to the large amount of taxi request estimated in the RASD and all the logged taxi drivers, users and call center operators, this project will use Amazon EC2 with Elastic Load Balancing enabled for each tier:

- Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. Amazon EC2 enables you to increase or decrease capacity within minutes, not hours or days. You can commission one, hundreds or even thousands of server instances simultaneously. Of course, because this is all controlled with web service APIs, your application can automatically scale itself up and down depending on its needs.
  - ➔ Using this technology this project will benefits of a better flexibility and the costs are directly related to the number of active instances, that can be easily increased or decreased thanks to the cloud-base architecture.
- Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances in the cloud. It enables you to achieve greater levels of fault tolerance in your applications, seamlessly providing the required amount of load balancing capacity needed to distribute application traffic.
  - ➔ Using this technology this project will benefits of a better availability thanks for the load balancing architecture that will automatically scale the needed server instances and will redirect the traffic to the less loaded instance. If an instance failure is identified, it will be automatically replaced with a new one by the load balancer, increasing the reliability.

InnoDB as storage engine for MySQL

InnoDB as storage engine for MySQL supports:

- Transactions
- Row-level locking, having a more fine-grained locking-mechanism that will result in a higher level of concurrency.
- Large buffer pool for both data and indexes

It also is more resistant to table corruption than other storage engine, for example like MyISAM.

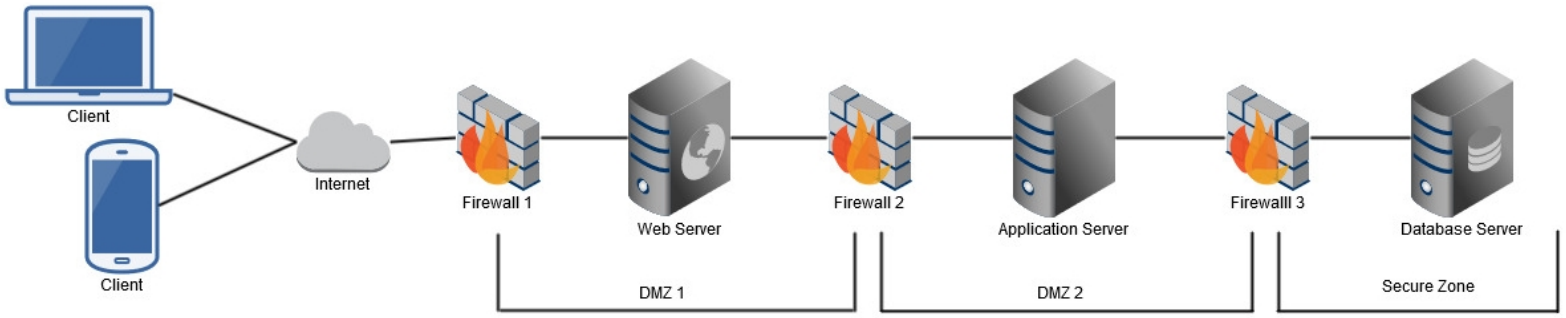
Two Factor Authentication for Administrator login

In order to have a good level of security and prevent non-authorized access to the administration panel, a code will be required to successfully complete the login phase for an administrator.
This code will be sent to the administrator mobile phone as a text message and will be valid only for 30 seconds.

Firewalls and DMZ

To accomplish a high level of security, there is the need of a firewall between each tier, and must be configured as follow:

- The firewall #3 (between Database Server and Application Server) must allow only the communication between the Application Server and the Database Server
- The firewall #2 (between Application Server and Web Server) must allow only the communication between the Web Server and the Application Server
- The firewall #1 (between Web Server and Clients) must allow the communication on the TCP port 80 and 443, that are required for the HTTP and HTTPS protocol.
- Is important that the policy of each firewall must be on a "DEFAULT DENY" base, allowing the needed communication with a properly designed rule.



Hashing Function and Salt for Password

Passwords are secrets. There is no reason to decrypt them under any circumstances. Helpdesk staff should be able to set new passwords (with an audit trail, obviously), not read back old passwords. Therefore, there is no reason to store passwords in a reversible form, so a hashing function is perfect for this purpose.

A Hash function creates a fixed length small fingerprint (or message digest) from an unlimited input string.

If the password's digest is stored in a database, an attacker should be unable to recover the password thanks to the preimage resistance. The only way to go past this would be a brute force attack, i.e. computing the hash of all possible passwords or a dictionary attack, i.e. computing all the often used password.

If each password is simply hashed, identical passwords will have the same hash. There are two drawbacks to choosing to only storing the password's hash:

- Due to the birthday paradox (http://en.wikipedia.org/wiki/Birthday_paradox), the attacker can find a password very quickly especially if the number of passwords the database is large.
- An attacker can use a list of precomputed hashes (http://en.wikipedia.org/wiki/Rainbow_table) to break passwords in seconds.

In order to solve these problems, a salt can be concatenated to the password before the digest operation.

A salt is a random number of a fixed length. This salt must be different for each stored entry. It must be stored as clear text next to the hashed password.

In this configuration, an attacker must handle a brute force attack on each individual password. The database is now birthday attack/rainbow crack resistant.

A 64 bits salt is recommended in RSA PKCS5 standard.

To slow down the computation it is recommended to iterate the hash operation n times. While hashing the password n times does slow down hashing for both attackers and typical users, typical users doesn't really notice it being that hashing is such a small percentage of their total time interacting with the system.

On the other hand, an attacker trying to crack passwords spends nearly 100% of their time hashing so hashing n times gives the appearance of slowing the attacker down by a factor of n while not noticeably affecting the typical user. A minimum of 1000 operations is recommended in RSA PKCS5 standard.
The stored password will look like this: *hash*(*hash*(*hash*(*hash*(…….*hash*(password||salt))))))))))))))
To authenticate a user, the operation same as above must be performed, followed by a comparison of the two hashes.
The hash function you need to use depends of your security policy. SHA-256 or SHA-512 is recommended for long term storage.

Polling Requests from Clients

The client of the driver will continuously send a request to the web server in order to update its position and the response to this update will contain a ride request, if present for this driver. Also, when a request is active for the driver, his/her client will continue to send those requests in order to update the request ETA.
When a passenger makes a request for a ride, his/her client (both mobile or web one) will continuously send a request in order to get the updated ETA for the taxi arrival.
The "continuously" request are made every 1-3 seconds, accordingly to the web server congestion. Each request is made only after receiving the response of the previous one. If no response arrives in less than 3 seconds, the request will be sent again.

Asynchronous Methods

For each action that needs to be verified after a certain amount of time, there will be an asynchronous method that will be invoked and run a control of a certain parameters after a certain amount of time.
This will prevent the timeout error on the client-side: the web server sends the response of a request, without waiting for the asynchronous method completion (it is not locking the resource waiting for a response).
An example of this method is provided in the algorithm design section of this document, that implements the *assignTaxi* method of the *RequestManager* component.

3. Algorithm design

3.1 Password Hash and Salt

This algorithm is a part of the AccountManager component and will be used for generating the hash of a password. The password must be combined with a randomly generated salt. In the database must be stored both the salt and the generated hash (and not the plaintext password).

```
import java.security.MessageDigest;

public byte[] hashPW(String pw, byte[] salt) throws NoSuchAlgorithmException {
	// Use sha1 with multiple iterations for generating the final hash
	MessageDigest digest = MessageDigest.getInstance("SHA-1");
	digest.reset();
	digest.update(salt);
	byte[] input = digest.digest(pw.getBytes("UTF-8"));
	// 1000 iteration will slow down an attacker
	for (int i = 0; i < 1000; i++) {
	    digest.reset();
	    input = digest.digest(input);
	}
	return input;
}
```

3.2 Insert a driver to a queue

This algorithm is a part of the QueueManager component and will be used to insert a driver into a queue.
The algorithm must take into account if the driver is busy or not and if the driver is already present in a queue, in order to perform the insertion.

```
// Enum for the Status type
public enum Status {
	BUSY, AVAILABLE, OFFLINE;
}

public void insertDriverToQueue(Driver driver, Queue queue) {
	int driverID = driver.getID();
	int queueID = queue.getID();
	// If the driver is busy must not be added to the queue
	if(TaxiManager.getDriverStatus(driver)).equals(Status.BUSY)) {
		return;
	}
	// Add the driver to the queue if he/she is not in one
	if(TaxiManager.getDriverQueue(driver).isEmpty()) {
		DatabaseManager.insert('driver_queue', 'queue_id = '+queueID+',
driver_id = '+driverID);
	}
}
```

3.3 Update Driver Status

This algorithm is a part of the TaxiManager component and will be used for update the status of the driver. When updating a status, the algorithm will add the driver to a queue or remove it from, accordingly to its new status.

```
public void setDriverStatus(Driver driver, Status status) {
        int driverID = driver.getID();
        // If the new status is not available, the driver must be removed from
        // its queue
        if(!status.equals(Status.AVAILABLE)) {
                Queue queue = this.getDriverQueue(driver);
                if(!queue.isEmpty()) {
                        QueueManager.removeDriverFromQueue(driver, queue);
                }
        }
        else {
        // The driver will be inserted to the queue
                ArrayList<Double> coordinates = this.getTaxiLocation(driver);
                Double latitude = coordinates.get(0);
                Double longitude = coordinates.get(1);
                Queue queue = LocationManager.getQueue(latitude, longitude);
                QueueManager.insertDriverToQueue(driver, queue);
        }
        // Update driver status
        DatabaseManager.update('driver', 'status = '+status.toString(), 'driverID =
'+driverID);
}
```

3.4 Assign a Taxi to a request

When a request is created, with either createUserRequest or createCallcenterRequest, the area's queue of the request is selected. After that, the request is assigned to the first driver of the previously selected queue. If no driver is available in the given queue, this method will compute the ETA to get to the passenger for both the first driver of the nearest area's queue and for the first driver that will end a request in the request's area, and assign the request to the driver with the less ETA. All of this is achieved with the help of the Queue Manager and the Location Manager.

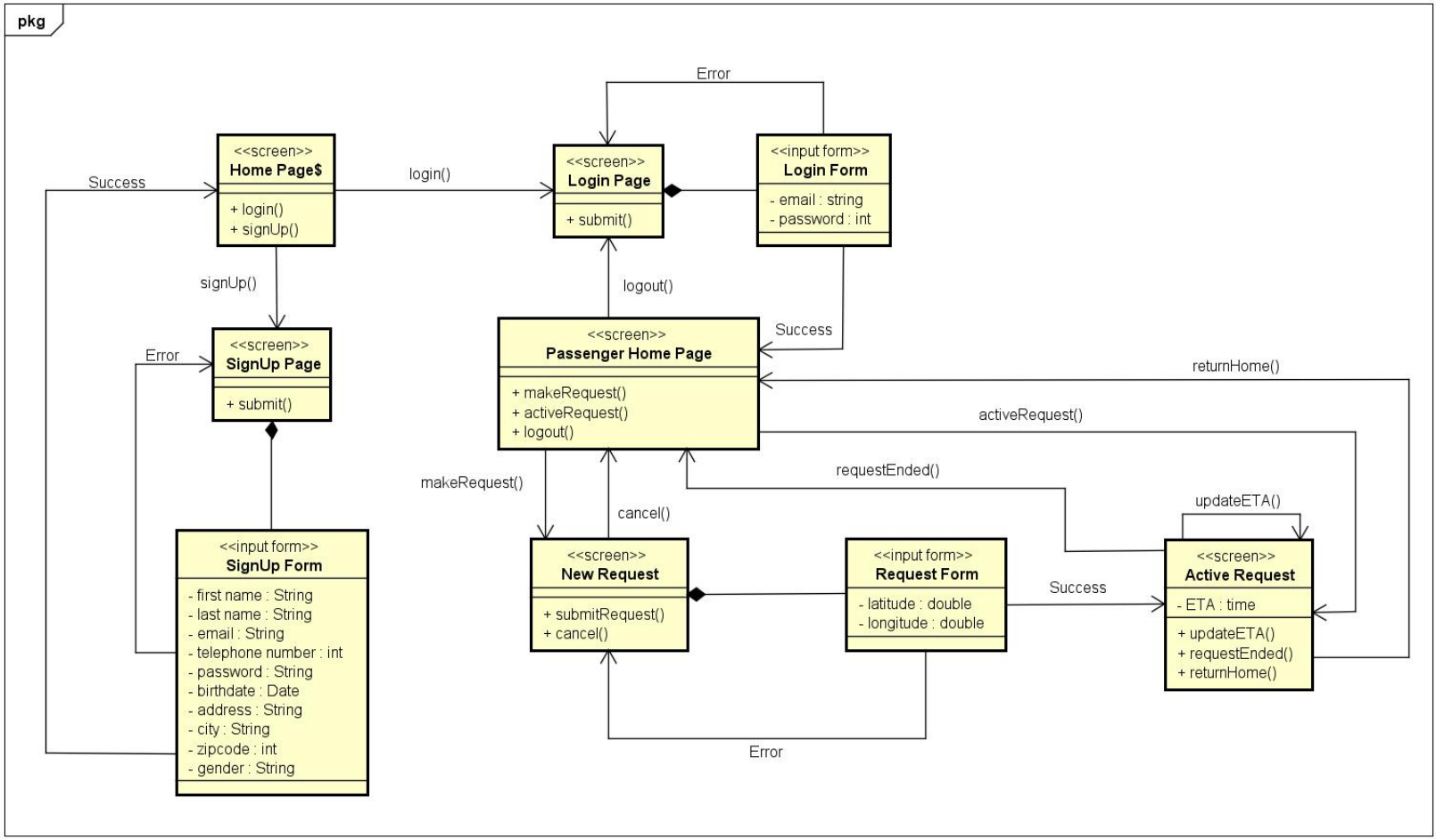
The following algorithm is a part of the RequestManager component and will be used for assign a request to a driver.
The component is composed by two different methods: the first (assignTaxi) will actually perform the assignment of the driver to a request, and the second (checkRequestAssignment) will check if the request is accepted or not after 30 seconds its assignment.

```
public void assignTaxi(Request request, Driver driver) {
        // If the driver is not available, cannot be assigned to a request
        // and the request must be assigned to a new driver
        if(!TaxiManager.getDriverStatus(driver).equals(Status.AVAILABLE)) {
                Queue queue = TaxiManager.getDriverQueue(driver);
                Driver newDriver = QueueManager.getFirstDriverInQueue(queue);
                this.assignTaxi(request, newDriver);
                return;
        }
        int driverID = driver.getID();
        int requestID = request.getID();
        DatabaseManager.update('request', 'driver = '+driverID, 'request =
'+requestID);
        // call the Async method for check the assignment after 30 sec
        this.checkRequestAssignment(request);

}


@Async
public Future<void> checkRequestAssignment(Request request) {
        try {
                Thread.sleep(30*1000); // 30 seconds
        } catch(InterruptedException ex) {
                Thread.currentThread().interrupt();
        }
        // If the request is not assigned, 30 seconds are passed so it must be
        // assigned to a new driver, and the assigned driver must be moved to the
        // bottom of the queue.
        if(!RequestManager.getRequestStatus(request).equals(RequestStatus.ACCEPTED)
) {
        Driver driver = RequestManager.getRequestDriver(request);
        Queue queue = TaxiManager.getDriverQueue(driver);
        QueueManager.moveDriverToBottom(queue, driver);
        driver = QueueManager.getFirstDriverInQueue(queue);
        RequestManager.assignTaxi(request, driver);
    }
}
```

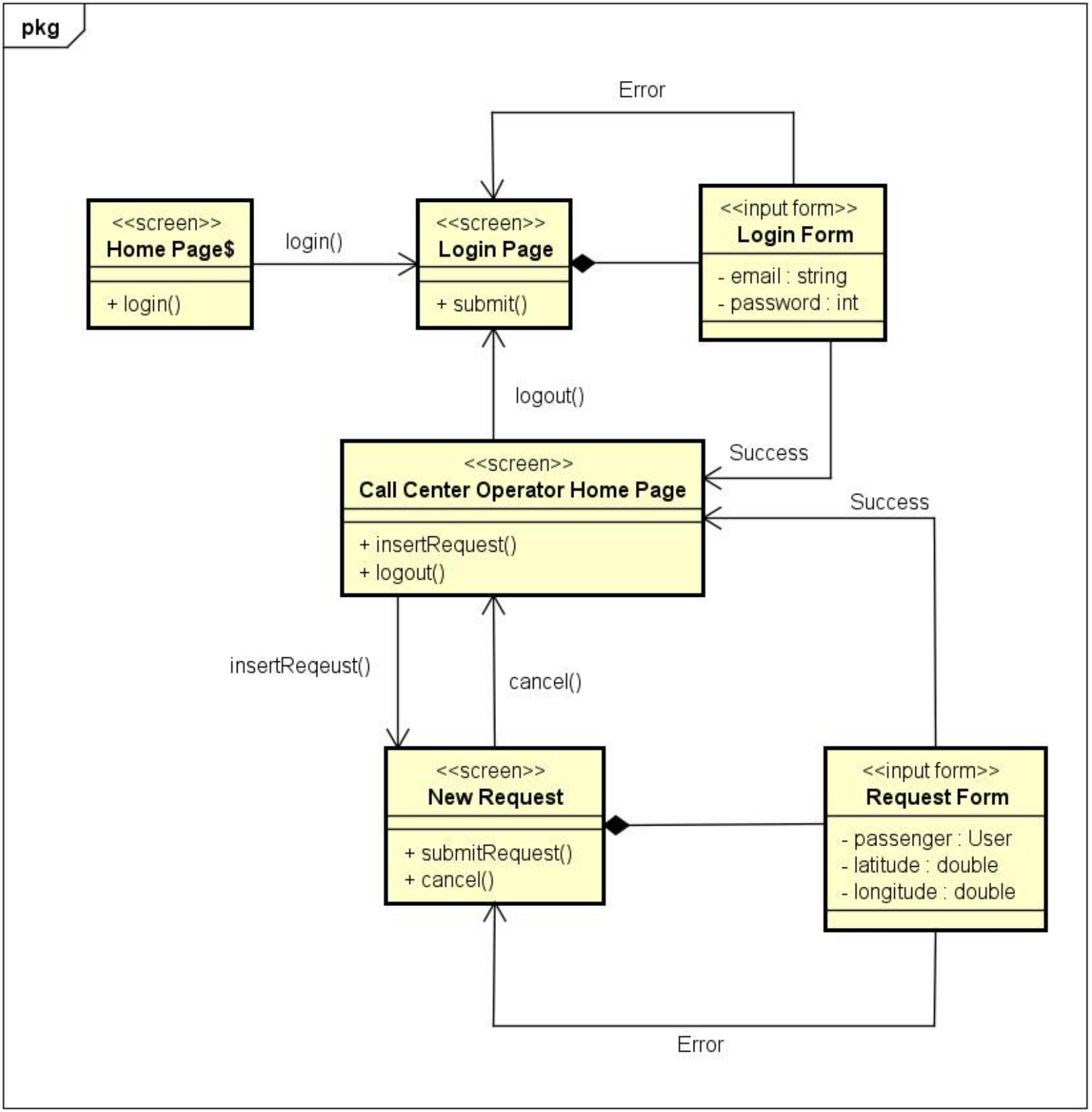
4. User interface design

The mockups of the user interface provided by both the web application and the mobile application are available into the RASD documentation provided, in the section 3.1.
In order to extend the provided mockups, here are provided the User eXperience models (UX) for each type of user that will interact with the system.

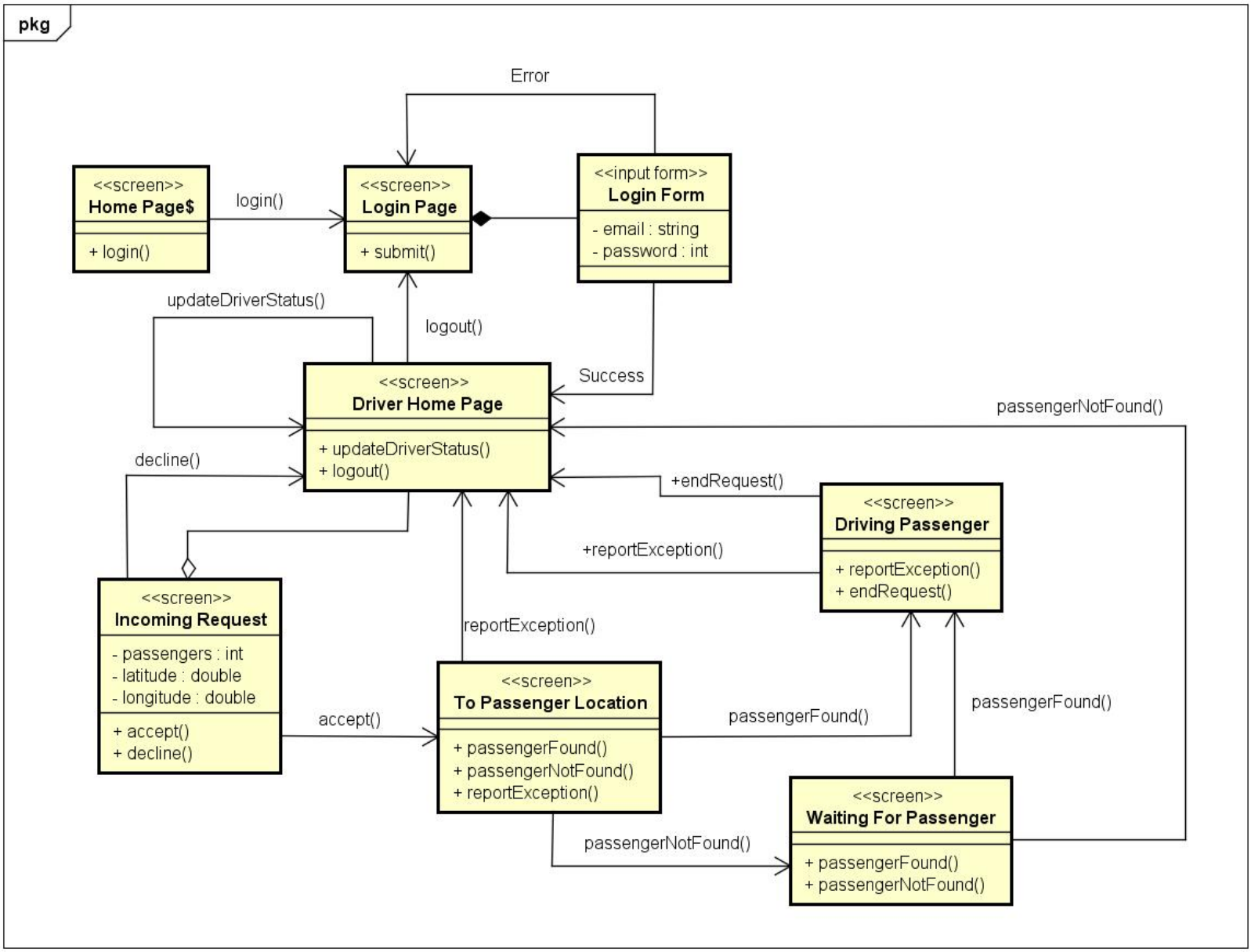
4.1 Passenger UX



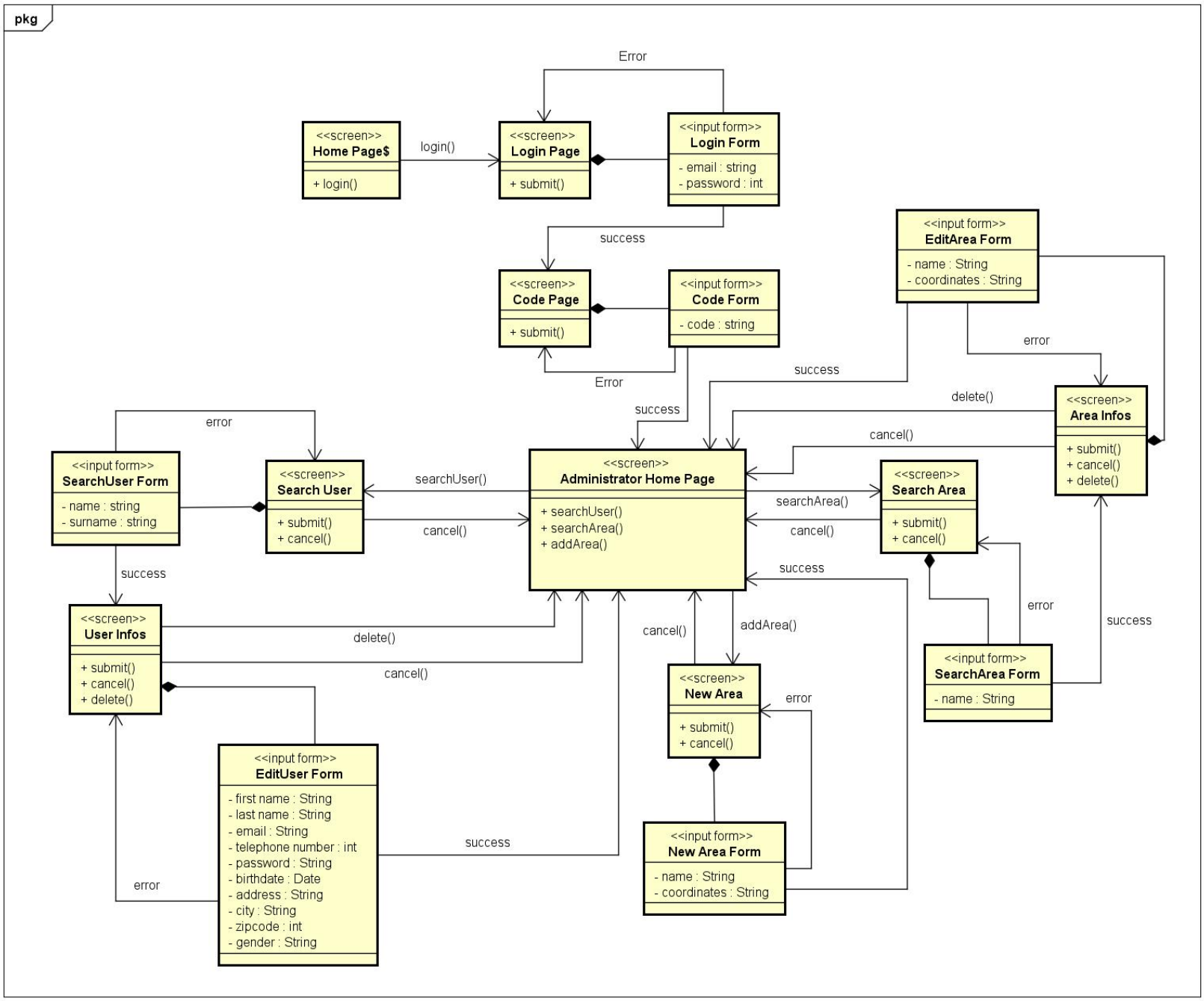
4.2 Call Center Operator UX



4.3 Driver UX


pkg
Error
<<screen>>
Home Page$
+ login()
login()
<<screen>>
Login Page
+ submit()
<<input form>>
Login Form
- email : string
- password : int
updateDriverStatus()
logout()
Success
<<screen>>
Driver Home Page
+ updateDriverStatus()
+ logout()
passengerNotFound()
decline()
+endRequest()
<<screen>>
Driving Passenger
+ reportException()
+ endRequest()
+reportException()
<<screen>>
Incoming Request
- passengers : int
- latitude : double
- longitude : double
+ accept()
+ decline()
reportException()
accept()
<<screen>>
To Passenger Location
+ passengerFound()
+ passengerNotFound()
+ reportException()
passengerFound()
passengerFound()
<<screen>>
Waiting For Passenger
+ passengerFound()
+ passengerNotFound()
passengerNotFound()

4.4 Administrator UX



5. Requirements Traceability

All the decisions made in this document are following the functional and non-functional requirements written in the RASD.

5.1 Functional Requirements Traceability

For each functional requirement specified in the RASD, in this list will be reported the component that will handle a required function. Is important to notice that the listed components are implementing the required logic, and the communication will be handled by the physical components of the system.

- **Guest can**:
  - Sign up: this action is handled by the *Account Manager*.

- **Passenger can**:
  - Login both on web application and mobile application: this action is handled by the *Account Manager*.
  - Modify his/her profile information: this action is handled by the *Account Manager*.
  - Request a taxi: this action is handled by the *Request Manager*.
  - Get the status of his/her request: this action is handled by the *Request Manager*.
  - Get the estimated ETA of his/her request: this action is handled by the *Request Manager* with the help of the *Location Manager*.
  - Logout from the application: this action is handled by the *Account Manager*.

- **Driver can**:
  - Login on the mobile application: this action is handled by the *Account Manager*.
  - Set his/her status as Available or Busy: this action is handled by the *Taxi Manager*.
  - See his/her queue position: this action is handled by the *Queue Manager.*
  - See his/her workday statistics: this action is handled by the *Taxi Manager*.
  - Receive taxi request: this action is handled by the *Request Manager*.
  - Accept or Decline a taxi request: this action is handled by the *Request Manager*.
  - Report if a passenger is found or not, after getting to the pick-up point: this action is handled by the *Request Manager*.
  - Report an exceptional event which prevent him/her to get to the pick-up point: this action is handled by the *Request Manager*.
  - Get the estimated arrival time to the pick-up point: this action is handled by the *Request Manager* with the help of the *Location Manager*.
  - Get indications on how to get to the pick-up point: this action is handled by the *Location Manager* and with the interaction with the *Google Maps API*.
  - Logout from the application: this action is handled by the *Account Manager*.

- **Call Center Operator can**:
  - Login on the web application: this action is handled by the *Account Manager*.
  - Receive calls for taxi requests: this action is possible thanks to the existing infrastructure of the Call Center.
  - Insert inside the system a taxi request: this action is handled by the *Request Manager*.
  - Get the status of the inserted request: this action is handled by the *Request Manager*.
  - Get the estimated ETA of the inserted request: this action is handled by the *Request Manager* with the help of the *Location Manager*.

    - Communicate request information to the client through the call: this action is possible thanks to the existing infrastructure of the Call Center.
    - Logout from the application: this action is handled by the *Account Manager*.

- **Administrator can**:
    - Login on the web application: this action is handled by the *Account Manager*.
    - Create, delete or modify an area: this action is handled by the Location Manager.
    - Modify or delete an user: this action is handled by the Account Manager.
    - Modify the user level (can be driver, call center operator): this action is handled by the Account Manager.

The system is not considered as an actor, but it must be able to:

- Assign a request for an area to the first taxi in the queue of this area: this action is handled by the *Request Manager* with the help of the *Queue Manager*.
- Organize drivers in queue: this action is handled by the *Queue Manager*.
- Assign exactly one queue for each area: this action is handled by the *Location Manager*.
- Automatically forward a request to the second driver in the queue if the first driver decline it: this action is handled by the *Request Manager.*
- Automatically forward a request to the second driver in the queue if the first driver doesn't accept or decline it within 30 seconds: this action is handled by the *Request Manager.*
- Assign a new driver to an accepted request if the assigned driver report an exceptional events that prevent him/her to get to the passenger: this action is handled by the *Request Manager.*
- If no driver is available in a queue, the system must be able to find the driver that will arrive to a fixed pick-up point in the less possible time. The driver can be one from another queue, or one that will finish a ride in the request's area: this action is handled by the *Request Manager* with the help of the *Queue Manager*.

5.2 Non-Functional Requirements Traceability

All the ***External Interface Requirements*** (RASD, 3.1) are respected:

- The design of the mobile application and web application is made to fulfill the User Interface Requirements.
- The software specified in the RASD (section 3.1.3) is the same that will be used to implement myTaxiService, as the port that will be used.

Performance Requirements, ***Availability Requirements*** and ***Maintainability Requirements*** are guaranteed by the cloud-computing based architecture, that is able to auto scale its performance up when needed more computation (and will auto scale its performance down when a lot of computational power is not needed, saving money for the company).

The ***Security Requirements*** are guaranteed by the designed network architecture and by the implementation of each component, that will filter inputs as required and will store the users password using its hash that will be salted (as described in the algorithm section of this documentation).

6. Appendix

6.1. Software and Tools used

- ***Microsoft Word 2013*** to redact and format this document.
    - Link: https://products.office.com/it-it/word/
- ***Astah Professional*** to create diagrams.
    - Link: http://astah.net/editions/professional/

6.2. Hours of Work

The redaction of the entire document took about 75 hours of work.

6.3. Revision

- **Revision 1:** initial document.
- **Revision 2:**
    - New policy for request assignment in the case of empty queue.
    - Removed cyclic dependency among software components.
    - Revised Web Server components and Application Server components.
- **Revision 3**:
    - Added actor “Administrator”.