

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.

Is also provided 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

[TODO]

1.4 Reference Documents

* Requirements Analysis and Specification Document

1.5 Document Structure

This document is structured in 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.
6. References: list the references used to redact this document.

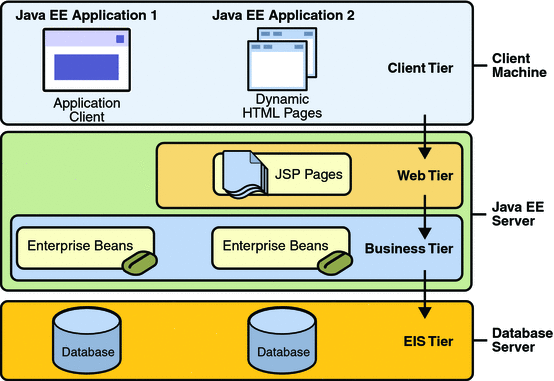
2. Architectural Design

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 through Remote Method Invocation (RMI).

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.

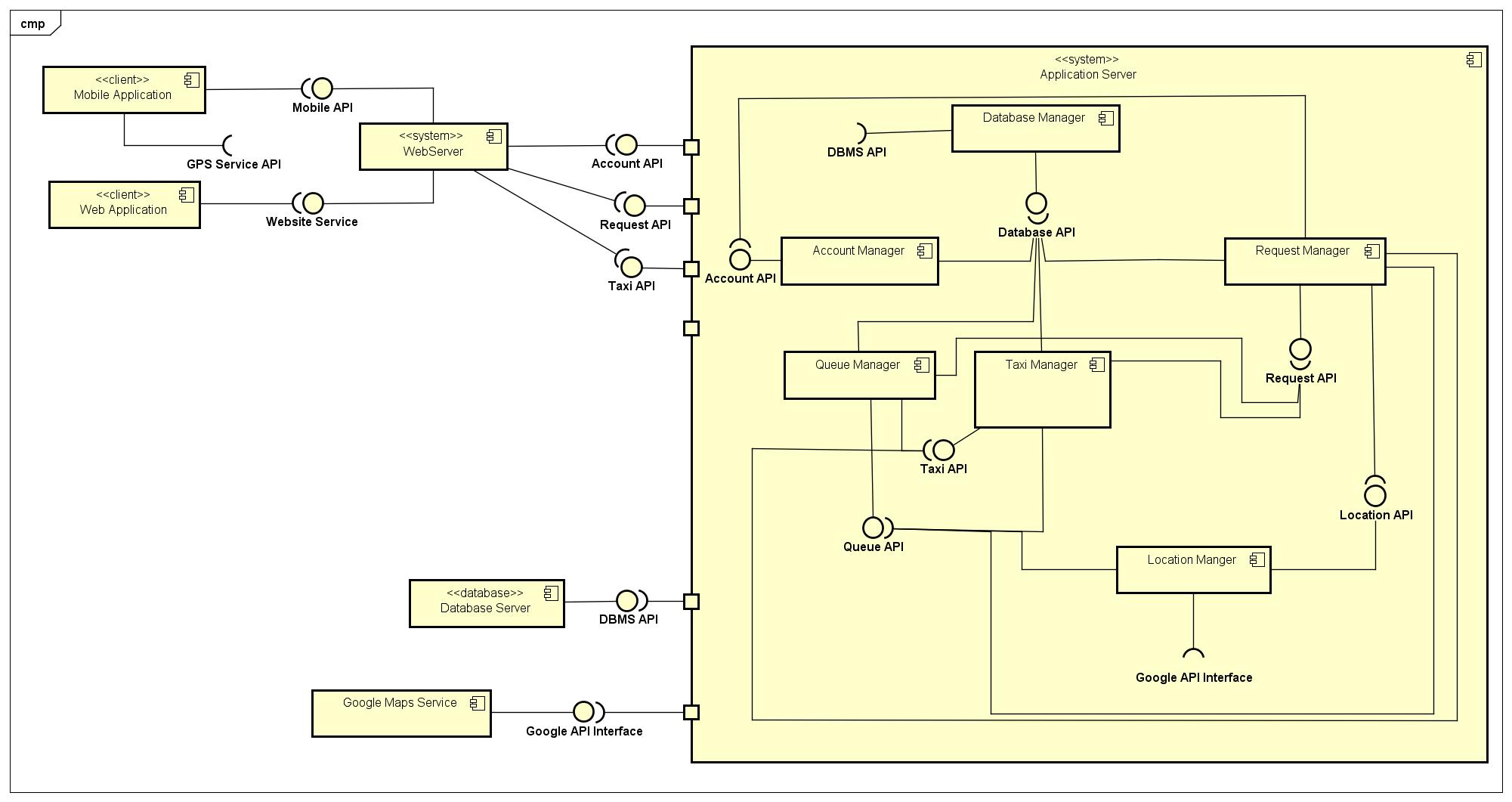
The database server will consists in MySQL Server as DBMS.

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 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, such as finding the associated queue 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, getting its location and more.
* Database Manager: this components provides all the functionalities required to interact with the database, maintaining an high level of abstraction.

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

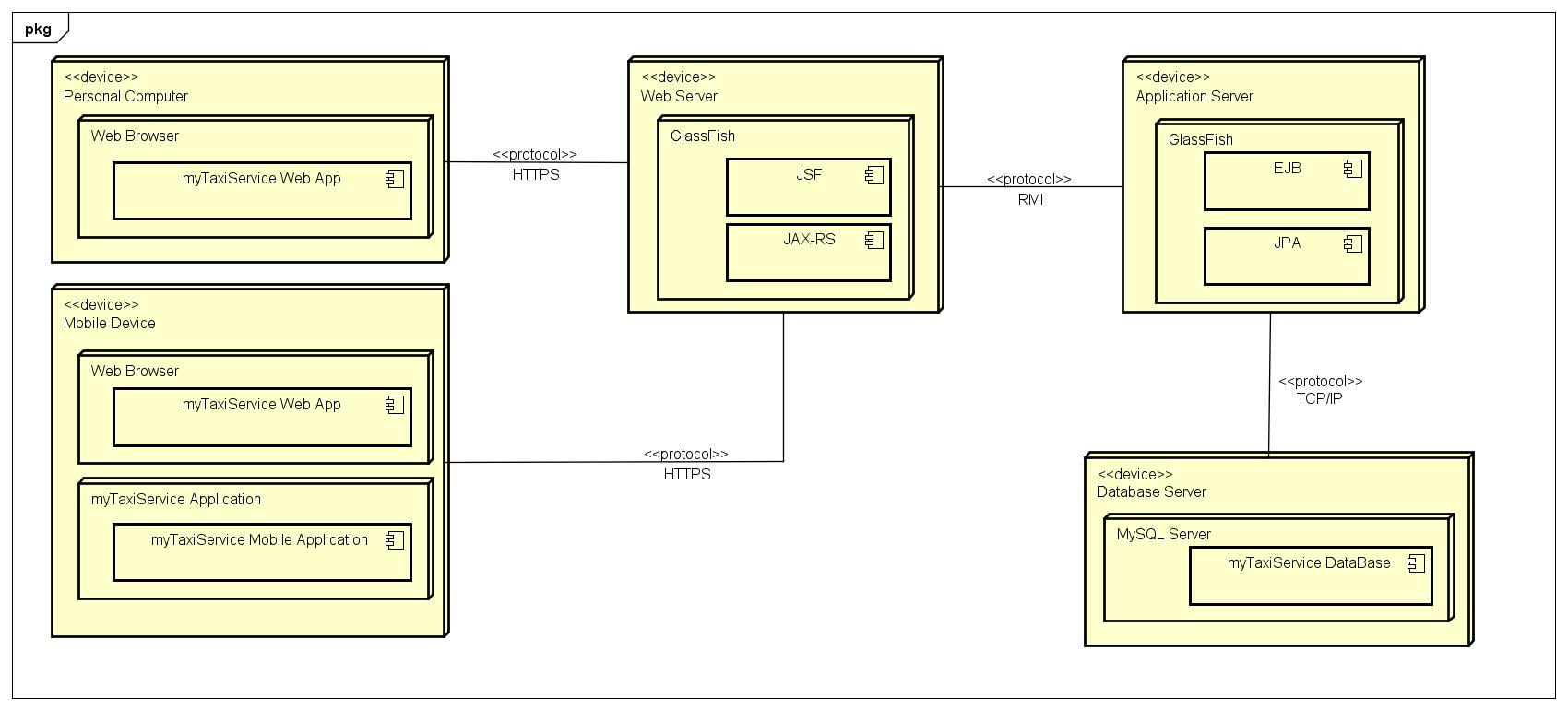
The Web Server Tier will be run using Glassfish Server and will be implemented using:

* 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.
  + 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.

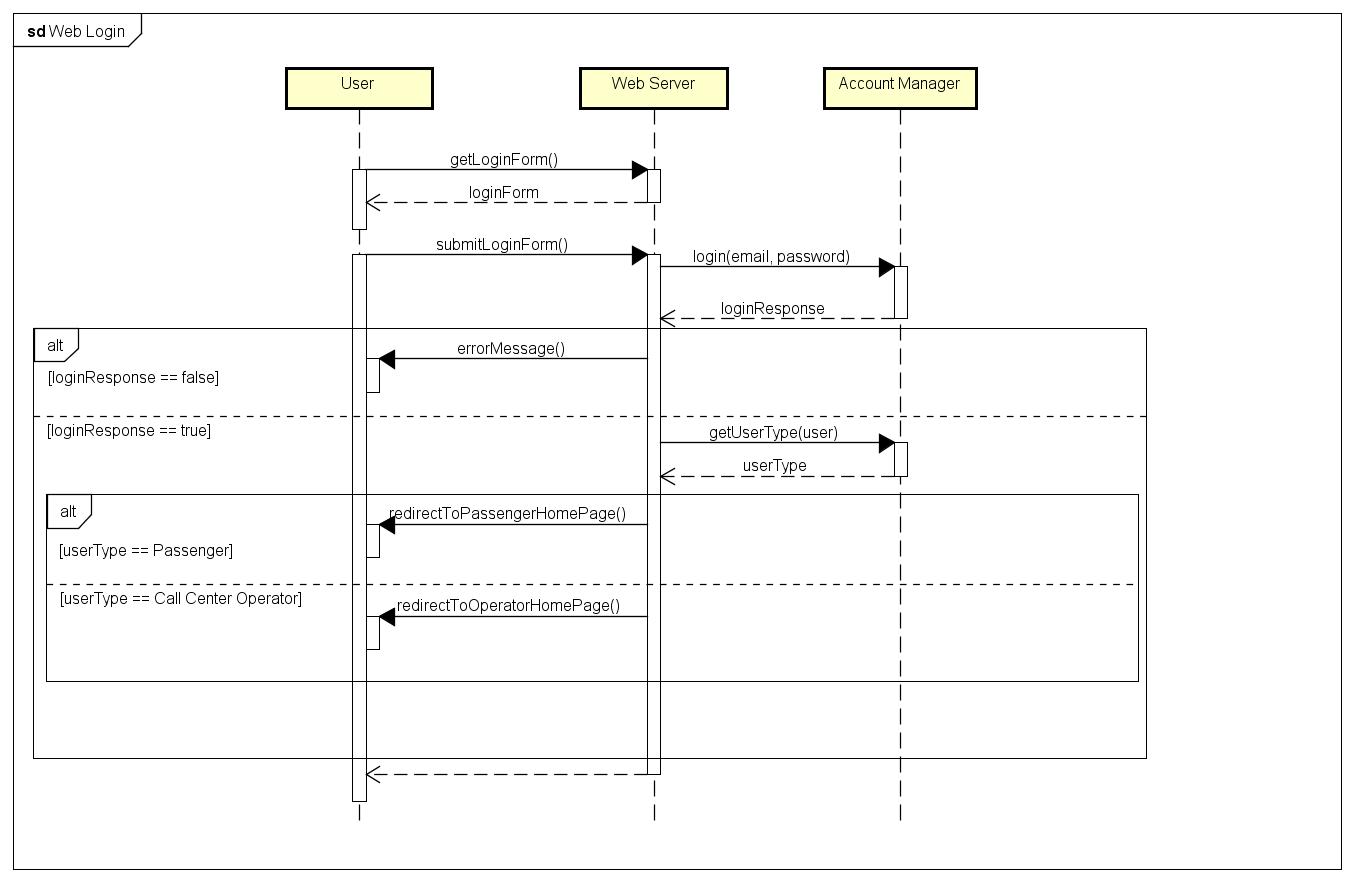
The Database Server Tier will be run using MySQL Server with the InnoDB storage engine.



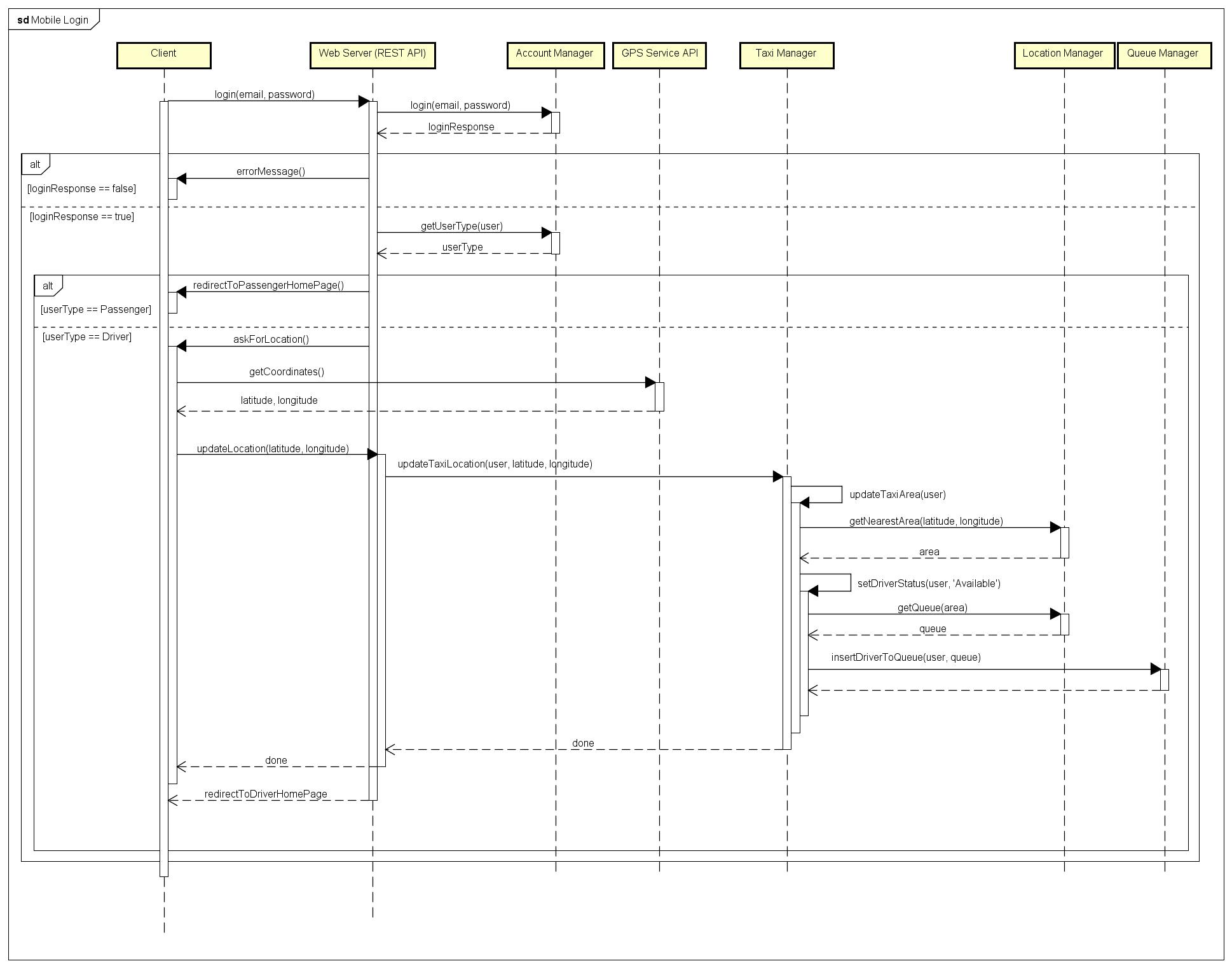
2.5 Runtime View

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.

Web Login



Mobile Login

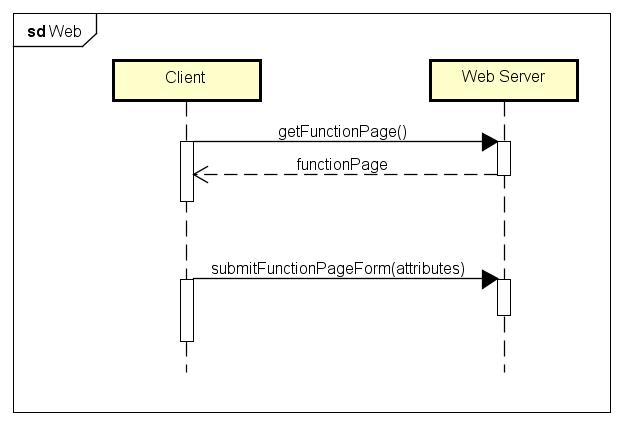
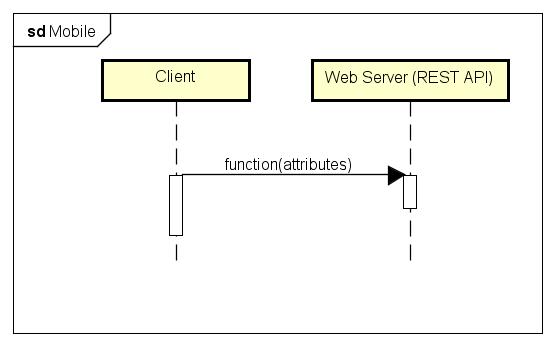


The only difference between the mobile application and the web application runtime view is the interaction with the webserver (it can be seen on the mobile and web login sequence diagram):

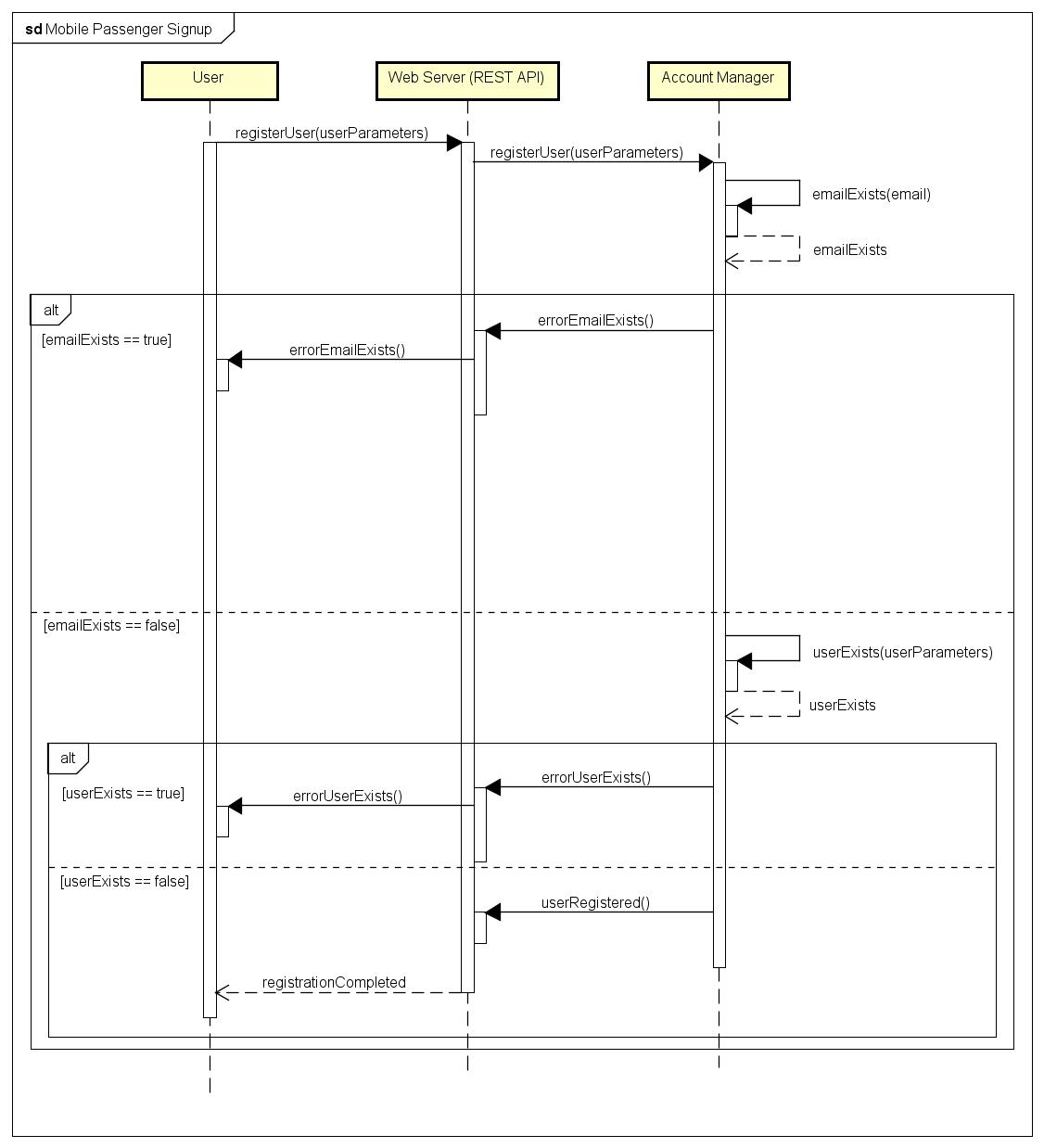
* The web application asks for a specific page and, once received it, send a POST request submitting a form.
* The mobile application skips asking for a page and directly send the request to the webserver, through the REST API provided.

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.

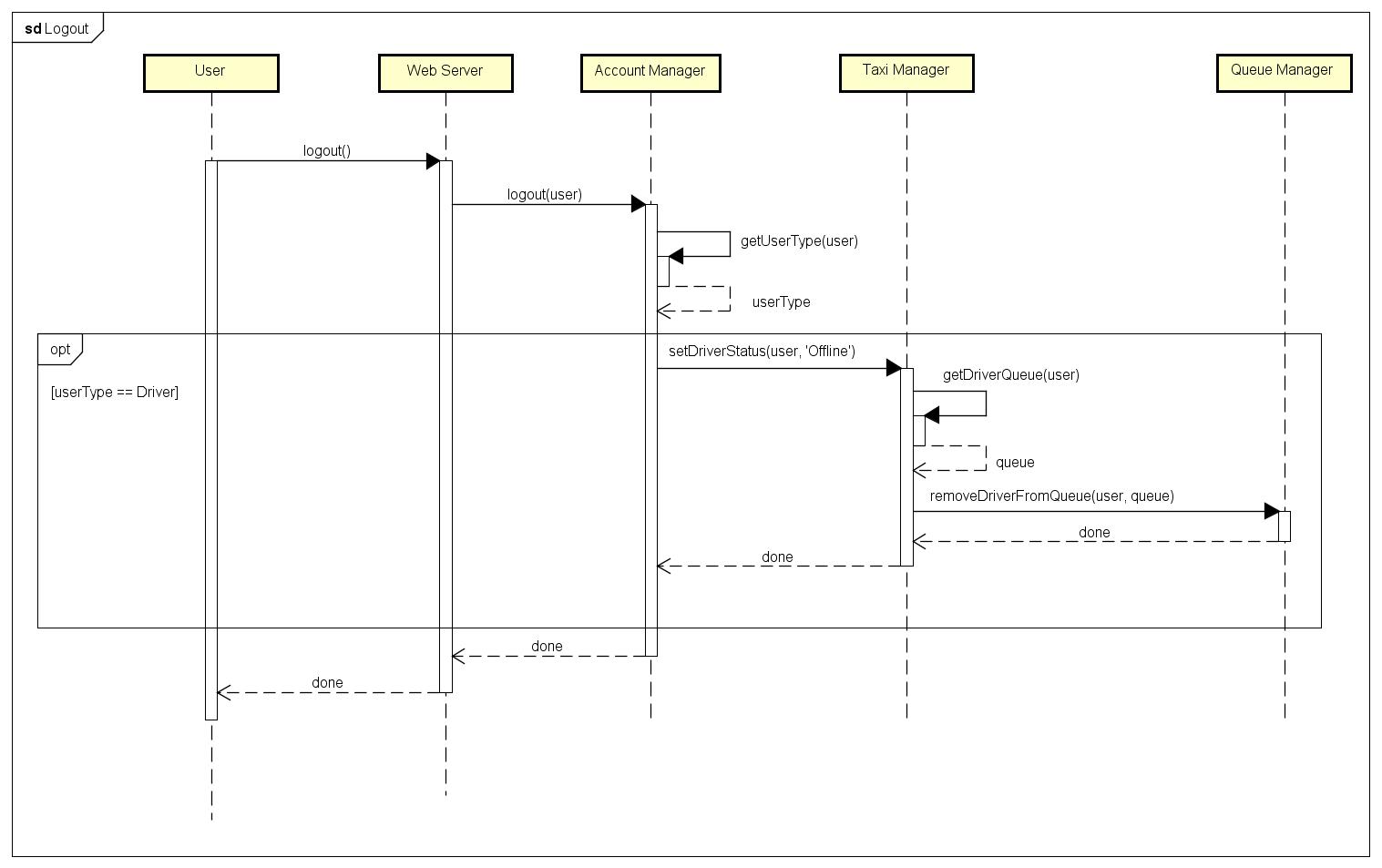
In order to consider the web application case of a given sequence diagram, for each request sent to the web server from a client must be considered the relative page request and wait for the response, as shown below.



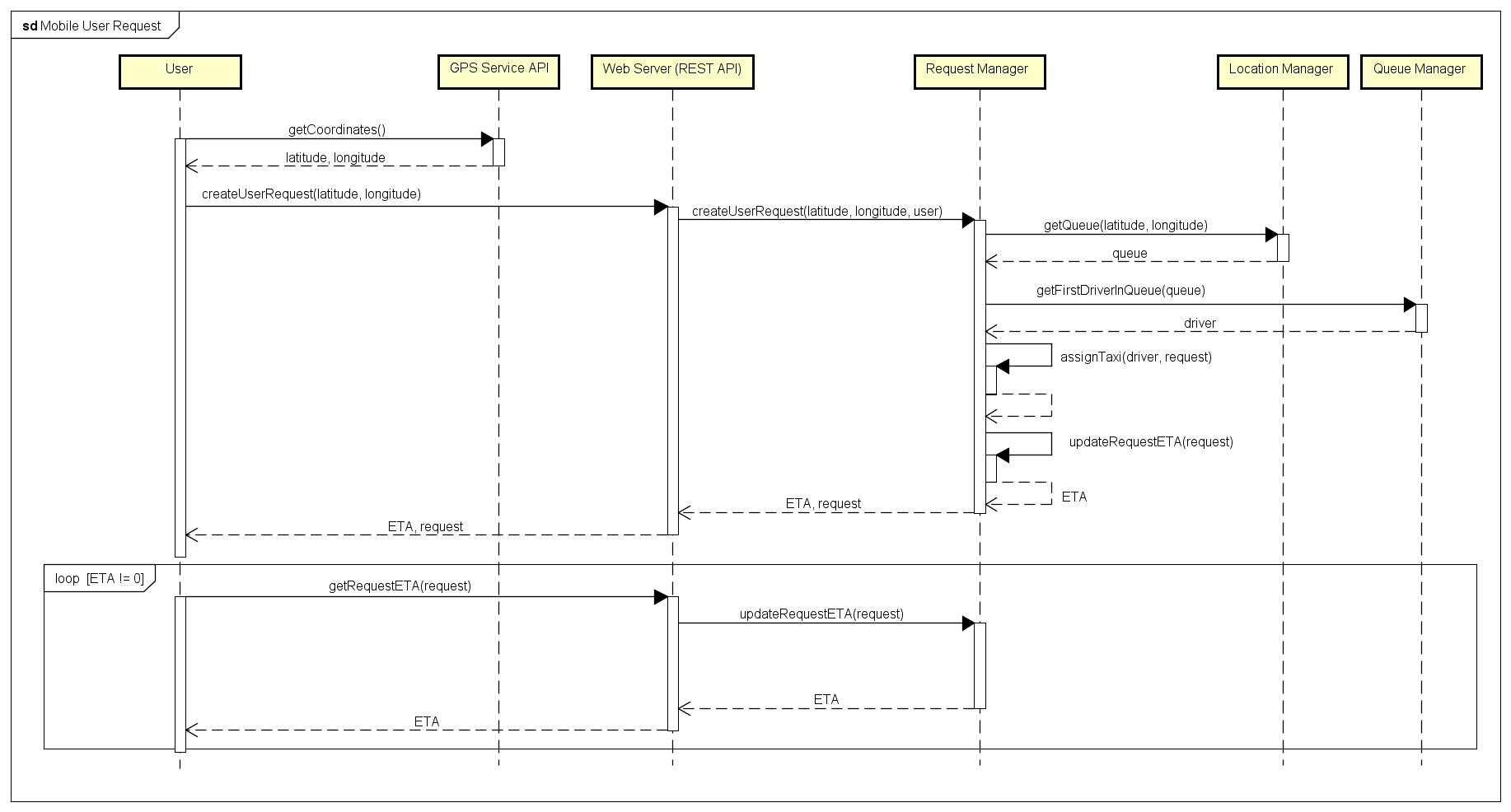
Mobile Passenger Signup



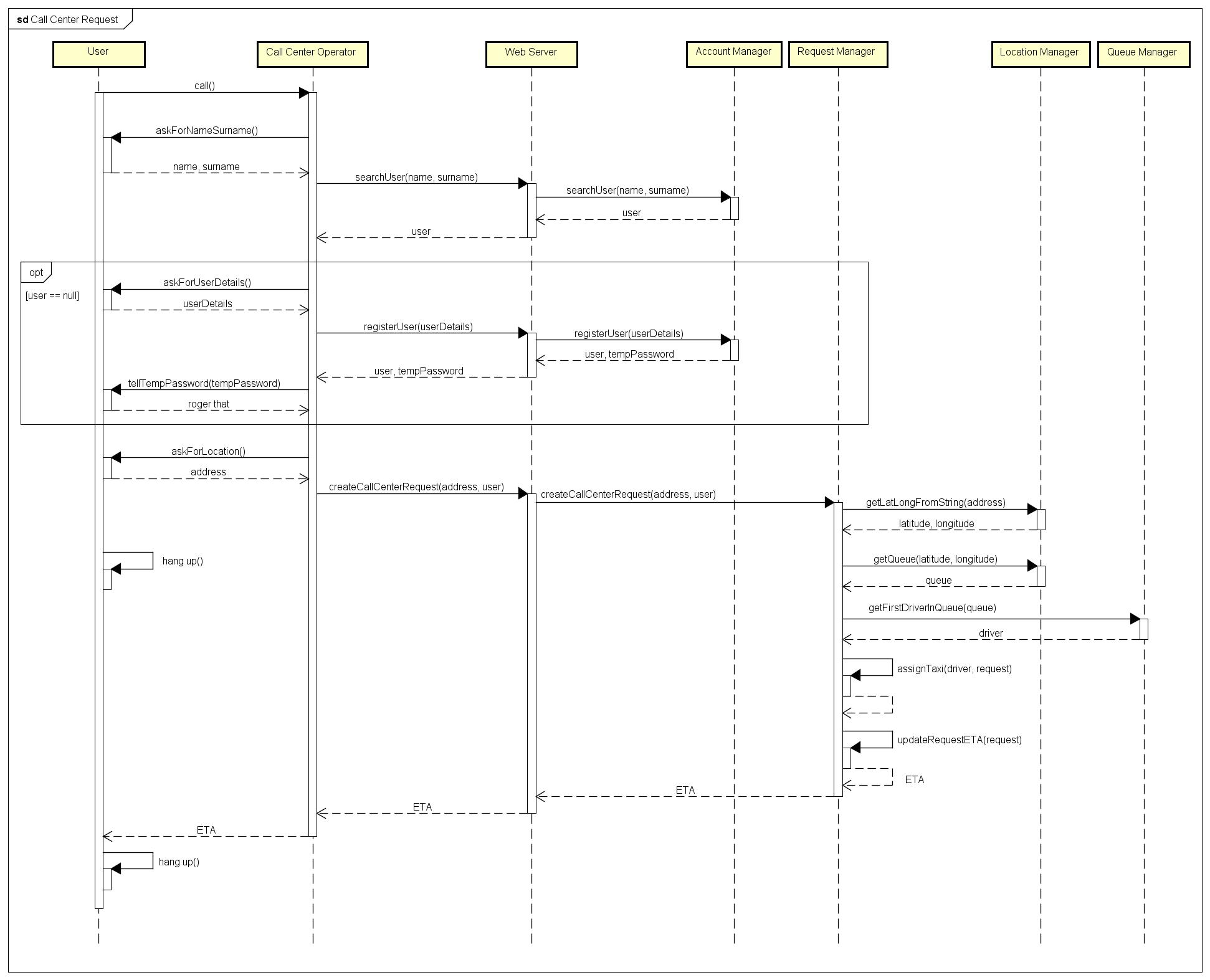
Logout



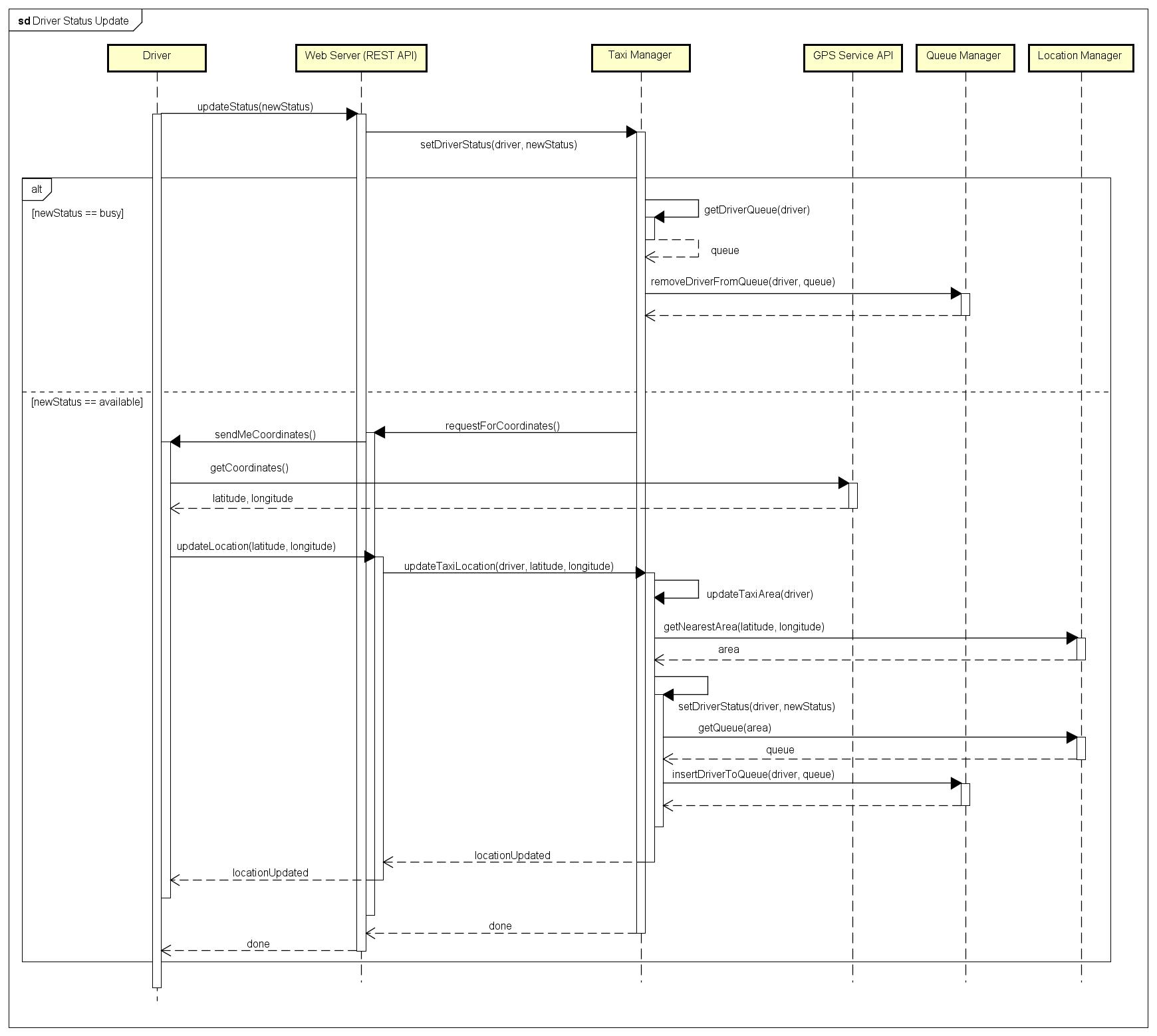
Mobile User Taxi Request



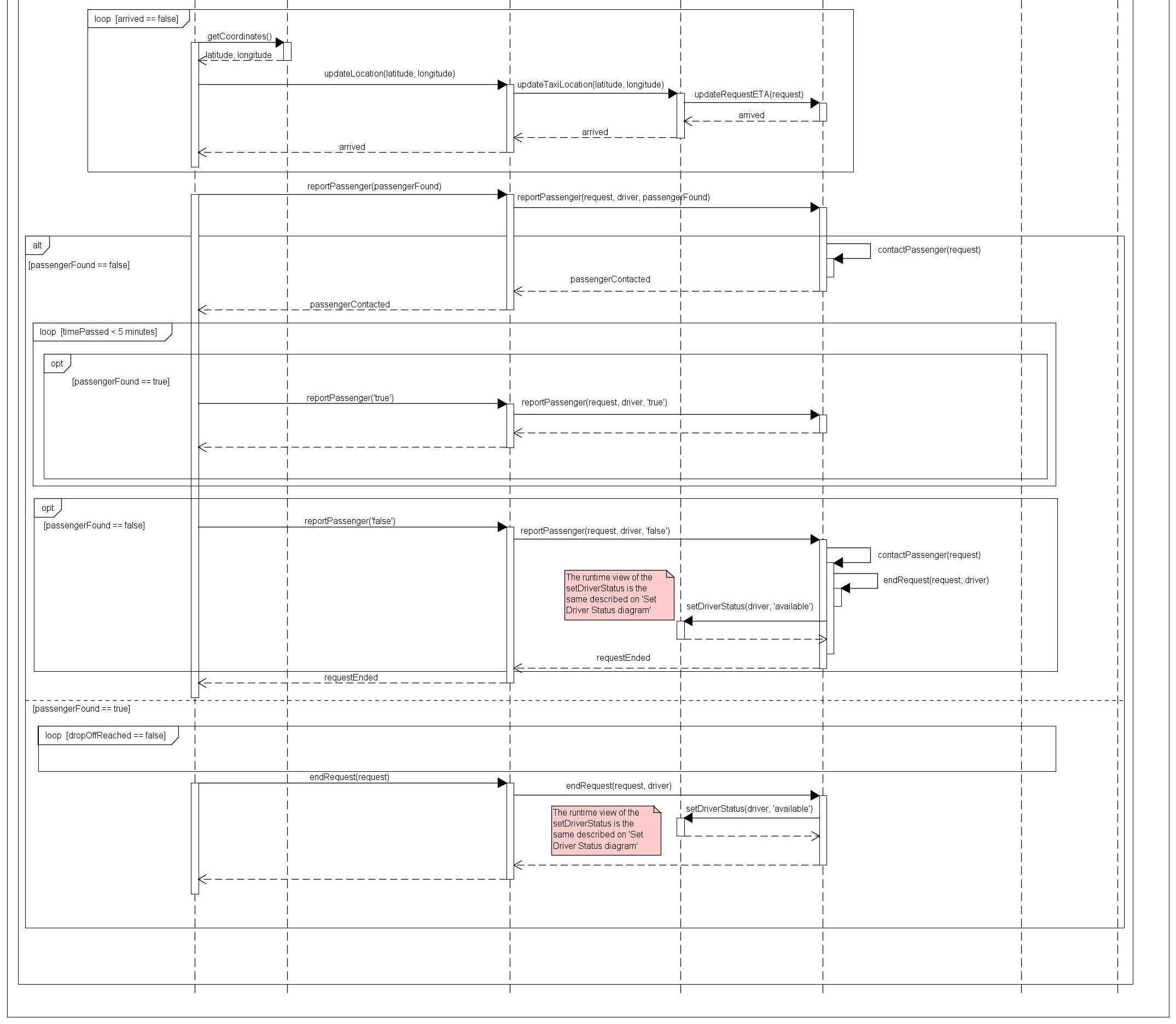
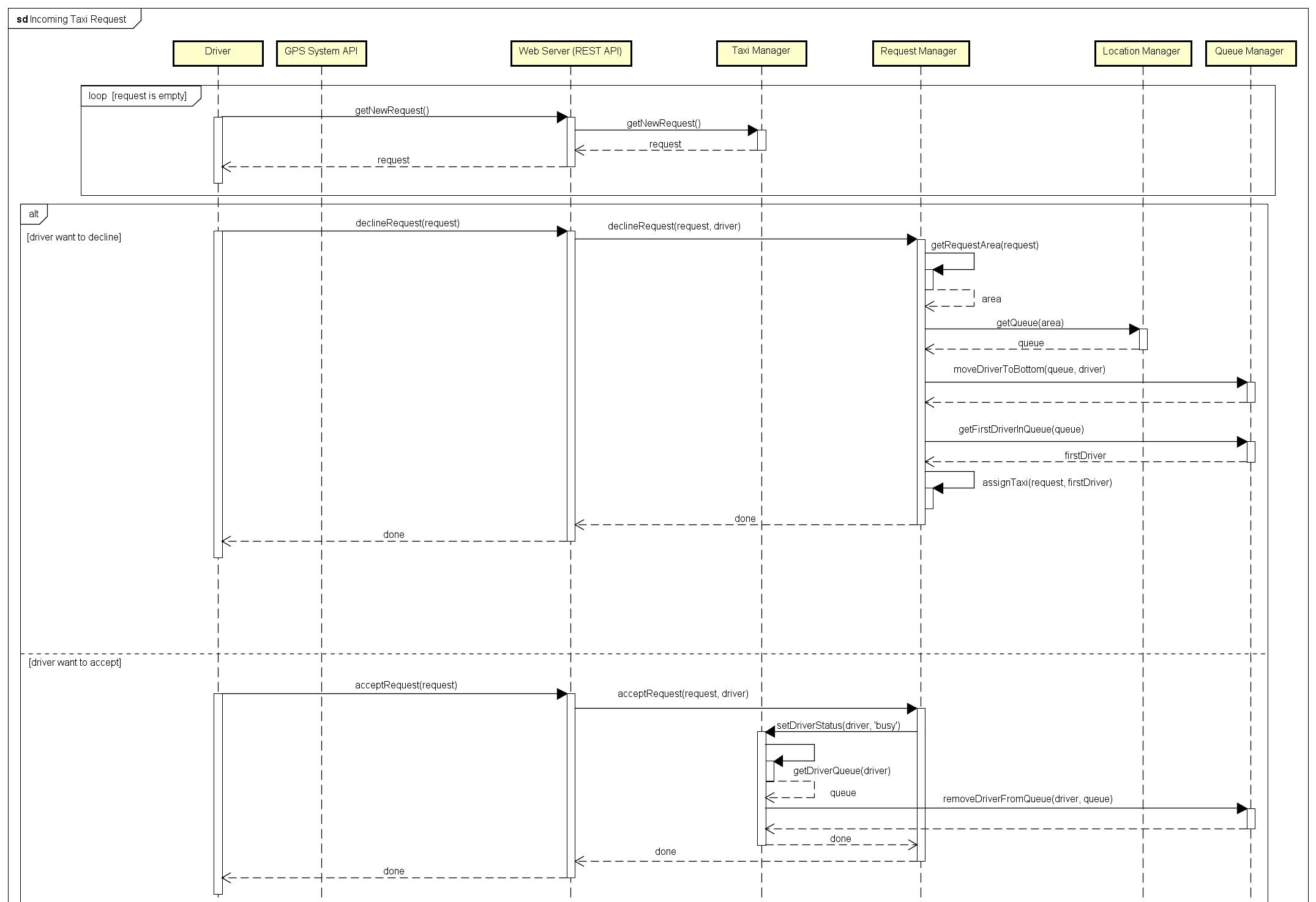
Call Center Request



Driver Status Update



Incoming Taxi Request



2.6 Component Interfaces

Account Manager

interface AccountManager {

public boolean login(String email, String password);

public boolean logout();

public boolean registerUser(ArrayList<String> parameters);

public boolean editUserData(User user, String data, 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);

}

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.
* editUserData: this method will edit a certain parameter (data) of a given user. The returned value corresponds to the success of the operation.
* getUserType: this method will return the type of a given user (Passenger, 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).

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 Driver getRequestDriver(Request request);

public User getRequestUser(Request request);

public void 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 void getRequestETA(Request request);

public void endRequest(Request request, Driver driver);

public Area getRequestArea(Request request);

public void setRequestArea(Area area, Request request);

public Date updateRequestETA(Request request);

public void contactPassenger(Request request);

}

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.
* getRequestDriver: this method will return the Driver data object associated to a request.
* getRequestUser: this method will return the User data object associated to a request.
* assignTaxi: this method will assign a given taxi to a given request.
* 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.
* getRequestArea: this method will return the area object assigned to the request.
* setRequestArea: this method will set the area associated to 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.
* contactPassenger: this method will contact a passenger of a given request. Will be invoked when a passenger is reported as not found.

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 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.
* getQueueArea: this method will return the Area data object associated to a given queue.

Location Manager

interface LocationManager {

public Area getNearestArea(Double latitude, Double longitude);

public Queue getQueue(Area area);

public ArrayList<Dobule> getLatLongFromString(String string);

public Queue getQueue(Double latitude, Double longitude);

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:

* getNearestArea: this method will return the nearest area to a given combination of coordinates.
* getQueue: this method will return the queue associated to a given area or to a given combination of coordinates.
* 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.
* getLatLongFromString: this method will return the latitude and longitude of a given street address.

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);

}

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.
* updateTaxiLocation: this method will update the coordinates representing the position of a given driver.
* 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.
* 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.

Database Manager

interface DatabaseManager {

public boolean insert(String table, ArrayList<Object> values);

public boolean update(String table, ArrayList<Object> newValues, String filters[]);

public ArrayList<Object> select(String table, String filters[]);

public boolean delete(String table, String filters[]);

}

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

* insert: this method will insert the given data inside a table
* update: this method will update the values of a given record of a given table.
* select: this method will select objects from a given table, accordingly to a given list of filters and return them.
* delete: this method will delete objects from a table accordingly to a list of given filters. The returned value corresponds to the success of the operation.

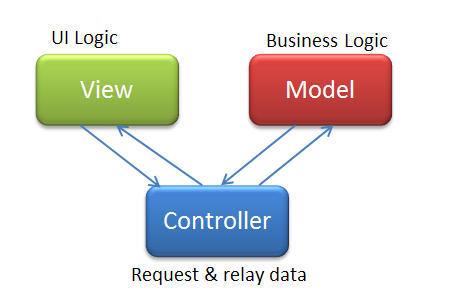
2.7 Selected architectural styles and patterns

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

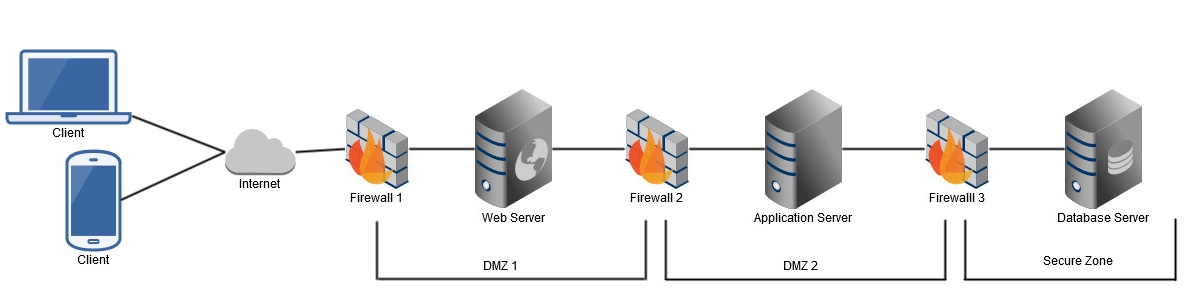
Due to the large amount of taxi request estimated in the RASD and all the logged taxi drivers, users and call center operators, this project will use Amazon EC2 with Elastic Load Balancing enabled for each tier:

* 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.

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.



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.

3. Algorithm design

Ideally

4. User interface design

Get from RASD, make UX

5. Requirements traceability

Easy

6. References

easy