

**Politecnico di Milano**

**A.A. 2015-2016**

**Software Engineering 2 project: MyTaxiService**

**Design Document**

Alessandro Pozzi (mat. 852358), Marco Romani (mat. 852361)

4 December 2015

Version 1.0

Sommario

[1. Introduction 3](#_Toc436653098)

[1.1 Purpose 3](#_Toc436653099)

[1.2 Scope 3](#_Toc436653100)

[1.3 Definitions, Acronyms and Abbrevations 3](#_Toc436653101)

[1.3.1 Definitions 3](#_Toc436653102)

[1.3.2 Acronyms 4](#_Toc436653103)

[1.4 Reference Documents 4](#_Toc436653104)

[1.5 Document Structure 4](#_Toc436653105)

[2. Architectural Design 5](#_Toc436653106)

[2.1 Overview 5](#_Toc436653107)

[2.2 Selected Architectural styles and patterns 5](#_Toc436653108)

[2.2.1 Three-Tier Architecture 5](#_Toc436653109)

[2.2.2 Even-based system 6](#_Toc436653110)

[2.3 High level components and their interaction 8](#_Toc436653111)

[2.4 Components View 9](#_Toc436653112)

[2.5 Deployment view 11](#_Toc436653113)

[2.6 Runtime view 12](#_Toc436653114)

[2.6.1 Request ride 12](#_Toc436653115)

[2.6.2 Reserve a ride 13](#_Toc436653116)

[2.6.3 Customer deletes a ride 14](#_Toc436653117)

[2.6.4 Driver refuses a request 15](#_Toc436653118)

[2.6.5 Login 16](#_Toc436653119)

[2.6.6 Registration 17](#_Toc436653120)

[2.6.7 Admin deletes a taxi driver account 18](#_Toc436653121)

[2.6.8 Taxi driver changes his availability 19](#_Toc436653122)

[2.7 Components Interfaces 20](#_Toc436653123)

[3. Algorithm Design 22](#_Toc436653124)

[4. User Interface Design 23](#_Toc436653125)

[5. Requirements traceability 24](#_Toc436653126)

[6. References 28](#_Toc436653127)

[6.1 Hours of work 28](#_Toc436653128)

[6.2 Software and tools used 28](#_Toc436653129)

# Introduction

## Purpose

The aim of the Design Document (DD) is to define guidelines that describes the architecture of a software project. Here the requirements specified in the Requirements Analysis and Specification Document (RASD) will be translated into a representation of software components, interfaces and others data necessary for the implementation phase. It is important to underline that this document will be the main reference during the code development. For this reason it won’t contain any detailed and extended code section (which would be an unnecessary constraint on programmers), but, instead, a set recommended design decision will be presented and theirs structures and advantages will be diffusely explained.

## Scope

In this document we will focus on the overall structure and architecture of the system, without going further into the details of the implementation. We will provide a high level description of the components involved and how they interact.

Components, connectors, interfaces are the main participants of the document.

We will explain the architecture styles adopted and the reasons behind them, trying to give a motivation for any choice taken.

We will also provide a few ideas concerning possible guidelines for the future implementations of the application’s main algorithms.

It is important to understand that all the content of the document is platform independent and the various architecture components will be mapped onto real hardware and software components only further in the implementation phase.

Since we have already provided a bunch of mockups for the graphical user interface in the RASD, we will only redirect you to them without showing them again.

## Definitions, Acronyms and Abbrevations

### Definitions

*Taxi* *Driver* – Employee of the taxi service with a driver account.

*Customer* –Registered user that may demand a taxi ride

*Guest* – Users that are accessing to MTS’s homepage (or other free services) not yet registered or not logged in

### Acronyms

MTS – MyTaxiService

GUI – Graphical User Interface

DB – Database

RASD – Requirements Analysis and Specification Document

## Reference Documents

* MyTaxiService’s RASD (by Alessandro Pozzi and Marco Romani)
* The IEEE standard 1016: Software Design Specification

## Document Structure

* *Section 1 - Introduction*
* *Section 2 – Architectural Design*: explains in details the architecture and design of MTS’s system, along with the chosen patterns and components identification.
* *Section 3* – *Algorithm Design:* shows a possible high-level implementation of some relevant application algorithms.
* *Section 4* – *User Interface Design*: shows indicatively how the user interface will look like.
* *Section 5* – *Requirements Traceability:* Shows how the requirements specified in the RASD have been satisfied in the design phase.
* *Section 6 – References:* Hours of works, software and tools used and others external information.

# Architectural Design

## Overview

MyTaxiService involves different users communicating over the internet with a single system. Such users may use different platforms (mobile and web) and can send requests of different types. The system must not only accept those requests and elaborate an answer in a short time, but it is required that it notifies multiple users of the occurring of some events. Usually a single event provide notifications for two types of users: taxi drivers and customers. Event notifications and users requests might also necessitate to access stored data, like taxi identifier or users information.

This brief analysis clearly highlight the need of implementing MTS as a client-server-like architecture, eventually subdivided into multiple physical tiers and logical layers: this will allow to model properly the request-answer requirement. The notification and updates part, instead, requires in our opinion a particular event-based paradigm: publish-subscribe. This allows users (the subscribers) to be notified by an entity (the publisher) on specific topics (a ride, for example).

These styles will be explained in detail in the following chapter.

## Selected Architectural styles and patterns

### Three-Tier Architecture

The image shows the tier architecture of the MTS system, composed by three physical tiers. We will now analyze every tier and explain its logical functions.

* *Top tier (Client)*

The users’ machines, that in our domain are mobile phones and computers, will have the only purpose to load the Graphical User Interface (GUI), which shows the services that can be requested from the MTS’s system. No application logic is involved at this level: Clients will only be able to send requests to the web server and application server.

Notice that users identified as clients are limited to the followings: Taxi Drivers, Customers, Guest.

* *Middle tier*

This tier encapsulate:

* The Web Server, which is the component of the system that manages the web requests sent by clients using the web application.

This component can handle such requests in two ways:

* if the request can be resolved with a static content page, the web server will generate and send the response itself
* if the request comport a dynamic content, the web server will delegate the dynamic response generation to the application server
* The Application Server, which provides access to the business logic, to be used by the client application programs. This component is the central part of MTS’s system, and will contain all the logic that provides MTS’s services. To accomplish this, it will be able to execute complex algorithms and access the Database tier.

The Application Server will also provide lightweight APIs to be used directly by mobile application clients. It will answer mobile’s requests by sending only the strictly necessary information, reducing the amount of data transiting over the mobile network and thus increasing the performance of the application.

Web application clients, instead, will be able to access this component only indirectly, through the Web Server.

* The Admin’s GUI, the specific interface for Administrators, is actually included and provided by the Application Server. It allows Admins to access to their exclusive functions dialoguing directly with the business logic of the system. It’s completely disjointed from the other users’ GUI and functions.
* *Bottom tier (Database)*

This tier, which will be separated from the previous one with a (possibly local) network, contains all the data that MTS needs to store, ranging from the users’ information to the city map.

### Even-based system

As anticipated, the core of MTS’s application logic is based on the publish-subscribe pattern.

Customers and taxi drivers have the role of subscriber: the system will automatically register them to specific topics, and they will receive update messages related to such topics. A topic is created every time customers request or reserve a ride. When a taxi driver is associated to that ride, he will be subscribed to the same topic too, and receive the relative notifications. Additionally, every taxi driver has an exclusive topic strictly bound to its status (available, busy…), which allows them to receive notification when the status is switched.

In this pattern we intend to use a broker, an intermediary component which performs the queue management and the filtering of the messages. The broker will allow to filter messages based on their content, so that taxi drivers and customers related to the same topic won’t receive necessary the same notification or messages.

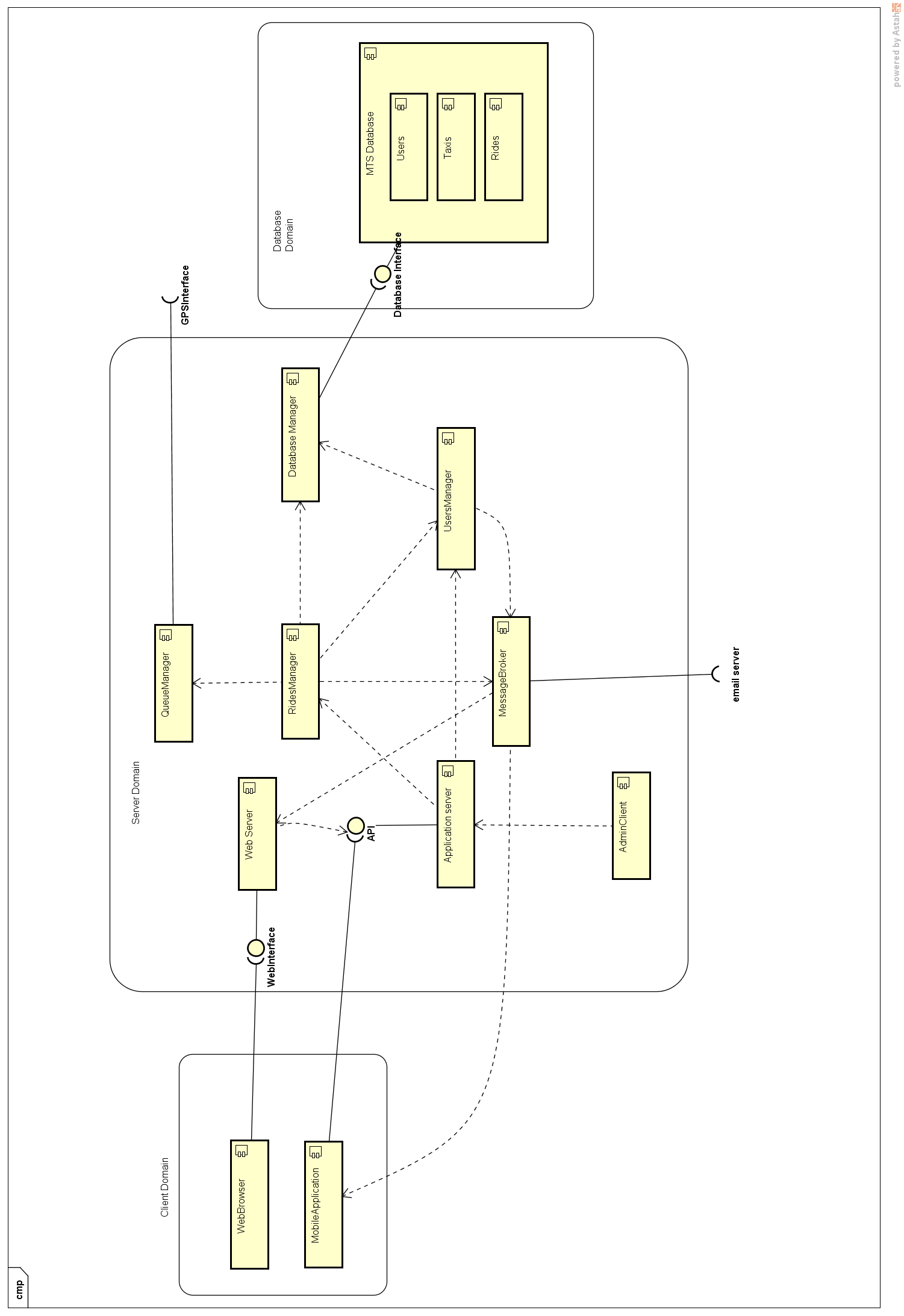
The role of the publisher, instead, is associated to the logic components of the system that manages the rides, the research of available taxis, etc. Basically, there are more components of the system that may generate an update for a certain topic, even users (for example, when a taxi driver change its status generates an update for its topic).

Despite the event-based pattern may not be strictly required to model the actual MTS system, it provides much space for future extendibility. For example, it allows to handle notification for multiple customer connected to the same ride (e.g. in a taxi sharing service), or permits to easily add new types of notifications.

## High level components and their interaction

## Components View

Component Diagram: this diagram describes the logical components that constitutes the physical tiers previously described.



* *Web Server*

This component represents the front-end of the system that interacts with the web application’s users.

* *Application Server*

Represents the front-end of the system that interacts with mobile application’s users and the Admin’s GUI.

* *Database Manager*

Handles the only access point of the middle tier to the DB tier.

* *Rides Manager*

Handles requests and reservations and, in general, everything that is connected with rides. It is a central part of the application, and is the only component that interacts with the *Queue Manager*.

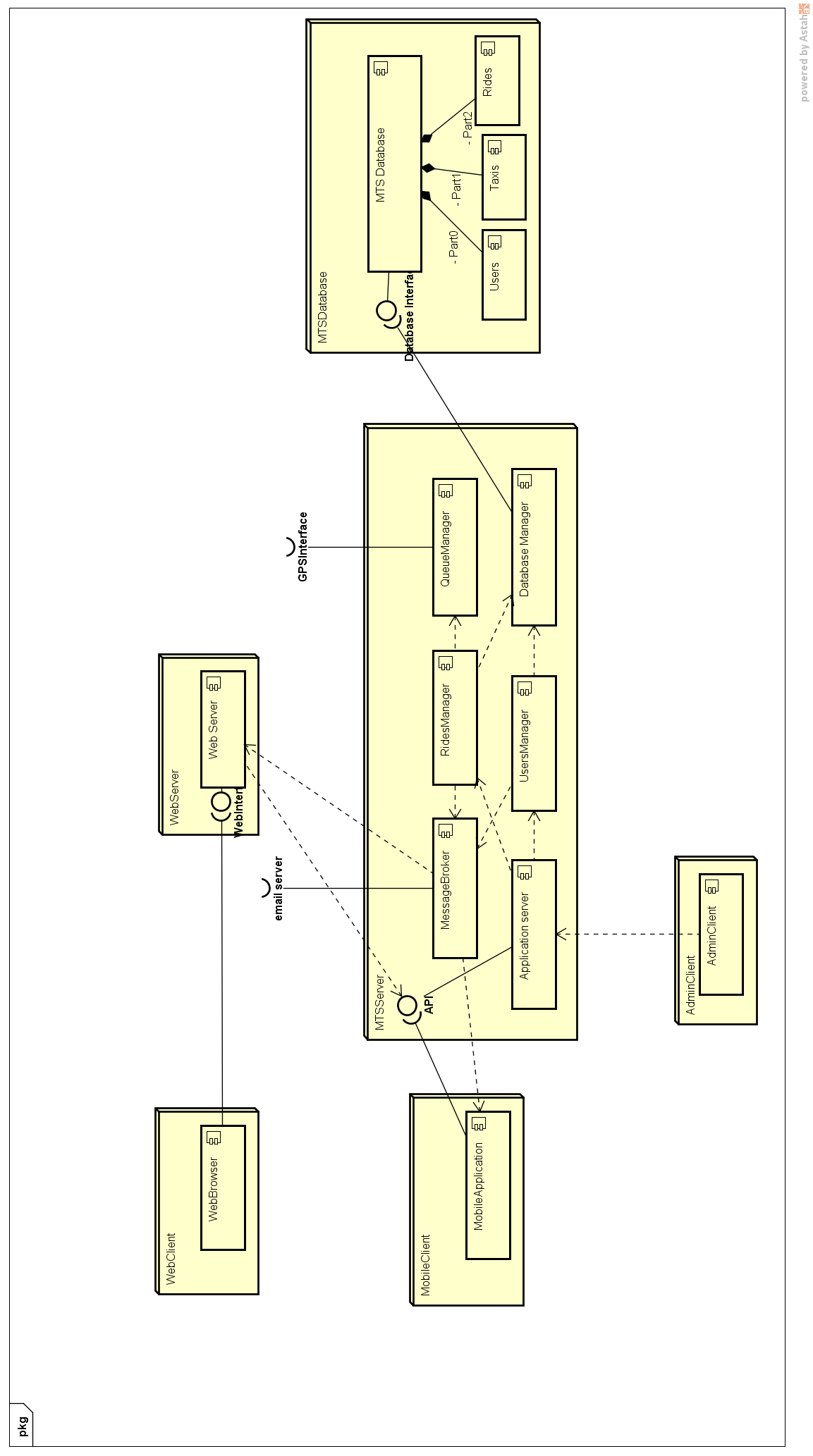
* *Queue Manager*

Manages the taxi zones and queues logic. It memorize (but does not store in the DB) the position of every active taxis received by the *GPS Interface*.

* *Message Broker*

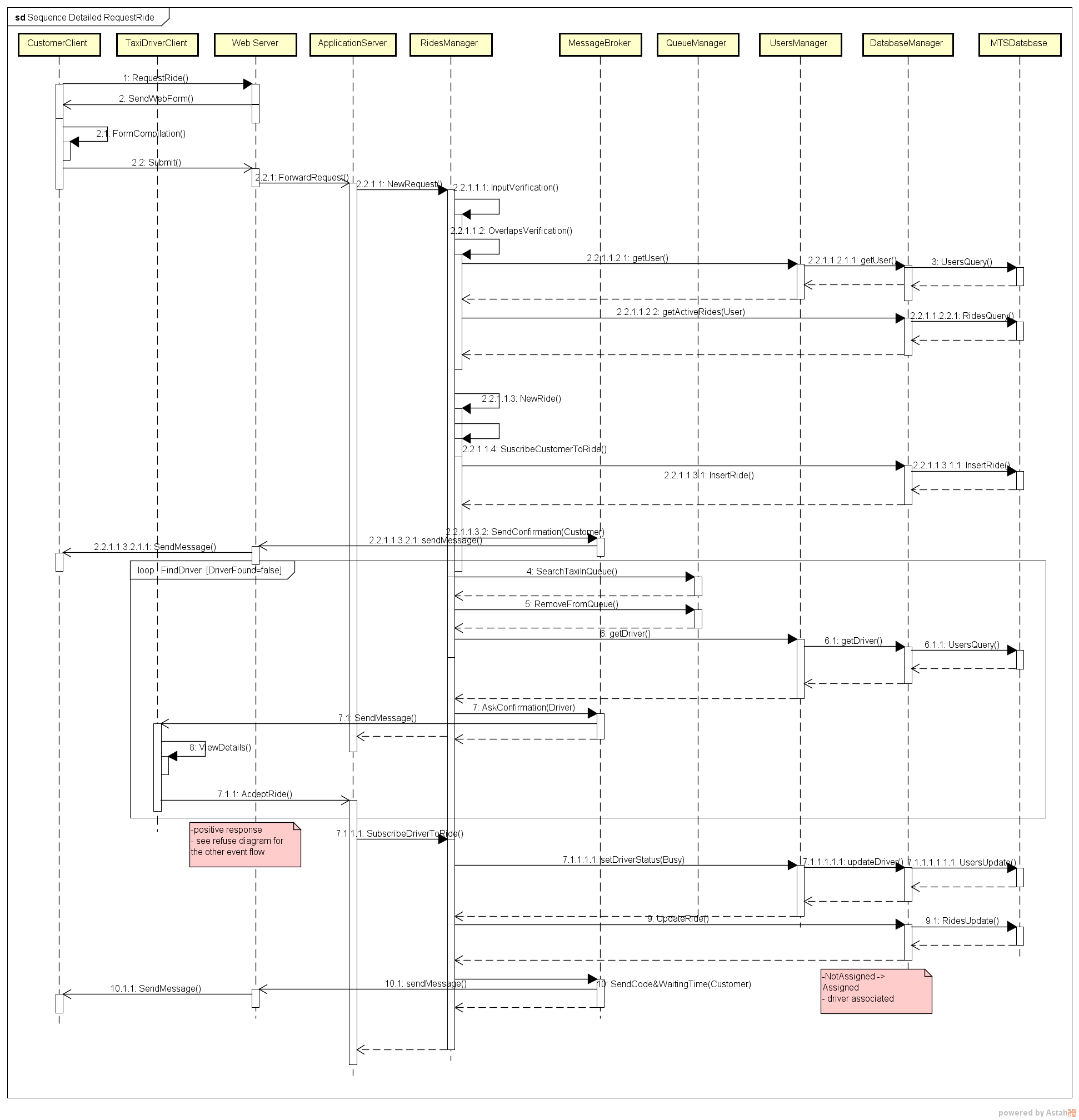
Implements the Broker role in the publisher-subscribe pattern. Receives the publication messages from the *Rides Manager* and *Users Manager* and forward the appropriate communications to the clients.

## Deployment view

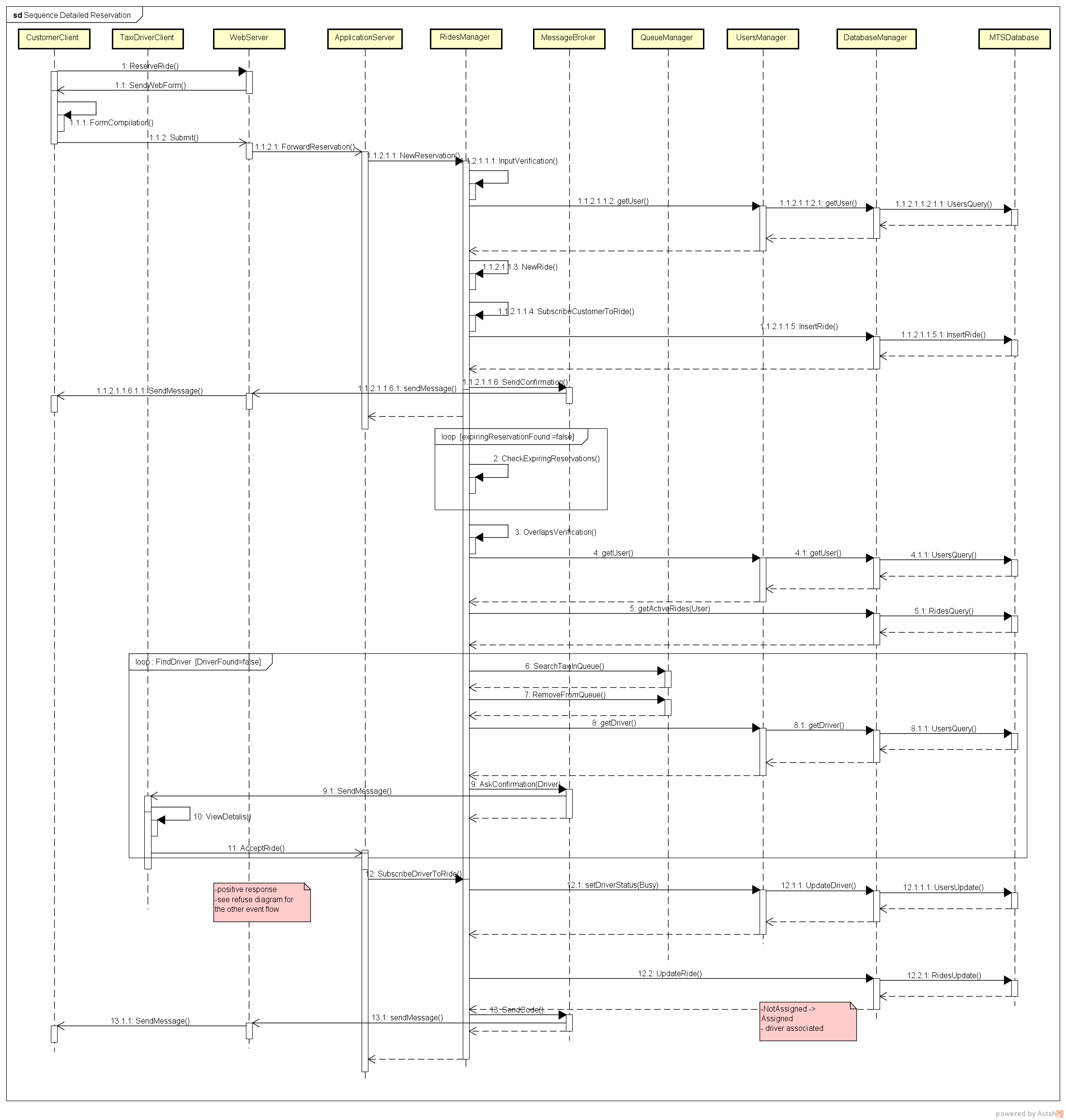


## Runtime view

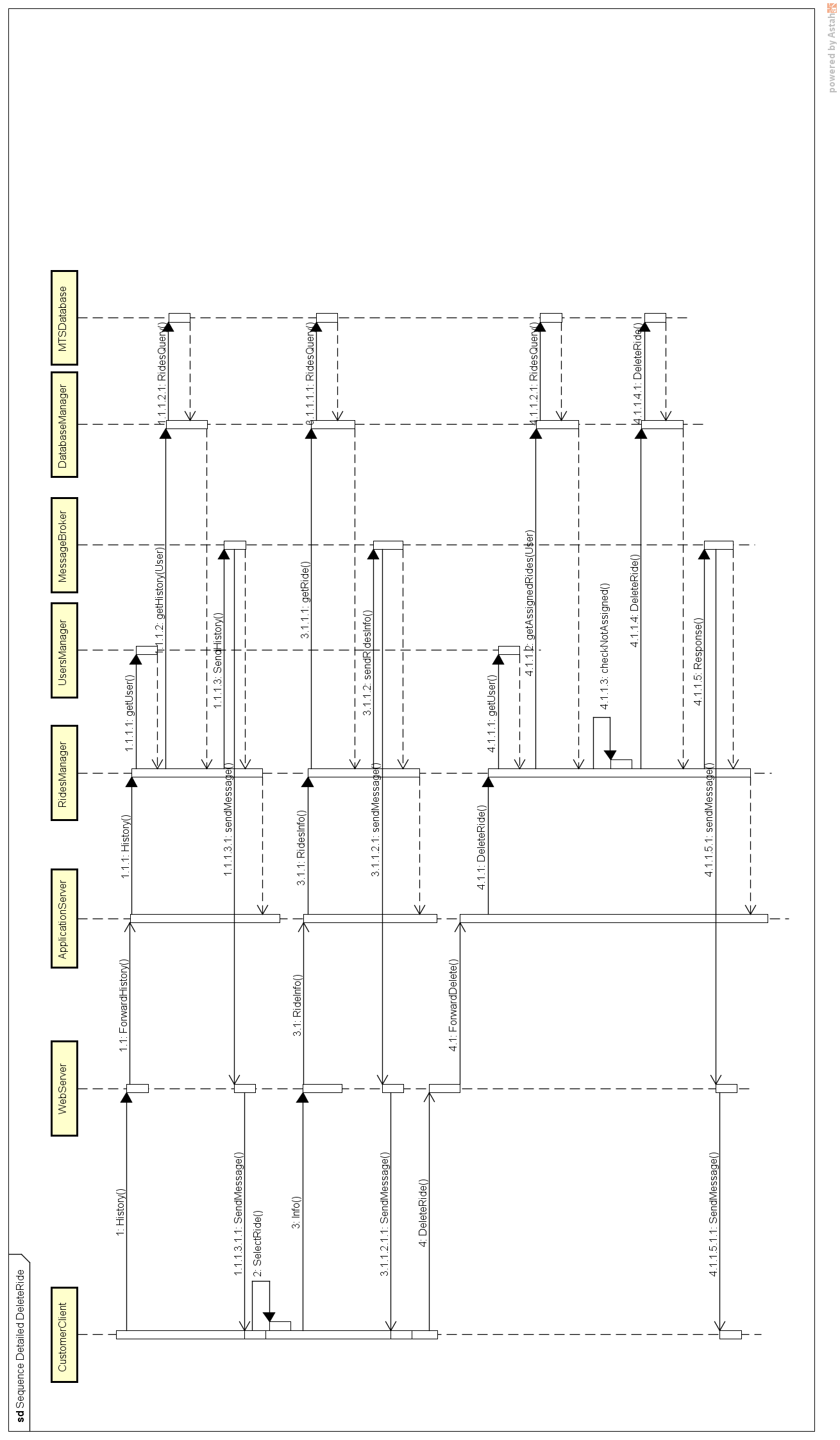
### Request ride



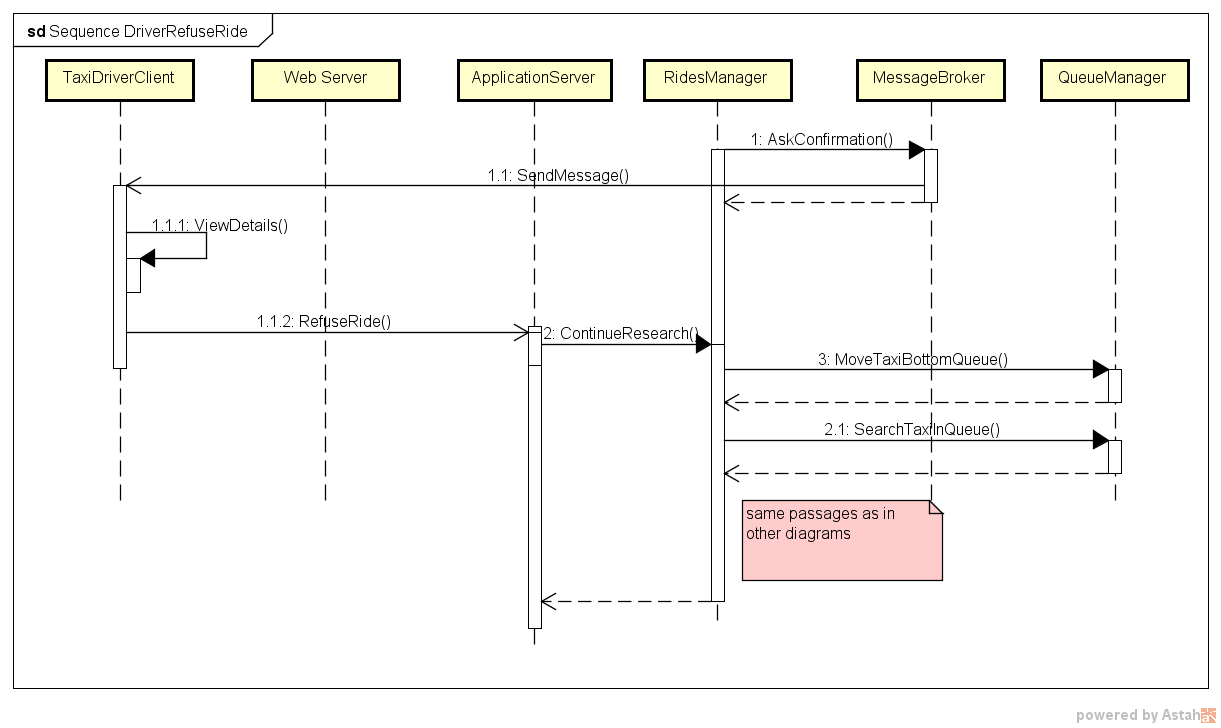
### Reserve a ride



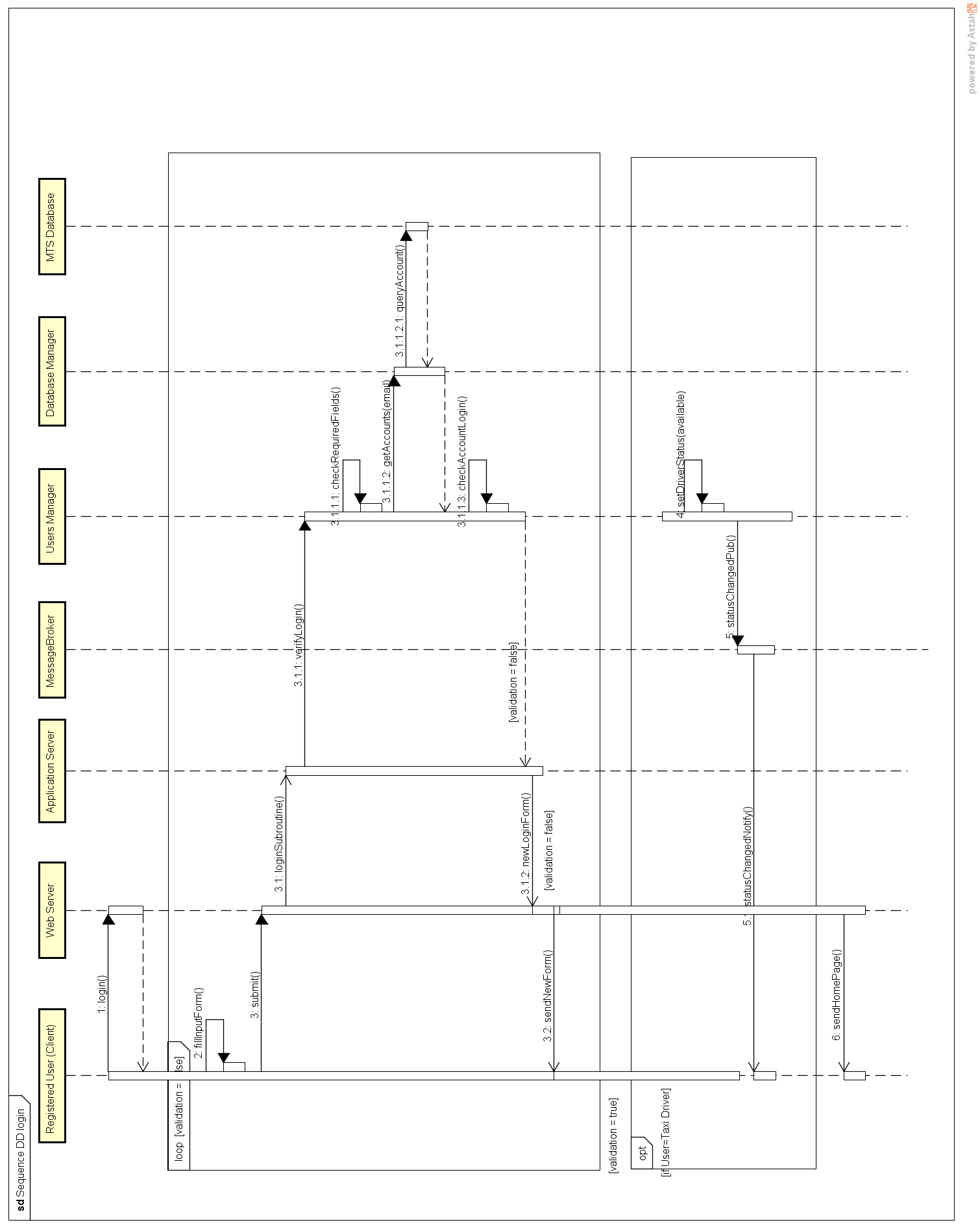
### Customer deletes a ride



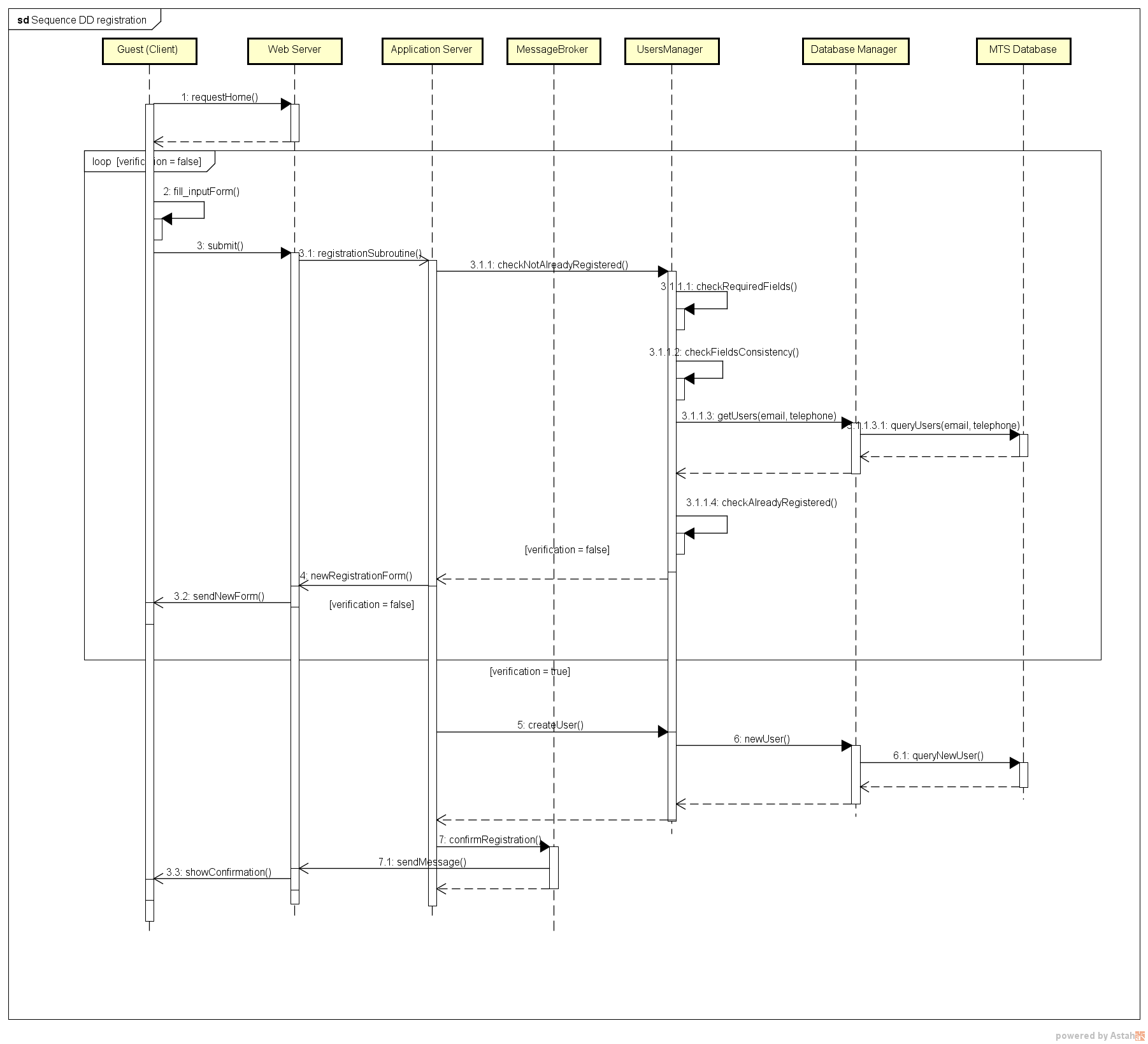
### Driver refuses a request



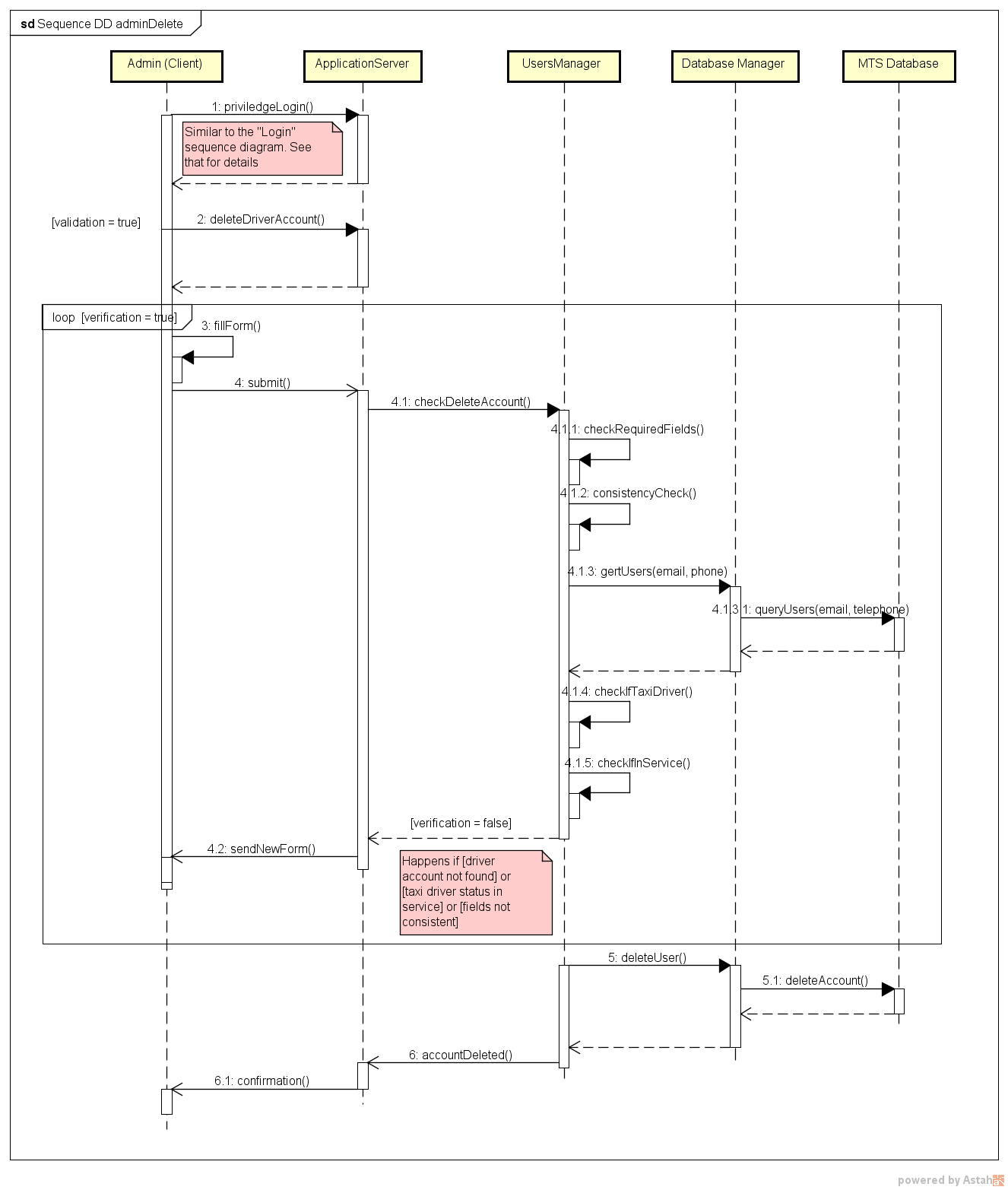
### Login



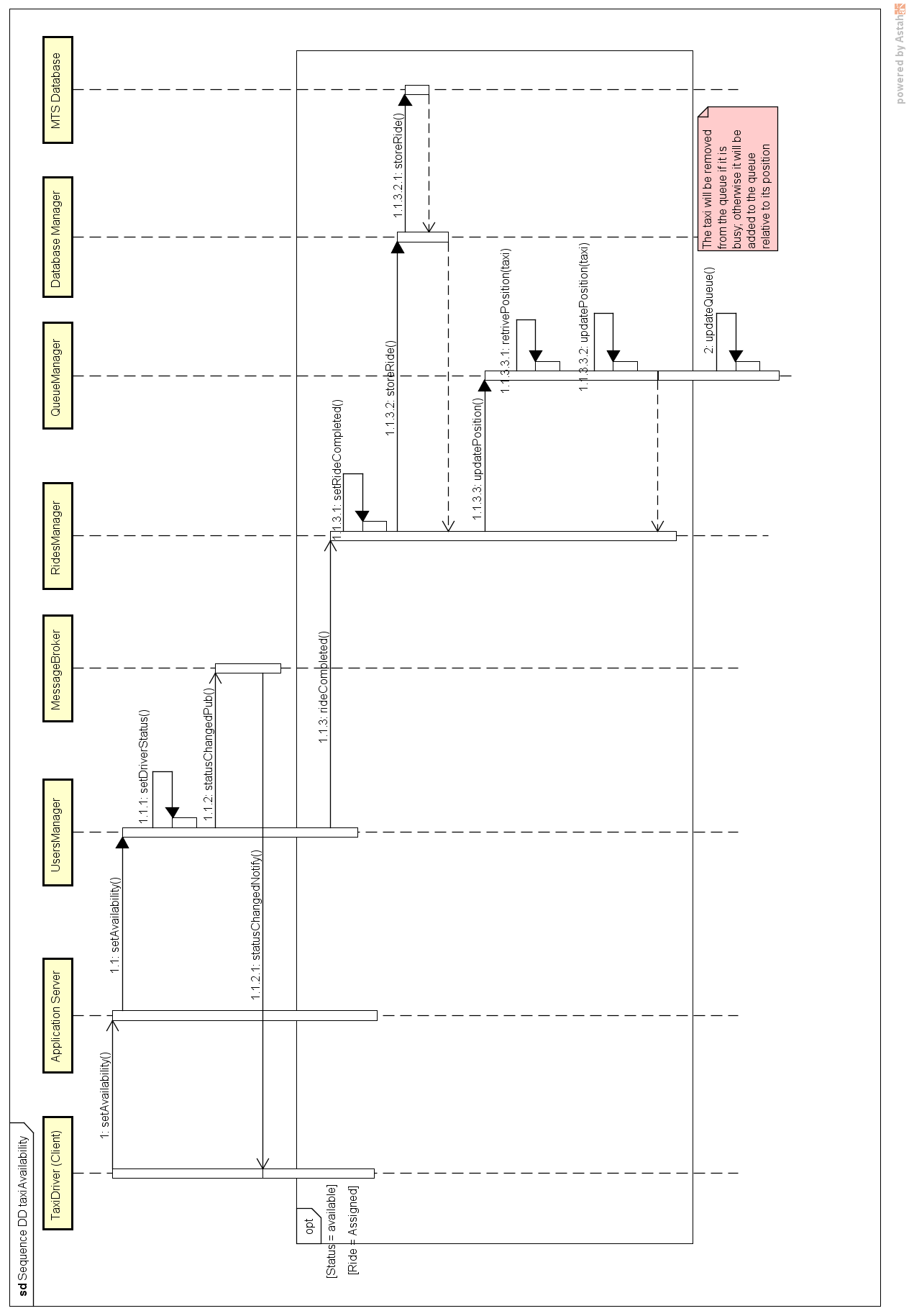
### Registration



### Admin deletes a taxi driver account



### Taxi driver changes his availability



## Components Interfaces

For each component defined in the previous diagrams we summarize which kind of interfaces link it to the rest of the system, specifying the nature of those interfaces.

* *Web Server*

Type of interface: exchange of web pages via http protocol.

Each service offered to web clients is mapped a customer service provided by the Application Server.

* *Application Server*

Type of interface: remote procedure call.

Summary of the main interfaces offered:

* Registration()
* Login()
* NewRequest()
* NewRegistration()
* DeleteRide()
* History()
* RidesInfo()
* AcceptRide() / RefuseRide() // only drivers
* setAvailability() // only drivers
* CreateDriver() / DeleteDriver() // only admin
* ModifyRide() // only admin
* ForceLogout() // only admin
* *RidesManager*

Type of interface: procedure call.

Summary of the main interfaces offered:

* NewRequest()
* NewReservation()
* RideCompleted()
* SubscribeCustomerToRide()
* SubscribeDriverToRide()
* DeleteRide()
* History()
* RidesInfo()
* *UsersManager*

Type of interface: procedure call.

Summary of the main interfaces offered:

* getUser()
* getDriver()
* setAvailability()
* checkAlreadyRegistered()
* createUser()
* verifyLogin()
* *QueueManager*

Type of interface: procedure call

Summary of the main interfaces offered:

* SearchTaxiInQueue()
* RemoveFromQueue()
* MoveTaxiBottomQueue()
* updatePosition()
* *DatabaseManager*

Type of interface: procedure call.

This component is a sort of stub acting the role of the real database.

The interfaces offered are basically all kind of query, inserts, deletes and updates addressed to the actual tables of the real database.

* *MessageBroker*

Type of interface: procedure call.

This component is responsible of communications with both mobile clients (directly via remote procedure calls) and web clients (indirectly via web server).

The interfaces offered are basically procedures that forward any kind of relevant message to the clients, possibly also to multiple clients due to the pub-sub architecture style.

* *MTSDatabase*

Type of interface: remote queries and standard database operations.

All kind of inserts, updates, deletes and queries are offered.

* *GPS Interface*

Type of interface: remote web service.

* RetrieveCoordinates()

# Algorithm Design

/\*\*\*\*\* QUEUE MANAGER \*\*\*\*\*/

---- searchTaxiInQueue SUBROUTINE ----

// Parameters given by the caller:

customerPosition

// Initialization:

initZone= findZone(customerPosition)

depth = getDEPTH() //The depth of the available taxi research should not be hard-coded so that future extensions may be easier.

//In our model DEPTH = 1.

G = getZonesGraph //gets the graph of the zones

//Algorithm core:

for each node n in G:

n.distance = INFINITY

n.parent = NIL

create empty queue Q

initZone.distance = 0

Q.enqueue(initZone)

while Q is not empty:

currentZone = Q.dequeue()

currentQueue = getQueue(currentZone)

if(NOTEMPTY(currentQueue)

return currentQueue.DEQUEUE

for each node n that is adjacent to currentZone:

if n.distance == INFINITY:

if (currentZone.distance + 1 < depth)

n.distance = currentZone.distance + 1

n.parent = currentZone

Q.enqueue(n)

---- moveBottomQueue SUBROUTINE ----

// Parameters given by the caller:

taxiToBeMoved

//Algorithm

zone = findZone(taxiToBeMoved)

queue = getQueue(zone)

queue.ENQUEUE(taxiToBeMoved)

/\*\*\*\*\* RIDES MANAGER \*\*\*\*\*/

---- when a taxi must be assigned to a driver SUBROUTINE ----

customerPosition = getCustomerPosition //extract the info from the ride

taxi = searchTaxiInQueue(customerPosition) //call the subroutine in the QUEUE MANAGER

driver = extractDriver(taxi)

askConfirmation(driver) //calls the MESSAGE BROKER to send the request to the driver

/\* ... here this algorithm is over. The rides manager will be called again when the driver answers

# User Interface Design

Please refer to the “*External Interface Requirements”* paragraph in the RASD document.

# Requirements traceability

The following table associate the requirements defined in the RASD document into the design component delineated in this document.

In the left column are listed all the requirements related to the goals, while in the right column there are the main components, as described in the component diagram and in the architectural style section, that allows that particular system functionality.

Note that components that are obviously required for almost all the functionalities, like the *Web Server* or the *Application Server*, sometimes will be omitted in order to preserve readability. The *Database Manager* won’t appear in this table for the very same reason.

|  |  |
| --- | --- |
| ***Goals / Requirements*** | **Design Components** |
| **GOAL G1** |  |
| [R1] Customers should be able to access the service through both the web and the mobile application, even at the same time. | Web Server  Application Server  (Three tier architecture) |
| [R2] Customers must be able to register to the taxi service from the mobile or web homepage. | Web Server  Application Server  (Three tier architecture) |
| [R3] Only registered customers can access MyTaxiService’s services. | Web Server  Application Server  Users Manager  (Three tier architecture) |
| [R4] The system should allow the log out functionality. | Web Server  Application Server  Users Manager  (Three tier architecture) |
| *Notes* | The servers specified in the architecture description and the internal component “Users Manager” verify and eventually allow the costumer to access the system functionalities. |
|  | |
| **GOAL G2** |  |
| [R1] Only registered customers can request a taxi ride. | Web Server  Application Server  Users Manager  (Three tier architecture) |
| [R2] Customers must insert a valid origin location in order to request a ride. | Rides Manager – Checks the syntactic correctness of the inputs |
| [R3] The system will not allow more than a request if the previous one (either request or reservation) has not been accomplished yet. | Rides Manager |
|  | |
| **GOAL G3** |  |
| [R1] The system should allow taxi reservations for a specific path communicated by the customer. | Rides Manager |
| [R2] The system must not allow overlaps between reservations (or requests) made by the same customer. | Rides Manager |
| [R3] The system allows reservations only 2 hours before the time and date specified by the customer. | Rides Manager |
| [R4] The system will assign a taxi driver for the reserved ride 10 minutes before the time and date specified by the customer. | Rides Manager |
|  | |
| **GOAL G4** |  |
| [R1] Taxi drivers should be able to communicate their current availability state to the system. | Message Broker – communicate the change of status |
| [R2] If available, taxi drivers should be able to receive incoming requests. | Ride Manager – provides the requests  Queue Manager – select the taxi driver  Message Broker – send the requests |
| [R3] After receiving an incoming request, the taxi driver should be able to either confirm or not his intention to take charge of the request. | Message Broker – receive and |
| [R4] Taxi drivers must be able to log in the mobile application with preassigned credential and be identified as drivers. | Application Server  Users Manager |
| [R5] At the end of their workshift, taxi drivers must be able to log out of the mobile application in order to communicate to the system that they are no longer active. | Application Server  Users Manager |
|  | |
| **GOAL G5** |  |
| [R1] The system should always search an available taxi giving maximum priority to the taxi zone related to the request and lower priority to the immediate near zones. Any other taxi zone should be ignored. | Queue Manager – uses the breadth first algorithm to find the zones and queues |
| [R2] If no taxis are available in the zones specified in the previous requirement, the system should put the request on hold and periodically check again the taxi availability. | Ride Manager  Queue Manager  Message Broker – tells the customer that no taxis are available |
|  | |
| **GOAL G6** |  |
| [R1] The system should send updates through email and/or in-app notification, as specified by the customer. | Message Broker |
| [R2] Absence of taxis available, reservations overlaps, taxi average waiting time and taxi assigned to customers are events that must be notified to the customer. | Ride Manager – provide the events  Message Broker – sends the events |
|  | |
| **GOAL G7** |  |
| [R1] Customers must leave a valid phone number in order to complete the registration phase. | Users Manager – check the consistency of the fields |
| [R2] Taxi drivers must be able to access to the customer’s phone number when the system has paired them. | Message Broker – sends the request and all the related information |
| [R3] Customers must receive the taxi drivers’ contact number after the system has paired them. | Message Broker – sends the updates of a request and the related data |
| [R4] Customers must receive the taxi code in order to be able to recognize its driver. | Message Broker – sends the updates of a request and the related data |
|  | |
| **GOAL G8** |  |
| [R1] Customers can cancel a request or reservation only if it has not been assigned to a taxi driver yet. | Rides Manager |
| [R2] Customers must be able to visualize the list of all their requests and reservations. | RidesManager  Database Manager |
|  | |
| **GOAL G9** |  |
| [R1] Administrators must be able to create a taxi driver’s account. | Application Server  Users Manager |
| [R2] Administrators must be able to delete a taxi driver’s account. | Application Server  Users Manager |
| [R3] Administrators must be able to change the status of taxi driver. | Application Server  Rides Manager  Users Manager |
| [R4] Administrators must be able to change the status of a ride. | Application Server  Rides Manager |
| *Notes* | The application server provide Admins the direct access to the Users Manager. |
|  | |
| **GOAL G10** |  |
| [R1] Customers and taxi drivers must be able to visualize, both in the mobile and web application, a support phone number which they can call to obtain assistance. | Web Server  Application Server |

Notice also that the component design has partially satisfied some of the *Non Functional Requirements*: for example, the ones about the reactivity of the application are now simply achievable through our light-oriented system.

# References

## Hours of work

Alessandro Pozzi ~22 hours

Marco Romani ~22 hours

## Software and tools used

* Microsoft Word (<https://products.office.com/it-it/word>) to redact and to format this document.
* Astah Professional (http://astah.net/) to create Use Cases Diagrams, Sequence Diagrams, Class Diagrams and State Diagrams.
* GitHub (<https://github.com>) to share the working material of this project.
* Notepad++ (<https://notepad-plus-plus.org/>) to write the algorithm’s code.