Politecnico di Milano

Software Engineering 2

MyTaxiService

Design Document

Author:
Andrea Tirinzoni

Author:
Matteo Papini

December 3, 2015

Contents

1 Introduction

1.1 Purpose

This is the Design Document for the MyTaxiService system. The aim of this document is to provide a detailed description of the system's architecture, its main components and their interaction. It is mainly addressed to developers and testers.

1.2 Scope

The aim of this project is to develop a system to improve the taxi service of Milan, accessible both via web and mobile applications. For further details see RASD, section 1.2.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

See RASD.

1.3.2 Acronyms

- DD: Design Document;
- RASD: Requirement Analysis and Specification Document;
- Java EE: Java Enterprise Edition;
- HTTP: HyperText Transfer Protocol;
- EJB: Enterprise Java Bean;
- JPA: Java Persistence API;
- JAX-RS: Java API for RESTful Services;
- P2P: peer-to-peer;
- MVC: Model-View-Controller;
- API: Application Programming Interface

1.3.3 Abbreviations

- app: application

1.4 Reference Documents

- RASD.pdf

1.5 Document Structure

We provide here the overall structure of this document:

- In section 1 the problem is introduced;
- In section 2 the architecture of the final system is proposed, focusing on the main components and their interaction;
- In section 3 the main algorithms used by the application are presented;
- In section 4 the user interfaces are detailed furtherly;
- In section 5 the requirements defined in the RASD document are mapped to components of the proposed architecture.

2 Architectural Design

2.1 Overview

This part of the document provides a detailed description of the system architecture. Section 2.2 illustrates the components at a high level, both from a physical and a logical perspective. Sections 2.3 and 2.4 furtherly detail the components, while section 2.5 focuses on their interaction. Section 2.6 specifies the interfaces provided by each component. Sections 2.7 and 2.8 explain and justify design choices.

2.2 High level components and their interaction

The architecture contemplates a central system, which provides all the required services and manages data, and three types of clients:

- Web application for passengers, entirely accessible via a web browser;
- Mobile app for passengers, a hybrid app accessing the same contents as the web application;
- Mobile app for drivers, a native application.

The system will be implemented using Java EE technologies.

2.2.1 Hardware architecture overview

Clients access the services through the Internet. Client side devices are:

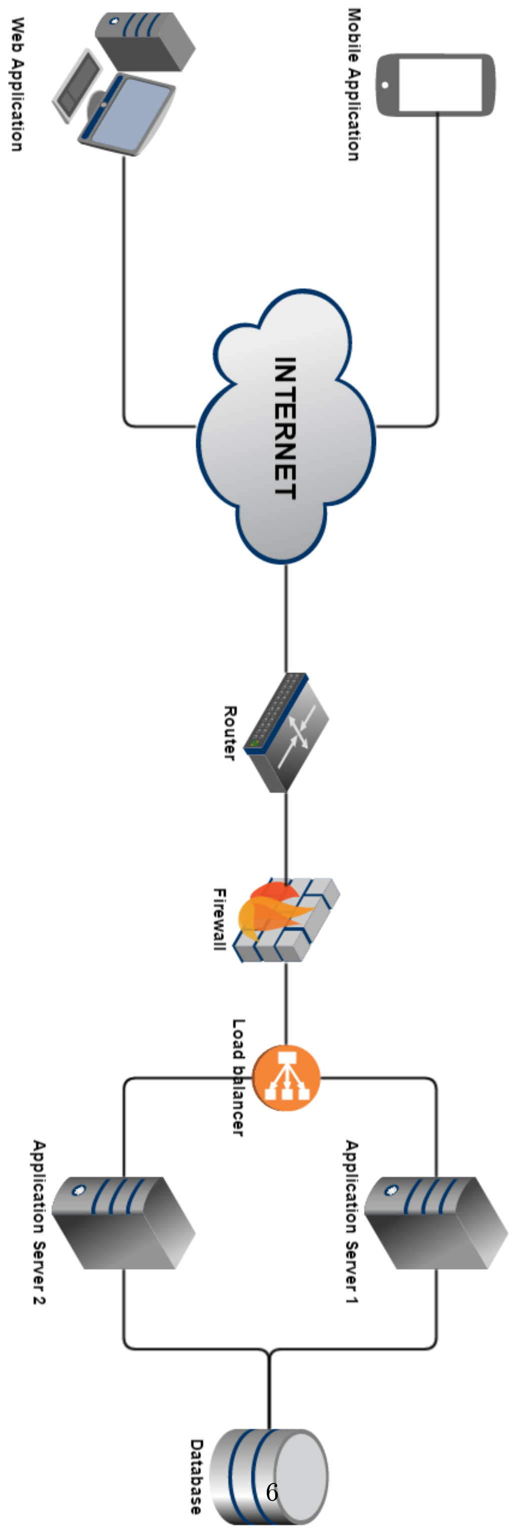
- Mobile phones running a standard operating system (Android, iOS or Windows Phone), on which the mobile applications are installed;
- Any device running a web browser, from which the web application is accessible.

Server side devices are:

- Router: connects the central system to the internet;
- Firewall: provides protection to the back-end system;
- Application server: a machine providing all the functionalities of a web server and hosting the business logic. It handles all application logic: client requests, queue management, taxi allocation... It communicates both with clients and with the Database Server. Considering the nonfunctional requirements specified in the RASD document, in particular availability and performance, a single machine could not be enough to properly manage the whole process. Thus, a cluster of two dedicated machines will host replicated instances of the server;

- Load balancer: redirect incoming requests to the machines in the cluster, in order to maximize performances;
- Database Server: the DBMS storing the system's data.

A graphical overview of the hardware architecture is provided on the next page.


Web Application
Mobile Application
INTERNET
Router
Firewall
Load balancer
Application Server 2
Application Server 1
Database

2.2.2 Software architecture overview

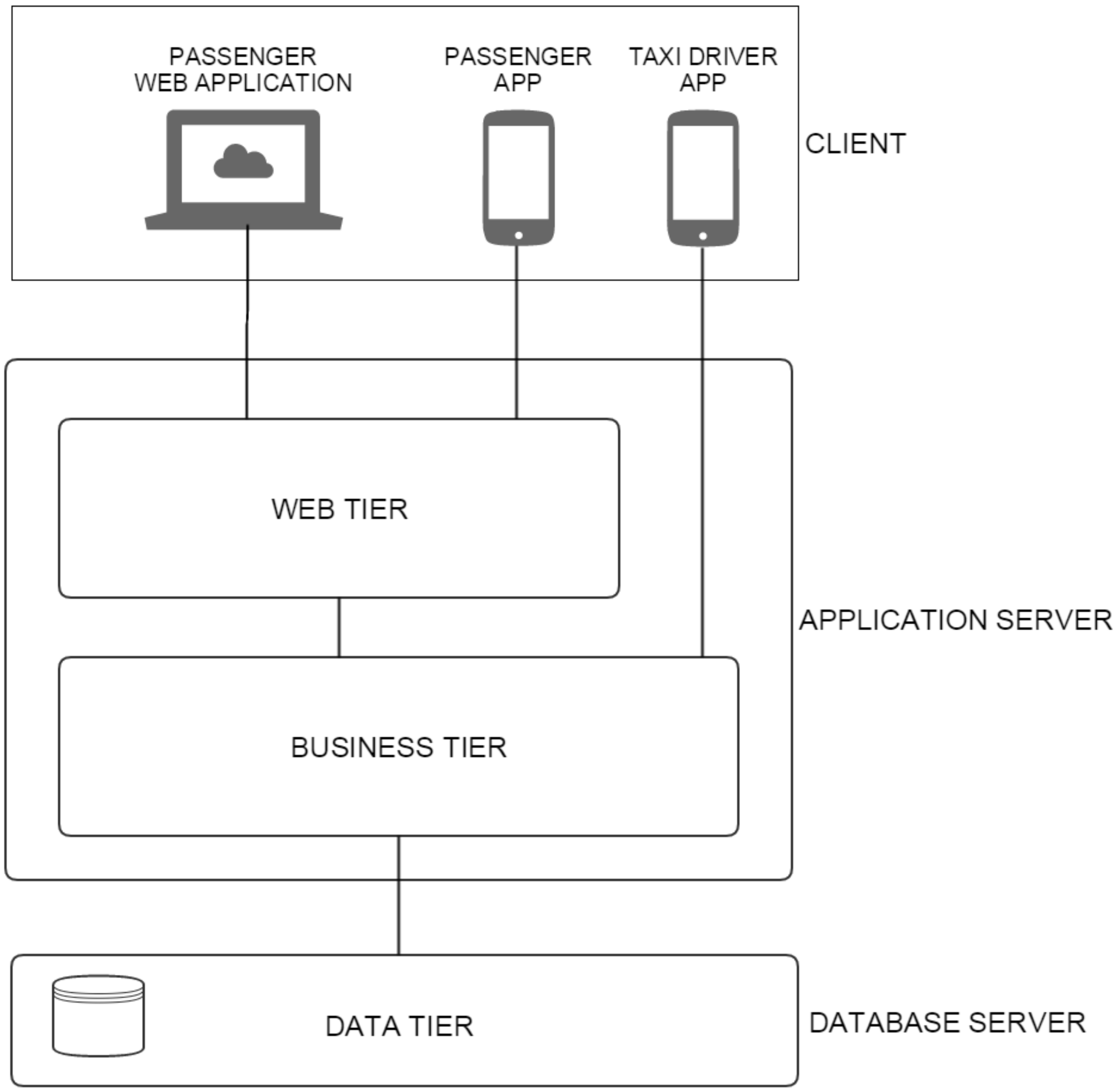
The central system is 3-tiered.
The server-side tiers are:

- Web tier: receives HTTP requests from the web application and the passenger mobile app, processes them by invoking services on the business tier and returns HTTP responses to the client;
- Business tier: includes several components providing the system functionalities, to be invoked both by the web tier and, remotely, by the driver mobile app. Some of these components are accessible as web services, so that external developers may use them in their application. The components access the database to store and retrieve data. A detailed description of this tier's components is provided in section 2.3;
- Data tier: receives query requests from the business tier and processes them.

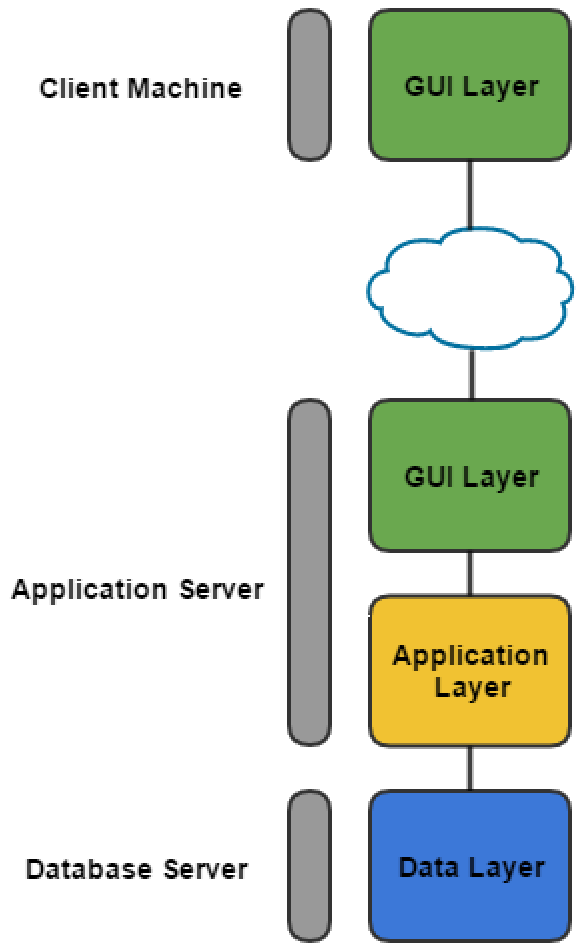
The client tier includes:

- Web application: a set of dynamic web pages accessed through a browser, which communicates through HTTP with the web tier;
- Passenger app: a hybrid mobile application (that is, an application directly displaying web contents), which communicates through HTTP with the web tier. Considering that the web application must also communicate using HTTP with the web tier and its functionalities are the same of those provided by the mobile one, a hybrid app has been chosen instead of a native. This choice will also simplify the deployment on different mobile operating systems;
- Taxi driver app: a native mobile application, which directly communicates with the business tier components according to a peer-to-peer paradigm (specified in section 2.3 and 2.7).

A diagram showing the tiers involved is provided below:



The software layers are distributed between the specified tiers according to the distributed presentation paradigm:



The application server, indeed, assembles the web pages, i.e. the client's view. Thus, a part of the GUI is handled by the application server. Note that this concerns only the web application and the passenger app. The driver app takes care of all the GUI layer by itself, thus the paradigm is, in this case, remote presentation.

2.2.3 Implementation technologies

As already mentioned, the system will be implemented with Java EE. More precisely, the following technologies will be used:

- Application Server: Glassfish 4.1.1;
- Database Server: MySQL;
- Web tier: JavaServer Faces;
- Business tier: EJB (for the components), JAX-RS (for the web services), JPA (for the database communication).

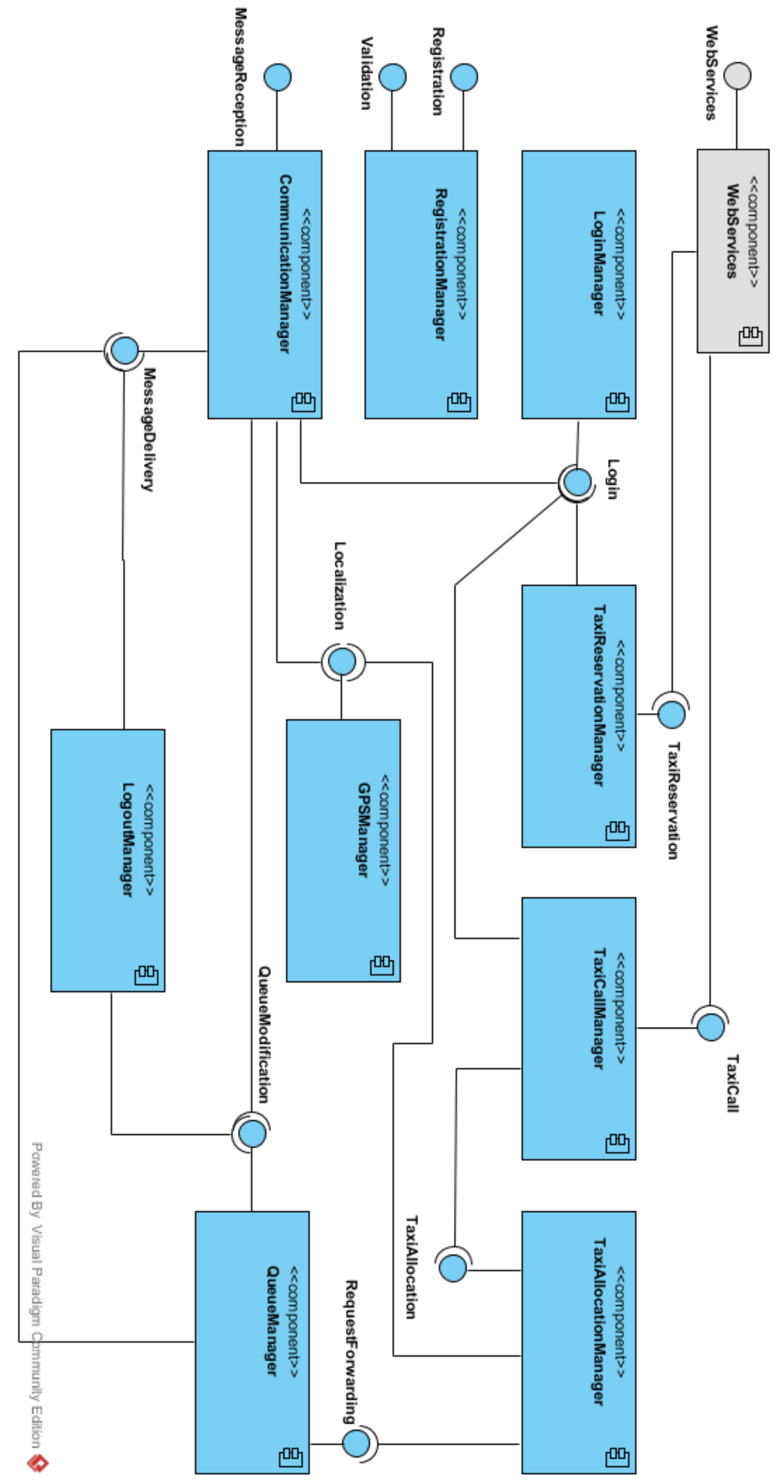
2.3 Component view

The business tier includes the following components:

- **Login manager**: handles all login requests. It distinguishes passenger logins from driver logins. The driver login function can be invoked remotely. The component checks the validity of user data, sets the new state accordingly and returns the result. It also provides a function to check whether a given user is already logged in.
- **Registration manager**: handles registrations of new passengers to the system. It can be divided into two sub-components. The first one generates validation codes and sends them (via SMS) through an external web service and checks the validity of submitted codes. The second one checks the validity of user data in a registration request, creates new users and returns the result to the invoker (the web tier).
- **Logout manager**: handles the automatic logout (due to timeout) of taxi drivers.
- **Taxi call manager**: handles all taxi calls. The component checks the validity of input data and forwards valid requests to the Taxi allocation manager. This component is also invoked by a public web service, in order to provide the taxi call function to external developers.
- **Taxi reservation manager**: handles all taxi reservations. The component checks the validity of input data and stores valid requests into the database. This component is also invoked by a public web service, in order to provide the taxi reservation function to external developers.
- **Taxi allocation manager**: processes call requests and periodically checks the database for reservations whose meeting time is in ten minutes. In both cases, forwards the request to the corresponding queue (in the queue manager).

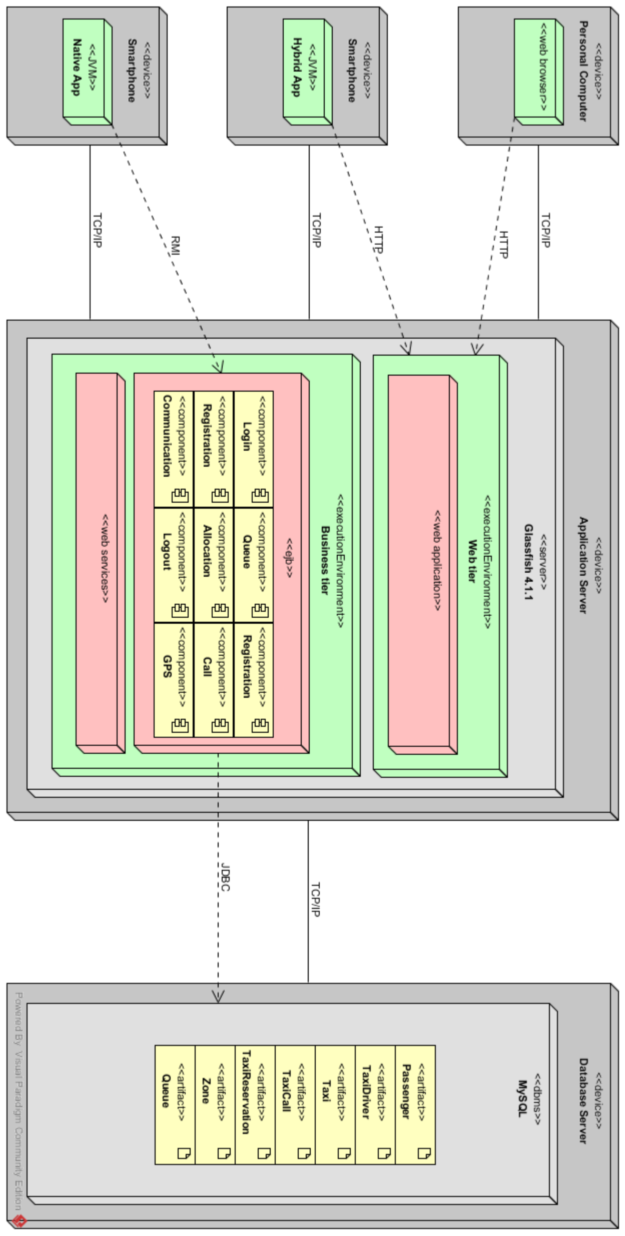
- **Queue manager**: manages all the queues. This component handles the insertion of a driver in a queue (after he sets his state to FREE) and his removal (after his state is set to BUSY). Moreover, it processes calls forwarding to drivers, according to the algorithm mentioned in the RASD document (see section 3 for more details).
- **Communication manager**: handles the message exchanges with the taxi drivers. The component receives messages and sends responses. It also sends asynchronous requests to the drivers. Thus, the overall architecture of this sub-system is p2p (see section 2.7 for further details).
- **GPS manager**: manages the communication with all taxi GPSs.

A component diagram is provided below. For a detailed description of the interfaces between components, see section 2.6.


WebServices
Registration
Validation
MessageReception
<<component>>
WebServices
<<component>>
LoginManager
<<component>>
RegistrationManager
<<component>>
CommunicationManager
MessageDelivery
Login
Localization
<<component>>
TaxiReservationManager
TaxiReservation
<<component>>
GPSManager
<<component>>
LogoutManager
<<component>>
TaxiCallManager
TaxiCall
QueueModification
TaxiAllocation
<<component>>
TaxiAllocationManager
RequestForwarding
<<component>>
QueueManager
Powered By Visual Paradigm Community Edition

2.4 Deployment view

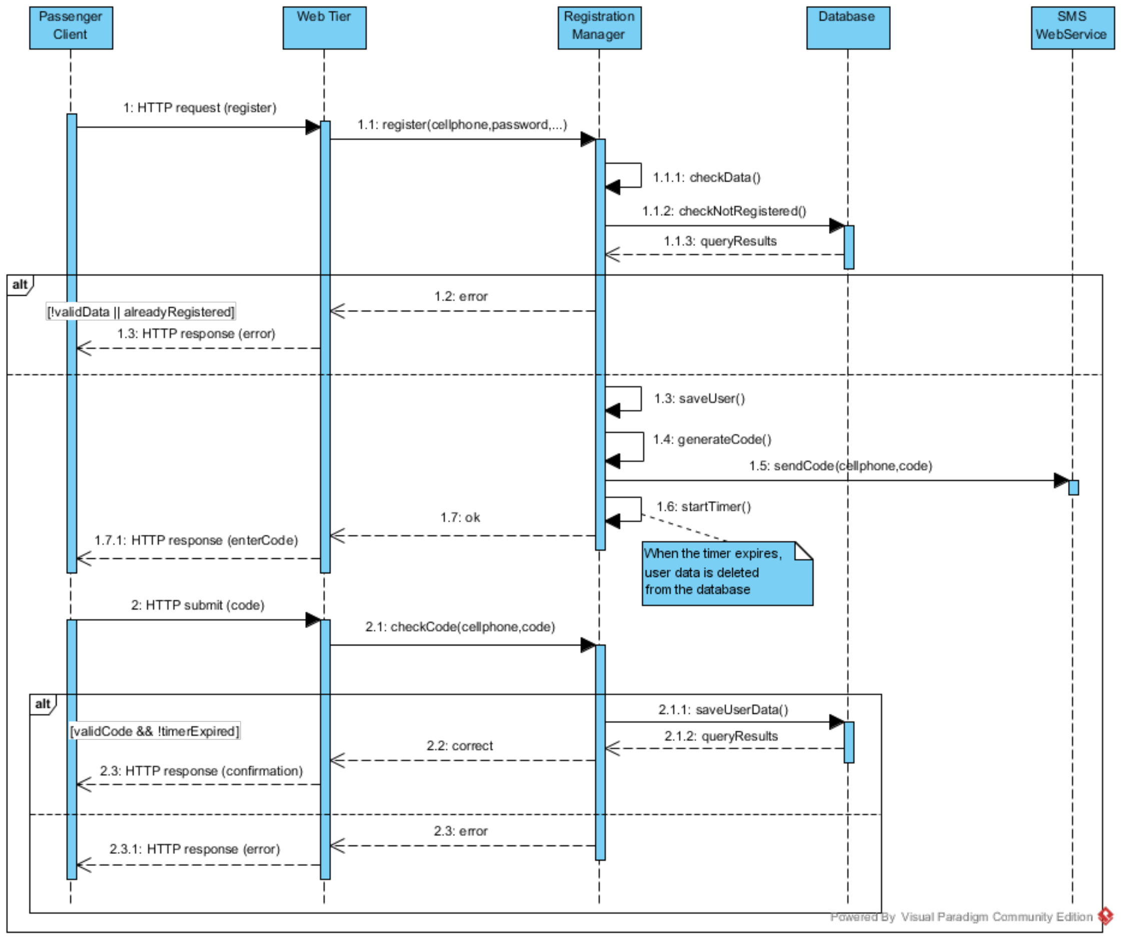
A deployment diagram is provided below.



2.5 Runtime view

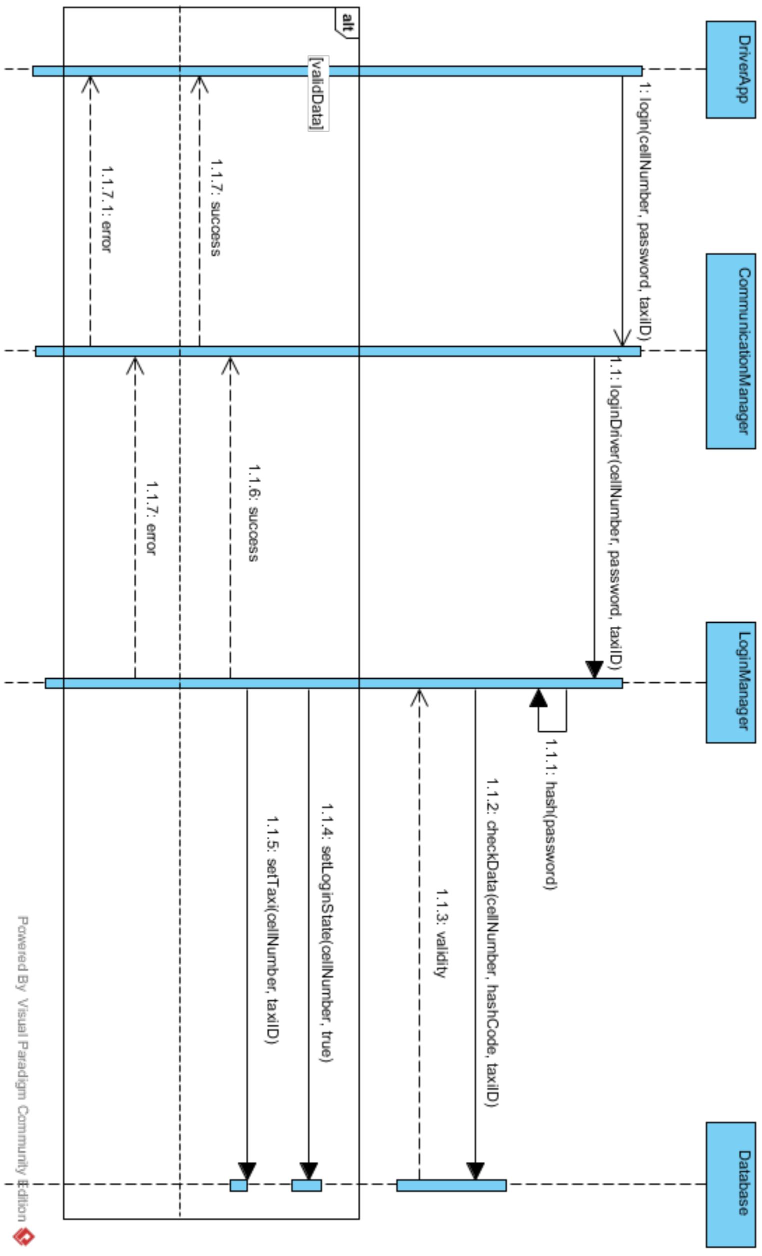
2.5.1 Registration

The following diagram shows the interaction between components in an attempt of registration.


Passenger Client
Web Tier
Registration Manager
Database
SMS WebService
1: HTTP request (register)
1.1: register(cellphone,password,...)
1.1.1: checkData()
1.1.2: checkNotRegistered()
1.1.3: queryResults
alt
1.2: error
[!validData || alreadyRegistered]
1.3: HTTP response (error)
1.3: saveUser()
1.4: generateCode()
1.5: sendCode(cellphone,code)
1.6: startTimer()
1.7: ok
1.7.1: HTTP response (enterCode)
When the timer expires, user data is deleted from the database
2: HTTP submit (code)
2.1: checkCode(cellphone,code)
alt
2.1.1: saveUserData()
[validCode && !timerExpired]
2.1.2: queryResults
2.2: correct
2.3: HTTP response (confirmation)
2.3: error
2.3.1: HTTP response (error)
Powered By Visual Paradigm Community Edition

2.5.2 Login

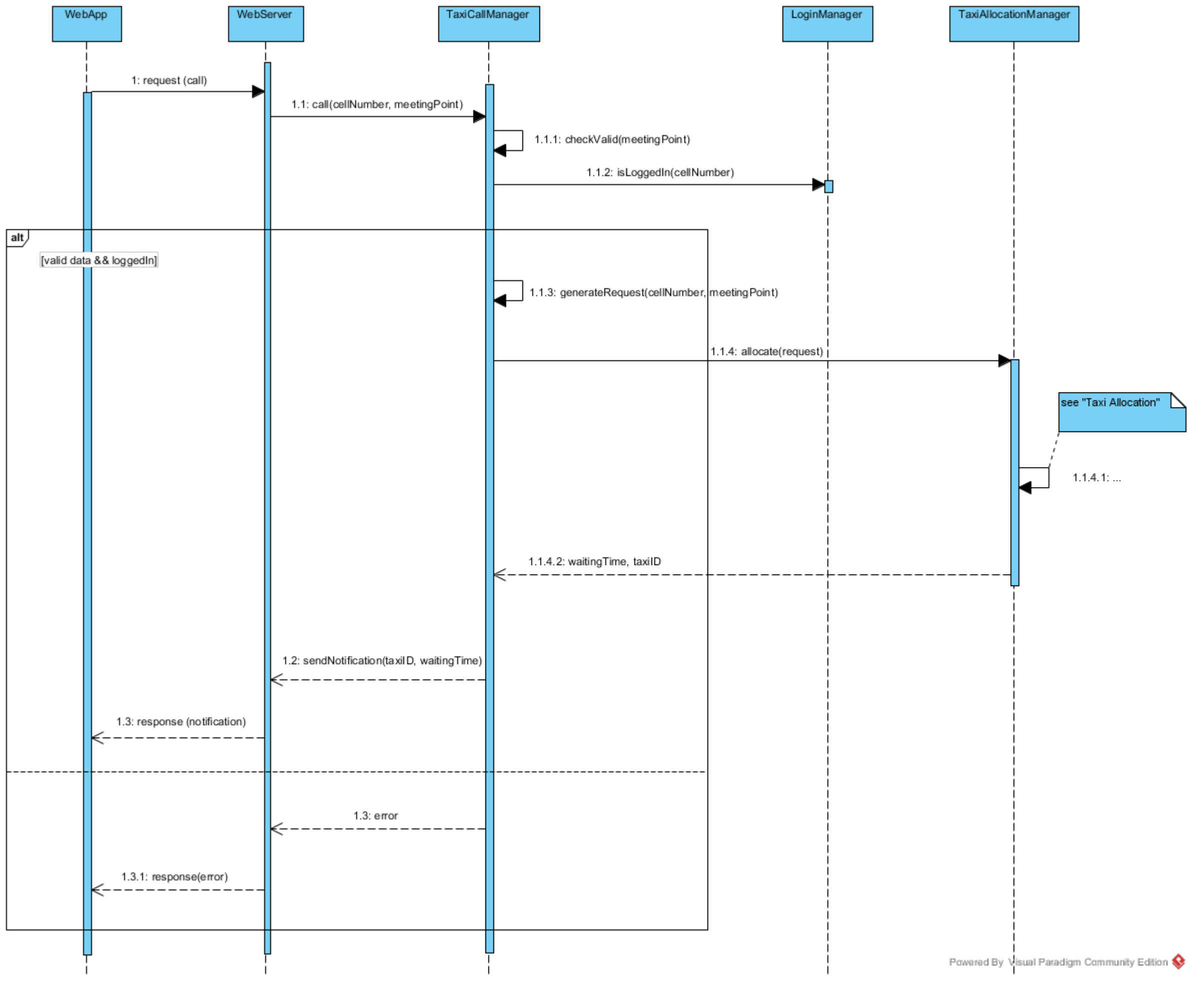
The following diagram shows the interaction between components in an attempt of login performed by a taxi driver.



DriverApp
CommunicationManager
LoginManager
Database
alt
[validData]
1: login(cellNumber, password, taxiID)
1.1: loginDriver(cellNumber, password, taxiID)
1.1.1: hash(password)
1.1.2: checkData(cellNumber, hashCode, taxiID)
1.1.3: validity
1.1.4: setLoginState(cellNumber, true)
1.1.5: setTaxi(cellNumber, taxiID)
1.1.6: success
1.1.7: success
1.1.7: error
1.1.7.1: error
Powered By Visual Paradigm Community Edition

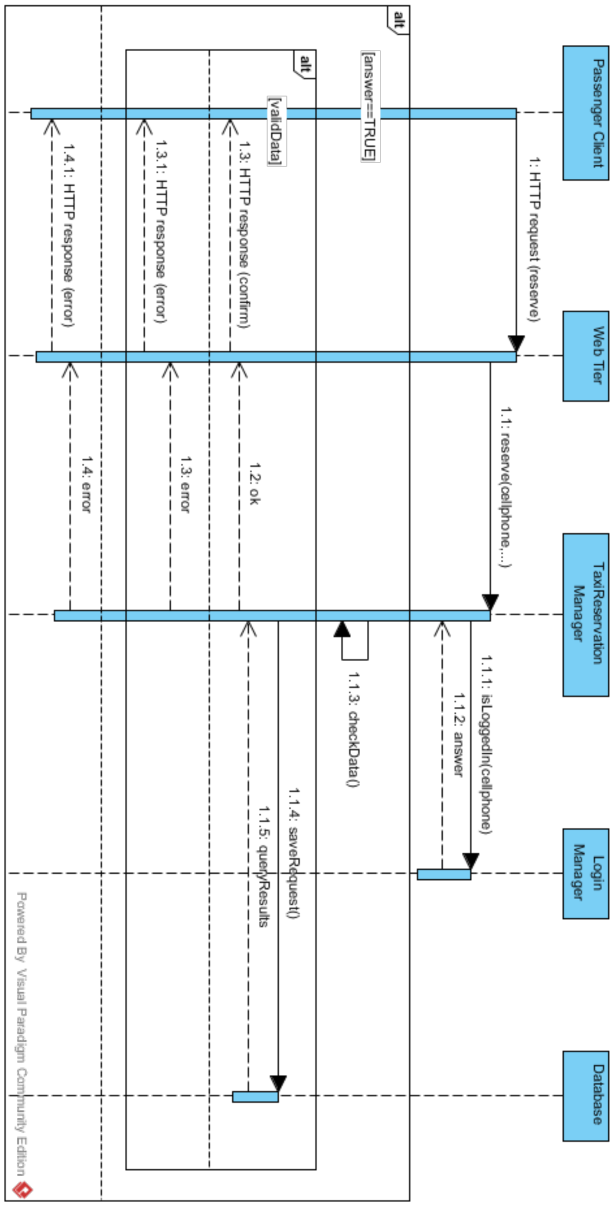
2.5.3 Taxi Call

The following diagram shows the interaction between components in a taxi call.


WebApp
WebServer
TaxiCallManager
LoginManager
TaxiAllocationManager
1: request (call)
1.1: call(cellNumber, meetingPoint)
1.1.1: checkValid(meetingPoint)
1.1.2: isLoggedIn(cellNumber)
alt
[valid data && loggedIn]
1.1.3: generateRequest(cellNumber, meetingPoint)
1.1.4: allocate(request)
see "Taxi Allocation"
1.1.4.1: ...
1.1.4.2: waitingTime, taxiID
1.2: sendNotification(taxiID, waitingTime)
1.3: response (notification)
1.3: error
1.3.1: response(error)
Powered By Visual Paradigm Community Edition

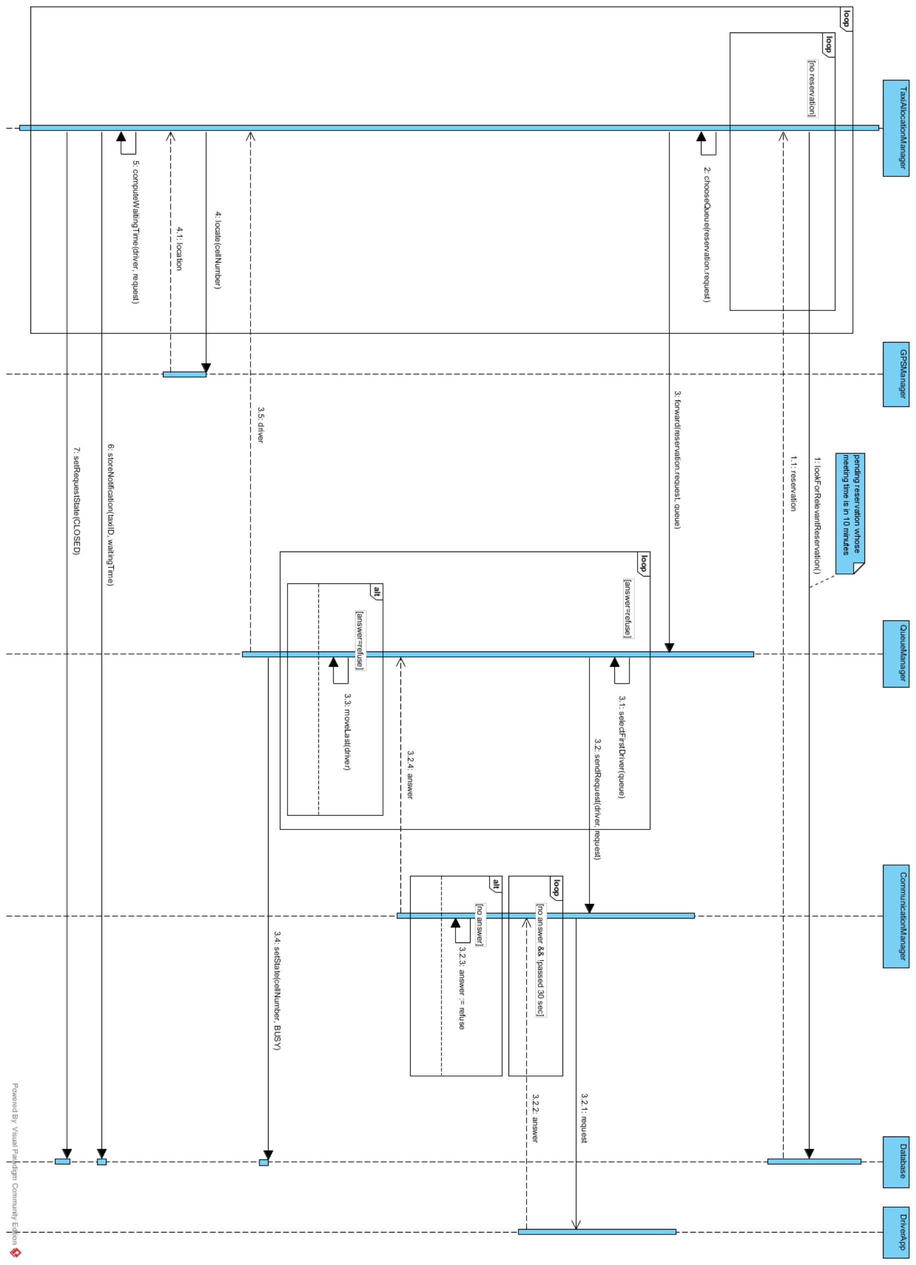
2.5.4 Taxi Reservation

The following diagram shows the interaction between components in a taxi reservation.


Passenger Client
Web Tier
TaxiReservation Manager
Login Manager
Database
alt
[answer==TRUE]
alt
[validData]
1: HTTP request (reserve)
1.1: reserve(cellphone,...)
1.1.1: isLoggedIn(cellphone)
1.1.2: answer
1.1.3: checkData()
1.1.4: saveRequest()
1.1.5: queryResults
1.2: ok
1.3: HTTP response (confirm)
1.3: error
1.3.1: HTTP response (error)
1.4: error
1.4.1: HTTP response (error)
Powered By Visual Paradigm Community Edition

2.5.5 Taxi Allocation

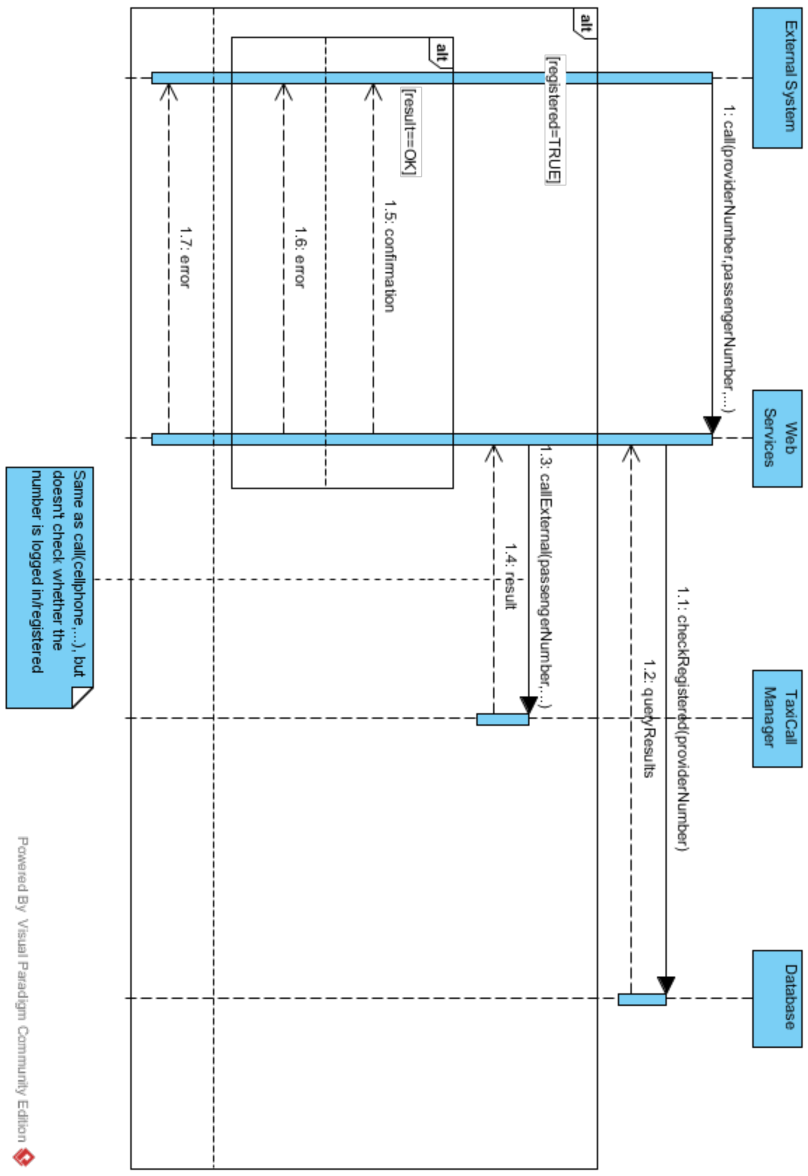
The following diagram shows the interaction between components in a taxi allocation.



TaxiAllocationManager
GPSManager
QueueManager
CommunicationManager
Database
DriverApp
loop
loop
[no reservation]
1: lookForRelevantReservation()
1.1: reservation
pending reservation whose meeting time is in 10 minutes
2: chooseQueue(reservation.request)
3: forward(reservation.request, queue)
loop
[answer!=refuse]
3.1: selectFirstDriver(queue)
3.2: sendRequest(driver, request)
loop
[no answer && !passed 30 sec]
3.2.1: request
3.2.2: answer
alt
[no answer]
3.2.3: answer := refuse
3.2.4: answer
alt
[answer==refuse]
3.3: moveLast(driver)
3.4: setState(cellNumber, BUSY)
3.5: driver
4: locate(cellNumber)
4.1: location
5: computeWaitingTime(driver, request)
6: storeNotification(taxiID, waitingTime)
7: setRequestState(CLOSED)
Powered By Visual Paradigm Community Edition

2.5.6 Taxi call through the web service

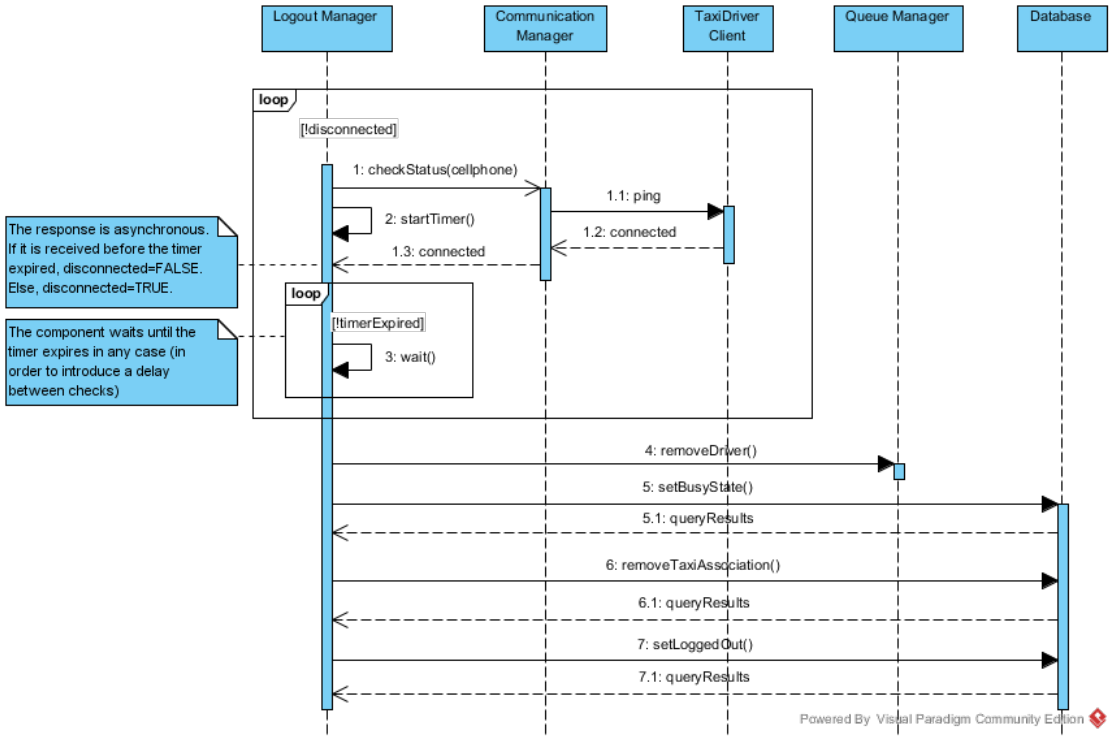
The following diagram shows the interaction between components when a taxi call is performed through the web service.



External System
Web Services
TaxiCall Manager
Database
1: call(providerNumber,passengerNumber,...)
1.1: checkRegistered(providerNumber)
1.2: queryResults
alt
[registered=TRUE]
1.3: callExternal(passengerNumber,...)
1.4: result
alt
[result==OK]
1.5: confirmation
1.6: error
1.7: error
Same as call(cellphone,...), but doesn't check whether the number is logged in/registered
Powered By Visual Paradigm Community Edition

2.5.7 Automatic Logout

The following diagram shows how the Logout Manager performs the automatic logout.


Logout Manager
Communication Manager
TaxiDriver Client
Queue Manager
Database
loop
[!disconnected]
1: checkStatus(cellphone)
1.1: ping
2: startTimer()
1.2: connected
1.3: connected
The response is asynchronous. If it is received before the timer expired, disconnected=FALSE. Else, disconnected=TRUE.
loop
[!timerExpired]
3: wait()
The component waits until the timer expires in any case (in order to introduce a delay between checks)
4: removeDriver()
5: setBusyState()
5.1: queryResults
6: removeTaxiAssociation()
6.1: queryResults
7: setLoggedOut()
7.1: queryResults
Powered By Visual Paradigm Community Edition

2.6 Component interfaces

Login Manager

Login

- loginPassenger(): performs passenger login. INPUT: cell phone number and password. OUTPUT: the result of the operation (either success or error).
- loginDriver(): performs driver login. INPUT: cell phone number, taxi ID, password. OUTPUT: the result of the operation (either success or error).
- isLoggedIn(): checks whether a given user is already logged in. INPUT: cell phone number. OUTPUT: either true (the user is already logged in) or false (vice-versa).

Registration Manager

Registration

- register(): performs passenger registration. INPUT: cell phone number, password, name, surname. OUTPUT: the result of the operation (either success or error).

Validation

- generateCode(): generates a random validation code. INPUT: none. OUTPUT: the generated code.
- checkCode(): checks whether the input code corresponds to the one generated for the given passenger. INPUT: cell phone number, code. OUTPUT: either true (the code is valid) or false (vice-versa).

TaxiCall Manager

TaxiCall

- call(): performs a taxi call operation. INPUT: cell phone number, meeting point. OUTPUT: the result of the operation (either success or error).
- callExternal(): performs a taxi call operation invoked through the web service (does not check whether the passenger is already registered). INPUT: passenger phone number, meeting point. OUTPUT: the result of the operation (either success or error).

TaxiReservation Manager

TaxiReservation

- reserve(): performs a taxi reservation. INPUT: cell phone number, meeting point, meeting time, destination. OUTPUT: the result of the operation (either success or error).
- reserveExternal(): performs a taxi reservation invoked through the web service (does not check whether the passenger is already registered). INPUT: passenger phone number, meeting point, meeting time, destination. OUTPUT: the result of the operation (either success or error).

TaxiAllocation Manager

TaxiAllocation

- allocate(): forwards a request to the corresponding queue. INPUT: request. OUTPUT: none.

Queue Manager

QueueModification

- addDriver(): adds the given driver to the specified queue. INPUT: cell phone number, queue. OUTPUT: the result of the operation (either success or error).
- removeDriver(): removes the given driver from the specified queue. INPUT: cell phone number, queue. OUTPUT: the result of the operation (either success or error).

RequestForwarding

- forward(): forwards the given request to the drivers in the given queue according to the already mentioned algorithm. INPUT: request, queue. OUTPUT: none.

GPS Manager

Localization

- localize(): localizes a taxi given its driver. INPUT: cell phone number. OUTPUT: either the taxi location or an error (if the taxi driver is currently not driving).

Communication Manager

MessageReception

- setFree(): invoked remotely by drivers. Forwards the request to the corresponding queue (computed through GPS Manager). INPUT: cell phone number. OUTPUT: the result of the operation (either success or error).
- setBusy(): invoked remotely by drivers. Forwards the request to the corresponding queue (computed through GPS Manager). INPUT: cell phone number. OUTPUT: the result of the operation (either success or error).
- login(): invoked remotely by drivers. Forwards the request to the Login Manager. INPUT: cell phone number, password. OUTPUT: the result of the operation (either success or error).

MessageDelivery

- sendRequest(): sends the given requests to the specified drivers and waits for a response. INPUT: cell phone number, request. OUTPUT: the driver's answer.
- sendQueuePosition(): sends the current queue position to the given driver. INPUT: cell phone number, queue position. OUTPUT: none.
- checkStatus(): checks whether the given driver is still connected. INPUT: cell phone number. OUTPUT: either true (the driver replied) or false (no response was received).

Public Web Services

WebService

- callService(): processes a taxi call request from external systems. INPUT: the developer's cell phone number, the caller's cell phone number, the meeting point. OUTPUT: the result of the operation (either success or error).
- reserveService(): processes a taxi reservation request from external systems. INPUT: the developer's cell phone number, the caller's cell phone number, the meeting point, the meeting time, the destination. OUTPUT: the result of the operation (either success or error).

2.7 Selected architectural styles and patterns

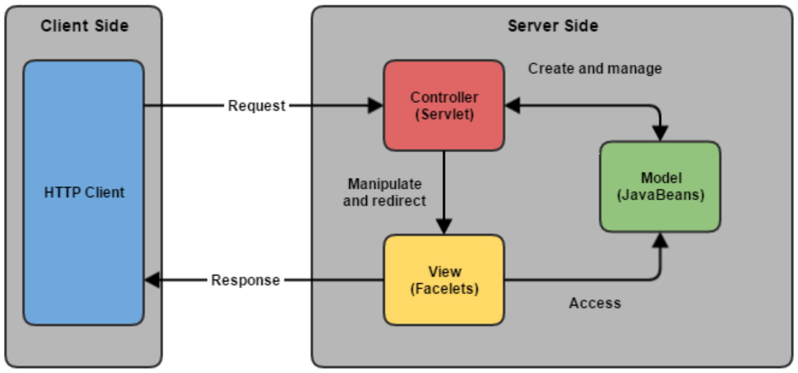
Considering that different types of client need different communication paradigms, the system is designed according to two different architectural styles:

- Client-Server: it is used for the communication with passenger clients (i.e., the web application and the hybrid mobile app). Clients make HTTP requests to the web tier. The latter processes them, invokes components of the business tier and finally produces responses. In this case, the server never opens a connection with the client by itself. Thus, the communication paradigm is a simple request-response.

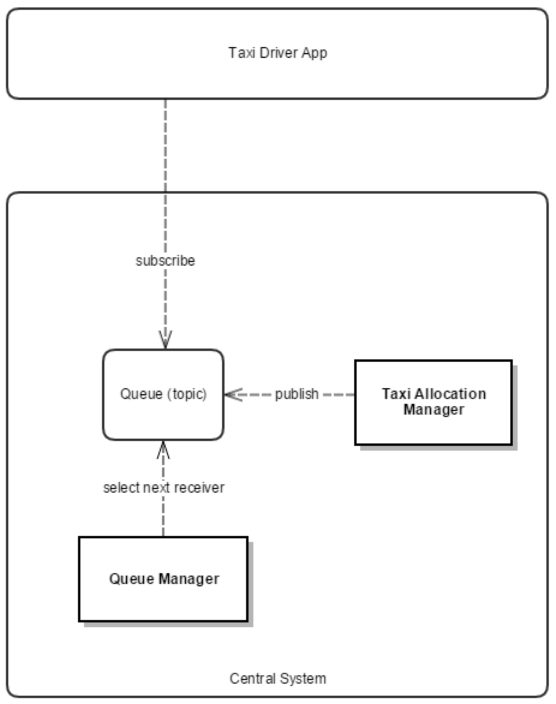
- Peer-to-peer: it is used for the communication with driver apps. In this case, both the server and the client can open a connection with each other. The server sends asynchronous messages to the clients (e.g. taxi requests) and, vice-versa, the client sends asynchronous messages to the server (e.g. set FREE or set BUSY). Thus, no server role is defined and the overall style is p2p (even though the clients do not communicate with each other).

The following architectural patterns have been considered:

- MVC: the client-server part of the system is designed according to the MVC pattern. JavaServer Faces, indeed, already implements this pattern. The view part is constituted by a set of facelets, whereas the model part is constituted by the java beans. The mediation between this two parts is managed by the faces servlet, which represents the controller of the MVC pattern. A better explanatory picture is provided below.



- Publisher-Subscriber: the p2p part design is inspired by the publisher-subscriber pattern. In fact, when a taxi driver sets his state to FREE, he is subscribing to the queue of his current zone. When the system allocates a taxi, it sends the request to the drivers in the corresponding queue. However, the system forwards the request to one taxi at a time and in a precise order, according to the already mentioned algorithm. Thus, this is just a variant of publisher-subscriber, in which the messages are not broadcasted and the system must know the identity of each subscriber. A diagram illustrating this pattern is provided below.



2.8 Other design decisions

The system provides APIs to external developers in the form of public web services. This choice has been made for several reasons:

- accessibility: web services are platform independent and they can be accessed by using standard protocols (HTTP in this case);
- extensibility: it is easy to add new web services that call the business tier functions;
- compatibility: the Glassfish application server directly supports web services.
- security: external developers must be registered user (as normal passengers) and the responsibility of every request is on them. Thus, security is guaranteed.

3 Algorithm Design

We provide in this section a description of the most relevant algorithms used by our application.

3.1 Queue management in taxi allocation

We provide here a detailed description of the already mentioned algorithm for the allocation of a taxi. Let us first define the following classes:

- Request: represents a request to a driver;
- Queue: represents a queue of Taxi. It provides the functions: enqueue(Taxi), which adds a Taxi object to the last position in the queue, dequeue(), which removes the first Taxi in the queue and returns it;
- Taxi: an object representing a single taxi.
- TaxiDriver: represents a taxi driver.

Let us define the following symbols:

- r: a Request object, representing the request to be forwarded;
- Q: a Queue object, representing the queue in which the request is going to be forwarded;
- response: a variable which contains the driver response at each cycle;
- nextTaxi: a variable containing the next taxi to which the request is going to be forwarded;

The pseudocode is the following:

```
forward (r, Q)
while(response == REFUSE)
    nextTaxi = Q.dequeue()
    response = CommunicationManager.sendRequest(nextTaxi.driver.cellphone, r)
    if(response == REFUSE)
        Q.enqueue(nextTaxi)
```

3.2 Waiting time estimation

We provide now a detailed description of the algorithm the application uses for estimating the waiting time for a taxi. The city is modeled as a graph, in which each node represents a cross and each arc represents a road connecting two crosses (of course, the graph is directed). To each node is associated a couple of coordinates. To each arc is associated a cost, computed as follows:

$$cost(i \rightarrow j) = \frac{distance(i \rightarrow j)}{averageSpeed} + trafficFactor(i, j)$$

In which:

- averageSpeed is the average speed of a taxi (estimated to be 40km/h);
- distance(i,j) is the length of the road connecting crosses i and j (obtained through Google maps services);
- trafficFactor(i,j) is a delay depending on the traffic condition in the area containing i and j (obtained through an external service of traffic estimation for Milan);

The TaxiAllocation Manager maintains this graph and periodically updates it. Let us define the following symbols:

- d: a TaxiDriver object representing the driver who accepted the request;
- r: a Request object representing the request for which the waiting time is going to be computed;
- G: the city graph;
- getUpdatedSubGraph(location): returns a subgraph of the last updated graph, including only the nodes in the same zone as the given location and the ones in adjacent zones;
- nearestNode(location): returns the nearest node in G to the specified location;
- s: the node in which the driver is located;
- t: the node in which the meeting point is located;
- S: a variable storing the nodes computed during Dijkstra's algorithm;
- d: a vector, of size the number of nodes, containing the cost of the minimum path from s to each node;

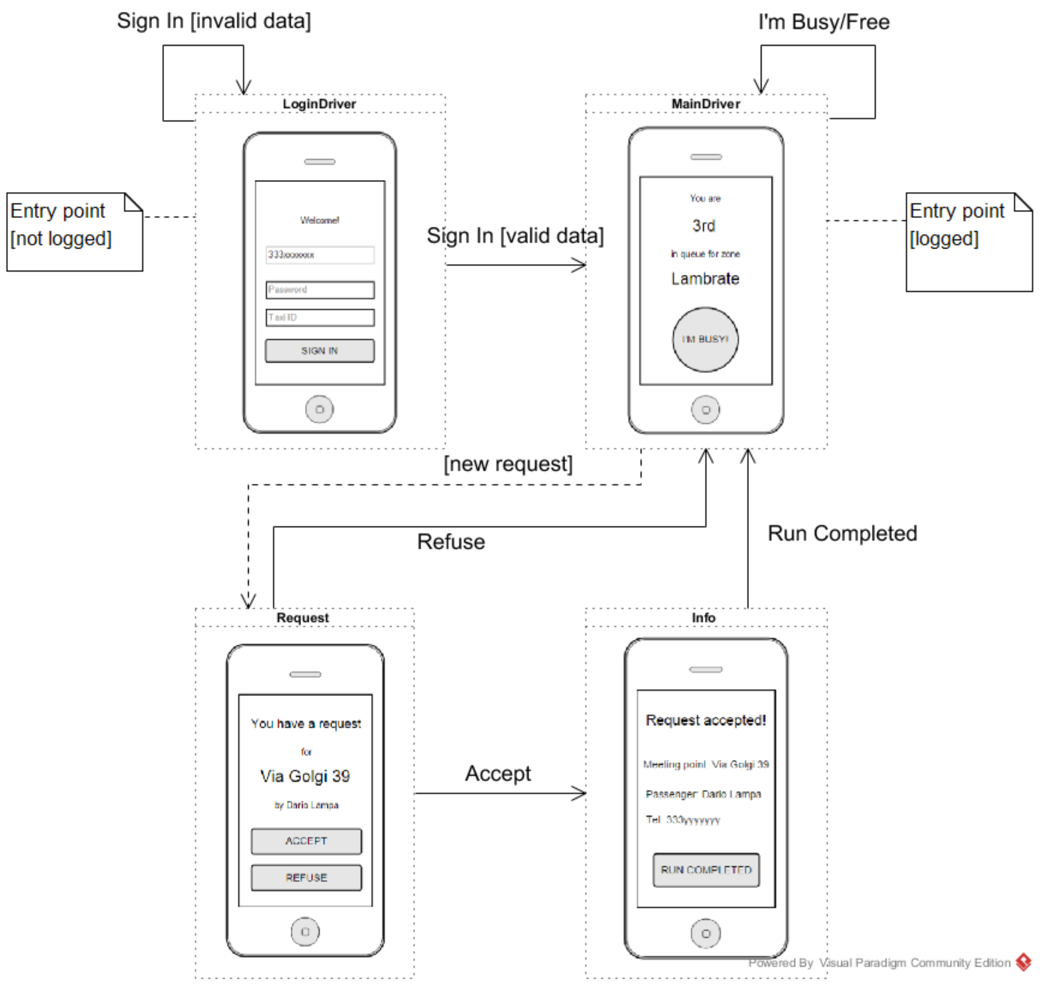
The algorithm proposed uses Dijkstra's algorithm to find the minimum path on the graph from the driver location to the meeting point and returns its cost (which is an estimation of the waiting time).

```
computeWaitingTime(d, r)
    G = getUpdatedSubGraph(d.taxi.location)
    s = nearestNode(G, d.taxi.location)
    t = nearestNode(G, r.meetingPoint)
    S = {s}
    d[s] = 0
    while(S!=G.nodes)
        find (i→j) s.t. i∈ S, j ∉ S that minimizes d[i]+cost(i→j)
        d[j] = d[i] + cost(i→j)
        S = S ∪{j}
    return d[t]
```

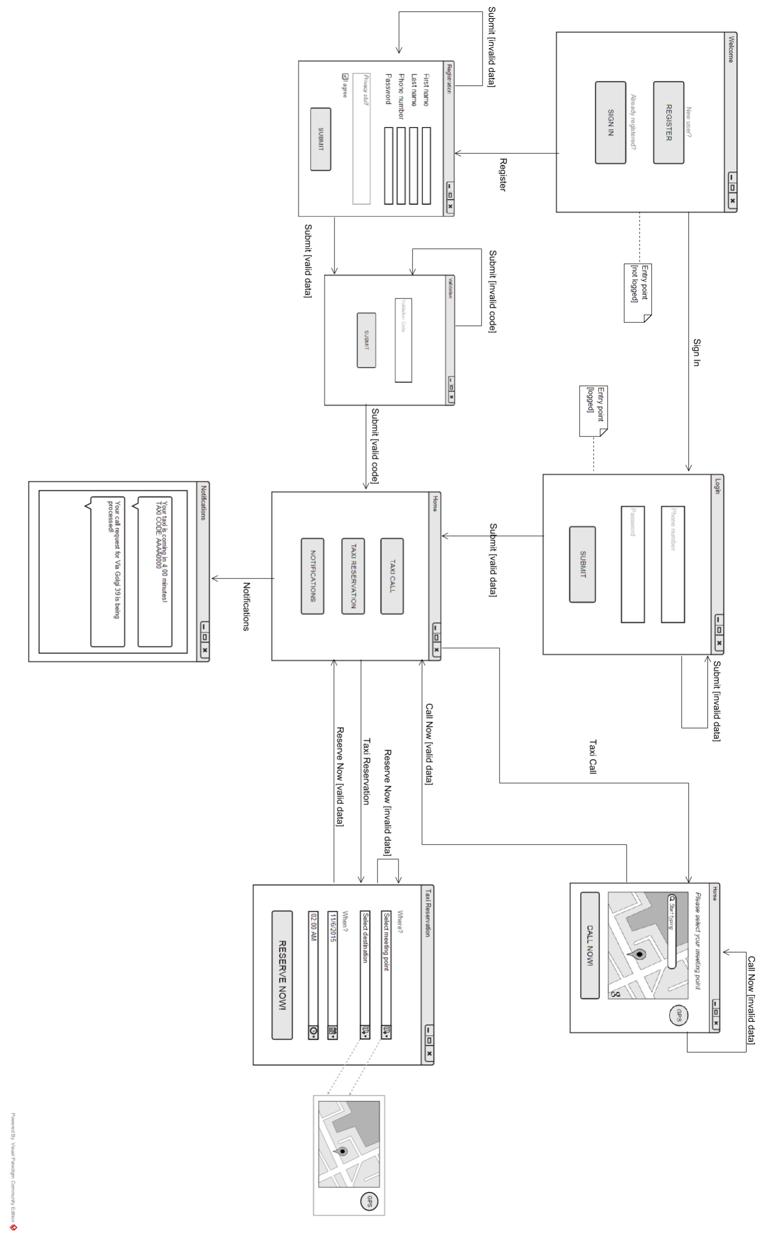
4 User Interface Design

For a complete description of user interfaces (with mockups), see the RASD document. In this section, we provide a graphical overview of the connections between screens.

4.0.1 Driver Interface



4.0.2 Passenger Interface



5 Requirements Traceability

The requirements defined in the RASD document are mapped to the components of the proposed architecture as follows:

Requirement	Components	Comments
[R1] Every taxi call to the central system is immediately forwarded to the queue associated to the zone containing the required meeting point.	TaxiCallManager, TaxiAllocation Manager	
[R2] For each reservation, a request is sent to the queue associated to the zone containing the required meeting point ten minutes before the meeting time.	TaxiAllocation Manager	
[R3] A request from the central system to taxi drivers always includes the meeting point, the passenger's name and her phone number.	Queue Manager, Communication Manager	
[R4] For each taxi allocation, the central system calculates the maximum waiting time based on the taxi GPS location and the meeting point.	TaxiAllocation Manager, GPS Manager	
[R5] Each time a taxi is allocated, the passenger is notified with the maximum waiting time **[R6]** Each time a taxi is allocated, the passenger is notified with the incoming taxi ID.	TaxiAllocation Manager	The notification is saved in the database and eventually retrieved after the next user HTTP request.
[R7] When a taxi driver sets her state to free, her taxi ID is inserted in the last position of the queue corresponding to her current location.	Communication Manager, Queue Manager	
[R8] When a request arrives at a queue, it is forwarded in order to the drivers of the taxis in the queue, beginning from the rst position, until someone accepts.	Queue Manager	
[R9] When a taxi driver accepts a request, her state is set to busy.	Queue Manager	

[R10] When the state of a taxi driver turns to busy, her taxi is deleted from the queue corresponding to her current location.	Communication Manager, Queue Manager, Logout Manager	The state of a driver is turned to busy either manually or automatically (following a disconnection).
[R11] When a taxi driver refuses a request, her taxi is moved in the last position of the queue corresponding to her current location.	Queue Manager	
[R12] When a taxi driver logs out from the central system, she is set to busy.	Logout Manager	
[R13] Each reservation request is accepted only if the submit time is at least two hours before the meeting time.	Reservation Manager	
[R14] The system provides public APIs for the call and reservation functions.	Public Web Services	
[R15] A request is automatically refused if the taxi driver does not accept within 30 seconds.	Communication Manager	
[R16] A taxi driver, whose application has not responded to the system solicitations for more than two minutes, is automatically logged out from the central system.	Logout Manager	
[R17] Every time a driver logs in, the application stores an association between her and the taxi she has specified.	Login Manager	
[R18] Every time a driver logs out, the association with her taxi is deleted.	Logout Manager	
[R19] The application rejects all driver login attempts in which the specified taxi is already associated to another driver.	Login Manager	

6 Appendix

6.1 Spent Hours

- Andrea Tirinzoni: ~22h
- Matteo Papini: ~22h