Design Document

for

Taxi Management System

Version 1.2

Prepared by Giuseppe Di Francesco & Domenico Iezzi

Summary

INTRODUCTION

A - Purpose

This software design document describes the architecture and system design of “Taxi Management System”, providing a complete overview of the architecture of the system, which functionalities and requirements are described in the R.A.S.D. It provides also an initial presentation of the algorithm and a more in-depth overview of the user interface, in order to give a complete idea of how this software will look like.

B - Scope

The aim of the project is to build an application for requesting taxi in a big city. User can queue for a taxi using the web or the smartphone application, and the systems answers with the code for the incoming taxi along with the waiting time. The system divides queues one for each zone of the city, while each of the taxies is assigned to a specific zone thanks to its GPS coordinates. Taxi driver uses a mobile application to inform the system of their availability and to confirm that they are going to take care of a certain call. When users queue to the app in a certain zone, system forwards the request to all taxis waiting in that zone. If the first one confirms, system will send a confirmation to the user. Else, the request will be moved to the second taxi in queue, and the first will be moved in the last position of the queue. Users can also reserve a taxi specifying origin and destination of their route, only if they do that at least two hours before the ride.

C - Reference Material

R.A.S.D. contains the requirements and the user interface ideas used to write this document.

D - Document Structure

This document is divided into three main parts:

- **Architectural design:** describes the architecture of the system in a high-level description. It also provides information about views and styles used during development.

- **Algorithm design:** contains some relevant algorithms in the project

- **User interface design:** this part continues the user interface ideas in the Non-functional requirements section of the R.A.S.D, providing detailed information on how the application will look like.

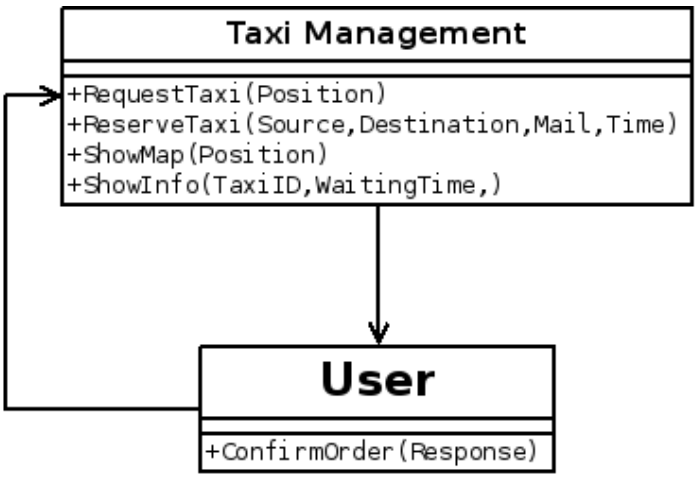
ARCHITECTURAL DESIGN

A - Overview

Here we will present the Architecture of our system, so we will analyze every component and their relations. We will talk about the choices we made about the architectural interfaces and styles, also motivating them.
Our choice is to use the **MEAN** (Mongo, Express, Angular, Node) framework to make our web applications. Then we will use **Cordova** to convert that web application into a smartphone application, adding some exclusive functions for notifications. This means that our application will be simply a client-server application, which is the best solution in our opinion.

B - High level component and their interaction

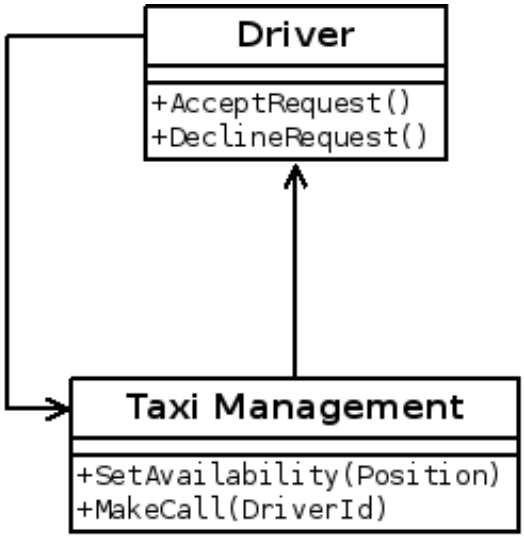
B.1 User



User interacts with the system mainly requesting a taxi, specifying the starting position, where the driver will pick him up. Once the system will compute the request and finds a taxi, it will notify the user about the taxi, sending informations about it. System will provide also a function to book a taxi for a specific path at a specific time, given source and destination position, and of course giving the time in which he will be picked up. User can give his mail to receive a confirmation by the system. There is also a function to retrieve the

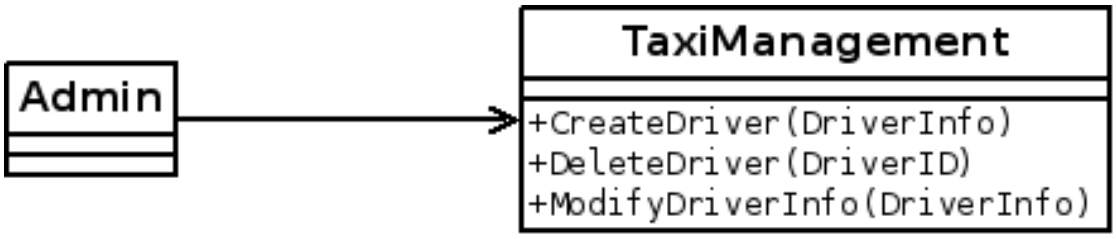
position of all available taxies, which will be shown on a map provided by a third-part API. Those functions will be implemented as asynchronous websocket messages, because in this way it will be more efficient, especially for the heavy “RequestTaxi” function.

B.2 Driver



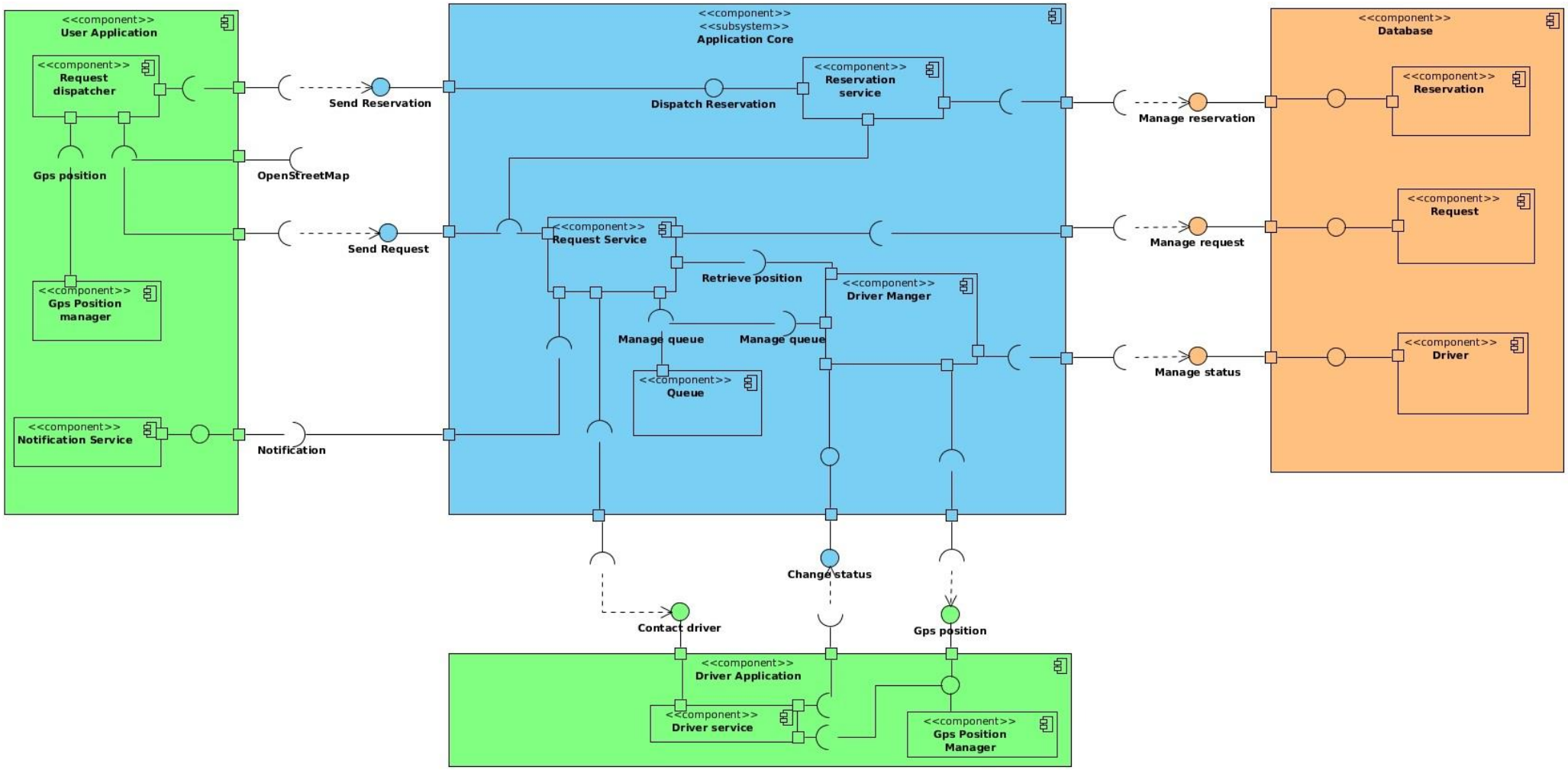
Driver can manage his availability through the terminal on his car, changing his status from available to not available or vice versa. The Client application will automatically give the system his taxi code. He should also be able to accept or decline requests coming from the system.

B.3 Admin



Administrator is capable of creating new Taxi entries in the system, modify their information and delete them through his control panel.

C – Component info



We identify two main subsystems, the **Application Core** and the **DBMS**. The **Application Core** manages URL and taxi requests, in fact it exposes *Send Request, Send Reservation, Change status* and it uses *Check availability, GPS, Change status, Manage status, Manage request* and *Manage reservation*. It has four subcomponents:

- **Request Service:** It is the service responsible of managing incoming client requests. It has the provided interface *Dispatch Request* . It uses the *Manage queue* interface, *Manage requests* of the database subsystem, *Contact Driver* interface provided by Driver Manager component and *Check availability* provided by the Driver Application.
- **Reservation Service:** It is responsible of managing client reservation and dispatch a reservation request some minutes before the scheduled time. It provides a *Dispatch reservation* interface and it uses the *Manage reservation* provided by the database subsystem.
- **Driver Manager:** it is responsible of managing driver status and the dispatching of requests to drivers. It provides *Contact Driver* interface, and it uses *Change status* provided by the Driver Application subsystem, *Manage status* interface provided by the database subsystem, *GPS* provided by the Driver Application and *Manage queue* provided by Request Service.
- **Queue:** it manages queue for every zone of the city. Provides the *Manage queue* interface used by Request Service and Driver Manager.

The **DBMS** subsystem has components that manages read write operation onto the database. It provides interfaces to manage data stored. These components are **Reservation, Request** and **Driver.**

As for the client side, there are Driver Application and User Application components:

- **Driver application:** it is the main application drivers access through the terminal in their vehicle. It provides *Notify* interface which will be used by the request service component of the core application subsystem to notify about requests, and the *GPS* interface used by the Driver Manager component. It uses the *Change status* interface provided by application core which will handle the driver status into the system and into the DB. There are 2 subcomponents: *Driver service* which contains main logic and the GPS position manager, and the *GPS position manager.*
- **User application:** it is the main logic for the user running in the web and mobile applications. It uses *Send request* and *send reservation interfaces* provided by the application core. It contains the GPS position manager component in order to retrieve the position and send it with the request. This component provides also the *Notify* interface, used buy the request manager for sending messages. There are *Gps position manager, notification service* and *request dispatcher* subcomponents.

D – Deployment View

To develop our application, as we anticipated before, we will use NodeJs server. We choose this solution because NodeJs does not have security issues, for similar applications it doesn't show performance issues, and is very easy to set up. It also has many security plugins, which we can add to our application in the future. Compared to JEE we will have maybe a little less reliability, but we will certainly gain speed in running our application, and also we will be able to develop it in a minor time, which means less money. One of the reasons for this last point is that with NodeJs we will use the same language, JavaScript, in both client and server.

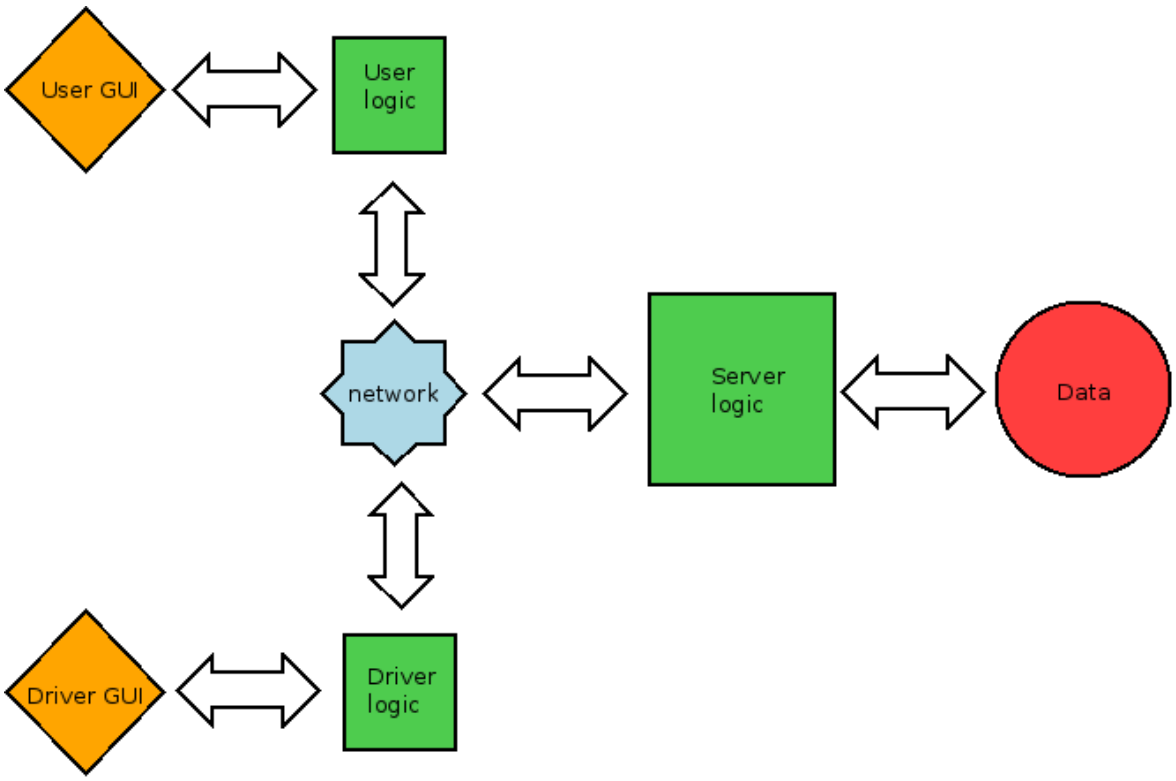
The request taxi function of our system needs a long computation by the main server, in order to contact every taxi in queue and wait for their response. So it would be better to handle all of the requests in an asynchronous way, without having to wait for the server response (we want our system to serve lots of request at the same time). Considering that NodeJs applications use non-blocking I/O calls in order to support many requests, we decided to adopt a two-tier architecture with a distributed logic, with a single machine running our main nodejs server and the DBMS service, and web/mobile application with AngularJs, which will handle the application logic.

- The server manages all HTTP request from clients. When a client connects, a websocket connection between client and server is established, and it will join a specific room on the socket server identified by his session id, to enable messages forwarding to user who made that specific request. We choose this technology to have a full-duplex communication between clients and server, and at the same time avoid latency and overhead introduced by HTTP requests in case of high network traffic. It holds also the taxi queues for every zone, along with interfaces to manage them.

- For the DBMS we choose MongoDB over classic SQL databases, because it integrates easily with NodeJs apps and it is easy to program with, allowing to develop application faster. Moreover, its flexible data model allows making easy changes to schemas if the model of our system will change in the future, and it can be scaled in case our system grows with multiple data centers.

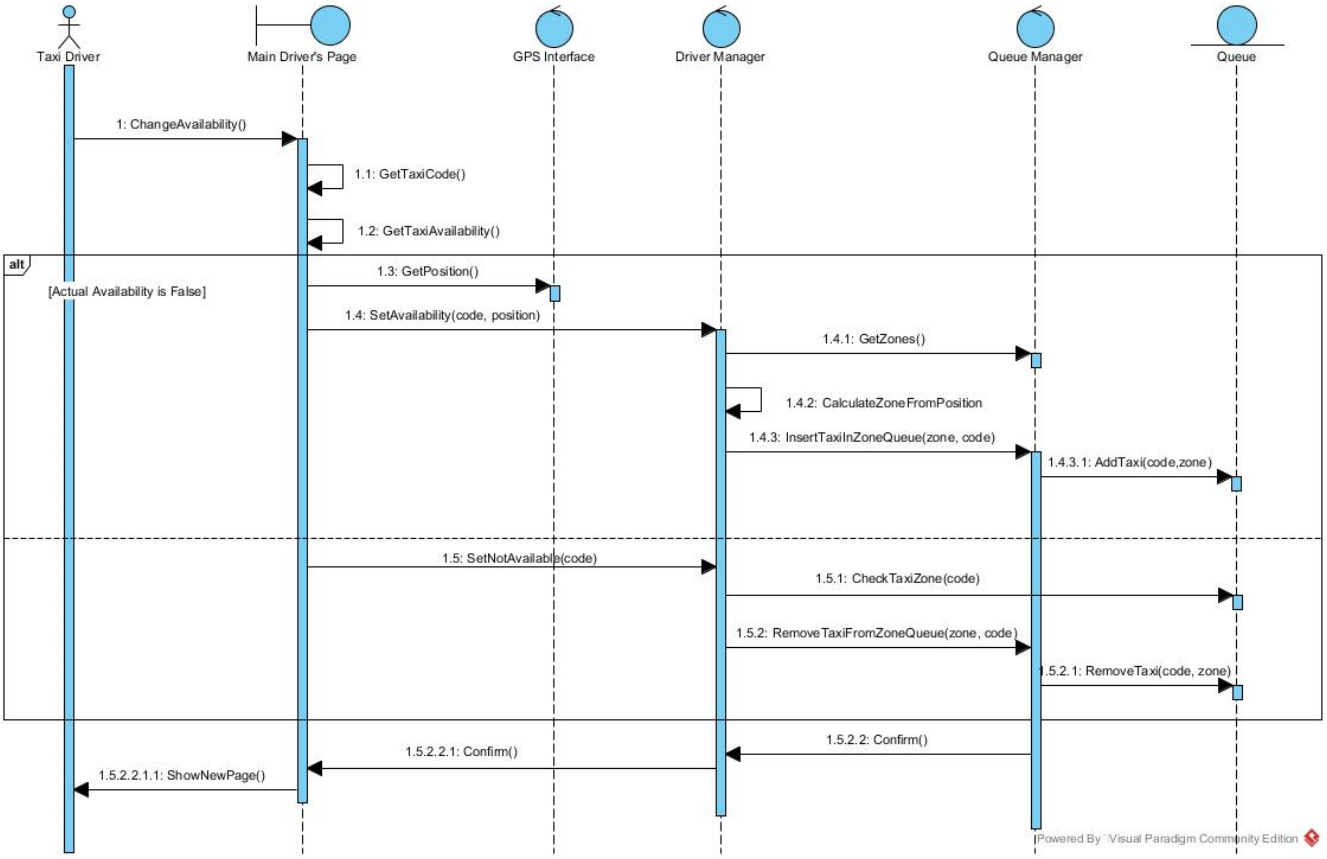
Web application will be built with AngularJS, using an MVC pattern in order to separate views from logic, and a Socket.IO for the web socket protocol implementation.

Then we will use Cordova to make our smartphone application. Apache Cordova is an open-source mobile development framework that allows you to use standard web technologies such as HTML5, CSS3, and JavaScript for cross-platform development, avoiding each mobile platforms' native development language. Applications execute within wrappers targeted to each platform, and rely on standards-compliant API bindings to access each device's sensors, data, and network status. We choose this solution because it is the fastest and more efficient way to convert a web app into a smartphone app.



E – Runtime View

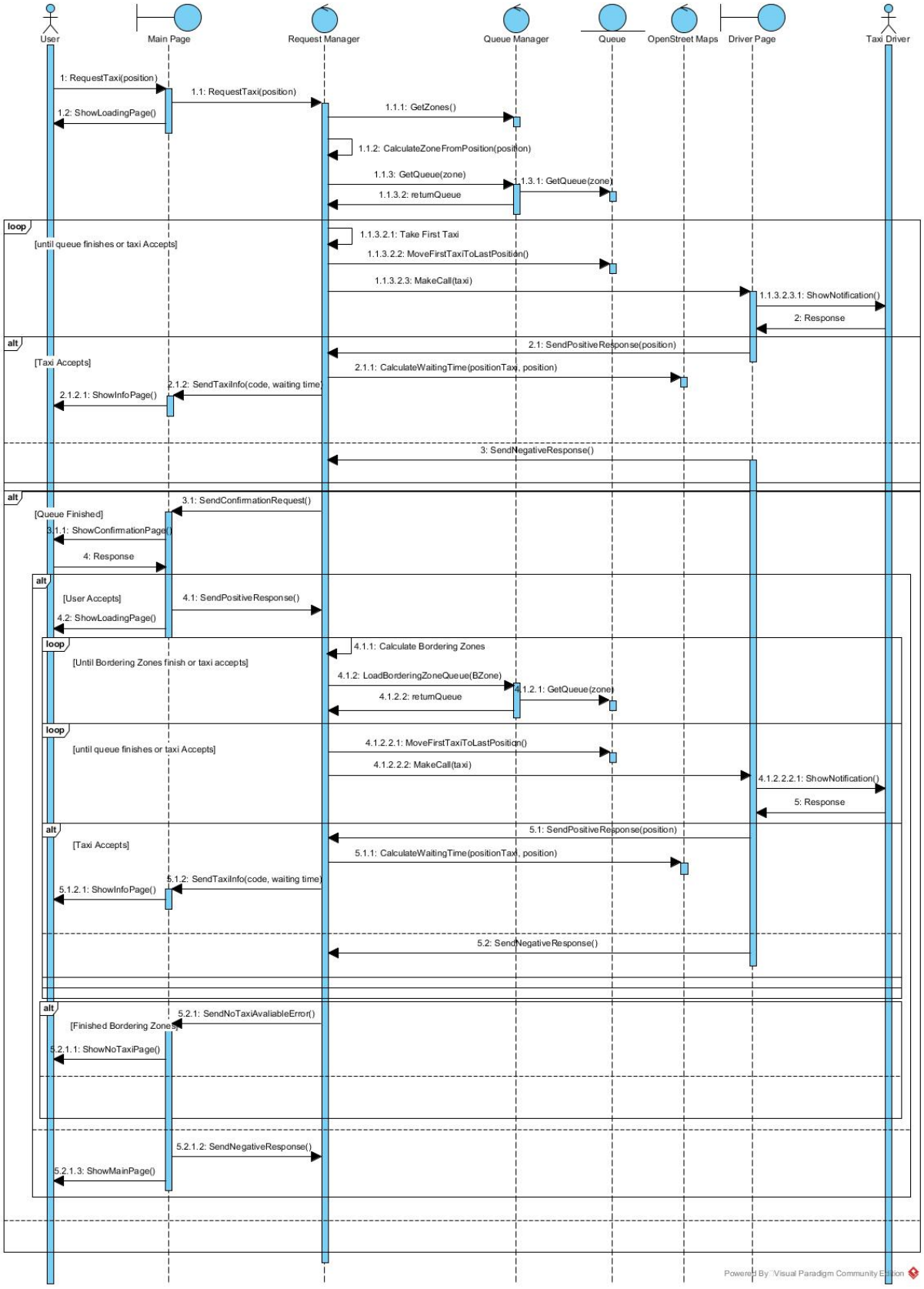
We will show the complete sequence diagrams that describes how the application components communicate to run the main functions. We will omit the sequence diagrams for the simplest activity, like driver login. Note that when the driver logs in or out, its availability is not changed, because he will set that after the login (and before the logout), so they are very simple operations and the R.A.S.D. version is enough to understand the runtime.
We also take into account the "Web App" version, so without considering the "Notification Service" of the Smartphone App, which means adding a simple service that communicates with the user page to create notifications.



E.1 Driver's Change Availability

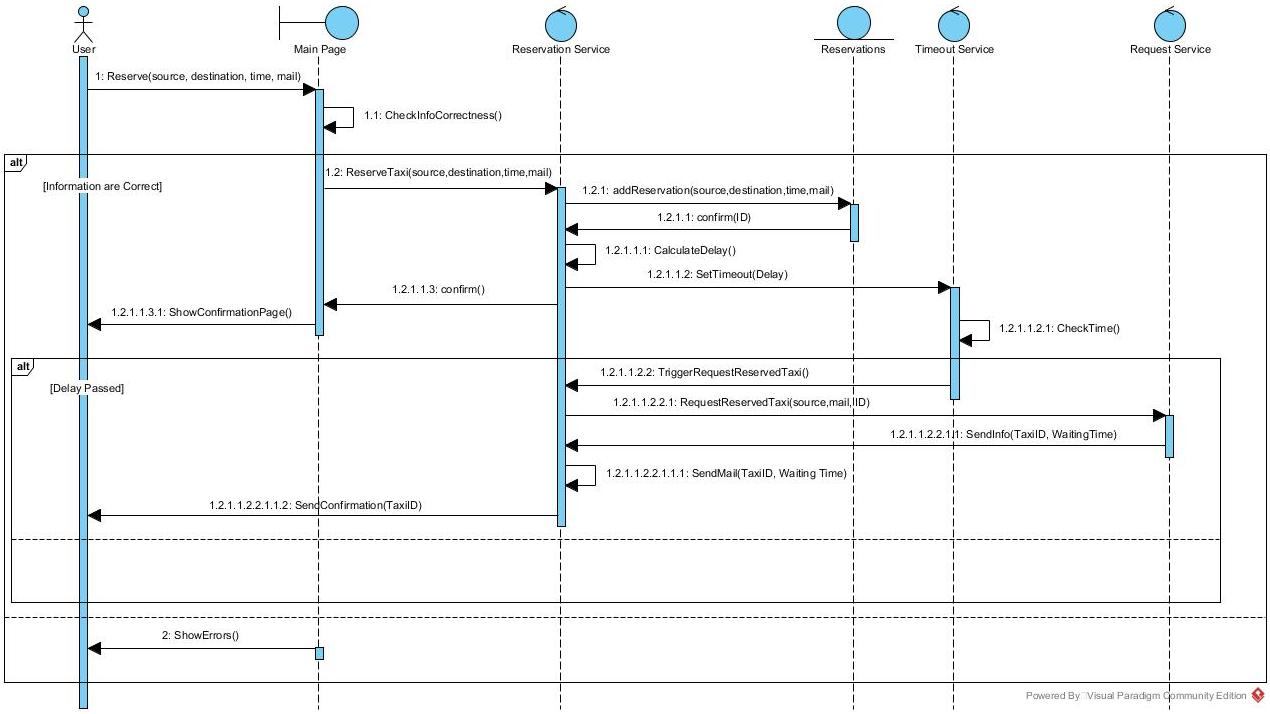
The Page in client side checks if the driver is actually available or not, and sets his new availability as the opposite. Therefore, the system will remove or add the taxi to his correct Queue. The Queue is also a Database entity, other than a system entity. If he turns on the availability, the system will localize his zone to insert him in that zone's queue.

E.2 Request taxi Diagram



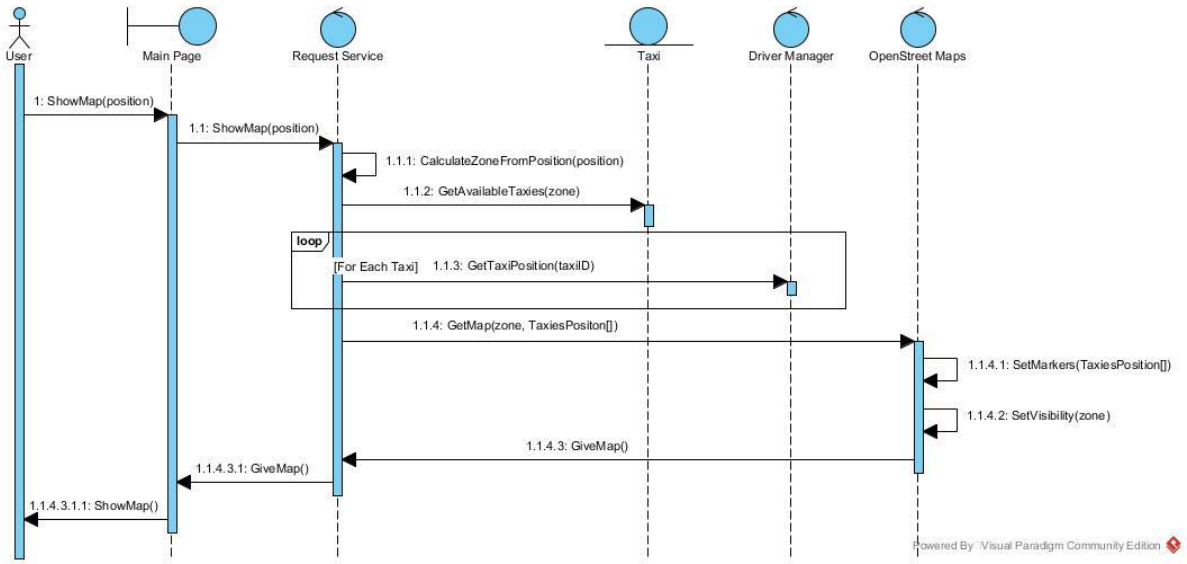
For a complete description of this diagram, see the algorithm design section.

E.3 Reserve Taxi Diagram



*Timeout service is our server's native service to trigger a function after a certain delay. "**RequestReservedTaxi**",is a function that we will not represent because it's very similar to "**ReserveTaxi**", but without the user interaction, and returns always a taxi. In the "**CheckInfoCorrectness**" there are all the checks on the user input, including the time of the reservation. In the algorithm section, this function will be presented in a more detailed way.*

E.4 Show Map Diagram



The Driver Manager will retrieve the Taxi Positions from their GPS, calling the taxi client.

F – Component Interfaces

Driver - Driver Service: Notify Interface

MakeCall (taxiId) - function to notify taxi specified by TaxiId about a request that he can accept or decline. When driver answers the call, a message is sent to the server to confirm driver's response.

Driver – Gps position manager: Gps position Interface

GetPosition() - returns an object representing position coordinates of the taxi, using the integrated GPS system built in the car's terminal.

User – Notification service: Notify Interface

SendTaxiInfo (code, waitingTime) - sends information about taxi that accepted request, along with the waiting time.

SendConfirmationRequest() - asks user for a confirmation of the request. After user accepts or declines, the response is sent to the server as a message.

SendNoTaxiAvailableError() - if system can't find any taxi available even in the bordering zones, it will call this function on the user page.

User – Gps position manager: Gps position Interface

GetPosition() - returns an object representing position coordinates of the user, using the integrated GPS system in his device or the HTML Geolocalization API.

Core – Reservation service: Dispatch reservation Interface

ReserveTaxi (source, destination, time, mail) – function to reserve a taxi from *source* to *destination.* It will save the time of the request and user mail for future notifications. It will trigger the RequestReservedTaxi function once the timeout has been reached. It will send to the client a message when the reservation has been successfully stored, and when a taxi has been found. When reservation cannot be processed, an error message is sent to the user.

Core – Request service: Dispatch Request Interface

RequestReservedTaxi(source, mail, ID, callback) - this function is called by the Reservation Service when the timeout of the reservation is finished. It requires *source* address of the reservation, *mail* of the user and reservation *ID.* When request service finds a taxi, a message is sent to the user, sending a JSON object containing *taxiID* data and *waitingTime.*

RequestTaxi(position) - function to request a taxi at a given position. When the server finds a taxi, a message is sent to the user containing a JSON object with *taxiID* data and *waitingTime.* If no taxi has been found in that zone, it will send a message to user to have a confirmation for search taxi in bordering zones. If no taxi has been found in any zone, sends an error message to user.

ShowMap(position) - this function calculates the zone from the position, gathering information from a third party map system, gets all Driver's position for the zone and sends a message to the user containing positions and map information to show on the browser's OpenStreetMap frame.

Core – Queue: Manage queue Interface:

GetZones() - this function returns an array of objects representing zones and their bounds.

GetQueue(zone) - return the queue object for the specified *zone.* The queue object is implemented as a typical queue data type, with other functions that will serve for our purpose, such as MoveFirstTaxiToLastPosition().

LoadBorderingZoneQueue(zone) - returns an array of objects representing zones bordering with the specified *zone.*

InsertTaxiInZoneQueue(zone, taxiId) - inserts taxi specified by *taxiId* in the queue for the *zone.*

Core – Driver Manager: Change status Interface:

SetAvailability(taxiId, position) - set the taxi specified by taxiID available for the zone which will be calcuated from the position parameter, and adds it to that queue.

SetNotAvailable(taxiId) - set the taxi specified by taxiId parameter not available anymore and removes it from the queue.

Core – Driver Manager: Contact driver Interface:

GetTaxiPosition(taxiId) - Returns an object representing the position of the taxi whose id corresponds to *taxiId*

Database – Reservation: Manage reservation Interface:

AddReservation(source, destination, time, mail) - saves reservation data provided as parameters into the Reservation schema of the database.

GetReservations() – returns an array of objects representing reservation saved on the database. It return only reservations which start time is greater than the actual time.

Database – Request: Manage request Interface:

AddRequest(position, sessionId) - saves request data provided as parameters into the Reservation schema of the database.

Database – Driver: Manage status Interface:

SetStatus(driverId, status) – writes the *status* for the driver specified by *driverId* into the database.

GetDrivers() – return the list of all drivers present in the database.

AddDriver(name, surname, taxiCode, username, password) - Adds a new driver into the database, with all information passed as parameters.

G – Selected Architectural Styles and Patterns

As previously mentioned, this system will have a two-tier architecture with logic distributed both on client side and server side. At first we will use a single dedicated server where our nodejs application and mongodb service will run. This will let us configure the machine following our needs, and have a strong initial solution which could tolerate high traffic. Moreover, cloud solution would be inefficent because our application is not easily scalable.
Each component in the application core will be implemented in nodejs with a module, following the Factory design pattern: every module will export a function which will manage the creation of the object along with its methods or attributes. Having one module for components allows the system to be easily extended in the future, in case other functionalities will be required.
In the main module, all the application components, libraries and routes are initialized, in particular the socket.io server, which will create a room for every different client session. For simplicity, we omitted in the last section that almost every function of the core modules require another parameter, that is the session ID of the user or the driver, otherwise it would be impossible to contact each client separately. With this session ID we can send messages only to client connected to a specific room identified by the session ID.

H – Other Design Decisions

We covered almost all the main aspects of the architecture design of our application in the previous chapters, but we want to implement another function in our application: the driver's page can be able to check if he is currently in the zone in which he set the availability, and if not it will display a message on his screen. All of this can be done Client-side, without using the server computation, considered that inside the driver page there is the current availability memorized. We will call this function "*checkCurrentZone"* and it will be called every amount of time, for example one minute (but we can change it later when we test it). This function will be something like this:

1. Retrieve the current GPS position of the Taxi
2. Check if the position is Inside the Availability Zone's boundaries
3. If not, display the message

As you can see, it is a very simple function, but it must be executed every minute or so, and would have been heavy for the server to do this for every taxi. For the client, instead, it is not a problem, as it can execute this very fast.

ALGORITHM DESIGN

Here we will present some interesting part of our code, in form of a “pseudo code” in the highest level possible. The italic words are variables, while the bold words are functions name.

First, we will introduce how we manage the Taxi Requests. Of course, the main function “**RequestTaxi**” will call other private functions, and we will show the most important ones. Then we will talk about the Reservation and the Queue Management by the availability change of taxies. We will also talk about the Restart Management, which explain what happens when the system reboots for some reason. This part will be exposed in depth because we think it is very important to understand our application’s logic.

We will not focus on the client side because we think that it is not interesting. Instead, we will talk about some relevant things on the observation part of each chapter.

We will also not mention the administrator classes, which are easy queries on the database and the login functions of the driver, which simply memorizes the work turn on the DB and give him access to the Taxi functionalities.

The “**Show Map**” function uses the OpenStreet Map API to show a map to the user pointing to the given position. It will retrieve the position of all available taxies in the user zone, and the API will place the markers on those positions.

To recognize the user and driver applications, we will use session’s IDs. We omitted the check on those IDs to make all the code more readable and shorter, but we must use the ID to retrieve the connection session every time we show or send a message to a Driver or User.

A – Taxi Requests Management

A.1 Function RequestTaxi (Position)

/ Pre Conditions: Position contains the X and Y position of the user in the map. This position belongs to one and only one zone. Every Zone has one and only one Queue. We have a Global Variable Array (Zones) that contains all the zones and one Map (Queues) for all the queues.*/*

START

- **Write the Request to the Database**
  1. Call the Function "**DBWriteRequest**" passing *Position*.

- **Identify the Zone by the given Position**
  1. For Each *Zone in Zones*
     a. IF the function "**ZoneContainsPosition**" passing *Position* and *Zone* returns TRUE
     b. THEN Memorize *Zone* as *User Zone* and BREAK the Cycle

- **Load The Queue for this zone**
  1. Load the *Queue* from *Queues* that have the *User Zone* as Index

- **Make calls to the Taxies in this queue until one of them accepts the call and move each called taxi to last position**
  1. Call The Function "**CallTaxiesInQueue**" passing the *Zone Queue* and Memorize the *Called Taxi*.

- **If somebody accepted, show the information of the Called Taxi.**
  1. IF *Called Taxi* is not null THEN
     a. Call the Function "**CalculateWaitingTime**" passing the *Called Taxi* position field and user position and memorize the *Waiting Time*.
     b. Call the Function "**ShowInfo**" passing *Waiting Time* and *Called Taxi* Code field, to show the result page to the user.
     c. Delete *Called Taxi* from the *Queue*.

- **If nobody accepted, show the Confirmation Dialog to the user.**

2. ELSE
    a. Call the Function "**ShowNoNearTaxiPage**" to Show the Confirmation Dialog To the User and Memorize *Response*

- **If his response is positive check other zones taxies, too, beginning each time by the nearest one.**

    a. IF *Response* is True
        i. Call the Function "**CallBorderingZones**" passing the *User Zone*, and memorize the *Called Taxi*.
        ii. IF *Called Taxi* is not NULL
            1. Call the Function "**CalculateWaitingTime**" passing the *Called Taxi* position field and *User Position* and memorize the *Waiting Time*.
            2. Call the function "**ShowInfo**" passing *Waiting Time* and *Called Taxi* Code.
            3. Delete *Called Taxi* from the *Queue*.
        iii. ELSE
            1. Call The Function "**ShowNoTaxiAvailablePage**" that tells the user there are no taxies.

- **If his response is negative return to the main page.**

    b. ELSE
        iv. Call The Function "**ShowMainPage**" that returns to the main page of the application.

END

A.2 Function CallTaxiesInQueue (Queue, Called Taxies)

*/*Pre Conditions: Queue contains the List of Taxies available in a zone. Called Taxies is empty at the beginning.*
*Returns the Taxi that has accepted, or NULL if nobody has. */*

START

- **Move the first to the last position, Call it and wait his response**
  1. Take the *First Taxi,* in the Queue's first position.
  2. Call Queue's Function "**MoveFirstToLastPosition**".
  3. Create a *Promise* with the function "**MakeCall**" passing *First Taxi* to make a call.

- **If a Taxi accepts the call we can exit and return the called taxi, otherwise we must memorize the Taxi in the Called Taxies array and do all of this again.**
  4. If the *Promise* is Successful
     a. RETURN the *First Taxi*
  5. ELSE
     a. Add the Index of *First Taxi* to the *Called Taxies* Array

- **If we encounter the same taxi again, we will stop and return NULL.**
     b. IF *Called Taxies* contains the index of the Taxi in the *Queue*'s first position OR *Queue* is empty
        i. RETURN NULL
     c. ELSE
        i. Recursively call "**CallTaxiesInQueue"** Passing Queue and Called Taxies and RETURN the returning value.

END

A.3 Function CallBorderingZones (Current Zone)

*/*Pre Conditions: Current Zone contains the Zone from where we must begin to calculate the bordering zones*
Returns the Taxi that has accepted, or Null if nobody has/*

START

- **Beginning to the start zone, find the bordering zones and take their queries, then call all the taxi in those queues.**
  1. Create a *Controlled Zones* Array and add the *Current Zone* in the 0 position.
  2. Make a *counter* that begins to 0
  3. REPEAT
     a. Take The *Zone* in the *counter* position of *Controlled Zones*
     b. Call The Function "**TakeBorderingZones**" to Find the *Bordering Zones* of the *Zone* Taken and Memorize them
     c. For Each *Bordering Zone* in *Bordering Zones*
        v. IF the *Bordering Zone* Isn't in *Controlled Zones*
           1. Load the *Queue* that have that *Bordering Zone* as Index
           2. Call The Function "**CallTaxiesInQueue**" passing the *Queue* and Memorize the *Called Taxi*.
           3. IF *Called Taxi* is not null RETURN it
           4. ELSE Add *Bordering Zone* to *Controlled Zones*
        vi. ELSE do nothing

- **If nobody still has not accepted, do this again controlling the uncontrolled bordering zones of each previous zone's borders until all the zones has been controlled. Then return null.**
     c. Increase the *Counter* by 1
  4. UNTIL *Controlled Zones* is NULL in the *Counter* Position
  5. RETURN NULL.

END

A.4 Function TakeBorderingZones (CurrentZone)

*/*Pre Conditions: CurrentZone contains informations about the Zone from what we must take the bordering Zones. We also have a Global Variable (Zones) that contains all the Zones and their boundaries.*
Returns an Array containing all the Bordering Zones of CurrentZone/*

START

- **Compare Each Zones Boundaries with the CurrentZone's ones to find out their Bordering Zones.**
  1. Create an empty array of *Bordering Zones*.
  2. For Each *Zone* in *Zones*
     a. IF the *Zone*'s Bottom Boundary equals the *CurrentZone*'s Upper Boundary OR
     b. IF the *Zone*'s Upper Boundary equals the *CurrentZone*'s Bottom Boundary OR
     c. IF the *Zone*'s Left Boundary equals the *CurrentZone*'s Right Boundary OR
     d. IF the *Zone*'s Right Boundary equals the *CurrentZone*'s Left Boundary
     e. THEN add *Zone* to *Bordering Zones*
  3. RETURN *Bordering Zones*

END

A.5 Other Used Functions

- The Function **CalculateWaitingTime** will simply use our API to calculate the travel duration from one position to another travelling by car.
- The Functions whose name begins by **Show..** will simply render the html page that shows that information to the user. The **ShowNoNearTaxiPage** has also a response that represent the user choice when the confirmation dialog appears. To do this asynchronously, we can use Promises functions.
- **MoveFirstToLastPosition** is a function that we will implement for the queues to pop the first object and push it at the end.
- **MakeCall** is a function that sends a notification to the Taxi Driver and waits for his response. Then returns as Successful only if there are no errors and the Taxi accepted the call. We will use the Promises function to do this asynchronously.
- **ZoneContainsPosition** simply check if the given Position is placed within the given Zone boundaries
- **DBAddRequest** take the request information and the reservation ID (null if not passed) and write them into the Database. It also memorizes the Time of this operation, which correspond to the Time of the Request.

A.6 Observations

- We decided to add a driver at the bottom of the queue when he will be available, but in that case he will be ignored by the pending requests. This because we stop the call iteration when we find an already called taxi, but in this time this will be above the new taxi in most cases. We can think about changing the condition to “when all the taxies in the queue are been called”, but this will not be efficient as it requires another control cycle. Therefore, we think that to miss the call from a pending request is not a big deal for the Taxi, because then in most cases the taxi will receive another call from a new request straight away, so we decided to leave the code in this way.
- It’s true that the driver can receive more calls at the same time doing this way: we will manage this anomaly in the Client side, so that he can accept only one request at a time.

B. Taxi Reservation Management

B.1 Function ReserveTaxi (Source, Destination, Time, Mail)

*/*Pre Conditions: Source is the position where the taxi will met the user, while destination is the arrival position. Time represent the time of the meeting, which is between 10 minutes and 2 hours from the reservation time. Mail contains the E-Mail of the User.*/*

START

- **Write the Reservation on the Database, calculate the Delay and show the Confirmation message to the user**

1. Call the "**DBAddReservation**" function passing *Source*, *Destination*, *Time* and *Mail*, and memorize the returning *ID*.
2. Calculate the Time *Delay* in milliseconds making the difference between *Time* and the Actual Time (accessible by the system).
3. IF there are no errors THEN call "**ShowConfirmationPage**" function to show the confirmation to the user.

- **Wait until 10 Minutes before the Reservation, then Request a Taxi**

4. After that *Delay* minus 10 Minutes execute the function "**RequestReservedTaxi**" passing *Source*, *Mail* and *ID*.

END

B.2 Function RequestReservedTaxi (ID, Source, Mail)

/ Pre Conditions: Source is the position where the taxi will met the user, while mail contains the E-Mail of the User. ID contains the ID of the Reservation*/*

START

- **Write the Request to the Database**
  2. Call the Function "**DBWriteRequest**" passing *Position* and *ID*.

- **Identify the Zone by the given Position**
  3. For Each *Zone in Zones*
     - IF the function "**ZoneContainsPosition**", passing *Position* and *Zone*, returns TRUE
     - THEN Memorize *Zone* as *User Zone* and BREAK the Cycle

- **Load The Queue for this zone**
  2. Load the *Queue* from *Queues* that have the *User Zone* as Index

- **Make calls to the Taxies in this queue until one of them accepts the call and move each called taxi to last position**
  2. Call The Function "**CallTaxiesInQueue**" passing the *Zone Queue* and Memorize the *Called Taxi*.

- **If somebody accepted, send a mail with the information of the Called Taxi.**
  3. IF *Called Taxi* is not null THEN
     a. Call the Function "**CalculateWaitingTime**" passing the *Called Taxi* position field and user position and memorize the *Waiting Time*.
     b. Call the Function "**SendMail**" passing *Waiting Time*, *Called Taxi* Code field, and "Mail" to send a Mail to the user.
     c. Delete *Called Taxi* from the *Queue*.

- **If nobody accepted, check other zones taxies, too**

4. ELSE
    a. Call the Function "**CallBorderingZones**" passing the *User Zone*, and memorize the *Called Taxi*.
    b. IF *Called Taxi* is not NULL
        i. Call the Function "**CalculateWaitingTime**" passing the *Called Taxi* position field and *User Position* and memorize the *Waiting Time*.
        ii. Call the Function "**SendMail**" passing *Waiting Time*, *Called Taxi* Code field, and "Mail" to send a Mail to the user.
        iii. Delete *Called Taxi* from the *Queue*.

- **If there isn't a taxi available, make a new Request.**

    c. ELSE
        i. Call "**RequestReservedTaxi**" function again passing the same initial variables.

END

B.3 Other Functions

- **SendMail** uses some plugins to send a Mail to the user. It calculates also if the waiting time is more than 10 minutes, so that in the mail can be specified if there will be a delay in the arrival. If the Mail is null or invalid, the function will do nothing.

- **DBAddReservation** is similar to **DBAddRequest**.

B.4 Observations

- It is very rare that at a certain moment there are no taxies that accepts a call, but in a reservation request we can't accept that fact, so we decided to make a new request, hoping that the second time there will be more taxies available, maybe also nearer, and they will accept the call. In addition, if the same taxi will be called again, we can hope that he accepts the second time.

- We have not considered the possibility that the user is using a Smartphone Application instead of the browser: in that case we will simply call a function that manage the notifications every time there is an important event. For example, he can use app notification instead of the E-Mail to receive the confirmation (we can simply set the Mail field to NULL and add another function with an "if" condition).

- The Time variable, as we said in the pre-condition, is already a valid time, because the client will make all the specific controls on the user data, as we also said in the diagram section. Therefore, it is useless to do that again Server-Side, although we will consider the possibility to do a double check if there will be security issues.

C – Taxi Queue Management

C.1 Function SetAvailability (Position, TaxiCode)

/ Pre Conditions: Position contains the actual position of the Taxi, retrieved by the GPS. TaxiCode contains the Code of the Taxi. The Taxi IS NOT available */*

START

- **Check the Zone of the Position**
  1. For Each *Zone in Zones*
     a. IF the function "**ZoneContainsPosition**", passing *Position* and *Zone*, returns TRUE
     b. THEN Memorize *Zone* as *TaxiZone* and BREAK the Cycle

- **Memorize the availability and his zone on the Database**
  2. Call the Function "**DBUpdateTaxiAvailability**" passing *TaxiZone* and *TaxiCode*, other than True for the Availability.

- **Add the Taxi on his Zone's Queue**
  3. Take the *Queue* by the *TaxiZone* index.
  4. Push the *TaxiCode* at the bottom of the Queue

END

C.2 Function SetNotAvailable (TaxiCode)

// Pre Conditions: TaxiCode contains the Code of the Taxi. The Taxi IS available

START

- **Retreive the Taxi Zone by the Database**
  1. Call the Function "**DBGetTaxiZone**" passing *TaxiCode* and *memorize TaxiZone*

- **Remove the Taxi from his Zone's Queue**
  2. Take the *Queue* by the *TaxiZone* index.
  3. Delete the *TaxiCode* from the Queue

- **Change the availability on the Database**
  4. Call the Function "**DBUpdateTaxiAvailability**" passing *TaxiCode* and False for the Availability.

END

C.3 Other Functions and observations

The remaining functions here are the ones starting with "**DB…**" which are simply calling the database and making some queries, so they are not relevant. The "*Pre Conditions*" will be respected because we will make those controls on the Client side, so that the Taxi cannot call those functions if those conditions are not respected. Note also that when a taxi will log out, the function "**SetNotAvailable**" will be automatically called from the Client.

D – Restore After Reboot

D.1 Function RestoreQueues

/ Pre Conditions: NONE. The system has been rebooted */*

START

- **Restore The Zones from the Database and Create the Queues**

1. Call the Function "**DBReadAllZones**" and Memorize the *Zones*
2. FOR EACH *Zone* in *Zones*
    a. Create a *Taxi Queue* and Add it in the *Queues* Map, with *Zone* as Index

- **Take all the Available Taxies from the Database and put them in the Queue according to their zone.**

3. Call the Function "**DBReadAllAvailableTaxies**" and Memorize the *Available Taxies*.
4. FOR EACH *Taxi* in *Available Taxi*
    a. Take the *Queue* that has *Taxi*'s Zone as Index
    b. Push the *Taxi*'s Code in that *Queue*

END

D.2 Function RestorePendingReservations

/ Pre Conditions: Queues are Restored */*

START

- **Read the Reservations from the Database, but only if their Time is greater than the actual Time.**

1. Call "**DBReadReservation**" and Memorize the *Reservations.*

- **For Each Reservation, Calculate the delay time. If it's more than 10 minutes, restore the previous command.**

2. FOR EACH Reservation in Reservations
    a. Calculate the Time *Delay* in milliseconds making the difference between Reservation's *Time* and the Actual Time (accessible by the system).
    b. IF *Delay* is more than 10 Minutes THEN
        i. 10 Minutes before Reservation's Time, execute the function "**RequestReservedTaxi**" passing Reservation's *Source*, *Mail* and *ID*.

END

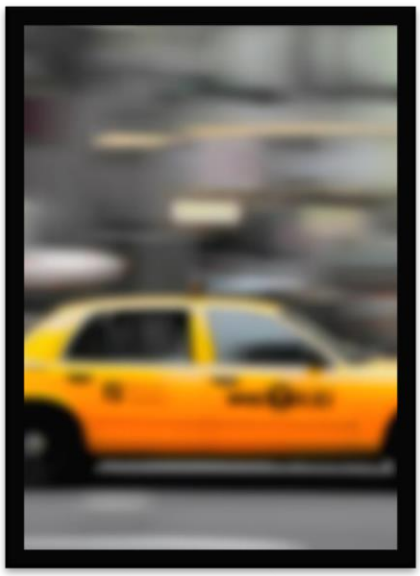
D.3 Other Function and Observations

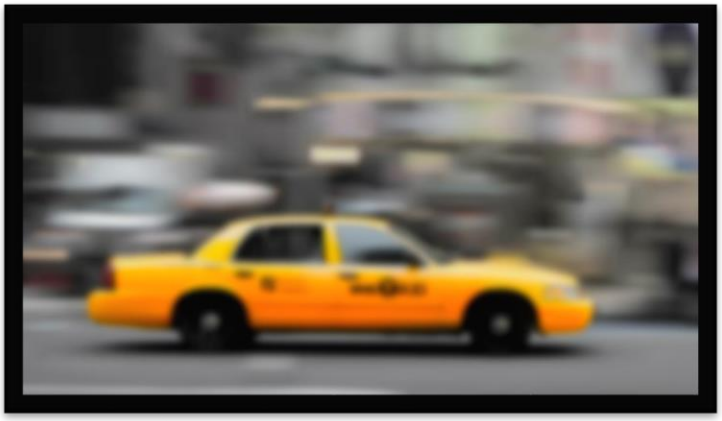
Note that no pending requests, either the ones deriving from a reservation or the ones made by the user as a Request, are restored when the system reboots. In fact the system considers all the requests as completed. This is a problem for a reservation, but note that this case occurs only when the system crashes DURING the calls, and it's rare. For the simple requests, in this case the user will not receive the confirmation page, so he understands that there is an error, and, hopefully, he will request the taxi again.

User Interface Design

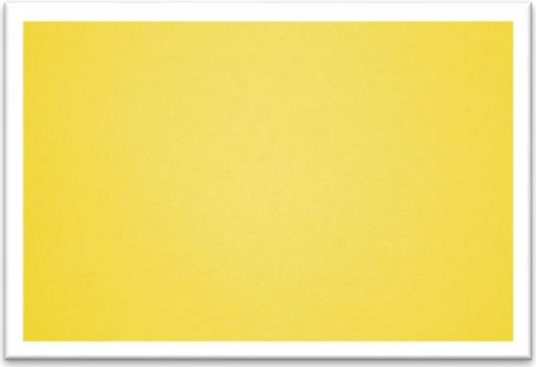
We already showed the main windows on the R.A.S.D. document. Our application is very simple and minimal, so there is not much to add. We found a good design for the background, instead of the light blue color, that will be like that for the User Application:

Portrait **Landscape**

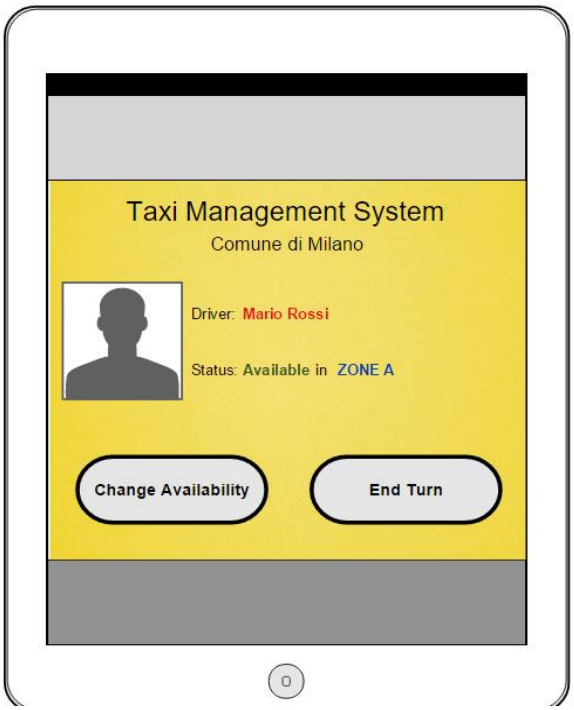




For the Drivers Application, instead, we will have a more flat background, so with a light Yellow texture and the Black Text. This can work well also for the administrator.



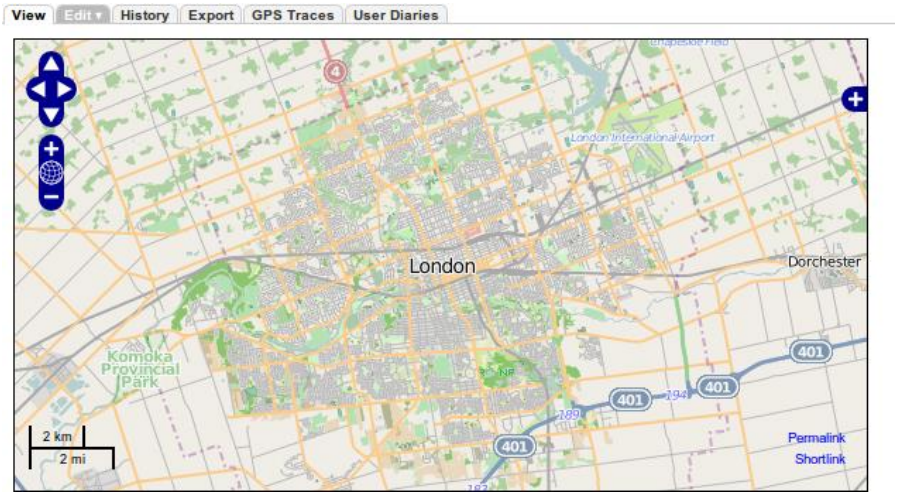
Driver's Yellow Background



On the left there is an example of the new main Driver's page, with a new information: we will show him the zone in which he logged in. This will be replaced with the message "*You're not in Zone A anymore, please go back there or* ***change your availability*** *again*" if he moves from the availability zone.

For the app notification's we will use the default style according to the OS, with our app's icon that will be a simple taxi icon like this one.

For the Map, instead, we will use our API with the default design, which is very good in our opinion.



Regarding to the functional part, we already explained in the R.A.S.D. that there will be only few buttons to click, either for the driver or for the user, so it's very intuitive, simple and fast, other than being effective.

For the Administrator part, there will be a "working" interface, so with rough and simply tables and forms, but still it will look easy and intuitive to use. This choice has been made because usually the administrator is an expert and must only do his work easily, without looking at the aesthetic, so it is not a priority to make it also looks good.

REQUIREMENTS TRACEABILITY

We already showed, in different ways, how we respected the non-functional requirements from our R.A.S.D. in our application. Therefore, we will focus on the functional requirements.

USER can:

- **Insert his Position (Using his GPS or manually)**

The *user interface* allows the user to insert his position manually. If he has the GPS or use the web localization, the client has the *GPS Manager* that stores his position, as seen in the Component Diagram.

- **Request a Taxi and see the Waiting Time**

We presented the **"***RequestTaxi"* function in the previous sections.

- **Make a Reservation for a certain time in a certain position**

We presented the *"ReserveTaxi"* function in the previous sections.

- **Show a Map with the Taxies locations**

We presented the *"ShowMap"* function in the previous sections.

TAXI DRIVER can:

- **Log In**

We talked about the "*Login"* function in the previous sections.

- **Report Availability to the System**

We presented the function *"ChangeStatus"* in the previous sections.

- **Accept or Decline the System Call**

We talked about the "*MakeCall*" function in the previous sections, so about how the driver can accept or decline that call.

- **See if he is in his zone or not**

  We added the availability zone information on the driver page, and the "*checkCurrentZone*" function, described in the "other design decision" chapter.

- **End Their Turn (Which means to Logout)**

  We talked about the *Logout* function in the previous sections.

For the **Administrator** we have simply created a function for each requirement (Login, Logout, Add, Modify and Cancel Driver) in the previous sections, without going into them so much because of their easiness.

REFERENCES

- Gianluca Guarini's Blog: NodeJS Push Notification Example
- Websocket.org: websocket performance compared to ajax
- PAWS Developer Guide: Synchronous VS Asynchronous calls
- Strongloop Blog: Promises as alternatives to callbacks
- MongoDB website: Why Mongo is better than SQL
- Stormpath Blog: NodeJS Sessions
- Infoworld article: JavaEE VS NodeJS
- Cordova Main Page: Cordova Overview
- Template: “Structure of the design document” PDF

USED TOOLS

- **Visual Paradigm 12.2**: Community Edition, for the Diagrams
- **Moqups** online software to build the user interface preview
- **Microsoft Word 2013** to write this document

Table of Changes

Version 1.1

Pages Number Bug fixed

Version 1.2

Corrected Component Diagram