Software Engineering 2
“myTaxiService”

Design Document Version 2.0

1/21/2016

Politecnico di Milano A.A. 2015-2016

Milica Jovanovic (mat. 835953);Pavle Vidanovic (mat. 854472)

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide a comprehensive description of the structure of the myTaxiService system. It will state and analyze the design decisions made in order to satisfy all the requirements stated in the Requirements Analysis and Specification Document (RASD). This document is meant mainly as a guideline for developers of the software in question.

1.2 Scope

The aim of this project is to develop and implement myTaxiService, an application similar to Uber, which makes the process of assigning an available taxi vehicle to possible passengers.

The developed system should allow new users to register. Users, once logged in, should be able to:

- request a taxi
- reserve a taxi
- cancel a ride
- check taxi availability around him
- receive a confirmation with information about the assigned vehicle and ETA once taxi is requested
- create/maintain user profile
- report a taxi driver

The developed system should allow new taxi drivers to register. Drivers, once logged in, should be able to:

- inform the system about their availability
- confirm/decline that they are going to take care of a certain call
- create/maintain taxi driver profile
- report a passenger

The system should keep information about new arrived requests, as well as the confirmed rides. A ride should have and id number, information about the passenger that requested the ride, as well as the code of the assigned vehicle and ETA. System should also keep information about taxi queues connected to particular zone of the city and ensure fair management of the queues. Developed system should keep information about the list of reservations made by passengers, such as id number of the reservation, information about the passenger that made the reservation and the time of reservation and time of the ride.

1.3 Glossary

The following are the definitions of some commonly used phrases throughout the document:

CSS	Cascading Style Sheets
DD	Design Document
DB	Database
DBMS	Database Management System
ER	Entity-Relationship
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secured
MVC	Model View Controller
RASD	Requirement Analysis and Specification Document
UML	Unified Modeling Language
ORM	Object Relational Mapper

1.4 Reference Documents

- IEEE Design Document template
- Specification Document: myTaxiService Project AA 2015-2016.pdf
- RASD myTaxiSevice

1.5 Document Overview

The document is essentially structured in six parts:

- Chapter 1: Introduction, gives description of document and some basic information about the software
- Chapter 2: Architectural Design, gives an overview of how and why the system was decomposed, and how the individual parts work together
- Chapter 3: Algorithm Design, description of the most relevant algorithms of the software system
- Chapter 4: User interface Design, overview on how the user interfaces of your system will look
- Chapter 5: Requirements traceability, gives an overview of how the requirements defined in RASD map into the design elements defined in DD.
- Chapter 6: References

2 Architectural design

2.1 Overview

Identify modules in the software system and explain the relationships between the modules to achieve the complete functionality of the system. This is a high level overview of how responsibilities of the system were partitioned and then assigned to subsystems. Identify each high level subsystem and the roles or responsibilities assigned to it. Describe how these subsystems collaborate with each other in order to achieve the desired functionality.

2.2 High level components and their interaction

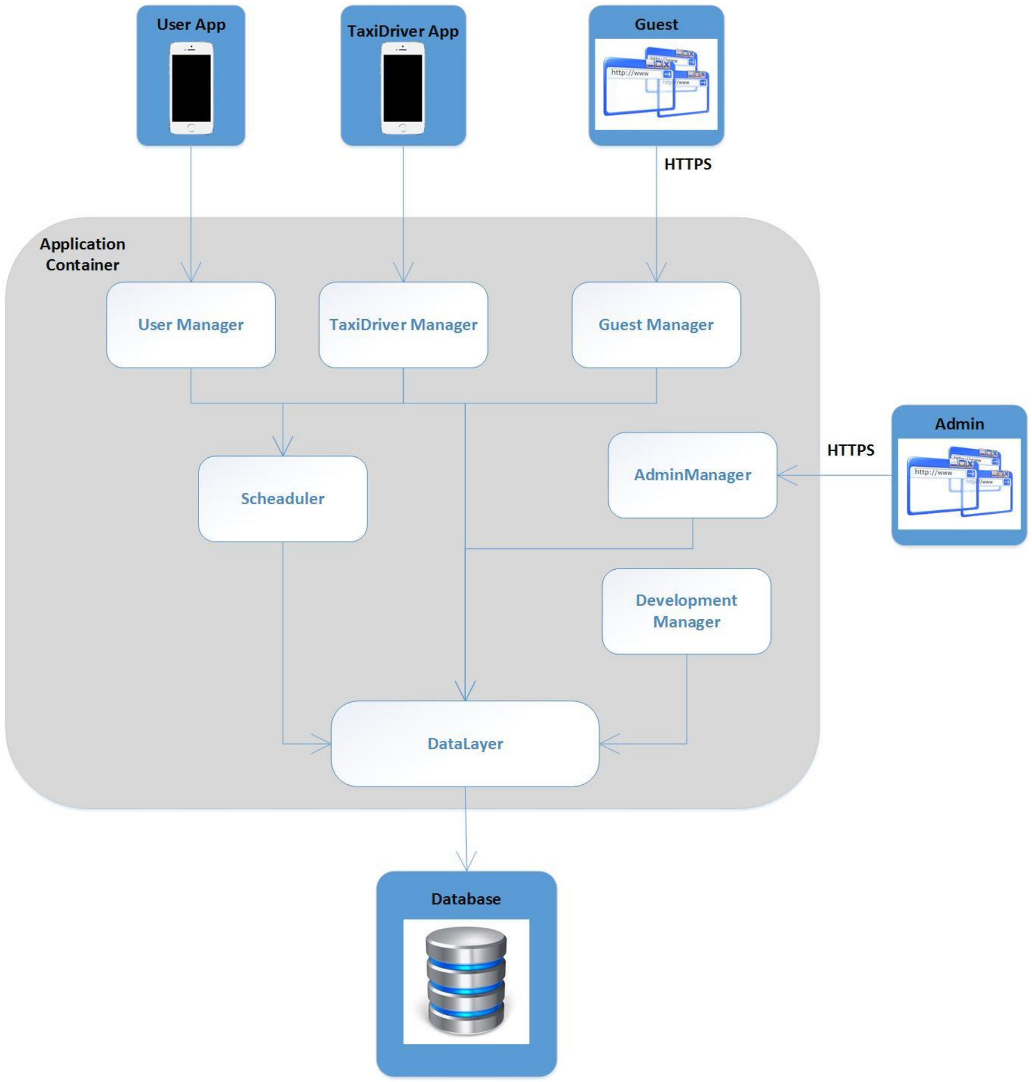

Figure 2.1 High level components

- *Guest Manager* is component that will provide interface for unregistered users or guest with an option to create a profile.
- *User Manager* is component of the system that will provide interfaces for users (passengers) with options of requesting, reserving taxi, managing profile, reporting driver, checking available taxis around him and canceling a ride.

- *Taxi Driver* Manager is component of the system that will provide interfaces for taxi drivers with options of confirming a ride, setting availability, reporting a user and canceling a ride.
- *Admin Manager* is component of the system that will provide interfaces for administrators with options of viewing reports and banning a user.
- *Scheduler* is component that is consisted of three subcomponents: Request Manager, Reservation Manager and Zone Manager. It provides interfaces for other components with options for requesting a taxi, finding zone, reserving a taxi as well as calculating ETA.

2.3 Component view

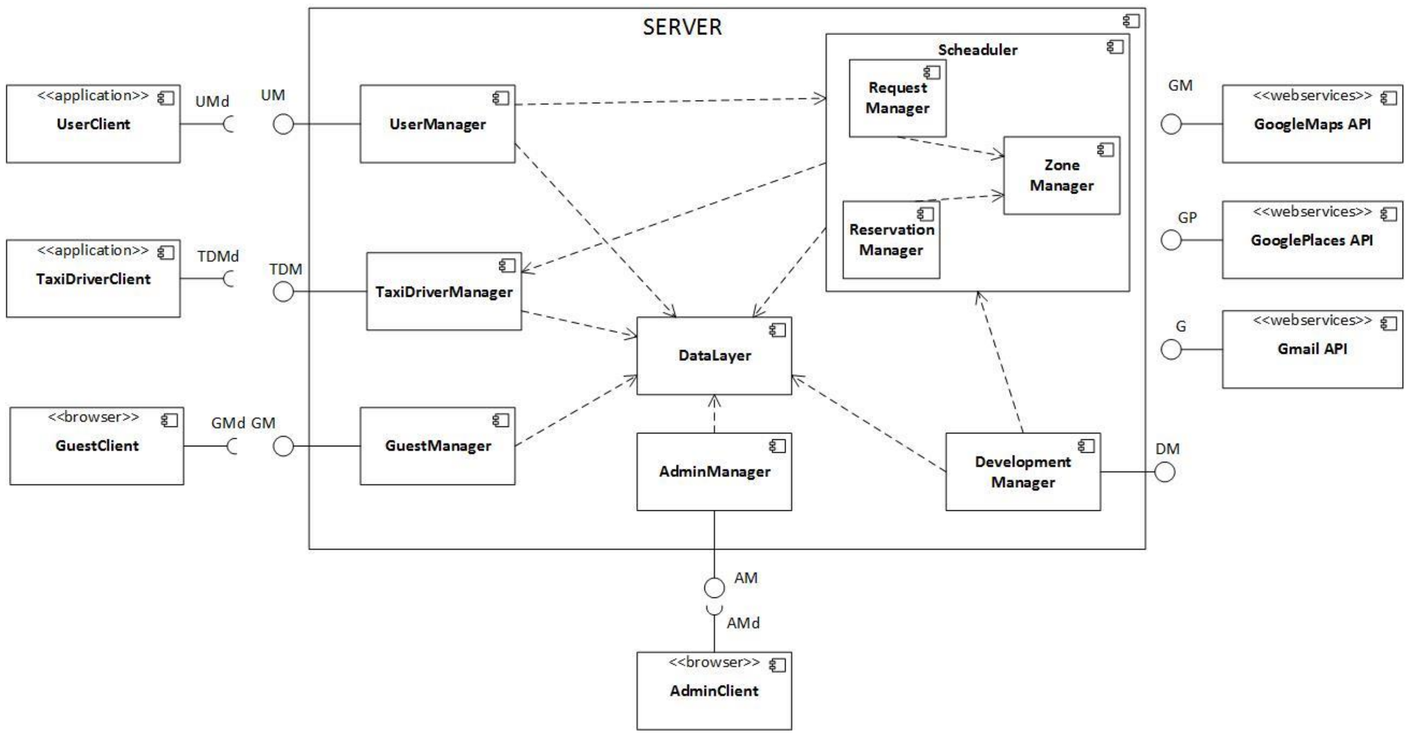

Figure 2.2 Component view

2.4 Deployment view

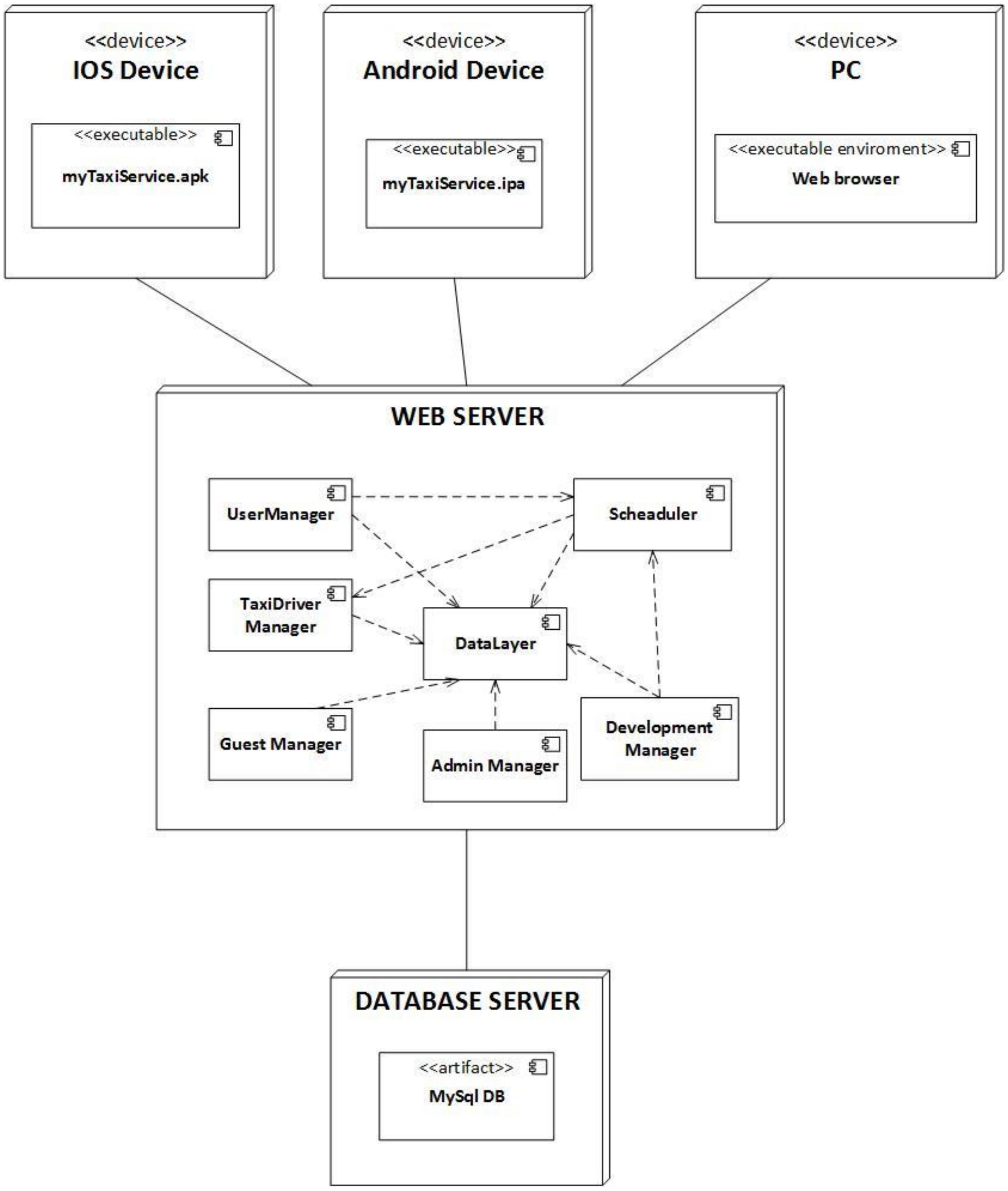

Figure 2.3 Deployment view

2.5 Runtime view

2.5.1 SignUp

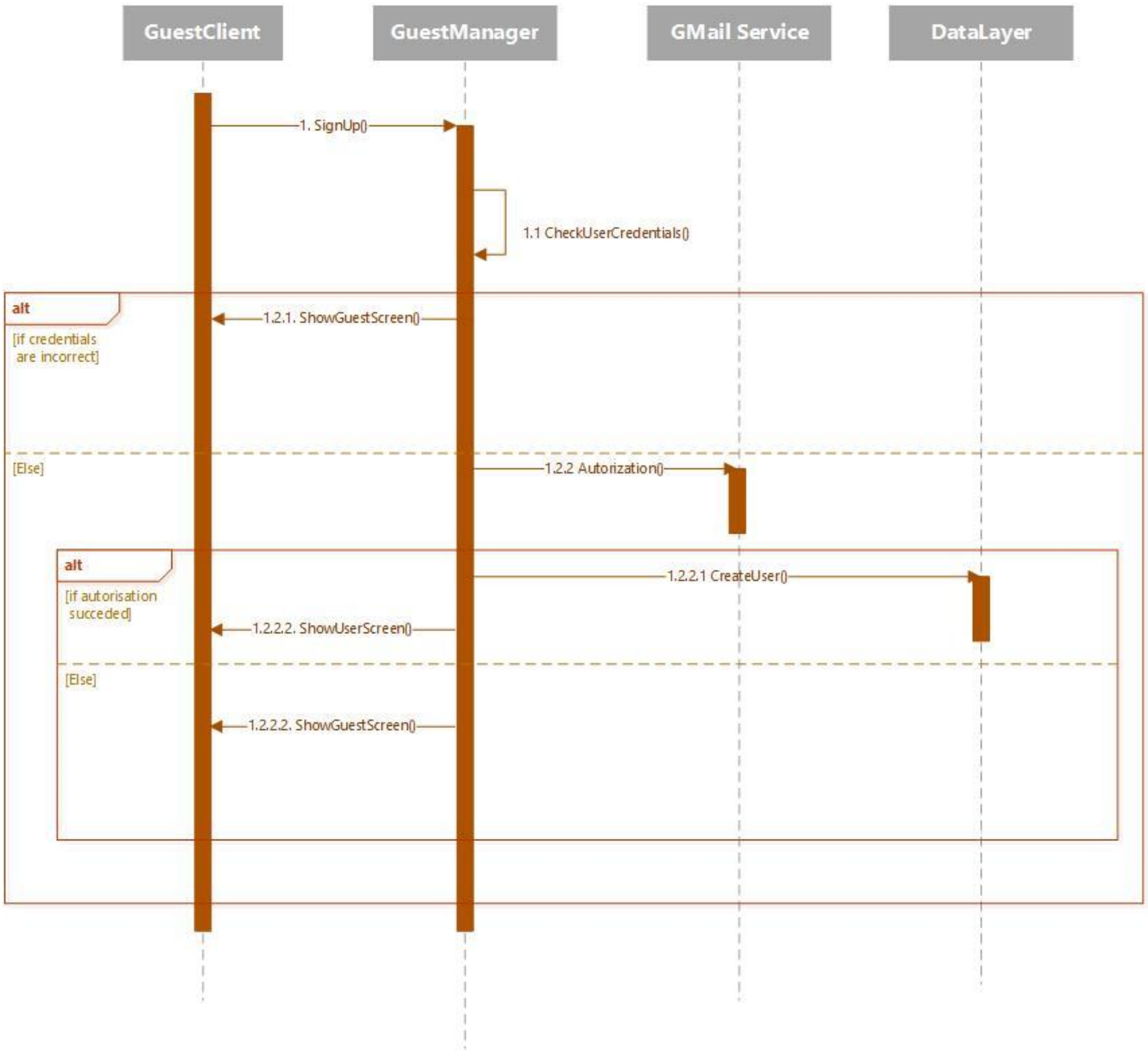

Figure 2.4 Runtime view for SignUp functionality

2.5.2 Request

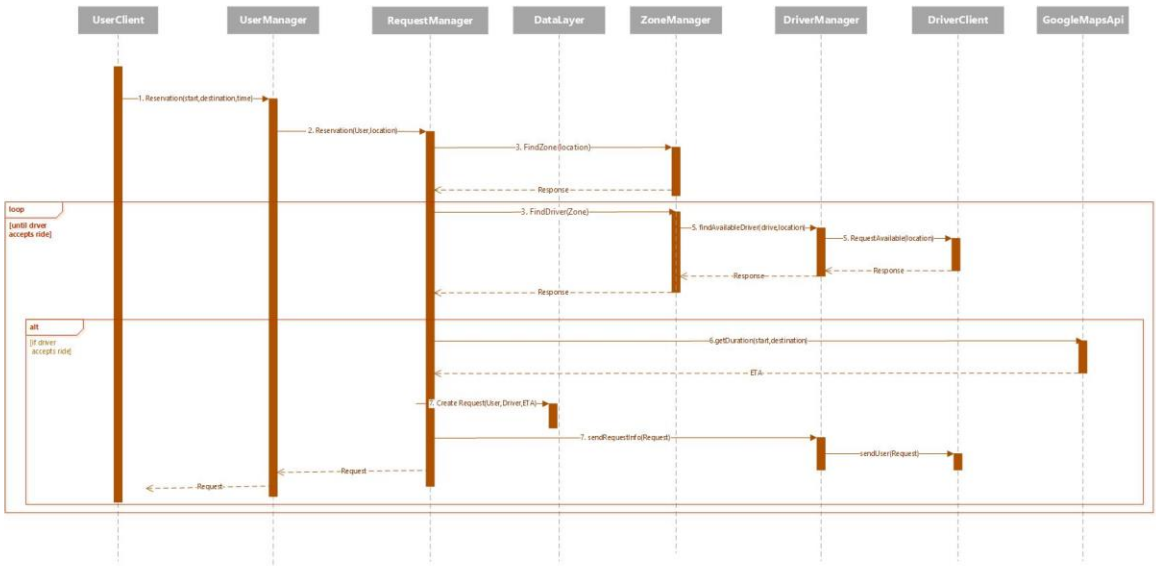
UserClient
UserManager
RequestManager
DataLayer
ZoneManager
DriverManager
DriverClient
GoogleMapsApi
1. Reservation(start,destination,time)
2. Reservation(User,location)
3. FindZone(location)
Response
loop
[until drver accepts ride]
3. FindDriver(Zone)
5. findAvailableDriver(drive,location)
5. RequestAvailable(location)
Response
Response
Response
alt
[if driver accepts ride]
6.getDuration(start,destination)
ETA
7. Create Request(User,Driver,ETA)
7. sendRequestInfo(Request)
sendUser(Request)
Request
Request

Figure 2.5 Runtime view for Request functionality

2.5.3 Reservation

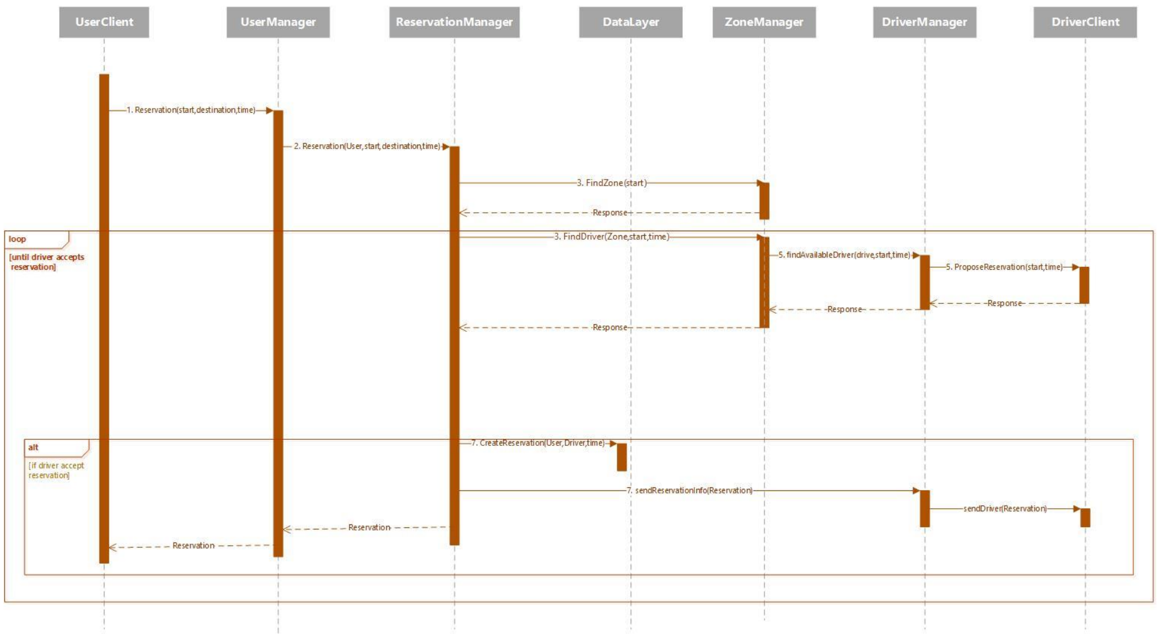
UserClient
UserManager
ReservationManager
DataLayer
ZoneManager
DriverManager
DriverClient
1. Reservation(start,destination,time)
2. Reservation(User,start,destination,time)
3. FindZone(start)
Response
loop
[until driver accepts reservation]
3. FindDriver(Zone,start,time)
5. findAvailableDriver(drive,start,time)
5. ProposeReservation(start,time)
Response
Response
Response
alt
[if driver accept reservation]
7. CreateReservation(User,Driver,time)
7. sendReservationInfo(Reservation)
sendDriver(Reservation)
Reservation
Reservation

Figure 2.6 Runtime view for Reservation functionality

2.5.4 Confirm/Decline a ride

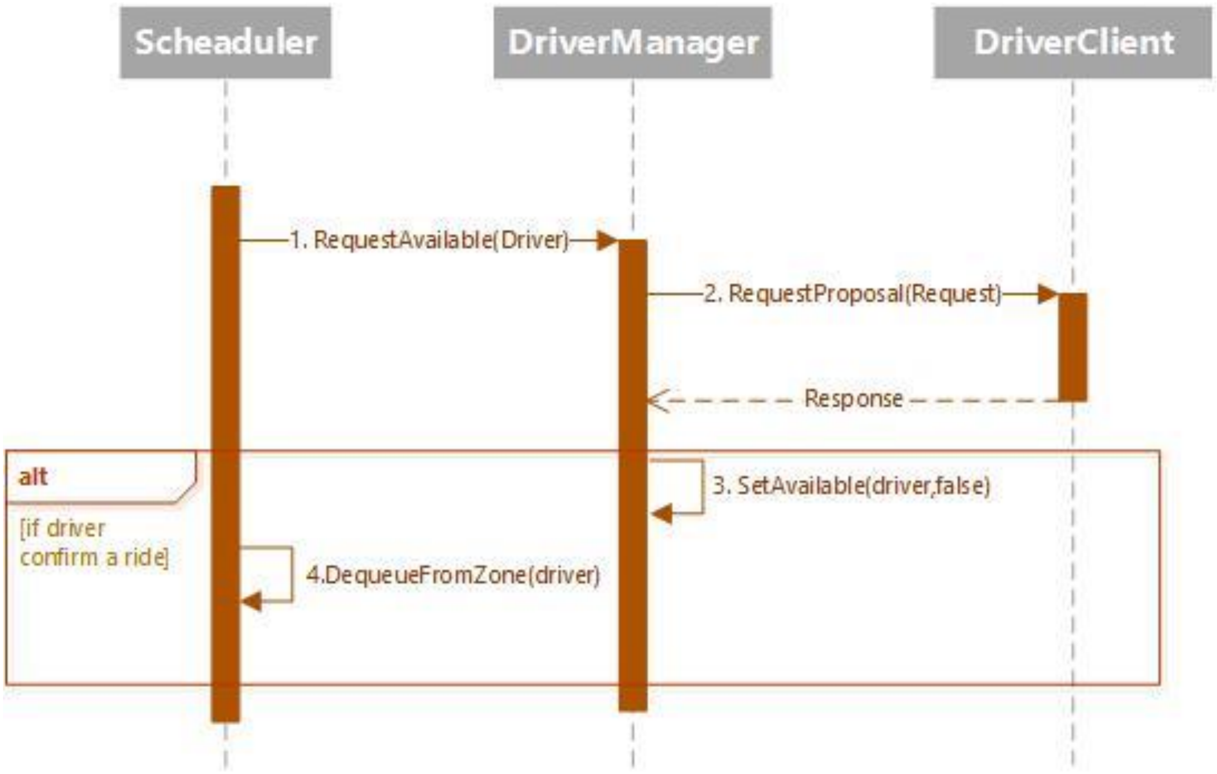
Scheaduler
DriverManager
DriverClient
1. RequestAvailable(Driver)
2. RequestProposal(Request)
Response
alt
[if driver confirm a ride]
3. SetAvailable(driver,false)
4.DequeueFromZone(driver)

Figure 2.7 Runtime view for Confirm/Decline a ride functionality

2.5.5 Ban User

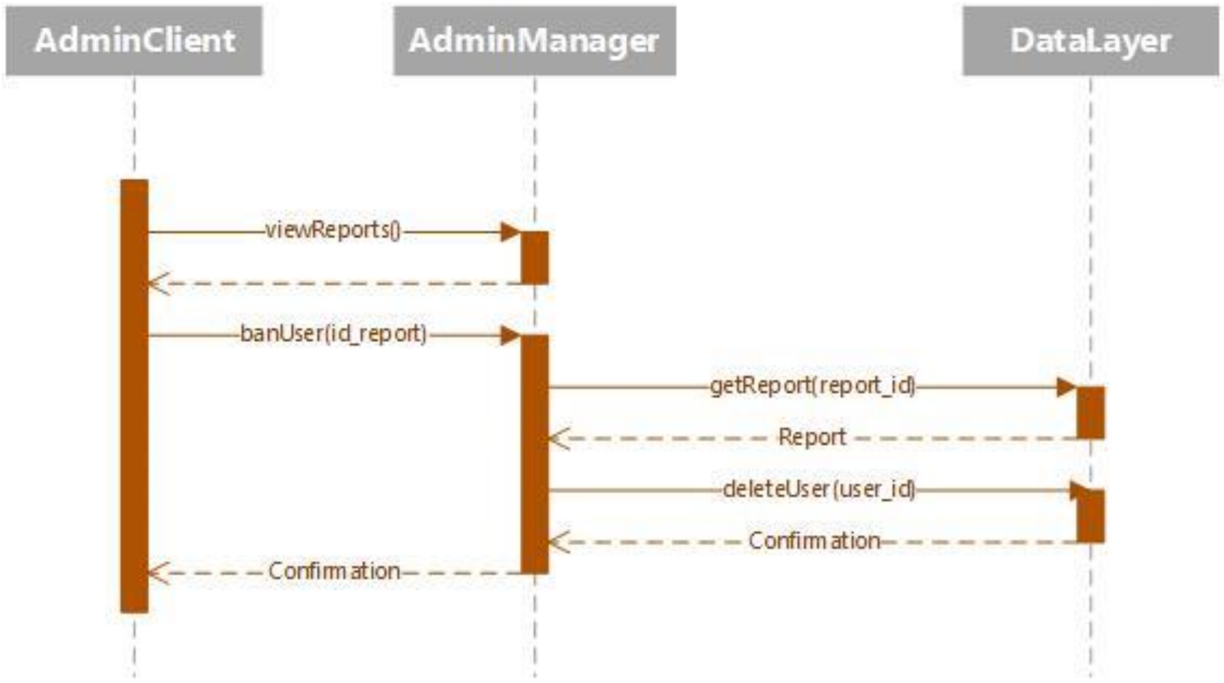

Figure 2.8 Runtime view for Ban user functionality

2.6 Component interfaces

2.6.1 Guest Manager

User signUp(name,surname,email,password,phone,img)

signUp function after submitting the inputted data sends it to the web server which checks the credentials if they already exist, uses Gmail API for email authentication. If received data is valid a new tuple is inserted in DB and sent back to the client app in form of JSON. After client app receives the string in encodes it to User object.

User signIn(email,password)

signIn function checks sent data in the server side. If the inputted credentials are valid i.e. tuple exists in DB an instance of User object is returned back to client app in the same way as in signUp function.

2.6.2 User Manager

Request makeRequest(source_address,user)

makeRequest function creates Request object, with user and source address initialized. Then it uses findZone(source_address) function of scheduler that returns zone of the user. Next, the Zone object contains queue of available taxi drivers, and the request is sent to first driver in the queue, and if his response is negative request is passed to next driver in queue. When driver accepts the ride, request instance is created and it is sent both to user and driver as confirmation.

Reservation makeReservation(source_address,time_of_ride,destination_address,user)

makeReservation function creates Reservation object, with user, source and destination address and time of the ride initialized. Then it uses findZone(source_address) function of Scheduler that returns zone of the user. Next, the confirmation of reservation is sent back to user. Moreover, the Zone object contains queue of available taxi drivers, and 30 minutes before the ride, the reservation is sent to first driver in the queue, and if his response is negative reservation is passed to next driver in queue. When the driver is found, the reservation instance is created and stored in DB and then 10 minutes before the ride the confirmation message is sent to user with drivers details.

String report(description,id_user,id_banned)

report function creates Report object with inputted data and stores it to DB.

String manageProfile(user)

manageProfile function receives modified User object and updates appropriate user tuple in DB.

TaxiDriver[] checkTaxisAvailable(source_address)

checkTaxisAvailable function contacts Zone Manager which returns a Zone object with the respect to inputted source address. Zone object contains queue of available taxi that is send back to client.

String cancelRide(request)

cancelRide function deletes particular request tuple of the database. After that, notification message is sent both to user and driver related to this request.

2.6.3 TaxiDriver Manager

String confirmDeclineRide(request)

confirmDeclineRide is function of TaxiDriver Client interface. Driver will accept or decline a ride and if his response is positive, the request attribute driver is instantiated and Google Maps API is invoked and provides a route. Otherwise, appropriate driver is moved to the end of a queue of appropriate zone (setAvailability(true, id_driver)).

String setAvailable(available_bool,id_driver)

setAvailable is function of TaxiDriver Manager interface. First, we contact the DB to get appropriate driver. Then, ZoneManager is contacted in order to provide driver's zone based on his current location. Based on available_bool value the driver is added or removed from the zone queue.

2.6.4 Admin Manager

String banUser(id_report)

banUser is function of Admin Manager component that uses id_report parameter to find particular user_id. After that, it deletes user tuple from DB. The confirmation message is sent back to admin.

Report[] viewReports()

viewReports is function of Admin Manager component which returns all reports from DB to admin.

2.7 Selected architectural styles and patterns

2.7.1 Architectural styles

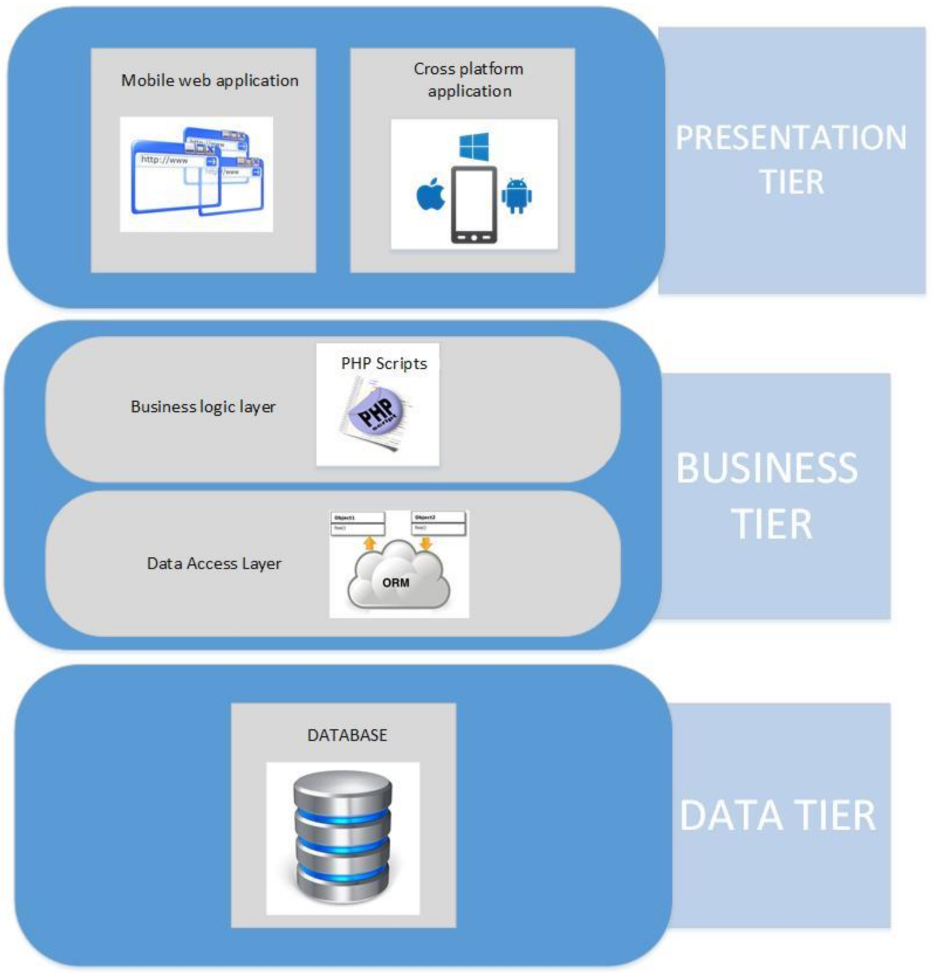

Figure 2.9 Scheme of 3-tier Architectural style of myTaxiService system

myTaxiService architecture is a 3-tier architecture, which consists of Presentation tier, Business tier and Data tier.

- **Presentation Tier**: it contains Cross platform application and Mobile Web application and it is the layer that interacts directly with the actors. As our project will be a web application, client will use a web browser to access pages.
- **Business Tier:** is consisted of two layers, Business logic layer and Data access layer. Business logic layer contains php scripts that are representing web services with application logic. Data access layer accepts the data from client and packs it in a form acceptable by DB server, as well as extracting data from DB and forms an object from them.
- **Data Tier**: Establishes a connection with MySql database and enables storing and retrieving all relevant data.

Client-Server

The client–server model of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Clients and servers exchange messages in a request–response messaging pattern: The client sends a request, and the server returns a response. This exchange of messages is an example of inter-process communication.

Three-tier architecture is a client–server software architecture style in which the user interface (presentation), functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.

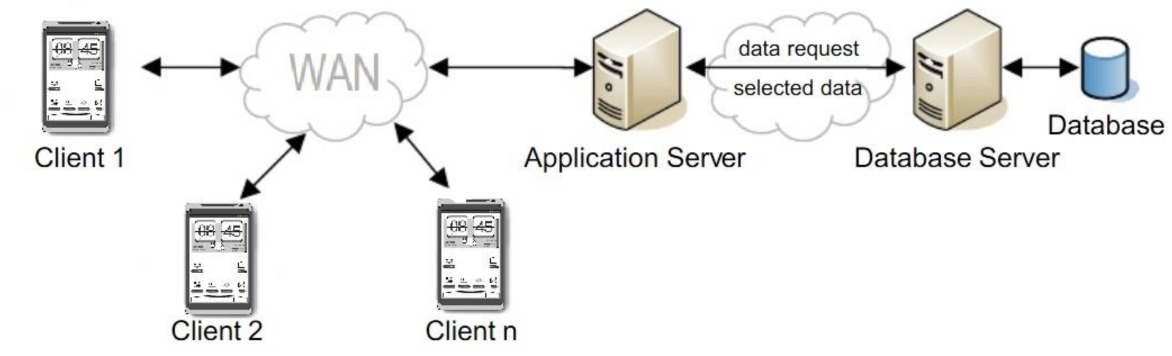

Figure 2.10 Scheme of 3-tier Client-Server Architectural style of myTaxiService system

myTaxiService system will be built on three tier client-server architectural style. Communication between the users and the taxi drivers goes through the server as a mediator. Purpose of the server in myTaxiSystem is to handle both users and drivers requests. Server processes the requests and queries the database. In this scenario, the database represents the third tier and is responsible for managing the stored data. After processing the request server sends a response back to the users or drivers. For example:

- user requests a taxi
- request is sent to the server
- server finds the zone and takes first available taxi from the queue
- server forwards the request to the taxi driver
- driver responds positive/negative
- response is sent to the server
- server calculate ETA and send response to user or he finds next taxi in the queue and send him the request

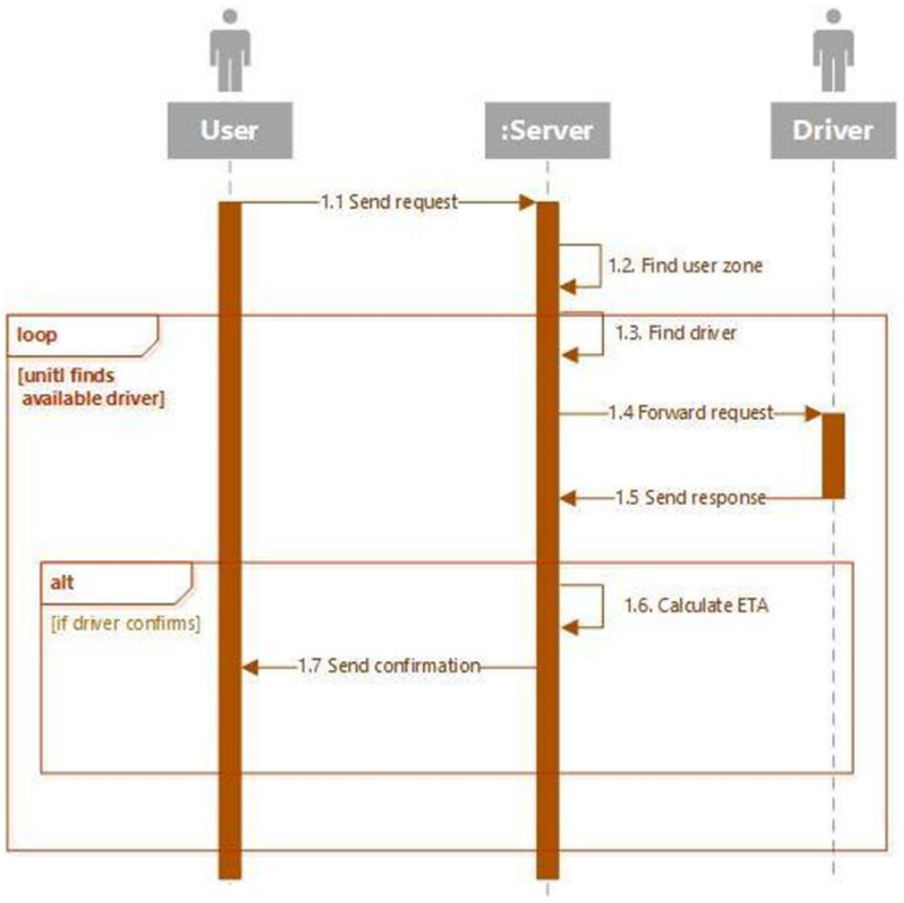

Figure 2.11 Example of flow for requesting a taxi and interaction between client and server

SOA

A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. The purpose of SOA is to allow users to combine fairly large chunks of functionality to form applications built from existing software services. If a service presents a simple interface that abstracts away its underlying complexity, then users can access independent services without knowledge of the service's platform implementation.

- When user fills the request form, system will contact the Google Places service for suggesting and autocompleting the input address. The suggested address is sent back to the user.
- When navigation page is requested by the taxi driver, Google Map service is contacted. Service provides the driver with map page with appropriate route.
- Google Maps Geocoding service provides coordinates for inputted address.
- Gmail API as email service of user’s authorization.

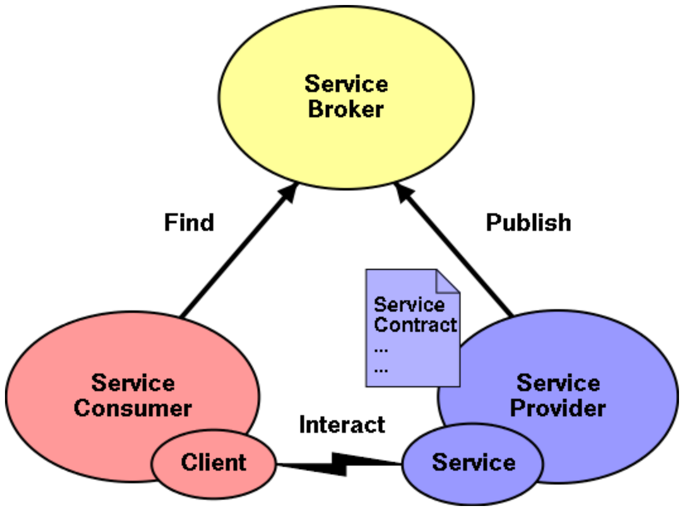

Figure 2.12 Scheme of SOA Architectural style

2.7.2 Patterns

Singleton

Singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

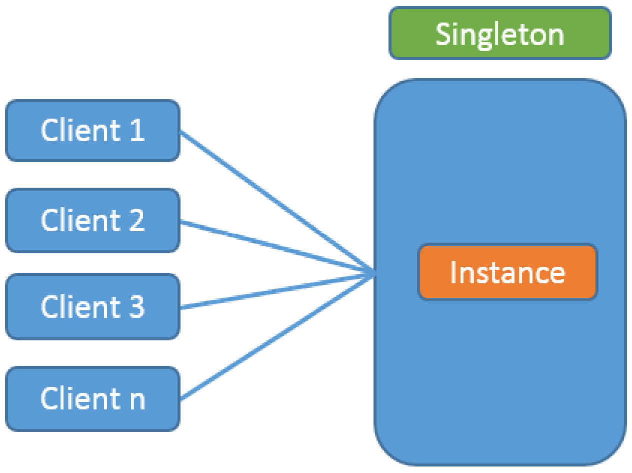

Figure 2.13 Scheme of Singleton Pattern

myTaxiService system will have one instance of Database. Every time access to the database is needed, system through this instance will open connection to the database. After the transaction the connection will be closed.

MVC

The model-view-controller pattern proposes three main components or objects to be used in software development:

- A *Model*, which represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface.
- A *View*, which is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth)
- A *Controller*, which contains the logic of system and is a mediator between the view i.e. user interface and his interaction with it and model which contains the data stored for our system.

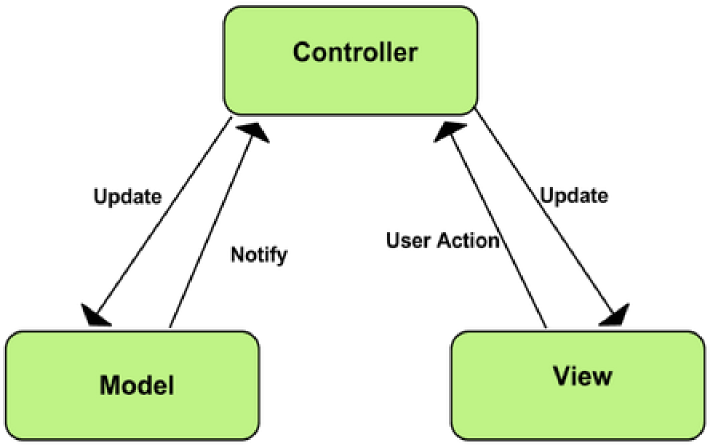

Figure 2.14 Scheme of MVC pattern

In our system the use of MVC design pattern in three tier architecture is presented in next picture:

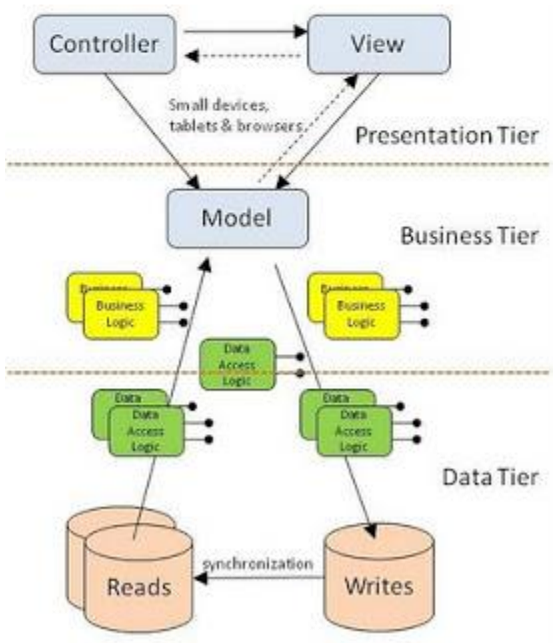

Figure 2.14 Scheme of MVC pattern as 3-tier architecture in myTaxiService system

When the MVC and three-tier approaches are brought together the View and Controller are considered the presentation tier, the Model exists in the business tier (and has access to many business and data tier modules). To a certain extent the Model could span both the business and data tiers. Writes take considerably more data storage resources than do reads, so separating these into different physical data stores can have huge benefit. It should also identify that the modules for the business and data logic can be numerous. And these exist for maintainability and scalability purposes. It is also reasonable that the Model may access the Data Tier directly without going through a Business Tier module.

2.8 Other design decisions

We choose cross-platform mobile development because in the business world, a growing trend called BYOD (Bring Your Own Device) is rising. BYOD refers to employees bringing their own personal mobile device into the workplace to be used in place of traditional desktop computers or company-provided mobile devices for accessing company applications and data. Because of BYOD, it has become necessary for businesses to develop their corporate mobile apps and be able to send them to many different mobile devices that operate on various networks and use different operating systems. Cross-platform mobile development can either involve a company developing the original app on a native platform (which could be iOS, Android, Windows Mobile, BlackBerry/RIM, etc.) or developing the original app in a singular environment for development that will then allow the app to be sent to many different native platforms

One of the big advantage of cross platform mobile development is the fact that they decrease cost and increase speed of application development. In addition, cross-platform mobile development tools are generally quite simple to use as they are based off of the common languages for scripting, including CSS, HTML, and JavaScript.

3 Algorithm design

3.1 Request Algorithm

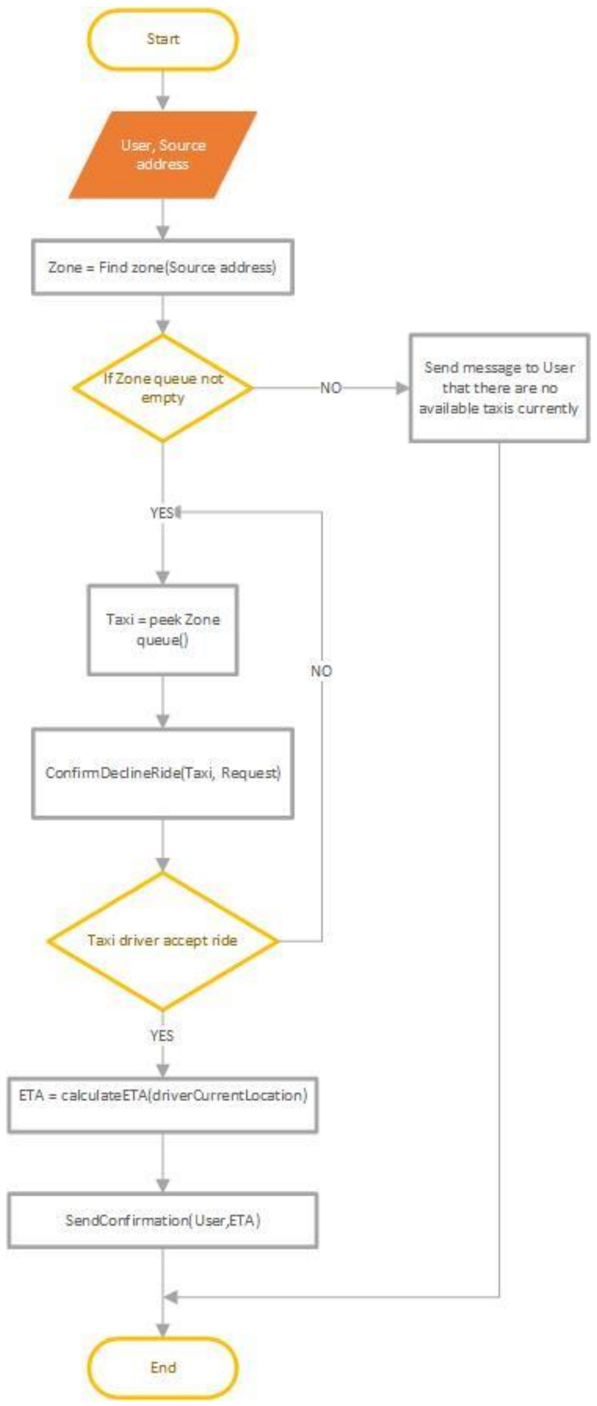
Start
User, Source address
Zone = Find zone(Source address)
If Zone queue not empty
NO
Send message to User that there are no available taxis currently
YES
Taxi = peek Zone queue()
NO
ConfirmDeclineRide(Taxi, Request)
Taxi driver accept ride
YES
ETA = calculateETA(driverCurrentLocation)
SendConfirmation(User,ETA)
End

Figure 3.1 Algorithm Request a taxi

3.2 Reserve Algorithm

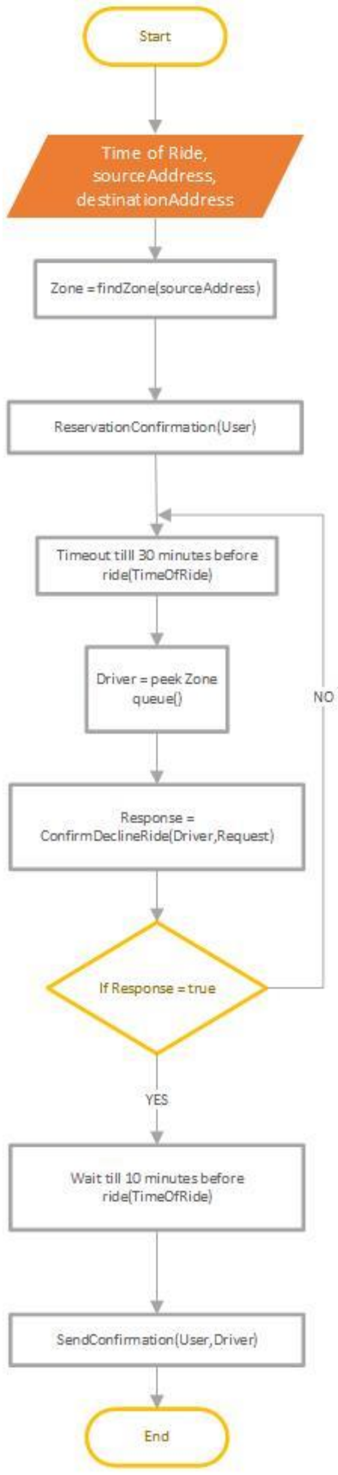
Start
Time of Ride, sourceAddress, destinationAddress
Zone = findZone(sourceAddress)
ReservationConfirmation(User)
Timeout tilll 30 minutes before ride(TimeOfRide)
Driver = peek Zone queue()
Response = ConfirmDeclineRide(Driver,Request)
If Response = true
NO
YES
Wait till 10 minutes before ride(TimeOfRide)
SendConfirmation(User,Driver)
End

Figure 3.2 Algorithm Reserve a taxi

3.3 ConfirmDecline Algorithm

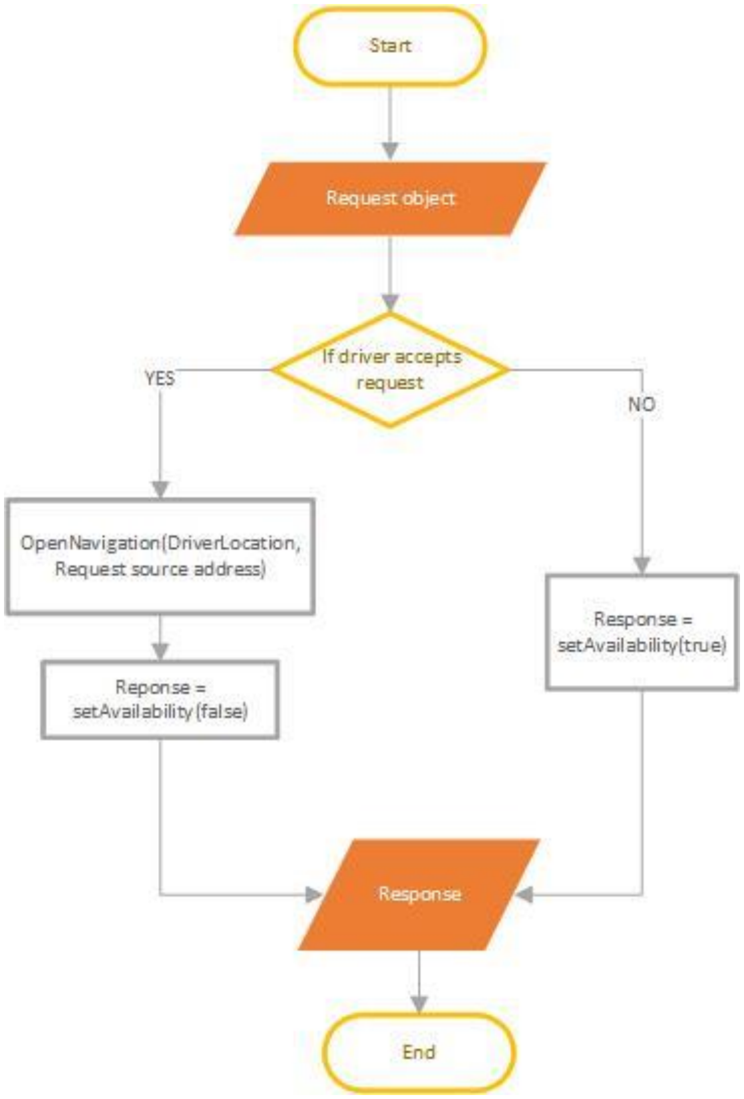
Start
Request object
If driver accepts request
YES
NO
OpenNavigation(DriverLocation, Request source address)
Reponse = setAvailability(false)
Response = setAvailability(true)
Response
End

Figure 3.3 Algorithm Confirm/Decline a ride

3.4 SetAvailability Algorithm

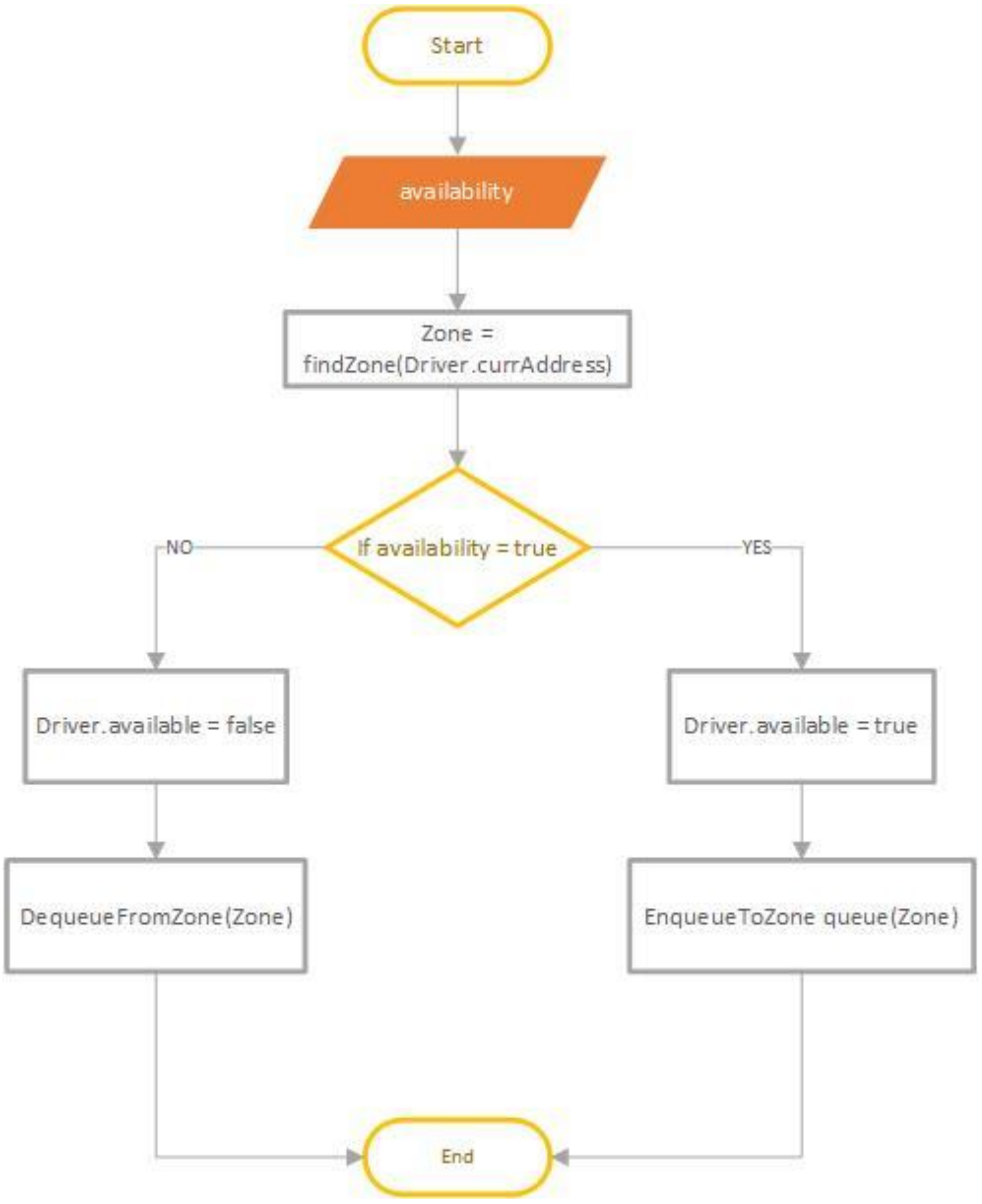

Figure 3.4 Algorithm for setting taxi availability

4 User interface design

Refer to the chapter 3.1 of RASD documents. In chapter 3.1 are shown user interfaces of myTaxiService system.

5 Requirement traceability

The non-functional requirements that we have presented in RASD document are accomplished through the overall architecture of our system (see Section 2.). In this section, we are going to provide a more precise list of all the functional requirements and their relative mapping to components in the architecture (the detailed interaction of most important components is presented in the *Runtime view* section). For more detailed view of how components are distributed in our system and how these components interact, see *High level components* section and component interfaces section respectively.

Functional requirements	**Components**
1.-2.UseCase: Accessing the application- Sign Up and Sign In	Guest Manager, Data Layer
3.UseCase: Manage Profile for user and driver	User Manager, Taxi Driver Manager, Data Layer
4.UseCase: Requesting a taxi	User Manager, Scheduler, Taxi Driver Manager, Data Layer, Google Maps API, Google Places API
5.UseCase: Reserving a taxi	User Manager, Scheduler, Taxi Driver Manager, Data Layer, Google Places API
6.UseCase: Canceling a ride	User Manager, Data Layer, Taxi Driver Manager, Scheduler
7.UseCase: Confirm/Decline a ride	Scheduler, Driver Manager , Google Maps API
8.UseCase: Check Taxis Available	User Manager, Scheduler, Google Maps API, Data Layer
9.UseCase: Report user/driver	User Manager or Driver Manager, Data Layer
10.UseCase: Ban user/driver	Admin Manager, DataLayer

Table 5.1 Mapping functional requirements into architecture components of myTaxiService system

6 References

- IEEE Design Document template
- MVC in a three-tier architecture, Peter Rawsthorne
- http://www.kony.com/resources/glossary/cross-platform-mobile-development
- https://msdn.microsoft.com/en-us/library/ee658117.aspx
- Singleton Design Pattern- Java Tech Guy must read!, Rudra Narayan Garnaik
- Software Design and Software Architecture, Politecnico di Milano
- Advanced Software Design and architectures: Architecture styles, Damian A. Tamburri