POLITECNICO DI MILANO

A.A. 2015-2016

SOFTWARE ENGINEERING 2: “myTaxiService”

Design Document

Andrijana Mirchevska (838622)

Marija Mavcheva(838647)

04 December 2015

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to provide a comprehensive description of the myTaxiService system. It's aim is to 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 project is about developing an application that will enable fast and optimized taxi services in the city. The application will allow users to register and then sign in into the app for using its services. Also taxi drivers can register and sign into the myTaxiService application with the purpose to manage their availability and duties.

Once registered into the app, the user can do the following actions:

- request a taxi
- make reservation
- cancel reservation
- find available taxi near
- receive confirmation about the vehicle, its code and estimated waiting time
- manage user profile
- report driver

Also the registered taxi driver once signed in can do these actions:

- inform the system about availability
- confirm/decline a request for taxi call
- manage driver profile
- report user

1.3 Glossary

MakeReservation	Passenger request a vehicle at least 2 hours before the ride
Request	Passenger request an immediate ride
RideConfirmation	Receive a confirmation about the confirmed ride with all the information about the particular ride
ReservationConfirmation	Notification that the reservation is successfully completed
ReportDriver	Passenger reports driver in case of any irregularities during the ride
ReportUser	Taxi driver reports user (particular passenger) in case of any irregularities during the ride
Guest	Not registered person that visits the app
User	A person that is already registered and signed in as user
TaxiDriver	A person that is already registered and signed in as driver
API	Application Programming Interface
GPS	Global Positioning System

1.4 Reference Documents

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

1.5 Document Structure

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

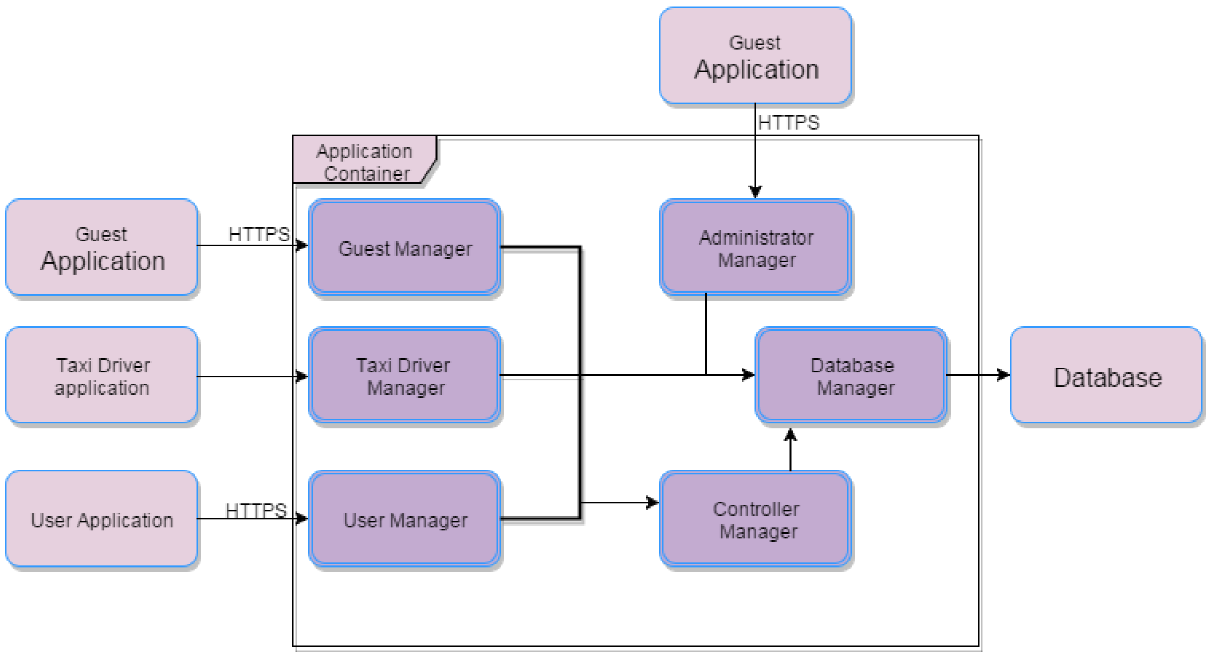
- Chapter 4: User interface Design, overview on how the user interfaces of your system will look
- Chapter 5: Requirements traceability, gives an overview of how the requirements defined in RASD map into the design elements defined in DD.
- Chapter 6: References

2. Architectural Design

2.1 Overview

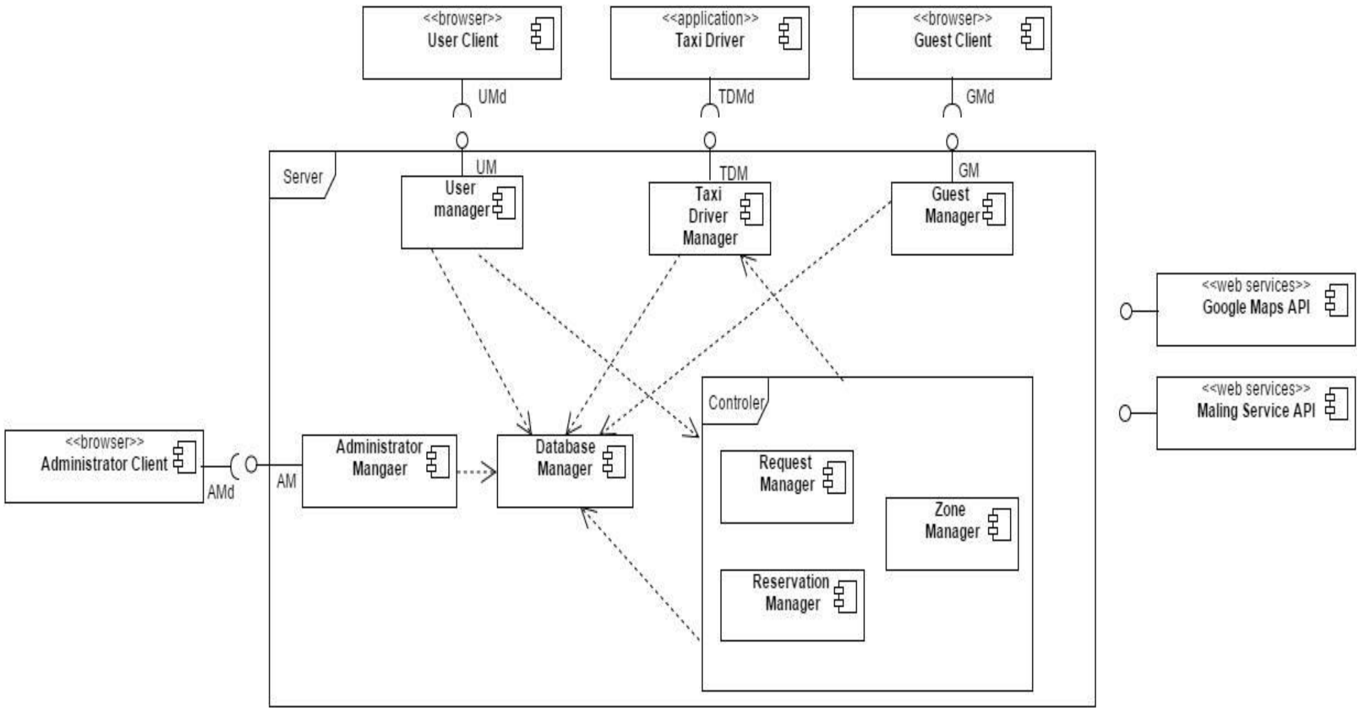
This is a high level overview of how responsibilities of the system were partitioned and then assigned to subsystems. Here we Identify the high level subsystem and the roles or responsibilities assigned to it, also the collaboration of these subsystems between each other in order to achieve the desired functionality.

2.2 High level components and their interaction

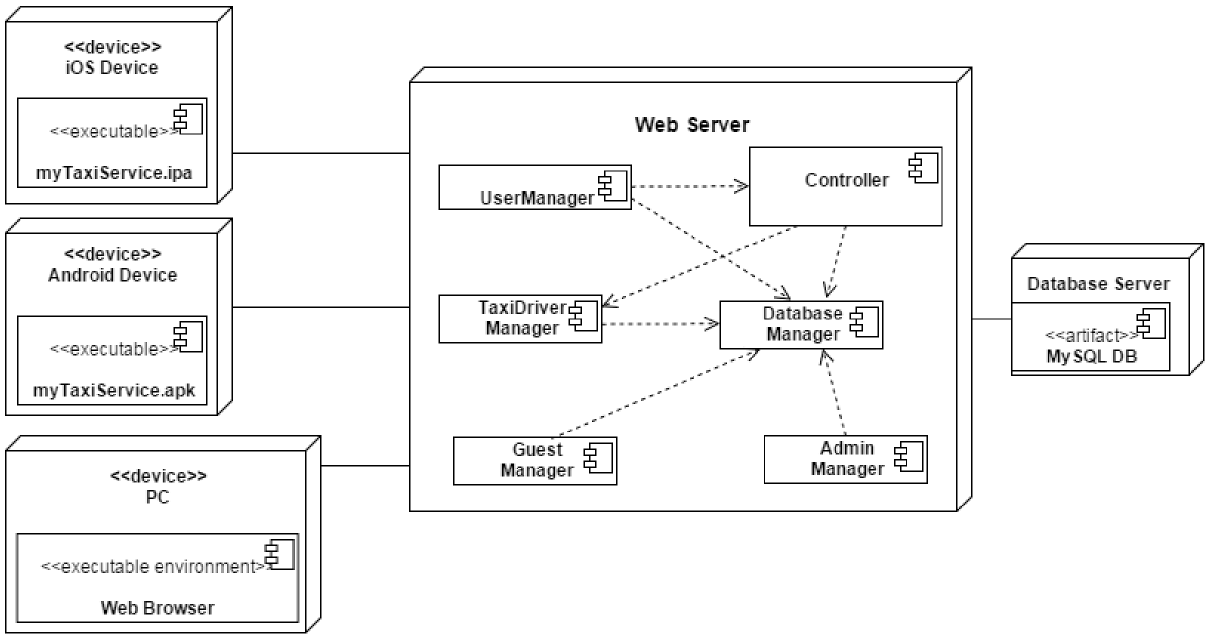


- *Guest Manager* will provide interface for users that are not registered or for guest that is planning to open a profile on the application.
- *User Manager* will provide interfaces for users (passengers) with options of requesting, reserving taxi, managing profile, and reporting driver.
- *Taxi Driver* Manager will provide interfaces for taxi drivers with options of confirming/declining a ride, reporting a user and canceling a ride.
- *Adminstator Manager* is component of the system that will provide interfaces for administrators with options of viewing reports and banning a user.
- *Controller Manager* is component that is structured of three subcomponents: Request Manager, Reservation Manager and Zone Manager. It provides interfaces for other components with options for requesting a taxi, finding zone, as well as reserving a taxi.

2.3 Component view

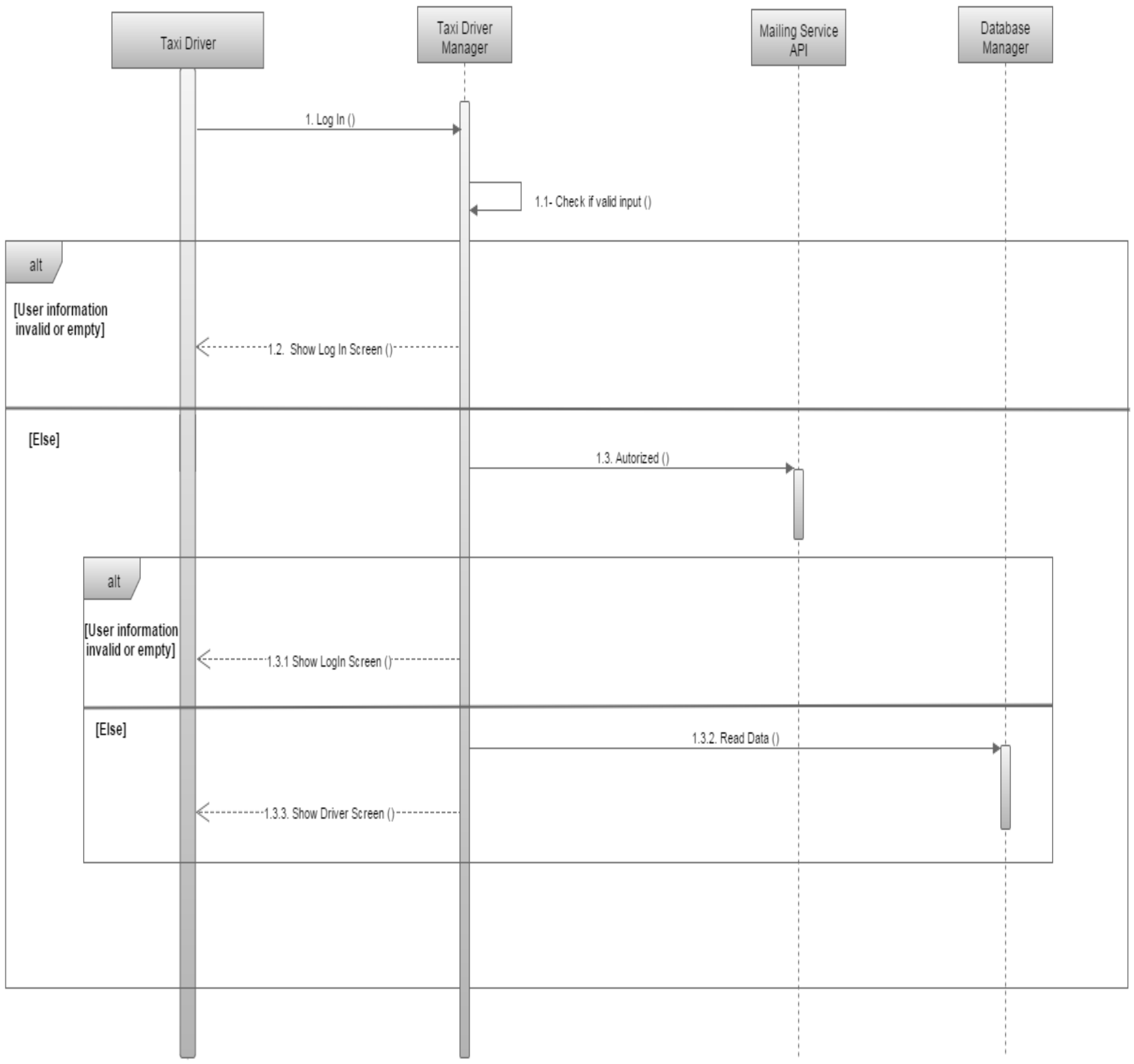


2.4 Deployment view

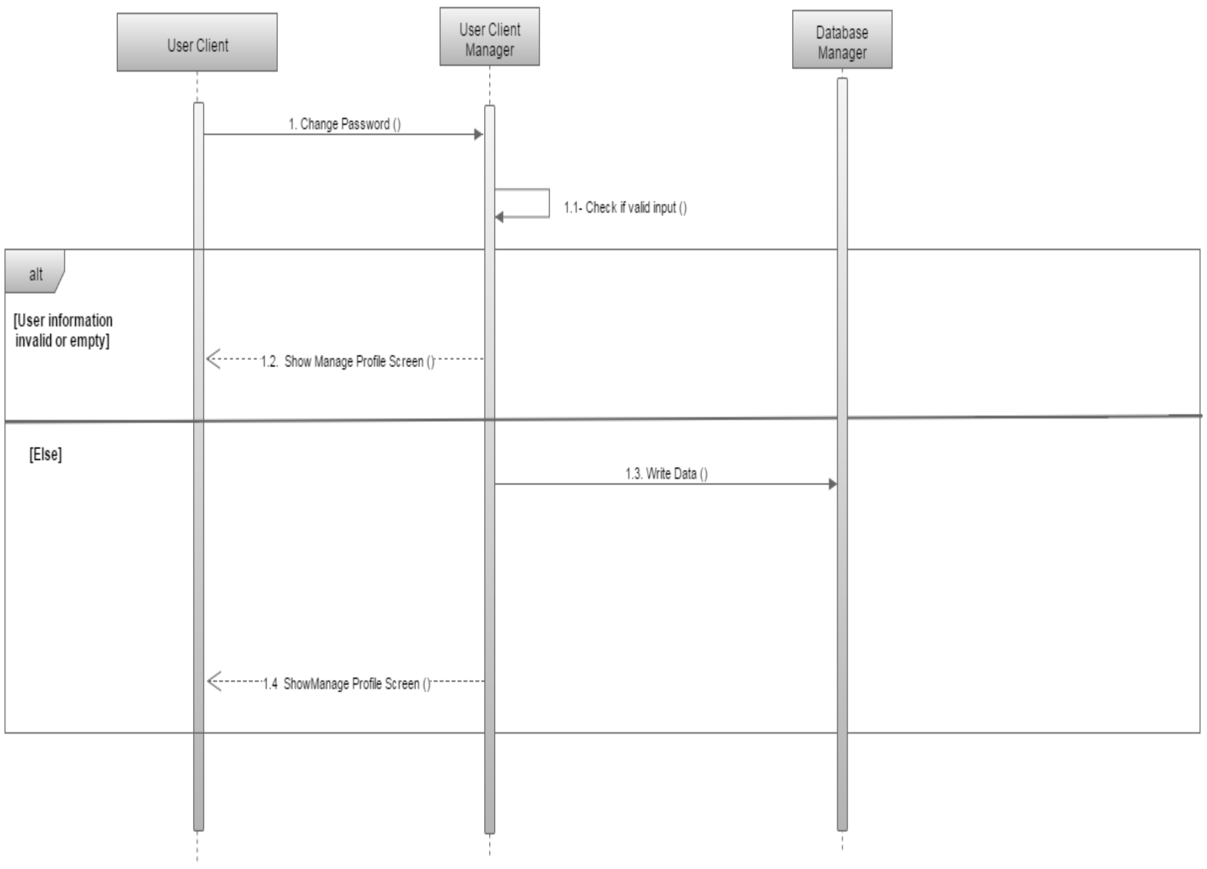


2.5 Runtime view

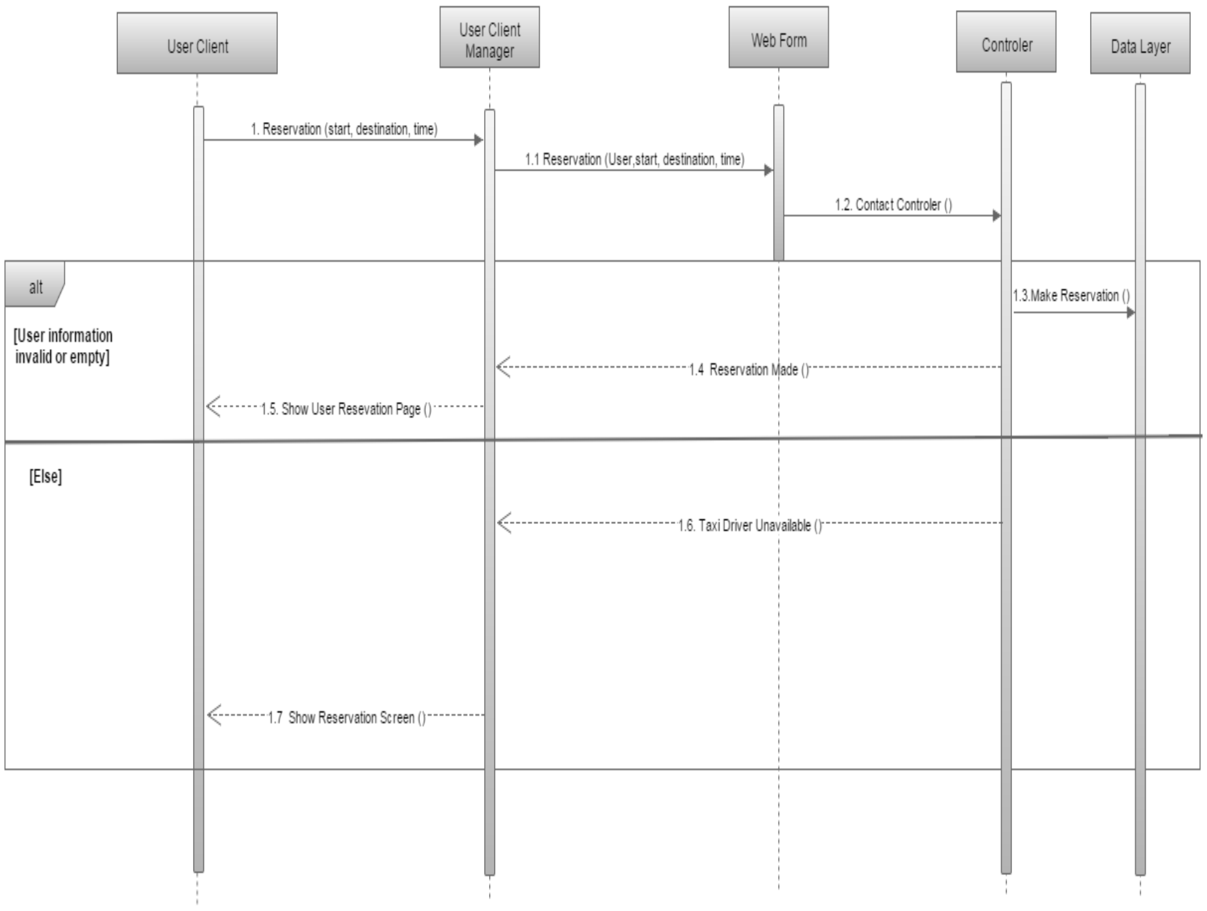
2.5.1 Log In


Taxi Driver
Taxi Driver Manager
Mailing Service API
Database Manager
1. Log In ()
1.1- Check if valid input ()
alt
[User information invalid or empty]
1.2. Show Log In Screen ()
[Else]
1.3. Autorized ()
alt
[User information invalid or empty]
1.3.1 Show LogIn Screen ()
[Else]
1.3.2. Read Data ()
1.3.3. Show Driver Screen ()

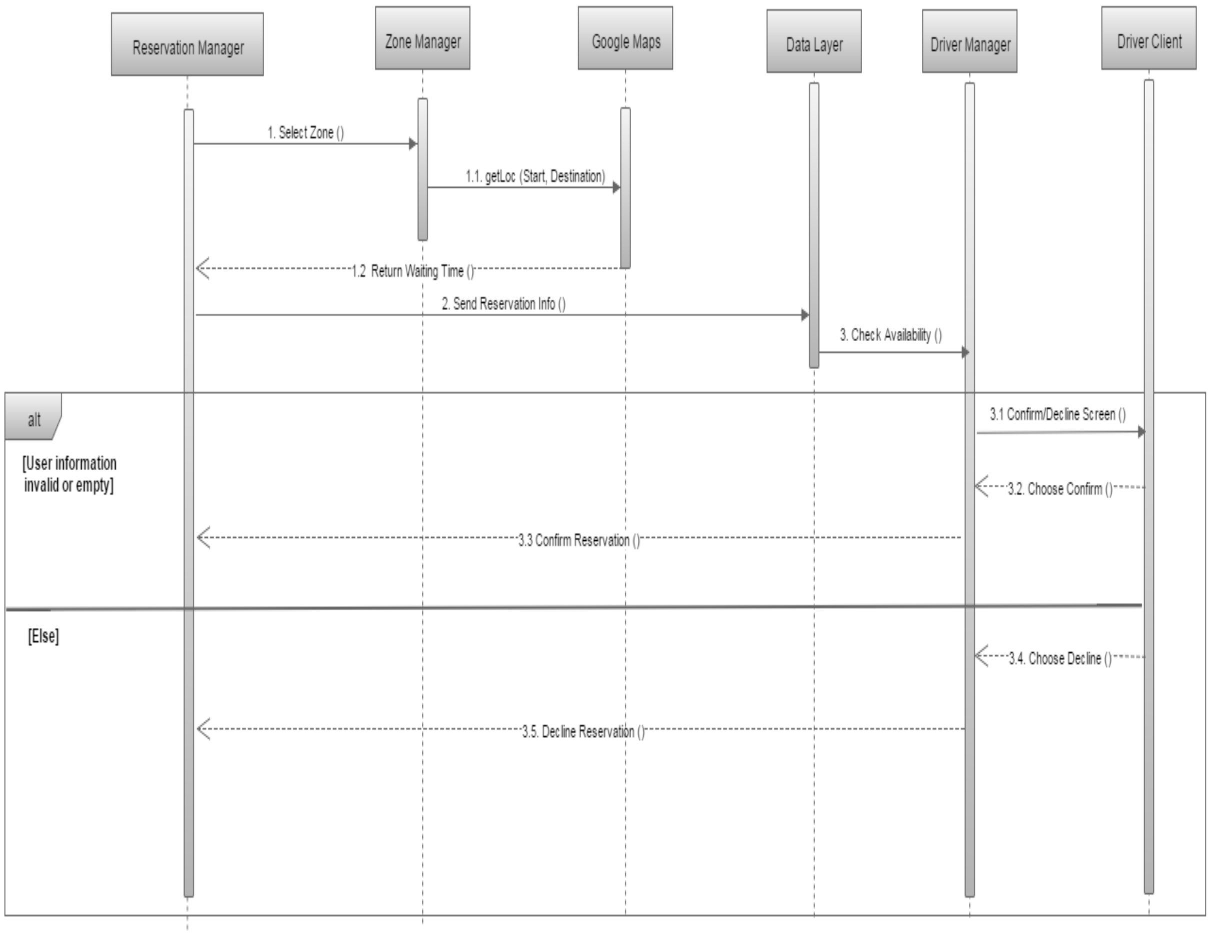
2.5.2 Manage Profile


User Client
User Client Manager
Database Manager
1. Change Password ()
1.1- Check if valid input ()
alt
[User information invalid or empty]
1.2. Show Manage Profile Screen ()
[Else]
1.3. Write Data ()
1.4 ShowManage Profile Screen ()

2.5.3 Reservation


User Client
User Client Manager
Web Form
Controler
Data Layer
1. Reservation (start, destination, time)
1.1 Reservation (User,start, destination, time)
1.2. Contact Controler ()
alt
[User information invalid or empty]
1.3.Make Reservation ()
1.4 Reservation Made ()
1.5. Show User Resevation Page ()
[Else]
1.6. Taxi Driver Unavailable ()
1.7 Show Reservation Screen ()

2.5.4. Reservation: Controller part



Reservation Manager
Zone Manager
Google Maps
Data Layer
Driver Manager
Driver Client
1. Select Zone ()
1.1. getLoc (Start, Destination)
1.2 Return Waiting Time ()
2. Send Reservation Info ()
3. Check Availability ()
alt
[User information invalid or empty]
3.1 Confirm/Decline Screen ()
3.2. Choose Confirm ()
3.3 Confirm Reservation ()
[Else]
3.4. Choose Decline ()
3.5. Decline Reservation ()

2.6 Component interfaces

2.6.1 Guest Manager

The Guest Manager interface has the following functions:

- ***User register(name, surname, email, password, avatar)***

  The *register* function after submitting the inputted data sends it to the web server which checks the credentials if they already exist. It uses the Mailing Service 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 logIn(email, password)***

  The *logIn* 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 register function.

2.6.2 Taxi Driver Manager

The Taxi Driver Manager interface has the following functions:

- ***String confirmDeclineRide(reservation)***

  The *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)).

2.6.3 User Manager

The User Manager interface has the following functions:

- ***Reservation makeReservation(zone, initialLocation, destination, time, user)***

  The *makeReservation* function creates Reservation object, with user, initialLocation and destination and time of the ride initialized. Then it uses findZone(initialLocation) function of Controller 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. When the reservation is sent to the 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.

- ***Request request(zone, initialLocation, destination, user)***

  The *request* function creates Request object, with user and initial location entered. Then it uses findZone(initialLocation) function of Controller 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.

- ***String report(description, id_user, id_banned)***

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

- ***String manageProfile(user)***

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

- ***String cancelRide(reservation)***

  The *cancelRide* function deletes particular tuple of the databese. After that, notification message is sent both to user and driver related to this reservation.

2.6.4 Administrator Manager

The Administrator Manager interface has the following functions:

- ***String BanUser(id_user)***

  The banUser is function of Administrator 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.

- ***See Reports(viewReports)***

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

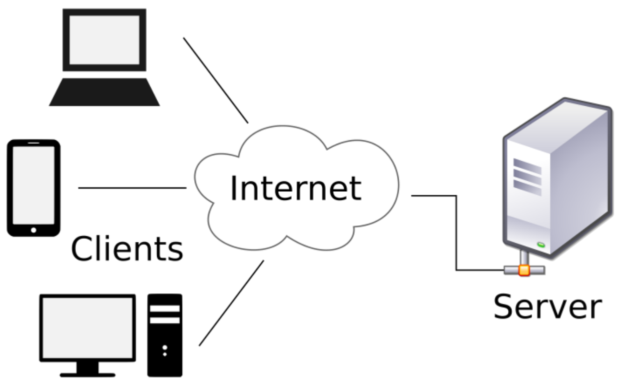
2.7 Selected architectural styles and patterns

2.7.1 Client/Server Architectural Style

The client/server architectural style describes distributed systems that involve a separate client and server system, and a connecting network. The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style.
More generally, the client/server architectural style describes the relationship between a client and one or more servers, where the client initiates one or more requests (perhaps using a graphical UI), waits for replies, and processes the replies on receipt. The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client.
Today, some examples of the client/server architectural style include Web browser—based programs running on the Internet or an intranet; Microsoft Windows® operating system—based applications that access networked data services; applications that access remote data stores (such as e-mail readers, FTP clients, and database query tools); and tools and utilities that manipulate remote systems (such as system management tools and network monitoring tools).
Other variations on the client/server style include:

- **Client-Queue-Client systems**. This approach allows clients to communicate with other clients through a server-based queue. Clients can read data from and send data to a server that acts simply as a queue to store the data. This allows clients to distribute and synchronize files and information. This is sometimes known as a *passive queue* architecture.
- **Peer-to-Peer (P2P) applications**. Developed from the Client-Queue-Client style, the P2P style allows the client and server to swap their roles in order to distribute and synchronize files and information across multiple clients. It extends the client/server style through multiple responses to requests, shared data, resource discovery, and resilience to removal of peers.
- **Application servers**. A specialized architectural style where the server hosts and executes applications and services that a thin client accesses through a browser or specialized client installed software. An example is a client executing an application that runs on the server through a framework such as Terminal Services.

The client-server architectural style has evolved into the more general 3-Tier (or N-Tier) architectural style which overcomes some of the disadvantages inherent in the 2-Tier client-server architecture and provides additional benefits.



2.7.2 N-Tier / 3-Tier Architectural Style

N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment. Each tier is completely independent from all other tiers, except for those immediately above and below it. N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality.

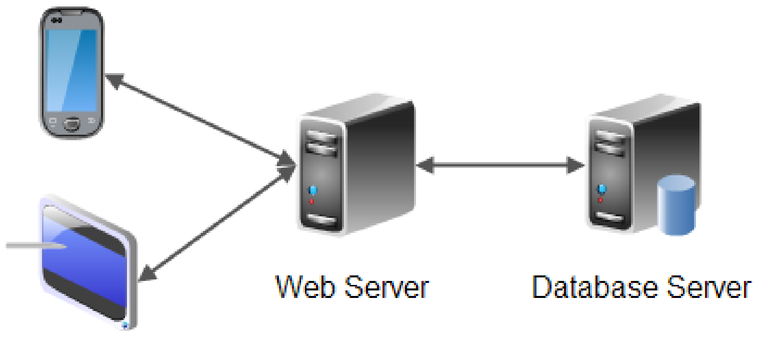
N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality. When using a layered design approach, a layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.

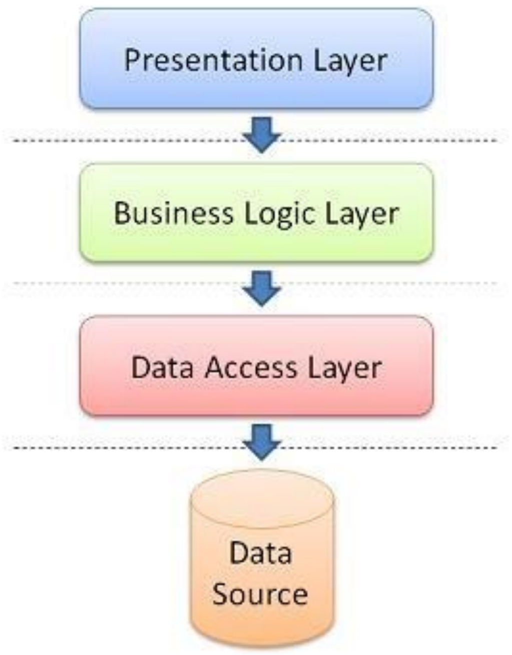
Three-tier architecture:

Presentation tier
This is the topmost level of the application. In simple terms it is a layer which users can access directly such as a web page, or an operating systems GUI

Application tier (business logic, logic tier, or middle tier)
The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing.

Data tier
The data tier includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. The data access layer should provide an API to the application tier that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms.





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.

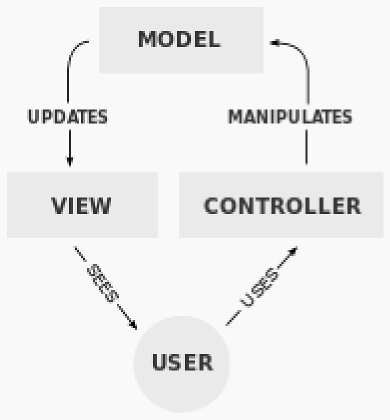
myTaxiService system will be built on three tier client-server architectural style. Communication between the users and the taxi drivers goes through the server as a mediator. Purpose of the server in myTaxiSystem is to handle both users and drivers requests. Server processes the requests and queries 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 reserve a taxi at least 2 hours before the ride
- reservation is sent to the server
- server finds the zone and takes first available taxi from the queue
- server forwards the reservation to the taxi driver
- driver responds confirm/decline
- response is sent to the server
- response is sent to user

2.7.3 Patterns

Model–view–controller (**MVC**) is a software architectural pattern mostly for implementing user interfaces (but not only for user interface). It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

Components



A typical collaboration of the MVC components.

The central component of MVC, the *model*, captures the behavior of the application in terms of its problem domain, independent of the user interface.[11] The model directly manages the data, logic and rules of the application.
A *view* can be any output representation of information, such as a chart or a diagram;

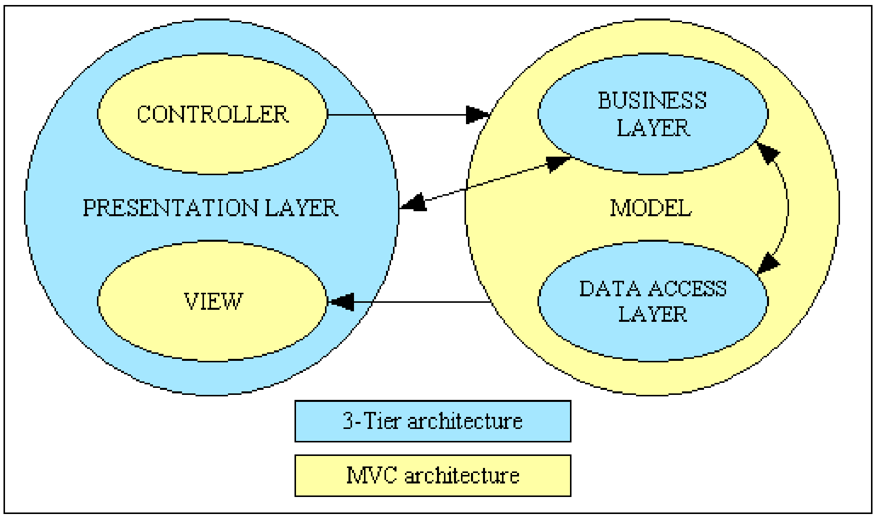
multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
The third part, the *controller*, accepts input and converts it to commands for the model or view.

2.7.4 Interactions

In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.

- A **controller** can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).
- A **model** stores data that is retrieved according to commands from the controller and displayed in the view.
- A **view** generates an output presentation to the user based on changes in the model.
- A **view controller** generates an output view and an embedded controller

The connection, in our system, between MVC design pattern and the 3-tier architecture is shown on the picture bellow:



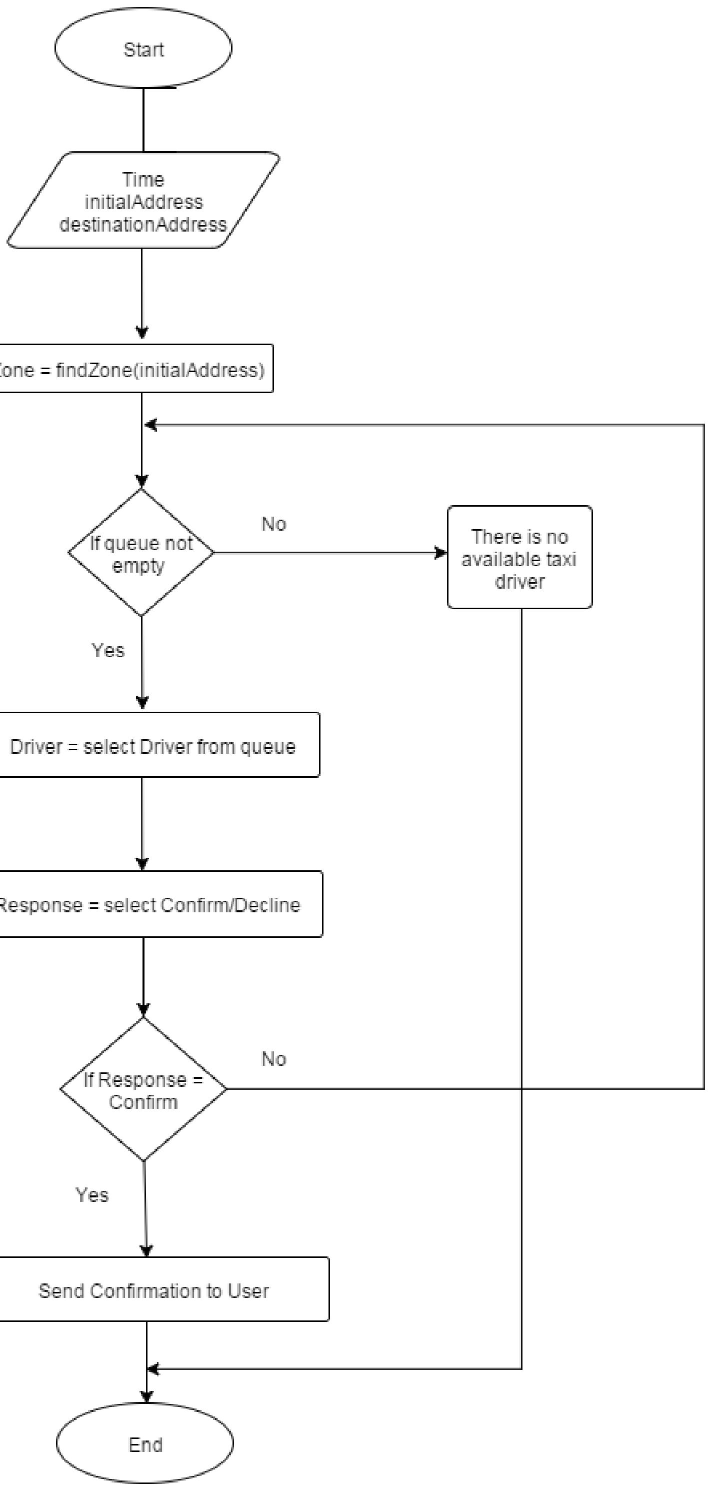
How do these two come together. From a MVC perspective the View and Controller exists in the presentation tier and the model spans the business and data tiers. From a three-tier perspectivethis means the Model is broken into modules which are optimized based on their activity and for their ability to be reused and altered for new business opportunities. The Model will span the business and data tiers. The Controller and View exist in the presentation tier and this is also optimal for it allows the software developer to build, render and respond to user interfaces best designed for the different devices (internet browsers and mobile devices).

2.7.5 Why the hybrid?

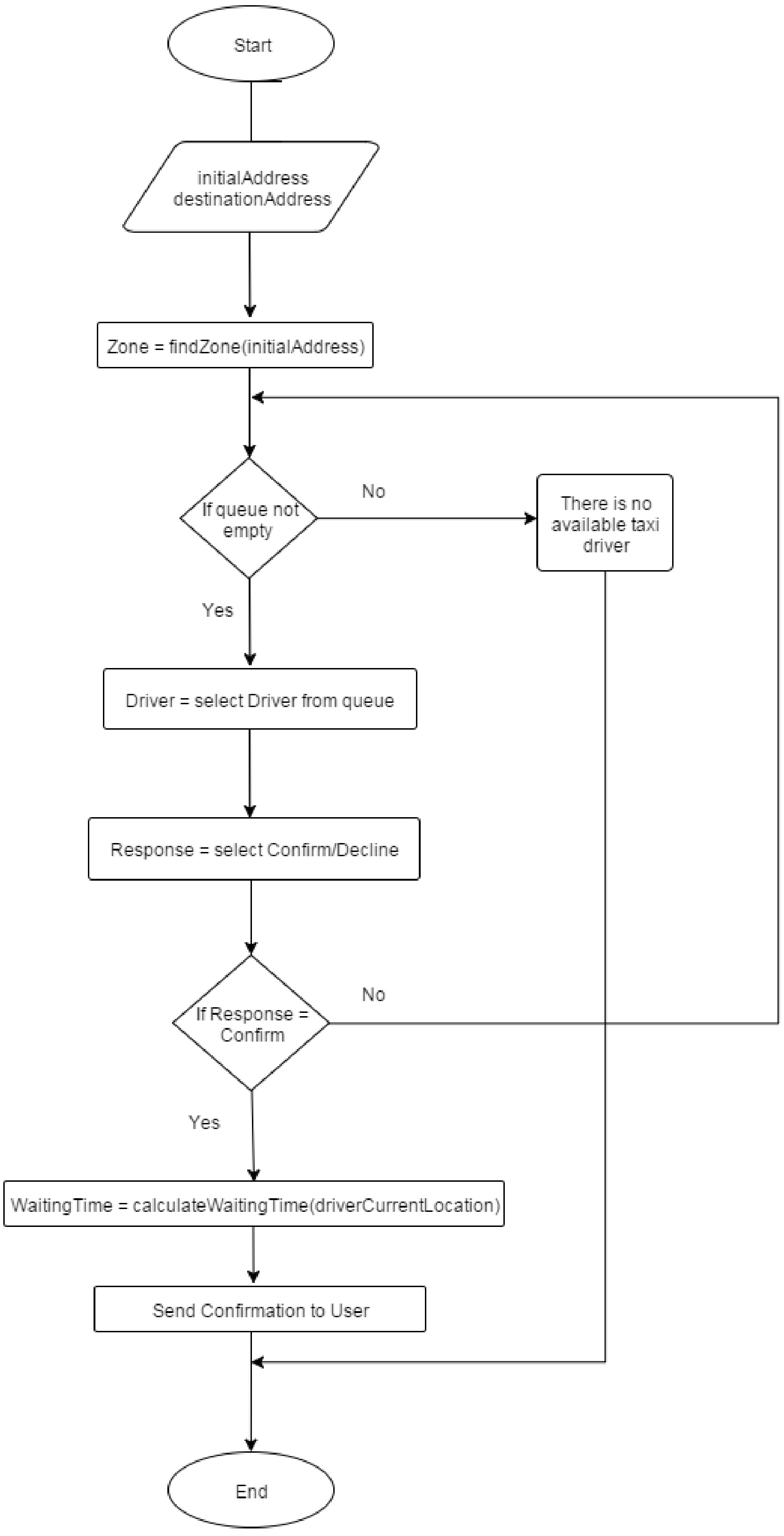
Because building websites can be complicated, particularly when the site engages users and have a lot of content that can be targeted toward and created by users. Why the hybrid of MVC and three-tiers? Two main reasons; First, because the MVC pattern does a great job of simplifying and managing the development of user interfaces over multiple devices and browsers, but it doesn't do a good job of defining how to build scalable server infrastructures. Second, The three-tier architecture does a great job of simplifying and managing the development of high performing, scalable,extensible and maintainable server infrastructures, but doesn't do a good job of defining how to build user interfaces across multiple devices and browsers. With a MVC three-tier hybrid you can utilize the best of both approaches without compromising either.

3. Algorithm design

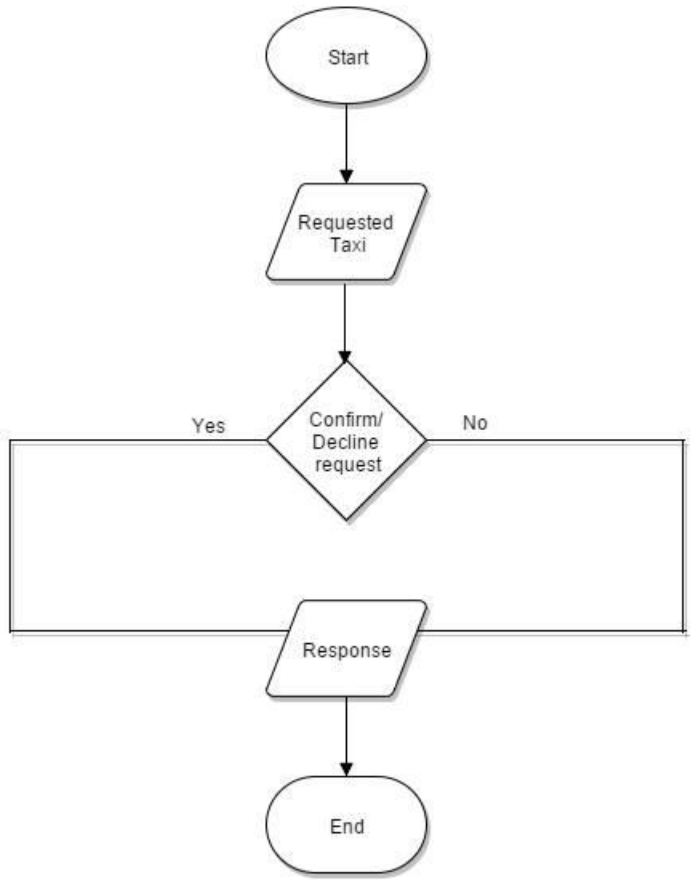
3.1 Reservation Algorithm


Start
Time
initialAddress
destinationAddress
Zone = findZone(initialAddress)
If queue not empty
No
There is no available taxi driver
Yes
Driver = select Driver from queue
Response = select Confirm/Decline
If Response = Confirm
No
Yes
Send Confirmation to User
End

3.2 Request algorithm


Start
initialAddress
destinationAddress
Zone = findZone(initialAddress)
If queue not empty
No
There is no available taxi driver
Yes
Driver = select Driver from queue
Response = select Confirm/Decline
If Response = Confirm
No
Yes
WaitingTime = calculateWaitingTime(driverCurrentLocation)
Send Confirmation to User
End

3.3 Confirm/Decline Algorithm



4. User interface design

Refer to the chapter 3.2 of RASD documents, specifically part 3.2.1 where we show the user interfaces of our application.

5. Requirement traceability

In this part is provided a more precise list of all the functional requirements and their mapping to the high level components in the architecture of our system.

- UseCase: Accessing the application - Register and LogIn is mapped to the Guest Manager and Database Manager Components.
- UseCase: Manage Profile for user and driver is connected to the User Manager, Taxi Driver Manager, Database Manager Components.
- UseCase: Requesting a taxi is mapped through User Manager, Controller, Taxi Driver Manager, Database Manager components, Google Maps API.
- UseCase: Reserving a taxi is connected with User Manager, Controller, Taxi Driver Manager, Database Manager components, Google Maps API
- UseCase: Canceling a ride is mapped with User Manager, Database Manager, Taxi Driver Manager, Controller
- UseCase: Confirm/Decline a ride is connected with Controller, Driver Manager, Google Maps API
- UseCase: Report user/driver is mapped with User Manager or Driver Manager, and Database Manager.

6. References

- Microsoft Application Architecture Guide, 2nd Edition - Chapter 3: Architectural Patterns and Styles
- MVC in a three-tier architecture from, Peter Rawsthorne
- Distributed Systems Book, 2nd Edition, Andrew S.Tanenbaum - Chapter 2: Architectural Styles
- IEEE Design Document template