

Politecnico di Milano

2015-2016

Software Engineering 2: “MyTaxiService”

Integration Test Plan Document

version 1.0

Author: Nenad Petrovic

13th January 2016

Contents

[1 Introduction 3](#_Toc437003990)

[1.1 Purpose 3](#_Toc437003991)

[1.2 Scope and document overview 3](#_Toc437003992)

[1.3 Definitions, acronyms, abbreviations 4](#_Toc437003993)

[1.4 Reference documents 7](#_Toc437003994)

[2 Architectural design 8](#_Toc437003995)

[2.1 Overview 8](#_Toc437003996)

[2.2 High level components and their interaction 9](#_Toc437003997)

[2.3 Component view 11](#_Toc437003998)

[2.4 Deployment view 14](#_Toc437003999)

[2.5 Runtime view 17](#_Toc437004000)

[2.6 Component interfaces 25](#_Toc437004001)

[2.7 Selected Architectural Styles and Patterns 40](#_Toc437004002)

[3 Algorithm desgin 46](#_Toc437004003)

[3.1 Scheduling algorithm 46](#_Toc437004004)

[3.2 Taxi zone determination 48](#_Toc437004006)

[3.3 Cost estimation algorithm 52](#_Toc437004007)

[4 User interface design 54](#_Toc437004008)

[5 Requirements traceability 66](#_Toc437004009)

[6 Database design 68](#_Toc437004010)

[7 Software and tools used 69](#_Toc437004011)

[7.1 Hours of works 70](#_Toc437004012)

[8 References 70](#_Toc437004013)

# Introduction

## Revision history

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Title | Version | Author | Date | Summary |
| Integration Test Plan Document | 1.0 | Nenad Petrovic | 13-01-2016 | The first version of the document |

## Purpose and scope

The Integration Test Plan Document (ITPD) that is going to be presented describes the plans for testing the integration of the created components.

This document is supposed to be written before the integration test happens, and takes the architectural description from Design Dociment as a starting point.

The purpose of this document is to test the interfaces between the components as described in Design Document.

This software implements a new online taxi scheduling service that would help the government of a large city to simplify the accesss of passengers to the taxi drivers, and would also guarantee a fair management of taxi queries at the same time by offering applications to both user and taxi-driver side with corresponding functionalities for each of them.

This document aims every team member who cooperates in the integration test, and should be read by them.

## Definitions, acronyms, abbreviations

1.3.1 Definitions

* Starting point: is a location where the drive should start from, determined by its GPS coordinates.
* Ending point: is a location where the current drive stops, determined by its GPS coordinates. At this point, taxi driver’s availability changes automatically to available.
* Request: is a message which consists of user’s desired destination for a taxi drive and, optionally, maximum waiting time . The user himself is a „sender“ of a request, while the taxi driver who receives the request is called „receiver“. If there is no taxi driver who can receive the request, the receiver part of the message is empty (user receives a message stating that there is no taxi available and gives him/her an option to send the same request again or change the desired destination), and in case of forwarding the message to another taxi driver, system changes the receiver to the taxi driver polled from a queue. The request constains starting point-determined by sender’s GPS location and ending point- which is, in fact the desired destination selected by user.
* Response: is a message which contains „y“-yes in case where taxi driver accepts the request or „n“-no in case where taxi driver rejects the request by user. This message also contains the estimated time needed for taxi driver to come to the requested starting point and estimated price. In this case, taxi driver has a role of sender, while the user who sent the request is receiver. Receiver part of this message can’t be blank. User can accept or reject the drive offer.
* Report:is a message written by user or taxi driver during the drive event, in order to mention bad behaviour of the other side. There is no strict definition for „bad behaviour“, so the one who reports has to write reason and describe the situation itself as close as possible. After that, administrators can view the reports and decide to delete user from system or not. User that is banned from system is prevented from using it and can’t login or register once again, because his/her fiscal code is on the „black list“. Single user can receive many reports. Reports are stored in database and could be viewed by administration.
* Taxi zone: a part of city (approximately 2km2), which is defined by its center point, and its boundaries are calculated and managed by the system. Each taxi zone has its taxi queue, which consists of car identification numbers of available taxi drivers whose current location belongs to its boundaries. The city has at least one taxi zone (in case of a very small town).
* Availability: could have value „y“(yes)- which means „available“ or „n“(no)- which means „unavailable“. If taxi driver is available, he/she is placed in taxi queue related to his/her taxi zone and is able to respond to requests belonging to this taxi zone. If taxi driver is unavailable, he/she can’t receive requests. When taxi driver accepts the request, his availability switches to „unavailable“. After finishing the drive, the availability switches to „available“ and is placed in a queue belonging to his/her current taxi zone. In certain cases, when something goes wrong (traffic rush, accident etc.),taxi driver can manually switch to „unavailable“. When taxi driver sends S.O.S signal, his/her status changes to „unavailable“ automatically.
* Drive event: is a system abstraction of a taxi drive, and consists of request by user, driver’s response and reports during a drive. There could be many reports, or no reports at all. Administrations can browse the database of drive events and see the actors of a drive event related to request and response. During active drive event, user can report driver, or driver can report user for bad behavior by describing the reason of report as close as possible. Drive event starts after the user accepts the offer by taxi driver. Sometimes, it is called simply “drive” in this text.
* Estimated waiting time: time needed for a taxi driver to come to the user’s current location.
* Maximum waiting time: maximum time that is user ready to spend waiting for a taxi to come.
* Bad behaviour: aggresion, avoiding payment, offensive beahviour etc.
* Black-box testing: assumes that the structure of the software being tested is unknown. Tests are based on knowledge of the system specification.
* White-box testing: based on the knowledge of the software and attemps to identify test sets that cause execution of all the instructions or all the branches or specfic paths in system.

1.3.2 Acronyms

* API: Application Programming Interface.
* DD: Design Document.
* DBMS: DataBase management system.
* DB: DataBase.
* EJBs: Enterprise JavaBeans (EJB) is a managed, [server](https://en.wikipedia.org/wiki/Server_(computing)) software for modular construction of [enterprise software](https://en.wikipedia.org/wiki/Enterprise_software), and one of several [Java APIs](https://en.wikipedia.org/wiki/List_of_Java_APIs). EJB is a [server-side](https://en.wikipedia.org/wiki/Server-side) [software component](https://en.wikipedia.org/wiki/Component_(software)) that [encapsulates](https://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)) the [business logic](https://en.wikipedia.org/wiki/Business_logic) of an application. The EJB specification is a subset of the [Java EE](https://en.wikipedia.org/wiki/Java_EE) specification.
* GUI: Graphical User Interface

HTTPS ( HTTP over SSL, and HTTP Secure) is a [protocol](https://en.wikipedia.org/wiki/Communications_protocol) for [secure](https://en.wikipedia.org/wiki/Network_security) communication over a [computer network](https://en.wikipedia.org/wiki/Computer_network) which is widely used on the [Internet](https://en.wikipedia.org/wiki/Internet). HTTPS consists of communication over [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP) within a connection encrypted by [Transport Layer Security or its predecessor, Secure Sockets Layer](https://en.wikipedia.org/wiki/Transport_Layer_Security). The main motivation for HTTPS is [authentication](https://en.wikipedia.org/wiki/Authentication) of the visited [website](https://en.wikipedia.org/wiki/Website) and protection of the [privacy](https://en.wikipedia.org/wiki/Information_privacy) and [integrity](https://en.wikipedia.org/wiki/Data_integrity) of the exchanged data.

* JPA:The Java Persistence API (JPA) is a  [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) specification that describes the management of [relational data](https://en.wikipedia.org/wiki/Relational_data_model) in applications using [Java Platform, Standard Edition](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition) and [Java Platform, Enterprise Edition](https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition).
* JAX-RS: Java API for RESTful Web Services (JAX-RS) is a [Java programming language](https://en.wikipedia.org/wiki/Java_(programming_language)) [API](https://en.wikipedia.org/wiki/Application_programming_interface) that provides support in creating [web services](https://en.wikipedia.org/wiki/Web_service) according to the [Representational State Transfer](https://en.wikipedia.org/wiki/Representational_State_Transfer) (REST) architectural pattern.
* JEE: Java Enterprise Solution
* JSF:

Java Server Faces (JSF) is a  [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) specification for building [component](https://en.wikipedia.org/wiki/Software_component)-based [user interfaces](https://en.wikipedia.org/wiki/User_interface) for [web applications](https://en.wikipedia.org/wiki/Web_application).[[1]](https://en.wikipedia.org/wiki/JavaServer_Faces#cite_note-1) It was formalized as a standard through the [Java Community Process](https://en.wikipedia.org/wiki/Java_Community_Process) and is part of the [Java Platform, Enterprise Edition](https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition).

JSF 2 uses [Facelets](https://en.wikipedia.org/wiki/Facelets) as its default templating system. Other view technologies such as [XUL](https://en.wikipedia.org/wiki/XUL) can also be employed. In contrast, JSF 1.x uses [JavaServer Pages](https://en.wikipedia.org/wiki/JavaServer_Pages) (JSP) as its default templating system.

* JVM: Java Virtual Machine
* OS: Operating System.
* MVC: Model View Controller
* REST (REpresentational State Transfer) is an architectural style, and an approach to communications that is often used in the development of [Web services](http://searchsoa.techtarget.com/definition/Web-Services-Glossary).
* Enterprise Bean[[1]](#footnote-1)

Written in the Java programming language, an enterprise bean is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called checkInventoryLevel and orderProduct. By invoking these methods, clients can access the inventory services provided by the application.

Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services such as transaction management and security authorization.

Second, because the beans rather than the clients contain the application’s business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant Java EE server provided that they use the standard APIs.

When to Use Enterprise Beans

You should consider using enterprise beans if your application has any of the following requirements:

The application must be scalable. To accommodate a growing number of users, you may need to distribute an application’s components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.

Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.

The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

Types of Enterprise Beans

Table  summarizes the two types of enterprise beans. The following sections discuss each type in more detail.

Table  Enterprise Bean Types

|  |  |
| --- | --- |
| Enterprise Bean Type | Purpose |
| Session | Performs a task for a client; optionally may implement a web service |
| Message-Driven | Acts as a listener for a particular messaging type, such as the Java  Message Service API |

## Reference documents

This document referecnces to previously created document – RASD and DD.

* Requirements and Analysis Specification Document ( „RASD 1.4.pdf“).
* Design Document ( „DD.pdf“ ).
* Integration Test Plan Document („ITPD.pdf“) [this document]

# Integration strategy

## Entry criteria

In order to begin integration test, it is necessary to provide:

-Design Document, which includes both architectural design and detailed design.

-Perform unit tests of functions that belong to corresponding components (unit test plan)[[2]](#footnote-2).

-Integration test plan (this document)

In what follows, the functions are going to be listed fo each of the subsystems:

Administrator

* 1. Browse users

void onClickBrowse(Event e)

Users[] browseUsers(string username, string firstName, string lastName, string fiscalCode)

* 1. View reports

void onSelectUser(Event e)

Report[] viewReports (string userid)

* 1. View drives

void onSelectUser(Event e)

Drive[] viewReports (string userid)

* 1. View drive and view report

void onSelectDrive(Event e)

void onSelectReport(Event e)

void viewDrive(string driveId)

void viewReport(string reportId)

Drive viewDrive (string userid)

Report viewReport (string userid)

* 1. Delete user

bool deleteUser(string userid)

1.6 Promote/downgrade

void onUpdateUser(Event e)

bool updateUser(userid,carId,carModel,licenseNumber)

Bool isEligible(string carId, string licenseId)

bool downgradeUser(userid)

bool isValidCarId(string carId)

bool isValidLicenseNumber(string licenseNumber)

2. Guest

2.1 Register

void onClickRegister(Event e)

User registerUser(string username, string firstName, string lastName, string fiscalCode, string password, string confirmPass, string picturePath, character Gender, string mobilePhone)

bool isValidFiscalCode(string fiscalCode)

void createSession(string username)

2.2 Log in

void onClickLogin(Event e)

User login(username, password)

void createSession(string username)

3. Guest

3.1 Edit profile

void onClickModify(Event e)

User ModifyUser(user.username, firstName, lastName, fiscalCode, password, confirmPass, picturePath, Gender, mobilePhoneNumber)

3.2 Report

void onClickReport(Event e)

Report newReport(string username1, string username2, string reason)

TaxiDriver getTaxiDriver(string username)

3.3 Log out

void onClickLogout(string username)

void destroySession(string username)

3.4 Request taxi

void onClickSubmitRequest(Event e)

Request makeRequest(User u, String maxWaiting, Location statpoint, Location endpoint)

3.5 Respond offer

Void onRespondOffer(Event e)

Drive handleResponse(Response response)

4. ITaxi

4.1 Change availability

void onAvailabilityChange(Event e)

bool changeAvailability(string taxiId)

4.2 Respond request

bool onClickButton(Event e)

TaxiDriver handleTaxiResponse(Response response)

float estimatePrice (startpoint, endpoint, timestamp)

time estimatedWaitingTime(startpoint, endpoint)

Response makeOffer(TaxiDriver taxidriver, User user)

6. GPS-related functions

float distance(location startpoint, location endpoint)

Location getUserLocation(string id)

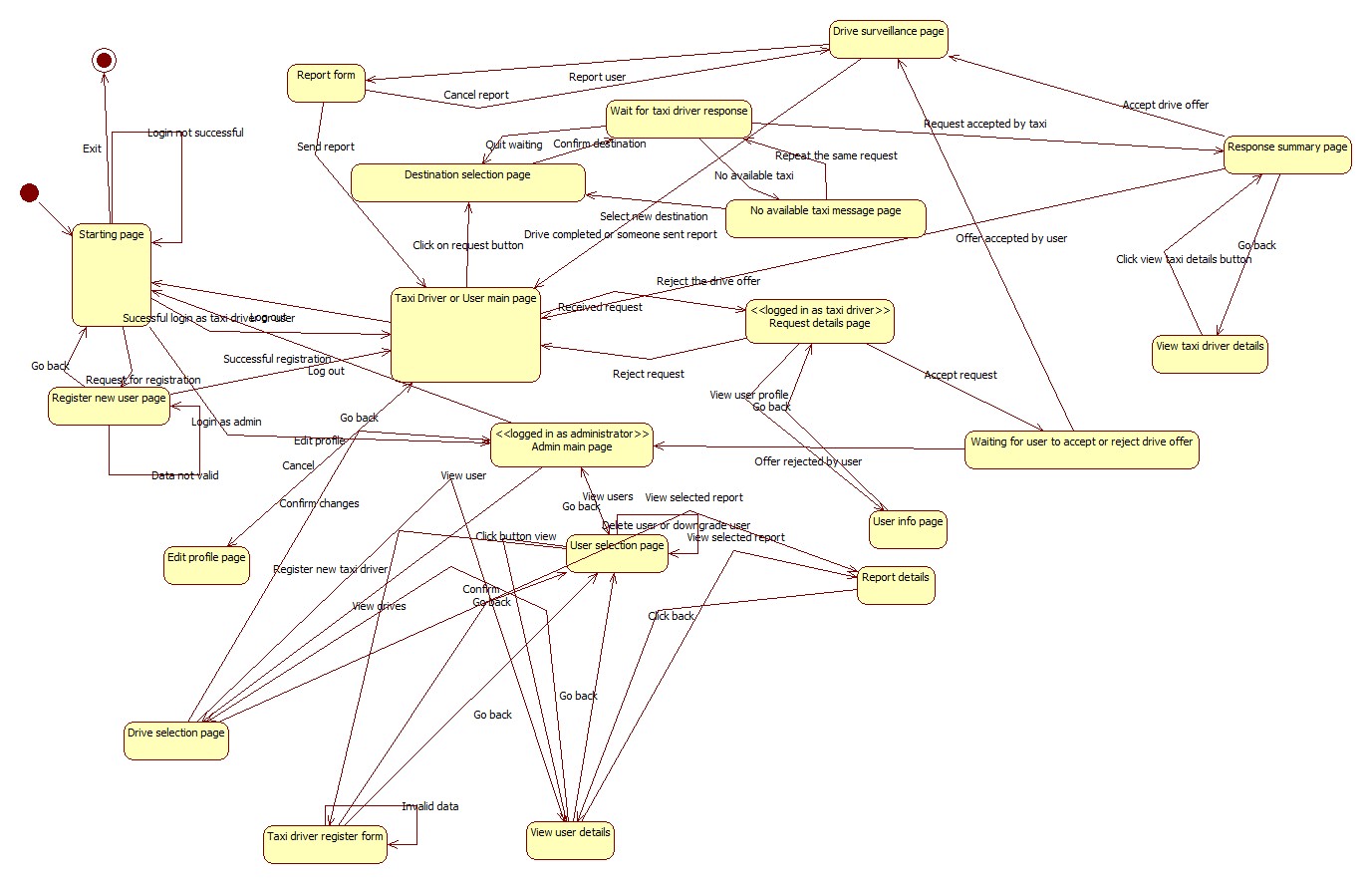
Time estimateTime(Location startpoint, Location endpoint)

3.Integration Test Plan Document [this document]

When it comes to unit tests, systematic testing is going to be used, that uses characteristics of the software artifacts and information about the beahviour of the system.

There are two main classes of test sets: white-box and black-box.

For unit testing, it is more common to choose the first approach, beacause covering of small portions of code is possible, while, for integration testing, it is more suitable to use black-box method, because it helps to identify the missing funcionalities.

Test cases are going to be derived analyzing the state diagram below:

## Elements to be integrated

This part of the document deals with elements that are going to be integrated and references to Design Document. These elements are described in details in Design Document and are going to be listed here, with brief overview.

In general, the system MyTaxiService consists of several subsystems:

-DataLayer subsystem

-Session manager subsystem

-Guest subsystem

-Scheduler subsystem

-User subsystem

-Taxi driver subsystem

-Admin subsystem

-Developer API subsystem

Some of the subsystems have components on both client and server side, which interact in order to achieve their goals.

The set of components is split into two subsets – the first one consists of all the elements from the server side, while another one is related to the elements contained on the client side and they are going to be presened in this order.

### 2.2.1 Server side elements

First, let’s take a look at the elements from the server side:

1.UnregisteredManager is an element which gives ability unregistered users to register or log in. This part of the server uses external city government databases through REST service in order to check the validity of fiscal code. It also uses DataLayer to check if there is same user already in a database or the user is banned and can’t be registered. In case of successful registration, this component uses DataLayer to add new users to database once they are registered. This element, as it can be seen, doesn’t use the scheduler, as unregistered users can’t make taxi requests.

It has two smaller components:

-register component

-login component

2.RegisteredUserManager is an element which enables users to logout , receives all the messages from users - drive requests, drive offer responses, reports, SOS signal, modify their profile, log out and all other functions that are present in RASD usecase diagrams. This component can access GPS coordinates service in order to determine the users location when necessary. In order to acomplish all of the operations, this component interacts with the Datalayer and visualises all the necessary elements when it is needed (maps, etc.). Also, this component uses Scheduler in order to serve the users in requests related to taxi drives.

Considering further, this component could be split into smaller components:

-profile management component, which includes functions related to profile editing

-taxi request component, which includes functions related to making a taxi request

-respond offer component, which is a part of negotiations between taxi driver and user

-report component, responsible for making reports and writing related reason during taxi drive

3.TaxiDriverManager is similar to previous one, but also includes some additional options, like accepting or rejecting a drive request and changing the availability status, in order to support TaxiDriver clients.

Considering further, this component could be split into smaller components:

-profile management component, which includes functions related to profile editing

-taxi request component, which includes functions related to making a taxi request

-respond offer component, which is a part of negotiations between taxi driver and user

-report component, responsible for making reports during taxi drive and writing related reason

-drive accept/reject component, which is the one of the component where drivers differ from typical users, and gives them ability to accept or reject drive

-availability change component, which is used for managing taxi driver availability

4.AdminManager is supporting AdminClients, and gives them ability, using the DataLayer, to browse users, read reports, promote users to taxi drivers, prevent some users from being drivers, delete users in case of bad behaviour, and log out. Notice that Admin manager doesn’t use Scheduler, as it isn’t necessary (Administrators can’t make taxi requests).

It could be split into smaller components:

-browsing and viewing users

-viewing drives related to users

-viewing reports

-deleting users

-promoting users to taxi drivers

5.Scheduler is the core element of the system that cross-references the location data and the system messages in order to dispatch a taxi from the corresponding taxi zone using the DataLayer. It uses GPS coordinates service in order to determine the taxi zone of the current user, deals with taxi queues and forwards the request to another taxi in current taxi zone if there is available taxi. It also receives the drive offer responses from uses and creates drive events. Scheduler uses results from time and cost calculator when makes drive offers to users.

In fact, this is an element used by three subsystems: user, taxi driver, developer.

Scheduler has three smaller components:

-Time and cost calculator. It estimates the time the time and price for a taxi drive. It uses data from DataLayer in order to have enough information to do its part of job. By the way, the Scheduler uses this component when it makes a drive offer to user- so it makes this system fair. Users can accept or rejecet the offer if the price is too high.

-Taxi queue manager – component which deals with avaiable taxi drivers and dispatching, adding and removin them from queue

-Zone determination – component that determines the taxi zone corresponding to a given location.Uses maps external GPS and Maps API.

6.DataLayer encapsulates the operations related to persistence of entities relevant to the system – report, drive, different types of users – User, Taxi Driver, Administrator and deals with their storage into database. It consists of:

-query constructor

-persistence

-database operations

Datalayer is an element used by many subsytems.

7.DevelopmentManager is a part of the system which offers access to basic system functions to developers that are outside the system via some kind of serivce (REST, for example) in order to give them ability to embed or extend MyTaxiService in their applications. They can externally request a taxi, for example by API or register or log in from external aplplication (for example, Facebook). This element uses all other components in order to achieve its functionalities.

8.SessionManager, a part of the system that is responsible for creating and destroying sessions (log in- create session, log out-destroy session). It is necessary to have this part of the system included, because it distincts logged in and logged out state.

This is an element used by many subsystems.

### 2.2.2 Client side elements

Now, let’s take a look at the elements which belong to the client side:

-GuestClient is a client component used by users before they register or log in. On the server side, this client type communicates with UnregisteredManager.

-UserClient is a client component used by users that have been already registered and logged in. This client communicates with RegisteredUserManager on the server side, which gives them a set of options – to select destination and make a taxi request, accept o reject taxi drive offer, modify their profiles, report another user during the taxi drive and all the functionalities previously mentioned.

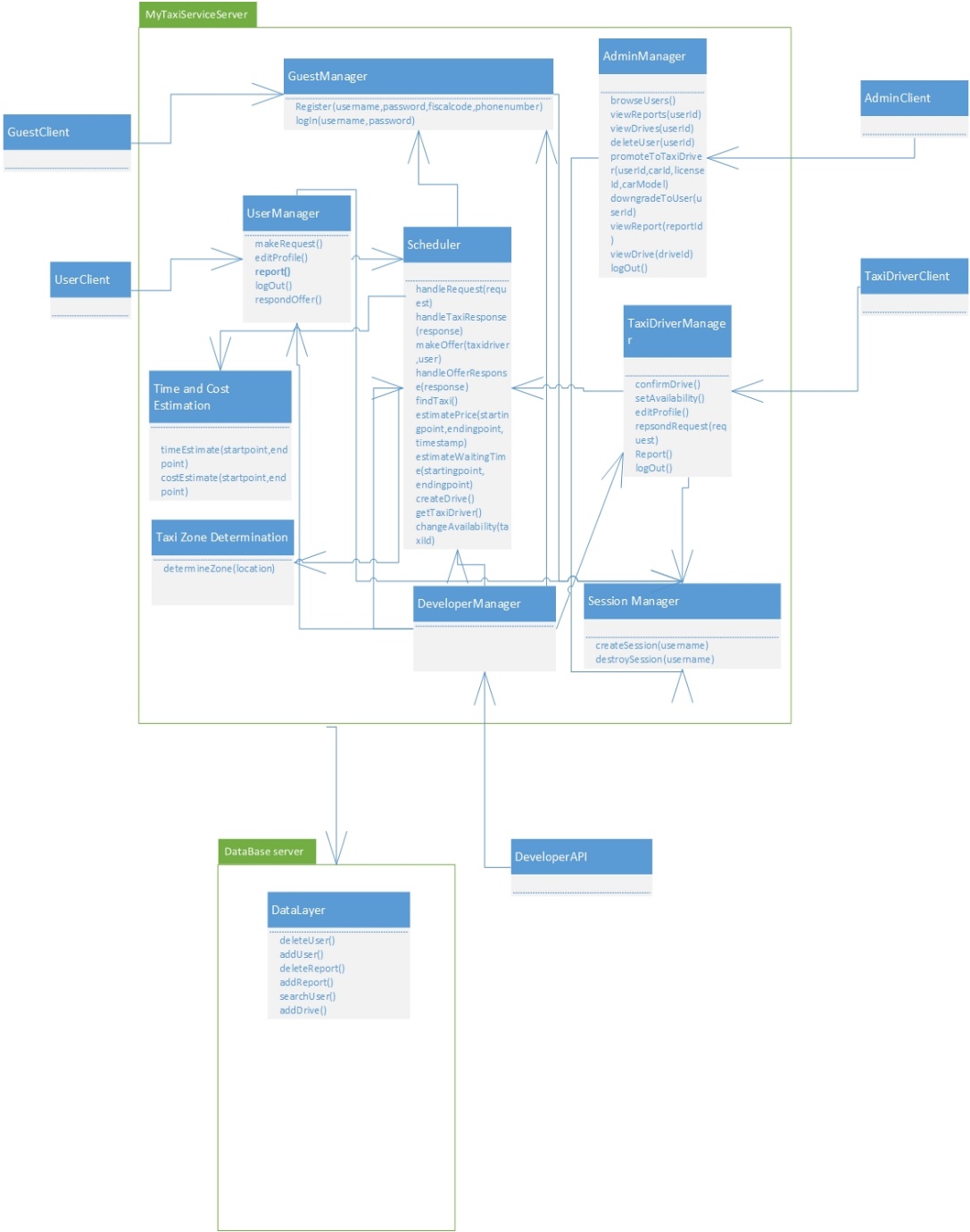
-TaxiDriverClient is client component used by users that have already been registered, logged in, and also promoted to taxi drivers by the administration. They have all the same functionalities and options as users, but also can accept or reject request for a taxi drive and change their availability status manually in case that some unexpected event occurs (accident,breakdown etc.). Taxi driver manager is a part of the server side which corresponds to this client.

-AdminClient is a client component used by the administration. This client offers them to browse users, promote them to taxi drivers by inserting the necessary data (license id, car id) , take their taxi driver privilegues in certain cases, read reports and block users. On the server side, we have AdminManager which deals with requests sent from AdminClient.

-DeveloperAPI is programmatic inteface component that gives ability to developers to use all the functionalities of MyTaxiSevice application, in order to make their own applications or integrate it into some bigger system. This element communicates with all of the server side elements, except the AdminManager, because it would be dangerous for service integrity and consistency.

As it could be seen, detailed descriptions of the smaller components are omitted here, because they are already mentioned in the part related to server side, and each of the client side components has its counterpart component on the server side, which is used to deal with the actions from the client side.

Once again, let us take a look at a system.



## Integration testing strategy

Topic of this part of the document is integration testing startegy selection. Possible approaches are going to be discussed and selected to most appropriate one, according to the specific application, its requirements and architecture.

In general, there are two different orientations:

1.Structural orientation, where modules are constructed,integrated and tested based on a hierarchical project structure. Here, we have the following approaches:

-top-down

-bottom-up

-sandwich

-backbone

2.Functional orientation, where modules are integrated according to application characteristics or features:

-threads

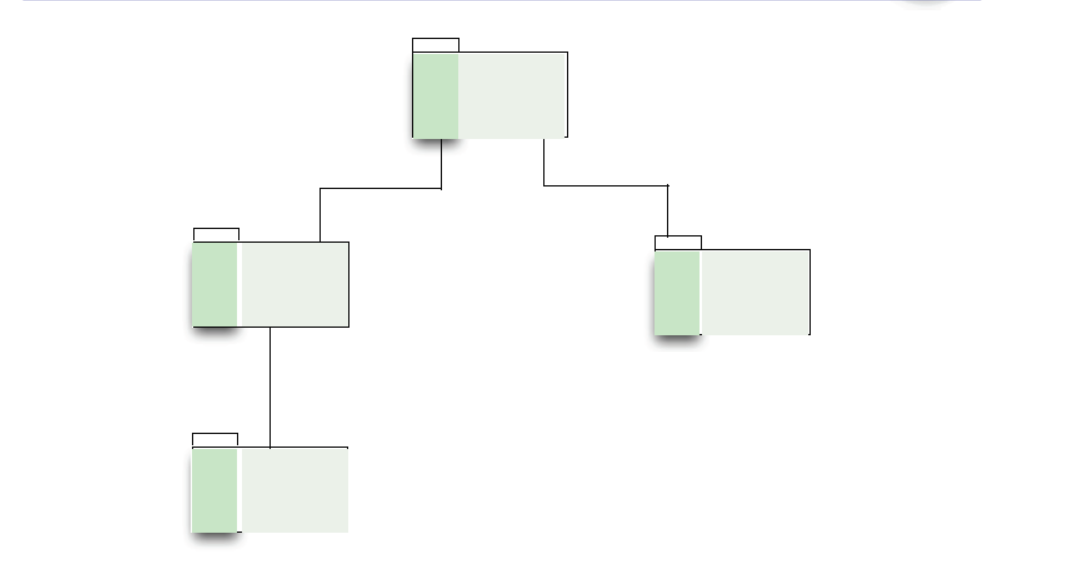
-critical module

Structural methods are more often used for simpler and smaller subsystems.

Combinations of thread and critical modules integration testing are often preferred for larger subsystems, and provide better process visibility. On the other side, structural methods are simpler.

In this case, a combination of different methods is going to be used – mainly a combiantion of thread and critical modules, when it comes to more complex components integration.

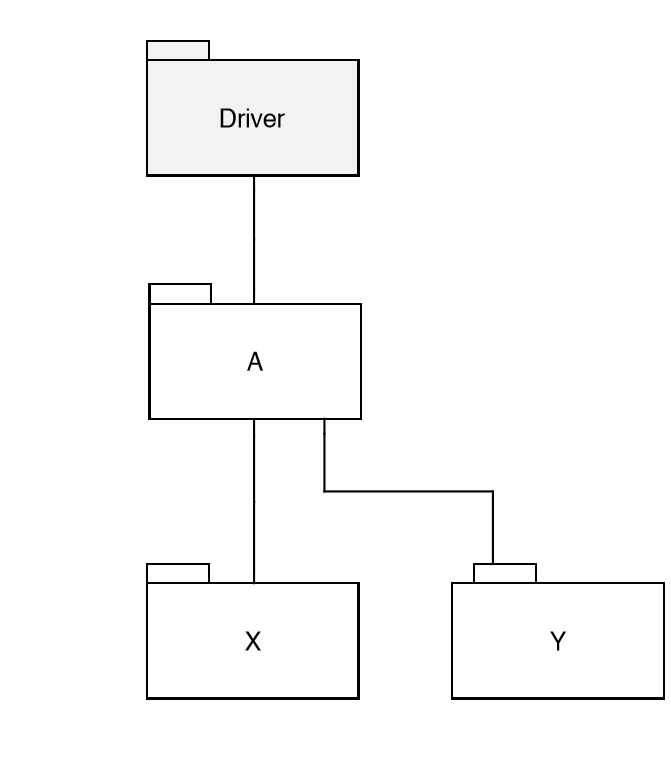
A thread is partion of several modules that together provide a user-visible feature. An illustration is displayed below:



Critical module strategy starts with riskiest modules in order to construct parts of one module to test funcionality in other.

This startegy (Critical module) is going to be used for following modules: DataLayer and Session Manager, because they are the modules which are the necessary for integration of other subsystems of MyTaxiService system, and without them, it is not even possible to develop other functionalities and integrate the elements properly. For example, without DataLayer it is not possible to login or register users. Without Session Manager, it is possible to register users, but not to log them in.

For simpler integrations, a variant of bottom-up approach is going to be used – it is going to be started from the smaller components (bottom-level) , going towards bigger components (top-level). A subset of integrated simpler components is changed with more complex one that could be further integrated (driver).



Threads strategy is going to be used for testing the subsystem related to specific users, which include a set of components and their interaction in order to achieve the functionality that is visible to the user. But, before that it is necessary to make sure that cirtical modules are working properly.

So, in general, it could be said that for more complex parts related to user-visible featues, when it comes to integration - threads (functional) strategy is going to be used, while, for smaller and simpler parts, bottom-up approach is going to be used (structural).

## Sequence of component/function integration

In what follows, a sequence of integration is going to be presented in the following order. For each subsystem the integration order is enumerated by I1,I2...IK. I1 stands for the first integration in a sequence, while IK stands for the last integration in this sequence.

2.4.1 Datalayer integration

First, the Datalayer subsystem is going to be integrated. It is used by all other subsystems in MyTaxiService and is the most critical part (critical module). Datalayer itself is not very complex component in terms of integration, so it could be integrated using some of the structural strategies (bottom-up).

The bottom component is the query execution subcomponent. It includes the functions related to CRUD operations.

Next level (upper level) is the component that deals with persistence – making objects using the query results.

And, on the top, is the component that deals with query constructor accroding to the actions taken or data required by client or server side of the system.

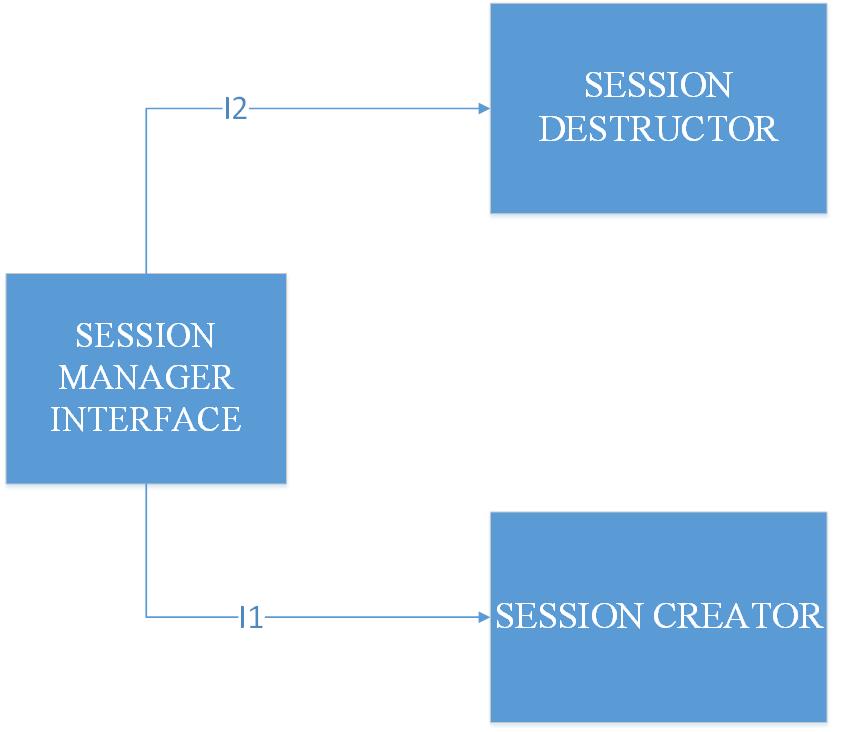


2.4.2 Session manager integration

It is a very simple element that deals with creating and destroying sessions, which enables users to log in and log out.

So, functions that are necessary for these operations are going to be integrated into this element, as it is one of the cirtical modules.

The integration itself is simple and consists of integration of session manager interface with modules related to creating and destroying sesssions.



Next step is to integrate this element with other, more complex elements, in order to make subsystems for different types of users (passengers, taxi drivers, administrators, developers, guests).

2.4.3 Guest subsystem integration

This is the first of the user-oriented subsystem that is going to be integrated.

Without this subsystem, it is not possible to register and log in users, so it is also very cirtical module, when it comes to overall MyTaxiService system.

As it is told previously, this subsystem consists of two important components – one for logging in, and one for registration.

Each of these parts is going to be integrated using threads strategy – parts of different modules are going to be integrated in order to achieve functionality that is visible to user (functional approach).

It is also necessary to integrate the components related to these functionalities with previous two in order to achieve the goal.

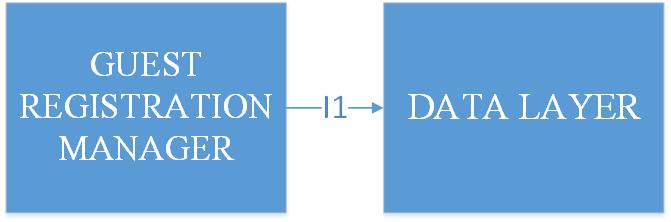
2.4.3.1 Guest registration integration

The first step for the potential users is to make profile.

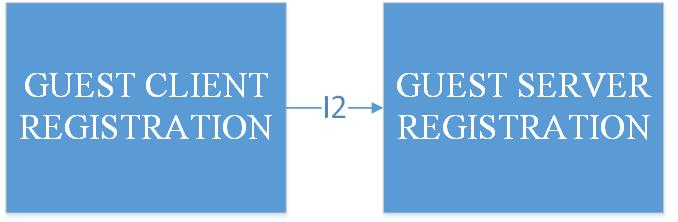
So, guest registation is going to be the first user-oriented function that is going to be integrated.

On the client side we have a component that accepts user input. On the server side, we have component connected with previous one which gives response to user actions using the DataLayer .

First, the server-side component is going to be integrated with the DataLayer (I1) in order to get module that can perform database operations on the server side.

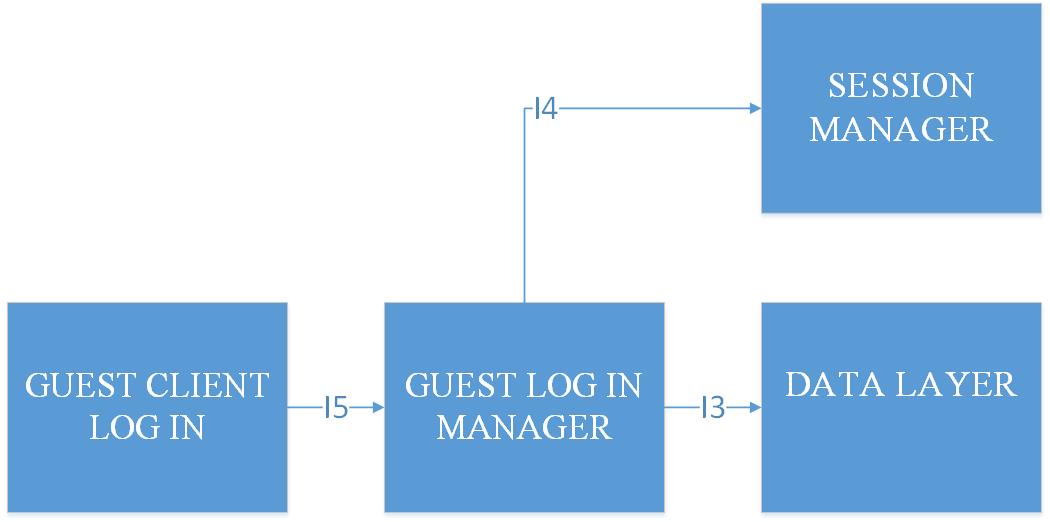


After that, user client is going to be integrated with this, bigger module from the server side(I2, bottom-up), so the clients can interact with the server side.



2.4.3.2 Guest log in integration

For log in integration, we use the same approach as for registration (bottom up in combination with functional). First, the modules of the server side are integrated – guest manager with datalayer (I3), and after that, with session manager (I4).

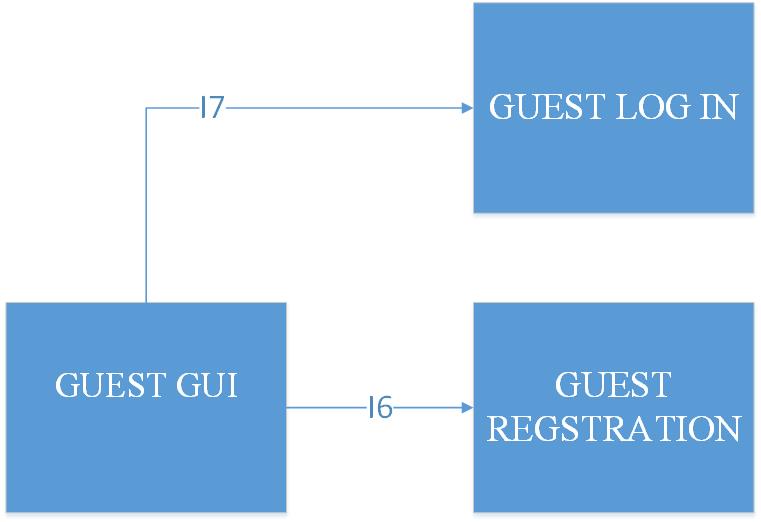


After that, the integration of these components is going to be performed with GUI component of the client side.

It is first necessary to register in order to log in, so, the sequence of integration is the following.

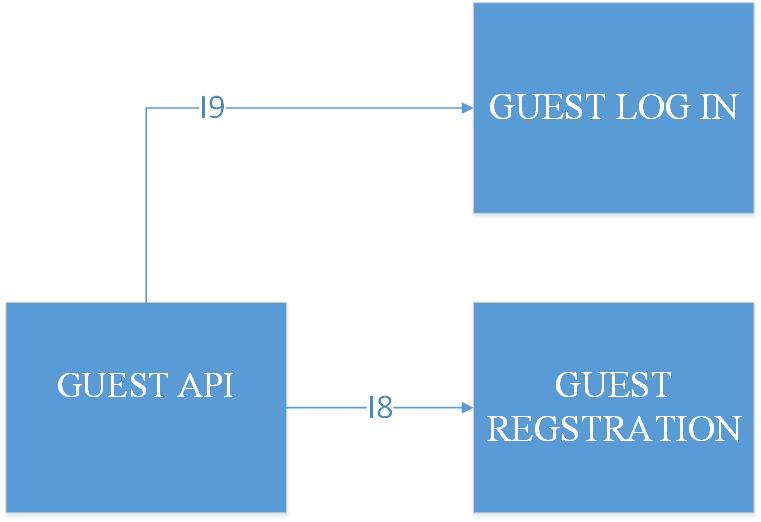
So, we first get the registration driver, and after that, we integrate it with log in module.

And now, we finally have completed the integration of Guest subsystem.



For this part of the system, Developer API is needed, so one more integration is going to be

performed:

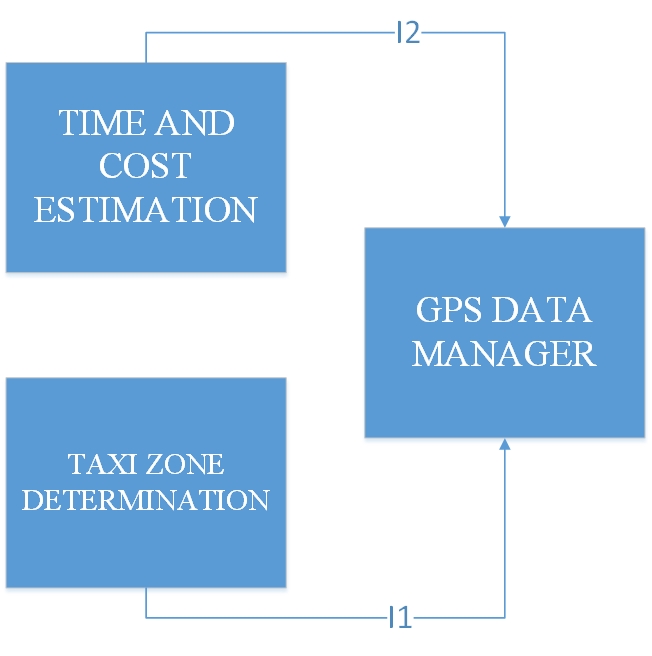


2.4.4 Scheduler integration

Again, one of the critical modules is going to be integrated. Scheduler is the core part of the system and is necessary for any type of system users in order to make taxi requests, respond offers, make estiamtions related to time and cost.

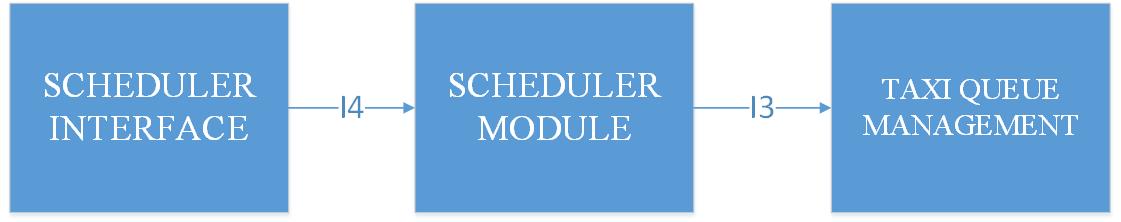
So ,the next step is to integrate the Scheduler. A variant of bottom-up integration is going to be used for integration of this element.

The first parts that ae going to be integrated are parts related to coordinates and taxi zone determination. No estimation is possible without having coordinates module. After that, coordinates module is going to be integrated with part of the scheduler that deals with time and cost estimation.



Here, the bottom-up approach is used. Now, we can change the integrated module with bigger module, that could be further integrated with part related to taxi queue management.

And, finally (bottom-up strategy here), it is going to be integrated with taxi queue manager in order to achieve the complete Scheduler that can deal with taxi requests (I3), and with component that gives an external interface to the rest of the system (I4).



2.4.5 Registered User and Taxi driver subsystem integration

For integration of this part of the system, threads approach is going to be used. Components are going to be integrated in such order, that each subset of integrated modules gives a user-visible functionlaity.

User-visible functionalities, related to this part of the system could be devided into several parts:

-profile management component, which includes functions related to profile editing

-taxi request component, which includes functions related to making a taxi request

-respond offer component, which is a part of negotiations between taxi driver and user

-report component, responsible for making reports and writing related reason during taxi drive

When it comes to taxi drivers, their subsystem is the same, but offers few more features:

-drive accept/reject component, which is the one of the component where drivers differ from typical users, and gives them ability to accept or reject drive

-availability change component, which is used for managing taxi driver availability

So, each of them is going to be integrated, and, after that, they are going to be integrated together into the user subsystem. For integration of bigger elements, a kind of structural strategy is going to be used.

2.4.5.1 Profile management integration

First, on the server side, Registered User manager is going to be integrated with the DataLayer in order to enable operations related to profile modifications (updates). For profile management, the part of the client related to these operations is going to be integrated with the corresponding part on the server side after that. The integration is going to be performed in a following sequence:



After this, we have profile editing feature completely available and integrated.

The integration is the same for both taxi drivers and users.

It is needed to perform taxi driver integration in parallel with user integration, in order to make request feature integration possible. In order to make taxi requests, it is necessary to have taxi drivers.



2.4.5.2 Taxi request integration

This is the key part of the whole MyTaxiService system. Functional strategy is going to be used for integration of this part. The goal is to enable users to request taxi (user-visible feature).

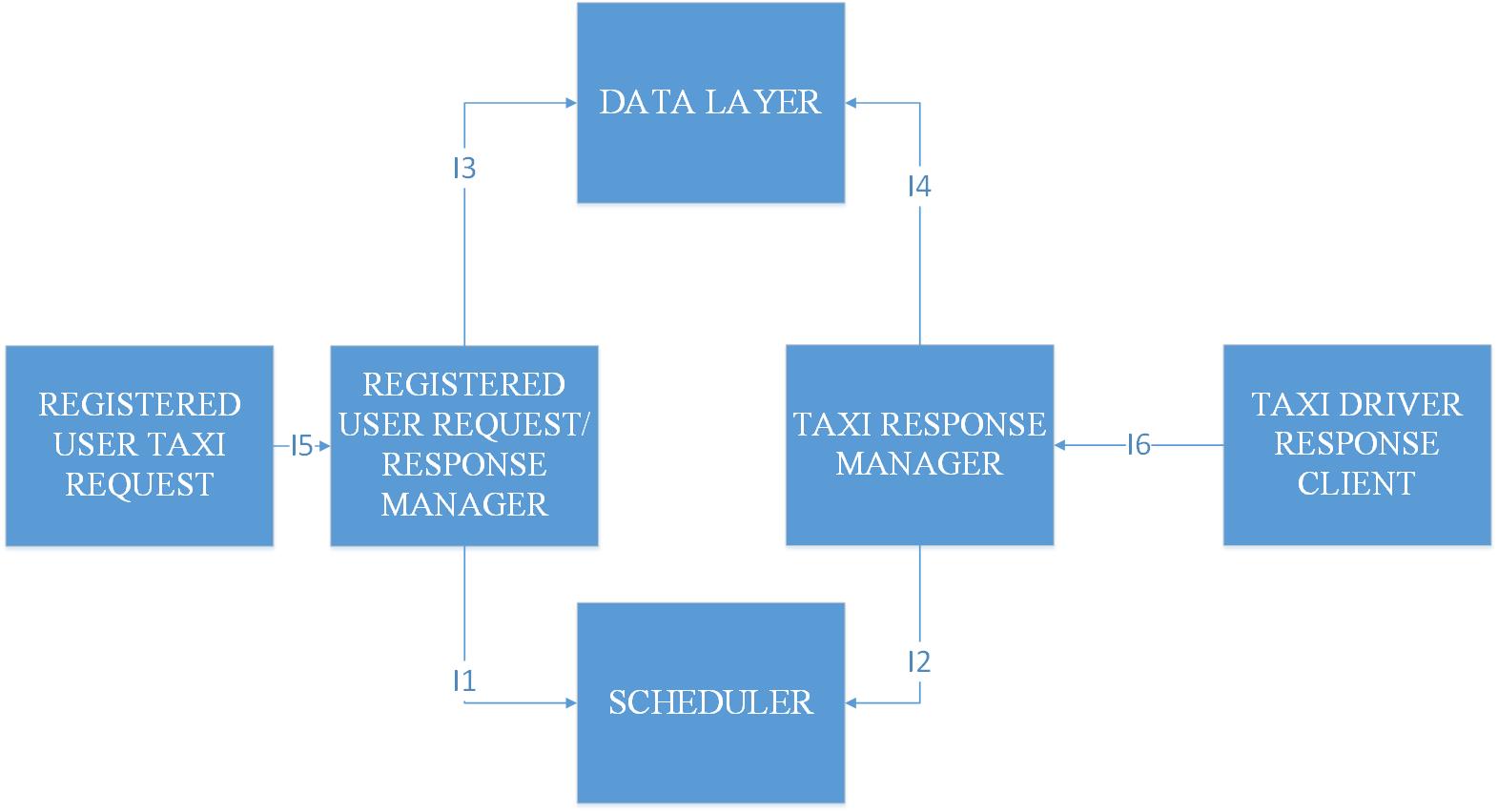
This is the most complex integration in the whole system, and is going to involve the integration of both client and server sides of Registered Users and Taxi drivers.

First, the request/response module on the server side of the users is going to be integrated with the scheduler. Scheduler is necessary for estimations and zone determinations, so making a request does not have any sense without usage of this module. The next step would be to do the same on the taxi driver side. After that, we can make taxi request, and respond them from the taxi side.

Further, the integration with DataLayer is performed in order to be able to keep track of drive events and store them into database.

And, finally, the integration with client side of the corresponding user types is performed (I5 and I6).

After performing these steps of integration, it is possible to state that we have a user-visible feature (taxi request) completely integrated.

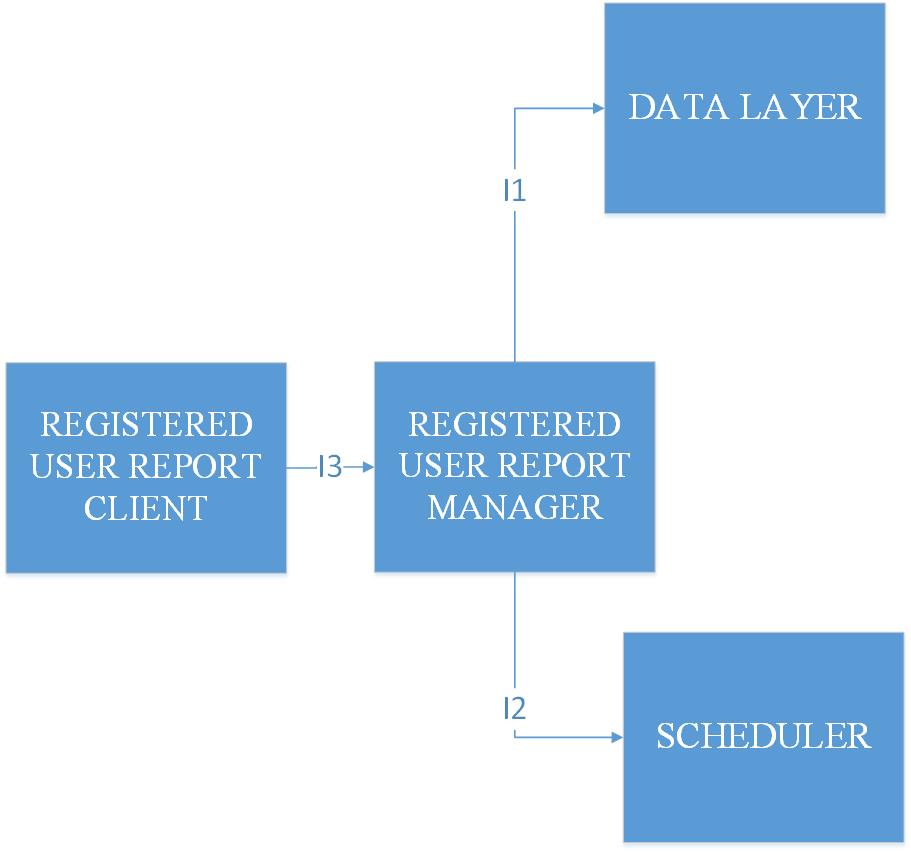


2.4.5.3 Report module integration

This part of system gives ability to users to make reports against other users (user to taxi driver or vice-versa).

This part of the system is symmetric for user and taxi driver. It is necessary in order to complete the taxi driver availability component, because the report event could affect taxi driver availability.

First, the server side report manager is integrated with DataLayer (I1) which is responsible for storage of the reports. After that, the manager on the server side is integrated with Scheduler, because writing reports affects the taxi driver availability and queue state. Finally, client is integrated with the server side part.



2.4.5.4 Taxi driver status management integration

The next feature that is going to be integrated is the module responsible for taxi driver status change. They can change their availability manually, or it can be changed automatically in case of certain events (report, end of the drive event).

First, the server-side components are integrated (I1), and after that, client component is integrated with server component (I2). Once again, we are integrating a user-visible feature from simpler module to more complex one.

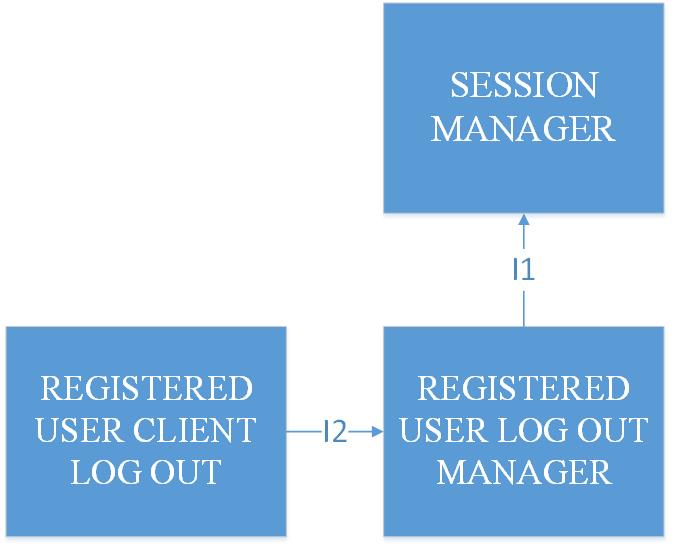


2.4.5.5 Log out integration

Log out feature is going to be achieved by integrating components from both client and server side, similar to log in which is previously mentioned.

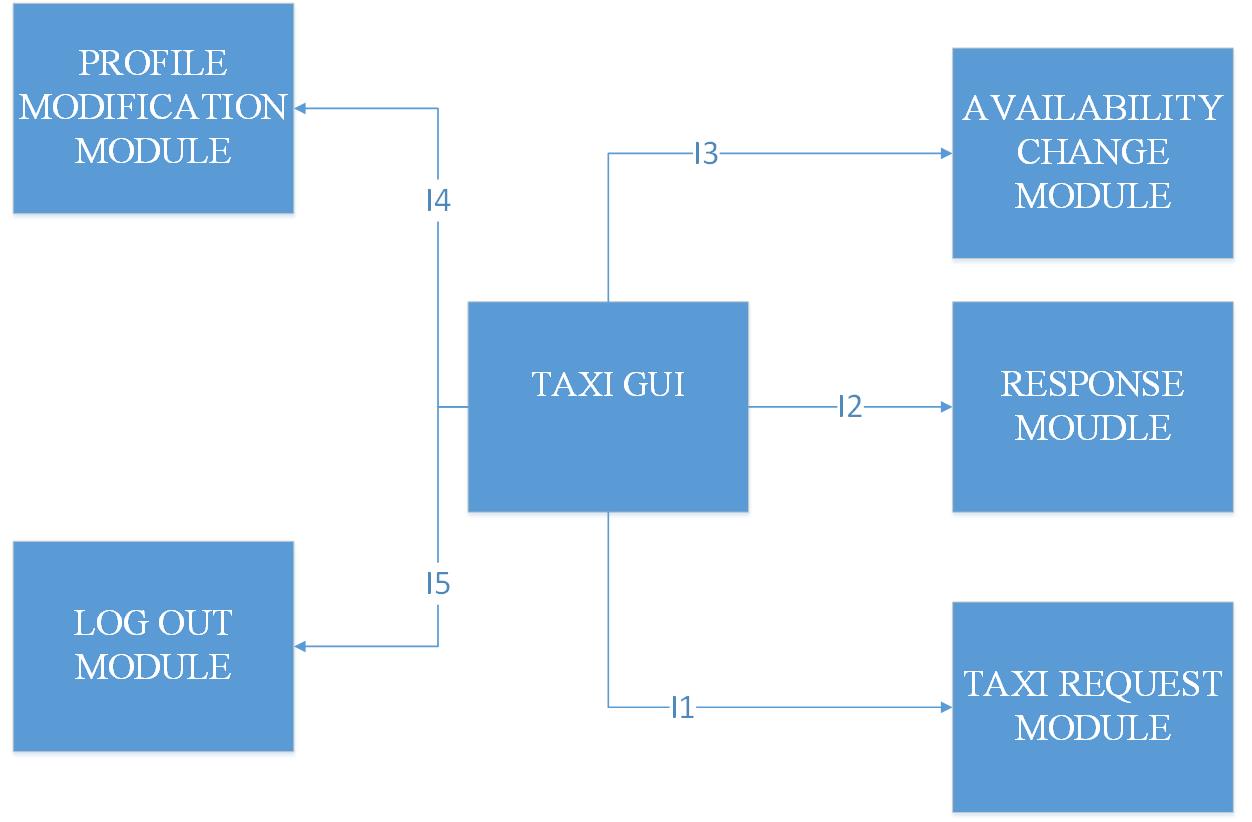
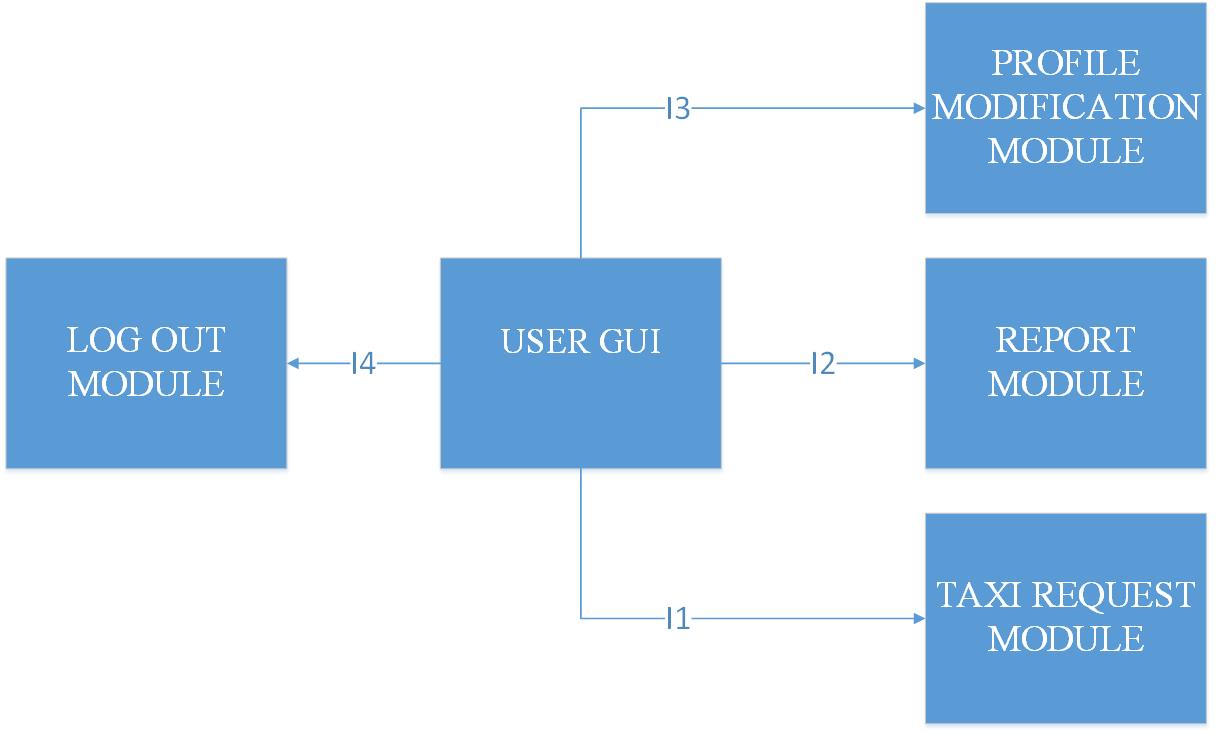
It is performed the same way for both user and taxi driver.

First, the server side-components are connected, and after that, the driver is connected with client component (I2).

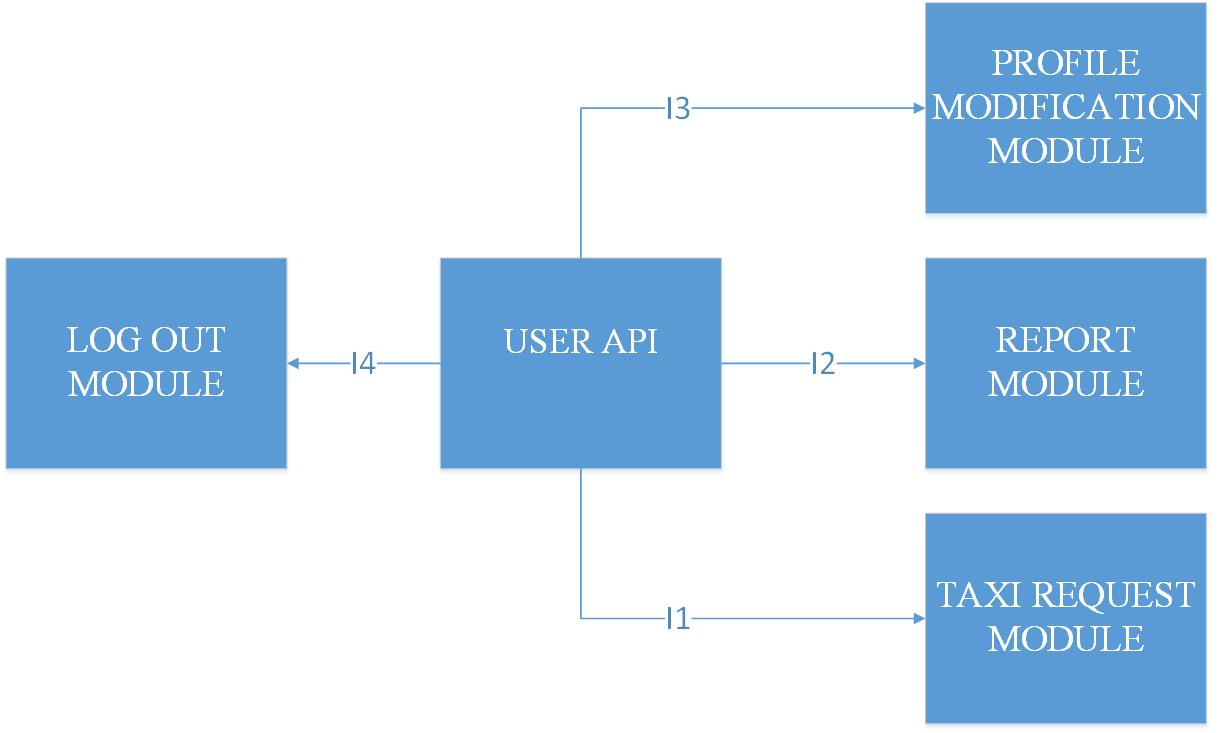


2.4.5.6 User/taxi driver Interface integration

And, finally the integration with GUI is performed in following order (for user and taxi driver):



Developer API will involve usage of User Subsystem, so the programmatic interface is also going to be constructed.



2.4.6 Admin subsystem integration

In what follows, admin subsystem is going to be integrated.

It offers the following features:

1.browsing and viewing users

2.viewing drives related to users

3.viewing reports

4.deleting users

5.promoting users to taxi drivers

6.log out

Features 1,2,3,4,5 are going to be integrated into data manipulation module, because CRUD operations are behind them, so heavy use of DataLayer is expected in this case.

Log out is going to be performed the same way as for users and taxi drivers.

2.4.6.1 Data manipulation module

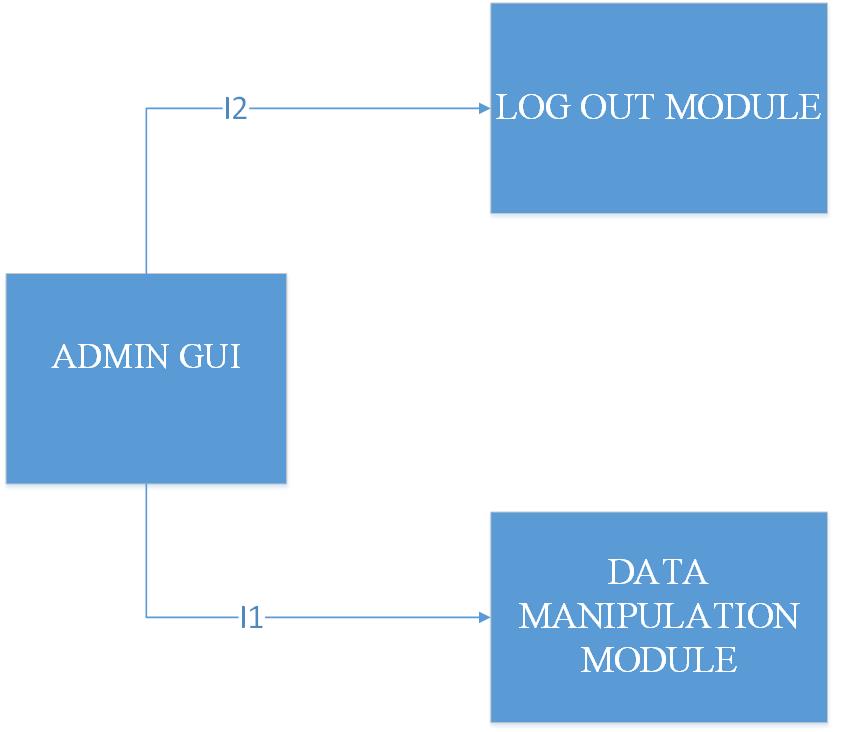
This module deals with browsing, editing, viewing and deleting data.

First, the server side part that deals with data manipulation is going to be integrated with data layer (I1). After that, client-side component is going to be connected with server side components.



2.4.6.2 Admin GUI module

And finally, admin GUI is going to be integrated with other modules.



2.4.7 Developer API integration

The API that is going to be created will combine various functionalities that are performed by guests and users together with scheduling.

-everything related to guests

-everything related to users

-scheduling

It is decided, that all other features are omitted for first version of the system, in order to avoid misuse of the API.

In further versions, it is expected to add taxi driver features for developers.

The integration is going to be performed in following sequence, when it comes to Developer API

(the picture below).

# C:\Users\Lenovo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\DEVELOPER API.JPG

# Individual steps and test description

In this chapter, individual steps in testing ae going to be described.

In order to perform a specific integration test, it is necessry to:

-Design the integation test

-Design a driver (if it was not made at the unit test)

-Design input test data (if it was not made at the unit test)

-Setting up a system, the components involved, the driver and the input test data

-Performing the integration test

In what follows, test cases are going to be specified.

## Test case specifications

3.1.1 Integration test case I1 - DataLayer

|  |  |
| --- | --- |
| Test case identifier | I1T1 |
| Test case items | Persistence Manager -> Query Executor |
| Input specification | Create typical Persistance Manager input |
| Output specification | Check if correct functions are called inside Query Executor |
| Environmental needs | Data Layer driver |

|  |  |
| --- | --- |
| Test case identifier | I1T2 |
| Test case items | Query Constructor -> Persistence Manager |
| Input specification | Create typical Query Constructor input |
| Output specification | Check if correct functions are called inside Persistence Manager |
| Environmental needs | I1T1 succeeded; Persist |

3.1.2 Integration test case I2 – Session Manager

|  |  |
| --- | --- |
| Test case identifier | I2T1 |
| Test case items | Session Manager Interface -> Session Creator |
| Input specification | Create Session Manager input that involves usage of Session Creator |
| Output specification | Check if correct functions are called inside Session Creator |
| Environmental needs | Session Manager driver |

|  |  |
| --- | --- |
| Test case identifier | I2T2 |
| Test case items | Session Manager Interface -> Session Destructor |
| Input specification | Create Session Manager input that will involve Session Destructor |
| Output specification | Check if correct functions are called inside Session Destructor |
| Environmental needs | Session Manager Driver |

3.1.3 Integration test case I3 – Guest Registration

|  |  |
| --- | --- |
| Test case identifier | I3T1 |
| Test case items | Guest Registration Manager-> Data Layer |
| Input specification | Create typical Guest Regsistration Manager input |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1T1 succeeded; I1T2 suceeded; Guest Registration driver |

|  |  |
| --- | --- |
| Test case identifier | I3T2 |
| Test case items | Guest Registration Client-> Guest Manager |
| Input specification | Create typical Guest Regsistration Manager input |
| Output specification | Check if correct functions are called inside Guest Registration Manager |
| Environmental needs | I3T1 succeeded; |

3.1.4 Integration test case I4 – Guest Log in

|  |  |
| --- | --- |
| Test case identifier | I4T1 |
| Test case items | Guest Log in Manager-> Data Layer |
| Input specification | Create Guest Log in Manager input that involves usage of Data Layer |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1T1 succeeded; I1T2 suceeded;Guest Login driver |

|  |  |
| --- | --- |
| Test case identifier | I4T2 |
| Test case items | Guest Log In Manager-> Session Manager |
| Input specification | Create typical Guest Log in Manager input |
| Output specification | Check if correct functions are called inside Session Manager |
| Environmental needs | I4T1 succeeded; I2T1 succeeded; I2T2 succeded |

|  |  |
| --- | --- |
| Test case identifier | I4T3 |
| Test case items | Guest Log In Client-> Guest Log in Manager |
| Input specification | Create typical Guest Log in Manager input |
| Output specification | Check if correct functions are called inside Guest Log in Manager |
| Environmental needs | I4T3 succeeded; |

3.1.5 Integration test case I5 – Guest GUI

|  |  |
| --- | --- |
| Test case identifier | I5T1 |
| Test case items | Guest GUI-> Guest Registration Manager |
| Input specification | Create typical Guest GUI input that involves Guest Registration Manager |
| Output specification | Check if correct functions are called inside Guest Registration Manager |
| Environmental needs | I3T1 succeeded; I3T2 succeeded |

|  |  |
| --- | --- |
| Test case identifier | I5T2 |
| Test case items | Guest GUI-> Guest Log In Manager |
| Input specification | Create typical Guest GUI input that involves Guest Log In Manager |
| Output specification | Check if correct functions are called inside Guest Log in Manager |
| Environmental needs | I4T1 succeeded; I4T2 suceeded; I4T3 succeeded; I5T1 succeded |

3.1.6 Integration test case I6 – Scheduler

|  |  |
| --- | --- |
| Test case identifier | I6T1 |
| Test case items | Taxi Zone Determination Module-> GPS Data Manager |
| Input specification | Create typical Taxi Zone Determination Module Input |
| Output specification | Check if correct functions are called inside GPS Data Manager |
| Environmental needs | GPS Data manager functions unit tests succeeded |

|  |  |
| --- | --- |
| Test case identifier | I6T2 |
| Test case items | Estimator-> GPS Data Manager |
| Input specification | Create typical Estimator input |
| Output specification | Check if correct functions are called inside GPS Data Manager |
| Environmental needs | GPS Data manager functions unit tests succeeded |

|  |  |
| --- | --- |
| Test case identifier | I6T3 |
| Test case items | Scheduler Submodule-> Taxi Queue Manager |
| Input specification | Create typical Scheduler Submodule input |
| Output specification | Check if correct functions are called inside Taxi Queue Manager |
| Environmental needs | I6T1 succeeded; I6T2 succeeded; Scheduler driver |

|  |  |
| --- | --- |
| Test case identifier | I6T4 |
| Test case items | Scheduler Interface-> Scheduler Submodule |
| Input specification | Create typical Scheduler Interface input |
| Output specification | Check if correct functions are called inside Scheduler Submodule |
| Environmental needs | I6T1 succeeded; I6T2 succeeded; I6T3 succeeded |

3.1.7 Integration test case I7 – Profile editing

|  |  |
| --- | --- |
| Test case identifier | I7T1 |
| Test case items | User Profile Editing Manager -> Data Layer |
| Input specification | Create typical User Profile Editing Manager input |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1 succeeded; User Profile Editing driver |

|  |  |
| --- | --- |
| Test case identifier | I7T2 |
| Test case items | User Profile Editing Client -> User Profile Editing Manager |
| Input specification | Create typical User Profile Editing Client input |
| Output specification | Check if correct functions are called inside User Profile Editing Manager |
| Environmental needs | I7T1 succeeded |

3.1.8 Integration test case I8 – Taxi Request

|  |  |
| --- | --- |
| Test case identifier | I8T1 |
| Test case items | Registered User Request/Response Manager -> Scheduler |
| Input specification | Create typical Registered User Request/Response Manager input that involves Scheduler |
| Output specification | Check if correct functions are called inside Scheduler |
| Environmental needs | I6 succeded; Taxi Request Driver |

|  |  |
| --- | --- |
| Test case identifier | I8T2 |
| Test case items | Taxi Response Manager -> Scheduler |
| Input specification | Create typical Taxi Response Manager input |
| Output specification | Check if correct functions are called inside Scheduler |
| Environmental needs | I6 succeeded; I8T1 suceeded; Taxi Response Driver |

|  |  |
| --- | --- |
| Test case identifier | I8T3 |
| Test case items | Registered User Request/Response Manager -> Data Layer |
| Input specification | Create typical Registered User Request/Response Manager input that involves Data Layer |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1 succeded; I8T1 succeeded; I8T2 succeeded; Taxi Request Driver |

|  |  |
| --- | --- |
| Test case identifier | I8T4 |
| Test case items | Taxi Response Manager -> Scheduler |
| Input specification | Create typical Taxi Response Manager input that involves Data Layer |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1 succeeded; I8T1 suceeded; I8T2 suceeded; I8T3 succeeded; Taxi Response Driver |

|  |  |
| --- | --- |
| Test case identifier | I8T5 |
| Test case items | Registered User Request/Response Client -> Registered User Request/Response Manager |
| Input specification | Create typical Registered User Request/Response Client input |
| Output specification | Check if correct functions are called inside Registered User Request/Response manager |
| Environmental needs | I8T3 succeded; |

|  |  |
| --- | --- |
| Test case identifier | I8T6 |
| Test case items | Taxi Response Client -> Taxi Response Manager |
| Input specification | Create typical Taxi Drive Response Client input |
| Output specification | Check if correct functions are called inside Taxi Response manager |
| Environmental needs | I8T4 succeded; |

3.1.9 Integration test case I9 – Report module

|  |  |
| --- | --- |
| Test case identifier | I9T1 |
| Test case items | Registered User Report Manager -> Data Layer |
| Input specification | Create typical Registered User Reporrt Manager input that involves Data Layer |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1 succeded; Report Driver |

|  |  |
| --- | --- |
| Test case identifier | I9T2 |
| Test case items | Registered User Report Manager -> Scheduler |
| Input specification | Create typical User Report Manager input |
| Output specification | Check if correct functions are called inside Scheduler |
| Environmental needs | I6 succeeded; I9T1 suceeded |

|  |  |
| --- | --- |
| Test case identifier | I9T3 |
| Test case items | Registered User Report Client -> Registered User Report Manager |
| Input specification | Create typical Registered User Report Client |
| Output specification | Registered User Report Manager |
| Environmental needs | I9T1 succeeded; I9T2 succeeded |

3.1.10 Integration test case I10 – Status change

|  |  |
| --- | --- |
| Test case identifier | I10T1 |
| Test case items | Taxi Driver Status Manager -> Scheduler |
| Input specification | Create typical Taxi Driver Status Manager Input |
| Output specification | Check if correct functions are called inside Scheduler |
| Environmental needs | I6 succeeded; Taxi Driver Status driver |

|  |  |
| --- | --- |
| Test case identifier | I10T2 |
| Test case items | Taxi Driver Status Client ->Taxi Driver Status Manager |
| Input specification | Create typical Taxi Driver Status Client input |
| Output specification | Check if correct functions are called inside Taxi Driver Status Manager |
| Environmental needs | I10T1 succeeded |

3.1.11 Integration test case I11 – Guest Log out

|  |  |
| --- | --- |
| Test case identifier | I11T1 |
| Test case items | Guest Log Out Manager-> Session Manager |
| Input specification | Create typical Guest Log out Manager input |
| Output specification | Check if correct functions are called inside Session Manager |
| Environmental needs | I2 succeeded; Guest Loug Out driver |

|  |  |
| --- | --- |
| Test case identifier | I11T2 |
| Test case items | Guest Log Out Client-> Guest Log Out Manager |
| Input specification | Create typical Guest Log Out Manager input |
| Output specification | Check if correct functions are called inside Guest Log out Manager |
| Environmental needs | I11T2 succeeded; |

3.1.12 Integration test case I12 – User GUI

|  |  |
| --- | --- |
| Test case identifier | I12T1 |
| Test case items | User GUI -> Taxi Request Module |
| Input specification | Create typical User GUI input that involves usage of Taxi Request Module |
| Output specification | Check if correct functions are called inside Taxi Request Module |
| Environmental needs | I8 succeded; |

|  |  |
| --- | --- |
| Test case identifier | I12T2 |
| Test case items | User GUI -> Report Module |
| Input specification | Create typical User GUI input that involves usage of Report Module |
| Output specification | Check if correct functions are called inside Report Module |
| Environmental needs | I9 suceeded; |

|  |  |
| --- | --- |
| Test case identifier | I12T3 |
| Test case items | User GUI -> Profile Editing Module |
| Input specification | Create typical User GUI input that involves Profile Editing Module |
| Output specification | Check if correct functions are called inside Profile Editing Module |
| Environmental needs | I7 succeded |

|  |  |
| --- | --- |
| Test case identifier | I12T4 |
| Test case items | User GUI->Log out Module |
| Input specification | Create typical User GUI input that involves Log out module |
| Output specification | Check if correct functionss are called inside Log out module |
| Environmental needs | I11 suceeded |

3.1.13 Integration test case I13 – Taxi Driver GUI

|  |  |
| --- | --- |
| Test case identifier | I13T1 |
| Test case items | Taxi Driver GUI -> Taxi Request Module |
| Input specification | Create typical Taxi Driver GUI input that involves usage of Taxi Request Module |
| Output specification | Check if correct functions are called inside Taxi Request Module |
| Environmental needs | I8 succeded; |

|  |  |
| --- | --- |
| Test case identifier | I13T2 |
| Test case items | Taxi Driver GUI -> Response Module |
| Input specification | Create typical Taxi Driver GUI input that involves usage of Response Module |
| Output specification | Check if correct functions are called inside Response Module |
| Environmental needs | I8 suceeded; |

|  |  |
| --- | --- |
| Test case identifier | I13T3 |
| Test case items | Taxi Driver GUI -> Availability Change Module |
| Input specification | Create typical Taxi Driver GUI input that involves usage of Availability Change Module |
| Output specification | Check if correct functions are called inside Availability Change Module |
| Environmental needs | I10 succeded |

|  |  |
| --- | --- |
| Test case identifier | I13T4 |
| Test case items | Taxi Driver GUI -> Profile Editing Module |
| Input specification | Create typical Taxi Driver GUI input that involves Profile Editing Module |
| Output specification | Check if correct functions are called inside Profile Editing Module |
| Environmental needs | I7 succeded |

|  |  |
| --- | --- |
| Test case identifier | I13T5 |
| Test case items | Taxi Driver GUI->Log out Module |
| Input specification | Create typical Taxi Driver GUI input that involves Log out module |
| Output specification | Check if correct functionss are called inside Log out module |
| Environmental needs | I11 suceeded |

3.1.14 Integration test case I14 – Admin Integration

|  |  |
| --- | --- |
| Test case identifier | I14T1 |
| Test case items | Admin Data Maniulation Manager -> Data Layer |
| Input specification | Create typical Admin Data Manipulation input |
| Output specification | Check if correct functions are called inside Data Layer |
| Environmental needs | I1 succeded; Admin Driver |

|  |  |
| --- | --- |
| Test case identifier | I14T2 |
| Test case items | Admin Data Manipulation Client -> Admin Data Manipulation Manager |
| Input specification | Create typical Admin Data Manipulation Client Input |
| Output specification | Check if correct functions are called inside Admin Data Manipulation Manager |
| Environmental needs | I1T4 suceeded; |

|  |  |
| --- | --- |
| Test case identifier | I14T3 |
| Test case items | Admin Client GUI -> Data Manipulation Module |
| Input specification | Create typical Admin Client GUI input that involves use of Data Manipuation Module |
| Output specification | Check if correct functions are called inside Data Manipulation Module |
| Environmental needs | I14T2 succeeded succeded |

|  |  |
| --- | --- |
| Test case identifier | I14T5 |
| Test case items | Admin Client GUI -> Log Out Module |
| Input specification | Create typical Admin Client GUI input that involves Admin Log Out |
| Output specification | Check if functions are called inside Admin Log Out Module |
| Environmental needs | I11 succeded |

3.1.15 Integration test case I15 – Guest API

|  |  |
| --- | --- |
| Test case identifier | I15T1 |
| Test case items | Guest API-> Guest Registration Manager |
| Input specification | Create typical Guest API input that involves Guest Registration Manager |
| Output specification | Check if correct functions are called inside Guest Registration Manager |
| Environmental needs | I3T1 succeeded; I3T2 succeeded;Guest API driver |

|  |  |
| --- | --- |
| Test case identifier | I15T2 |
| Test case items | Guest API-> Guest Log In Manager |
| Input specification | Create typical Guest API input that involves Guest Log In Manager |
| Output specification | Check if correct functions are called inside Guest Log in Manager |
| Environmental needs | I4T1 succeeded; I4T2 suceeded; I4T3 succeeded; I5T1 succeded |

3.1.16 Integration test case I12 – User API

|  |  |
| --- | --- |
| Test case identifier | I16T1 |
| Test case items | User API -> Taxi Request Module |
| Input specification | Create typical User API input that involves usage of Taxi Request Module |
| Output specification | Check if correct functions are called inside Taxi Request Module |
| Environmental needs | I8 succeded; |

|  |  |
| --- | --- |
| Test case identifier | I16T2 |
| Test case items | User API -> Report Module |
| Input specification | Create typical User API input that involves usage of Report Module |
| Output specification | Check if correct functions are called inside Report Module |
| Environmental needs | I9 suceeded; |

|  |  |
| --- | --- |
| Test case identifier | I12T3 |
| Test case items | User API -> Profile Editing Module |
| Input specification | Create typical User API input that involves Profile Editing Module |
| Output specification | Check if correct functions are called inside Profile Editing Module |
| Environmental needs | I7 succeded |

|  |  |
| --- | --- |
| Test case identifier | I16T4 |
| Test case items | User API->Log out Module |
| Input specification | Create typical User API input that involves Log out module |
| Output specification | Check if correct functionss are called inside Log out module |
| Environmental needs | I11 suceeded |

3.1.17 Integration test case I17 – Developer API Integration

|  |  |
| --- | --- |
| Test case identifier | I17T1 |
| Test case items | Developer API -> Guest Subsystem |
| Input specification | Create typical Developer API input that involves use of Guest subsystem |
| Output specification | Check if correct functions are called inside Guest Subsystem |
| Environmental needs | I15 succeded; Developer API Driver |

|  |  |
| --- | --- |
| Test case identifier | I17T2 |
| Test case items | Developer API -> User Subsystem |
| Input specification | Create typical Developer API input that involves use of User Subsystem |
| Output specification | Check if correct functions are called inside User Subsystem |
| Environmental needs | I16 suceeded; |

|  |  |
| --- | --- |
| Test case identifier | I17T3 |
| Test case items | Developer API -> Scheduler |
| Input specification | Create typical Admin input that involves use of Scheduler |
| Output specification | Check if correct functions are called inside Scheduler |
| Environmental needs | I6 succeeded succeded |

## Test procedures

|  |  |
| --- | --- |
| Test procedure identifier | TP1 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP2 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP3 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP4 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP5 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP6 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP7 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP8 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

|  |  |
| --- | --- |
| Test procedure identifier | TP9 |
| Purpose | Developer API -> Scheduler |
| Procedure steps | Create typical Admin input that involves use of Scheduler |

# Tools and test equipment required

# Program stubs and test data required

In this part,the overall architecture of the system is going to be considered using the top-down approach in order to identify all the subsystems. This approach is used in order to give a brief overview , to make it easier to understand how the system would look like and to separate the functional units in order to make deriving component, deployment and runtime view more understandable.

In what follows, relying on identified usecases in RASD document, the decomposition of application into subsystems is going to be presented – starting from the biggest functional unit – the MyTaxiService itself, and getting closer and closer to the smaller units.

System is separated into following subsystems:

-Unregistered user subsystem – includes login and registration related functions encapsulated into Login and Registration subsystems.

-Registered user subsystem – includes sending taxi request, reports, responding to drive offers, profile modifications and log out. Each of the operations mentioned corresponds to a subsystem that deals with related problem.

-Taxi driver subsystem – the same as previous one, but also includes status availability change and accpeting/rejecting drive requests.

-Scheduler subsystem – determines the taxi zones and forwards requests and responses to corresponfing queue. This part of the system also acts a mediator between users and taxi drivers.

-Time and cost calculator subsystem – this subsystem is used in order to calculate the estimated waiting time for the taxi and cost. It is used by scheduler during negotiations in order to make drive offers to users.

-Taxi zone determination – calculate the corresponding taxi zone for a given location.

-Developer subsystem – operations related to external API offered to developers. They can virtuallz access any function from external app.

-Administrator subsystem – gives ability to browse users, edit them, delete them, view their drives and reports or promote them to taxi drivers or downgrade them

-Data subsystem – stores the classes of the necessary entities, deals with database and operations performed over database.

The application will use client-server architectural style in general, with some variations. So, in what follows a brief overview of the separation is given.

Unregistered user, Registered user,Taxi driver, Developer user and Administrator subsystems have their separated counterparts on the both client and server sides.The client side of these subsystems are in interaction with users, while the server-side counterparts handle the user actions and requests. Scheduler, Time and cost estimation and Data subsystems are present on the server side in order to to deal with these requests, create responses according to the enviornment conditions (negotiations between users and taxi drivers, availability status changes,their decisions etc.).

Data subsystem could be located on a separated server, in order to fulfill the security and other non-functional requirements, but this part will be explained in details later, when deployment and architectural style are considered.

Session subsystem is also necessary, as it is needed to handle log in and log out operations. Once users log in, they can use service, without need to log in again after each action. When they finish the use of the application, they can log out.

## High level components and their interaction

Taking into consideration previous conclusions, the high level components and their interactions are derived and descirbed.

As it can be seen, the system consists of clients that belong to different types of users (unregistered/registered, taxi driver, administrator and developer), server which consists of parts that handle the requests from corresponding clients, and data layer part, which is separated from the application part of the server and deals with database queries.In fact, the application part of the server uses data layer part, in situations when it is needed to deal with database – add users, delete users, show reports, update etc.

Unregistered user’s actions could be either log in or register. The corresponding part of the server deals with these requests. In case of log in, the Unregistered user manager consults the data layer to check if the user really exists. When register action is taken, application server invokes insert database query and adds new user (assume that data is correct and valid).

When user is logged in, the client changes its appearance, corresponding to the type of user. User that is logged in can send taxi requests, modify profile, send offer responses, logout. The part of the server that deals with typical users is UserManager. If the user that is logged in is taxi driver, the person gets interface that allows to change availability status and also respond the drive requests. Part of the server that deals with taxi drivers is TaxiDriverManager.

Administrators can browse users, view reports, drives, promote to taxi drivers or even revert back taxi drivers to users.

Developers have programmatic interface that lets them call any function from MyTaxiService, except the functions related to administration. Component that handles the API calls is DeveloperManager and belongs to server part of the application.

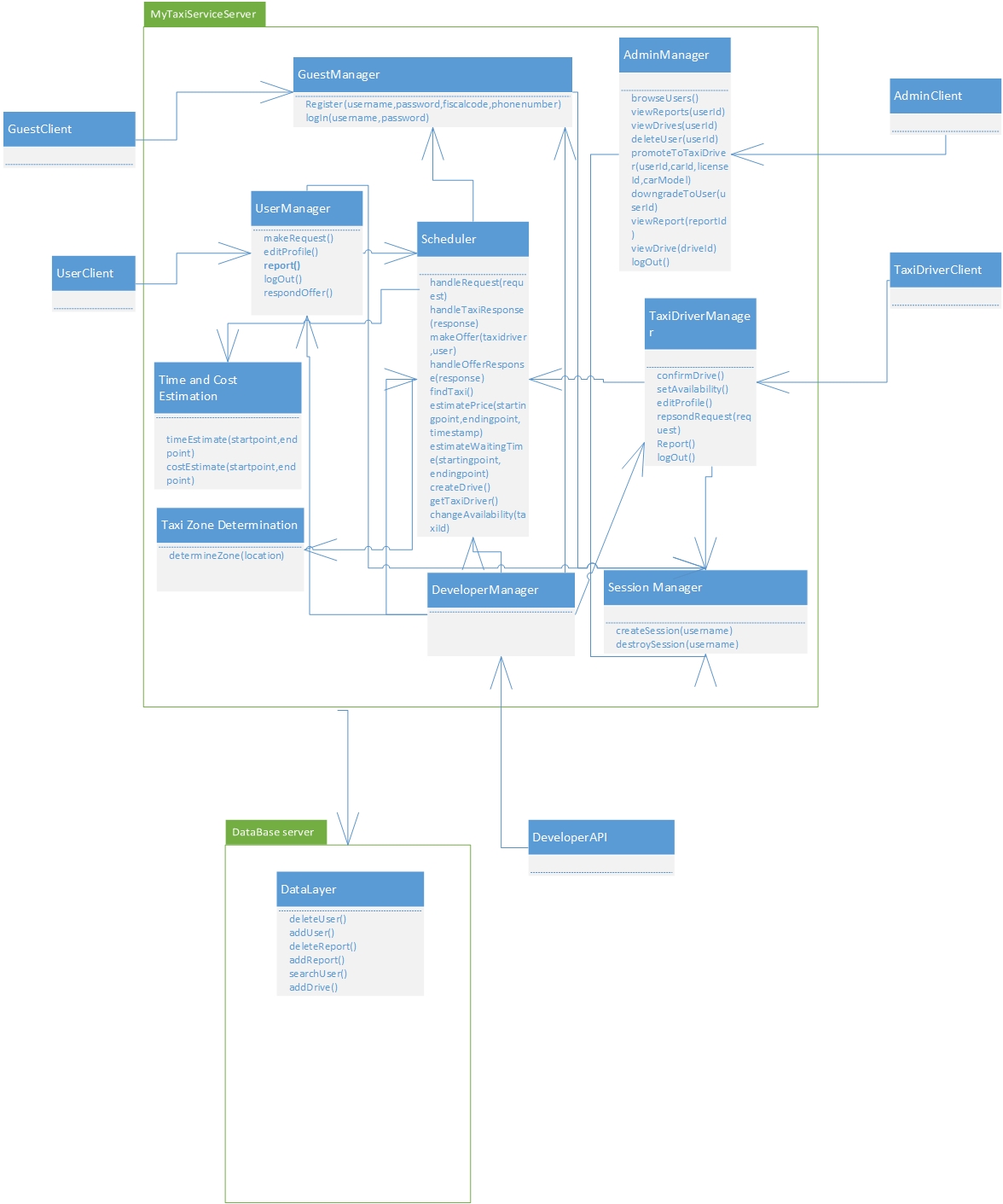
Scheduler is responsible for handling requests, responses, taxi scheduling and estimation of waiting time and price. This part is used by other components of the system, such as UserManager and TaxiDriverManager in situations when it is necessary to handle the negotiations between them.

DataLayer is responsible for actions taken over database and is used by application server of the MyTaxiService. It encapsulates necessary actions that interact with database: search, insert, delete and update queries, so the application server has to use it in order to satisfy requests that need access to the database. For example, to check if there is already user with same username during registration, validity during log in, checking car id validity and fiscal code during taxi driver promotion and registration, adding and viewing reports, browsing users etc.

Session manager is responsible for log in and log out. It creates session, in case of log in, and destroys it, in case of log out. Each client manager uses it in log out, and Guest Manager uses it for log in and register (when new user is registered, he/she is automatically logged in).

On the next page, high level components and their interaction are presented in a diagram.

Notice that some parts are omitted (parameters and attributes), and only the most relevant among them are listed in order to make image easier to understand. For example,Taxi driver Manager contains all the same interfaces as Registered User Manager plus additional interfaces. Only additional interfaces are displayed. Parameters are going to be explained later in component interfaces part.



## Component view

The system follows client-server style for what concerns the management of the interaction with the users (including system administrators, taxi driversn and developers) and interacts, through a service interface (REST interface, for example) with GPS coordinates service and city government service. At the same time, the server offers REST interface to developers who want to use functionalities of the MyTaxiService in their application in order to extend it.The developers are outside of the system, so their client is not considered.

The server contains main logic and is in charge with interacting with tfour kinds of clients – RegisteredUserClient, TaxiDriverClient, AdminClient and UnregisteredClient. Moreover, the part of the server that is related to data that needs to be check if it is valid (fiscal code, driving license, car id) uses external access to city government databases. Furthermore, the clients of users and scheduler use GPS coordinates service (like Google Maps API etc.) to send location data, receive it, manipulate it.

The server consists of folowing parts:

-UnregisteredManager is a component which gives ability unregistered users to register or log in. This part of the server uses external city government databases through REST service in order to check the validity of fiscal code. It also uses DataLayer to check if there is same user already in a database or the user is banned and can’t be registered. In case of successful registration, this component uses DataLayer to add new users to database once they are registered. This component, as it can be seen, doesn’t use the scheduler, as unregistered users can’t make taxi requests.

-RegisteredUserManager is a component which enables users to logout , receives all the messages from users - drive requests, drive offer responses, reports, SOS signal, modify their profile, log out and all other functions that are present in RASD usecase diagrams. This component can access GPS coordinates service in order to determine the users location when necessary. In order to acomplish all of the operations, this component interacts with the Datalayer and visualises all the necessary elements when it is needed (maps, etc.). Also, this component uses Scheduler in order to serve the users in requests related to taxi drives.

-TaxiDriverManager is similar to previous one, but also includes some additional options, like accepting or rejecting a drive request and changing the availability status, in order to support TaxiDriver clients.

-AdminManager is supporting AdminClients, and gives them ability, using the DataLayer, to browse users, read reports, promote users to taxi drivers, prevent some users from being drivers, delete users in case of bad behaviour, and log out. Notice that Admin manager doesn’t use Scheduler, as it isn’t necessary (Administrators can’t make taxi requests).

-Scheduler is the core component of the system that cross-references the location data and the system messages in order to dispatch a taxi from the corresponding taxi zone using the DataLayer. It uses GPS coordinates service in order to determine the taxi zone of the current user, deals with taxi queues and forwards the request to another taxi in current taxi zone if there is available taxi. It also receives the drive offer responses from uses and creates drive events. Scheduler uses results from time and cost calculator when makes drive offers to users.

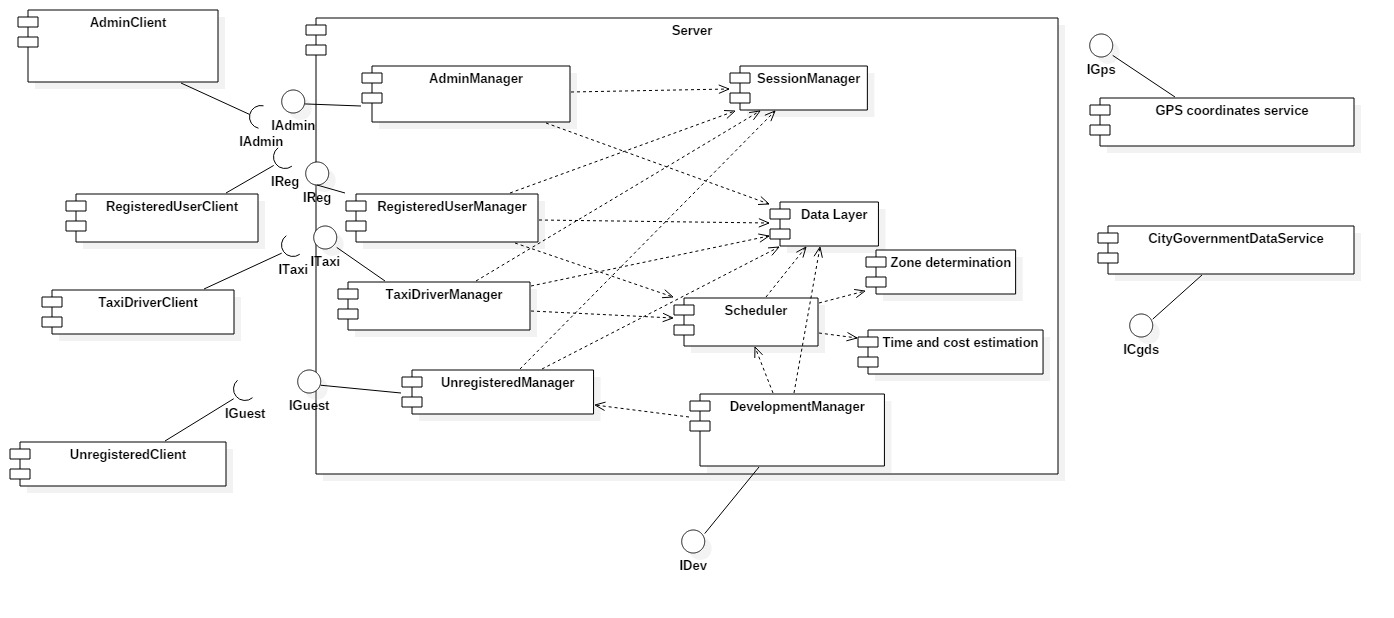
-Time and cost calculator. It estimates the time the time and price for a taxi drive. It uses data from DataLayer in order to have enough information to do its part of job. By the way, the Scheduler uses this component when it makes a drive offer to user- so it makes this system fair. Users can accept or rejecet the offer if the price is too high.

-Zone determination – component that determines the taxi zone corresponding to a given location.Uses maps external GPS and Maps API.

-DataLayer encapsulates the operations related to persistence of entities relevant to the system – report, drive, different types of users – User, Taxi Driver, Administrator and deals with their storage into database.

-DevelopmentManager is a part of the system which offers access to basic system functions to developers that are outside the system via some kind of serivce (REST, for example) in order to give them ability to embed or extend MyTaxiService in their applications. They can externally request a taxi, for example by API or register or log in from external aplplication (for example, Facebook).

-SessionManager, a part of the system that is responsible for creating and destroying sessions (log in- create session, log out-destroy session). It is necessary to have this part of the system included, because it distincts logged in and logged out state.



## Deployment view

In this section, the deployment of the components and processes is presented. Using this information, reader can find out where each of the components is located and executed and details about communication between nodes. As it is already told, the application consists of server and client parts. Server side has web and application tier separated. Data tier is also separated from them and is used to handle the database operations. This kind of separation gives option for better security implementations, as it has been told in RASD.

In what follows, the system deployment is going to be described and illustrated.

Client could be either mobile phone which runs mobile application or personal computer running web browser.

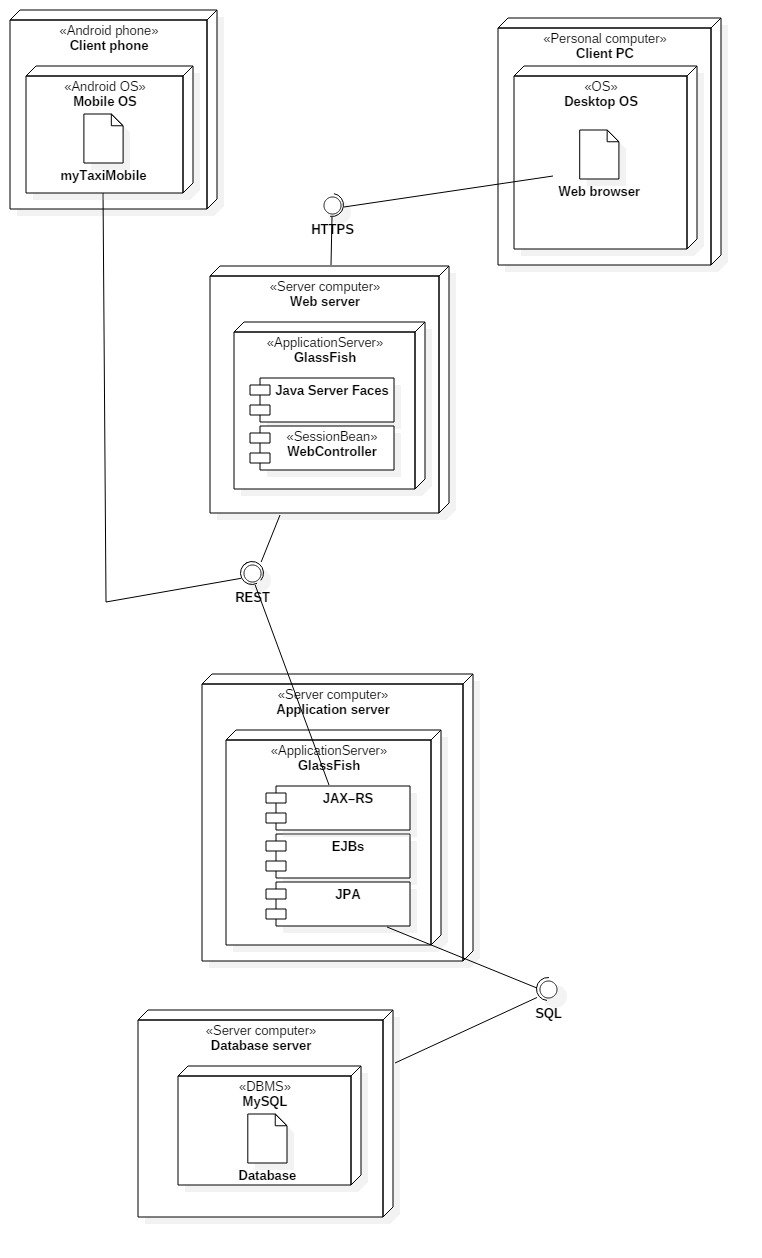
In case of mobile phone the application communicates directly with application server via REST[[3]](#footnote-3) service. JAX-RS[[4]](#footnote-4) is used for providing REST service in the application.

In case of web browser, we have HTTPS[[5]](#footnote-5) communication with WEB server which is running Glass Fish with Java Server Faces and has Session Bean.

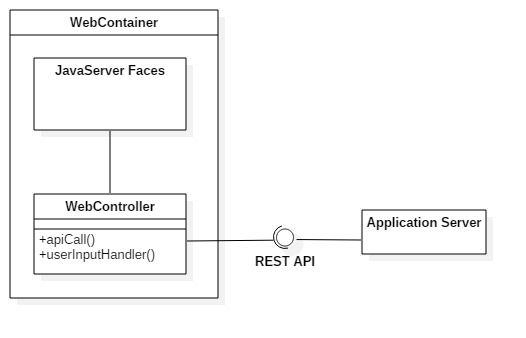
Then, the web server communicates via REST with application server. JAX-RS is on top layer for communication, while the business logic itself is contained in Enterprise Java Beans. JPA[[6]](#footnote-6) is used for persistence and provides object-relational mapping which gives ability to translate SQL query results into objects and use them.

And finally, there is a database server which runs MySQL and is responsible for database operations performed by the application server. Queries are formed and executed according to the business logic, and results are transformed into objects using object-relational mapping. So, in this form (objects), it is easier to use the query results and present them to users.

I would like to mention that there are different approaches when it comes to deployment view (UML 1.X, UML 2.0), but I used the approach that is a combination of both and that would give a better overview of the system and introduce information not previously given by other diagrams (component view, high-level component) and is based more on 1.X approach. In section related to interfaces between components, another variation of component and deployment diagrams is used in order to illustrate communication and deployment of the components via their interfaces.



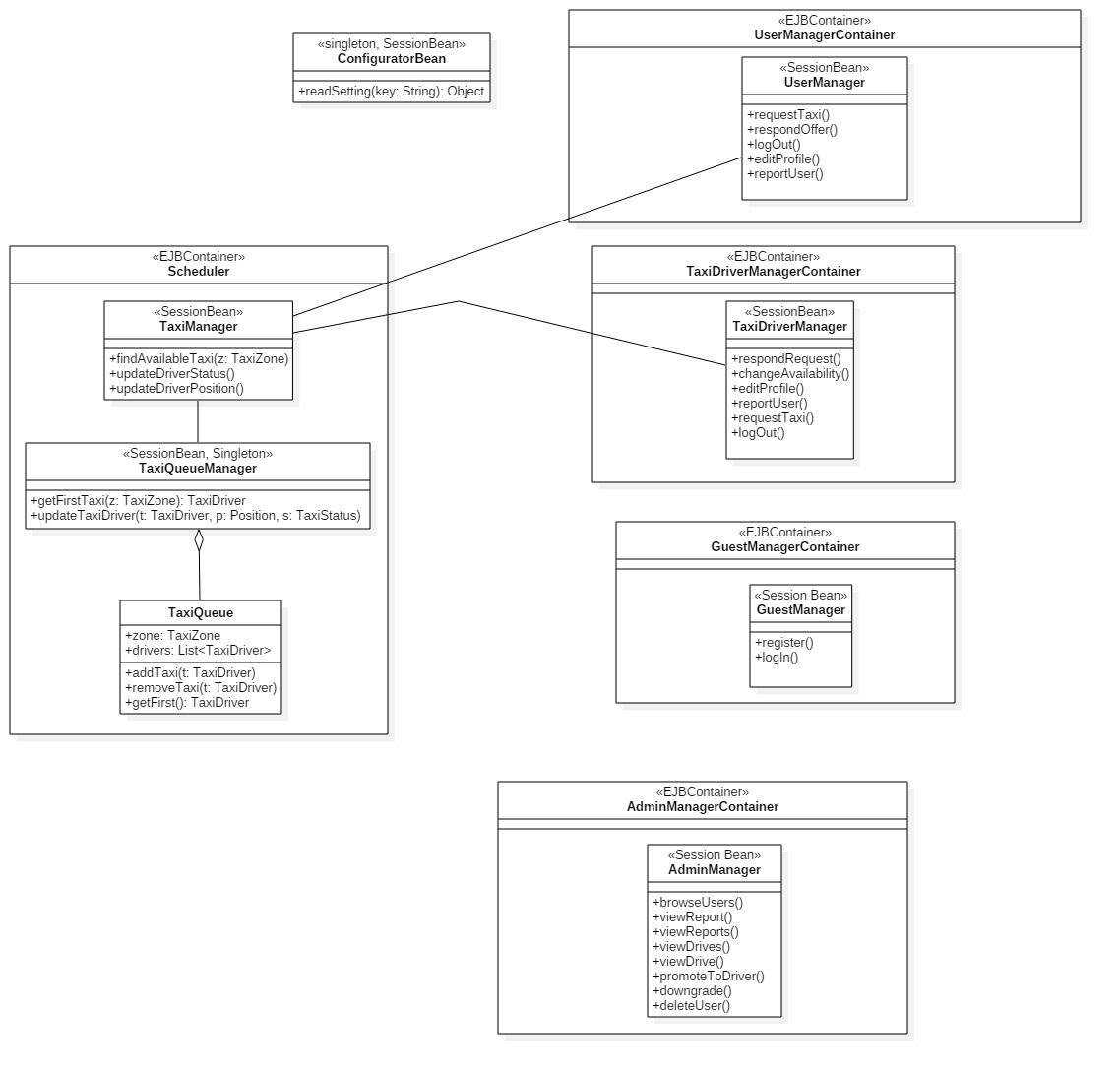
## Now, let’s consider connection between components in more details.



As it can be seen, user takes actions on pages (inputs data, clicks buttons, scrolls through the lists, chooses available options, etc.). Web controller handles this actions using event handlers (on click, on submit, on typing etc.). After that, action is interpreted by these handlers, and corresponding API call is made via REST API to the application server, as it can be seen above. After that, application server communicates with database server and queries are executed (in case that database access is needed for triggered service).

In the picture below, deployment of Session Beans and their operations are presented.

Different types of user managers receives actions from Web Controller controler. After that, Session Beams take their role. Session Beams operations are executed, and in some cases, they interact with database. Taxi and User Managers interact with Scheduler. Database interactions are not present in this diagram.

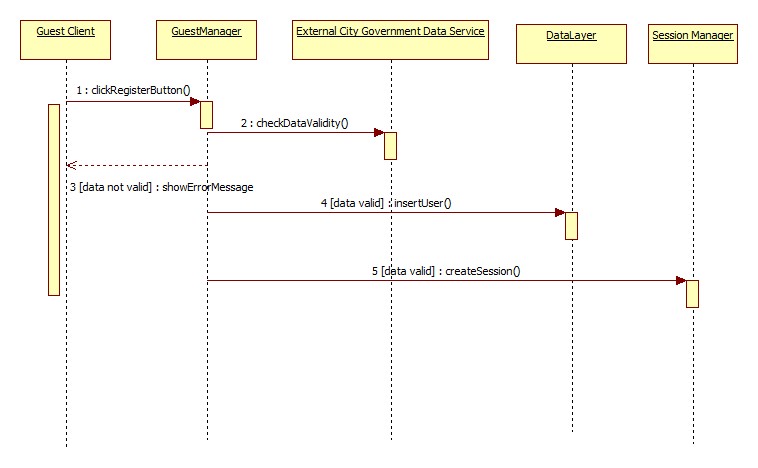


## Runtime view

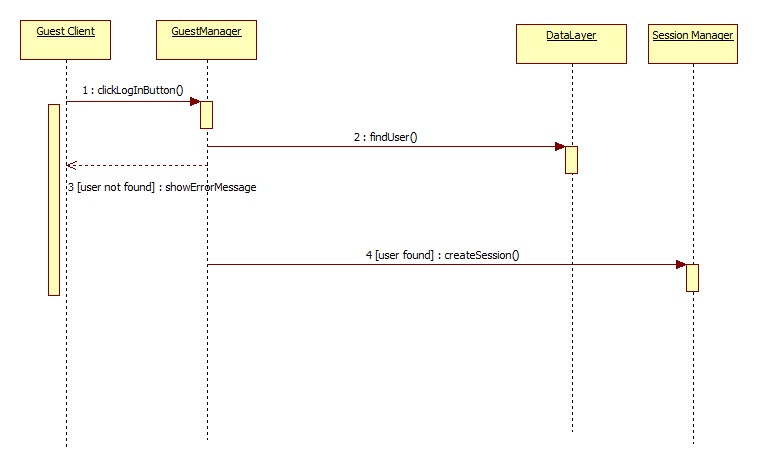
This section deals with the components interaction during runtime. Each time some action is performed in the application, it is translated to a set of component interactions. To illustrate runtime component interactions, sequence diagrams are going to be used.

Register user

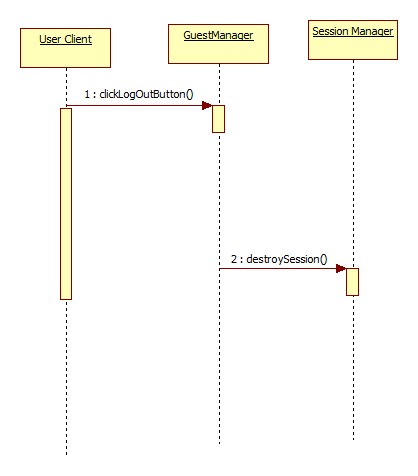
New users can register by entering the necessary data. When users finish entering their data,

they click on register button. GuestManager then checks if data is valid (fiscal code and identity correspondence) using external database provided by City Government. If data is not valid, guest is informed in his/her client. Otherwise, GuestManager interacts with DataLayer and inserts new user into database. After that, user is automatically logged in, so the new seesion is created in interaction with Session Manager.

Log in

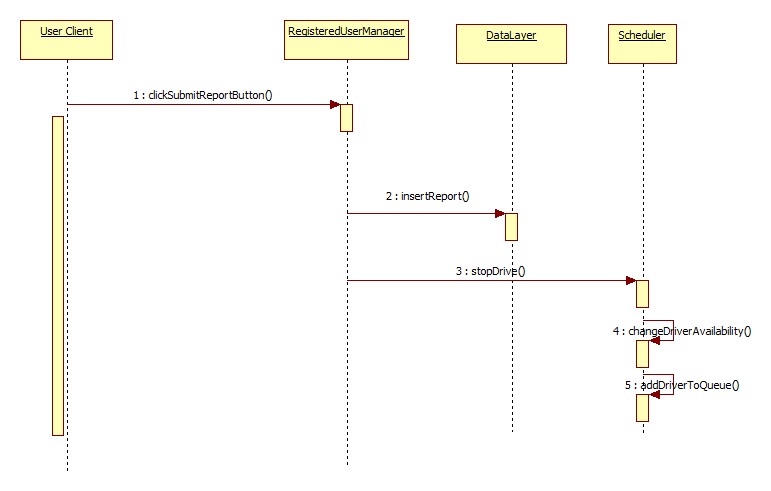
If user enters valid combination of username and password, he/she is going to be logged in and new session is created.

Log out

If user decides to log out, session is destroyed.

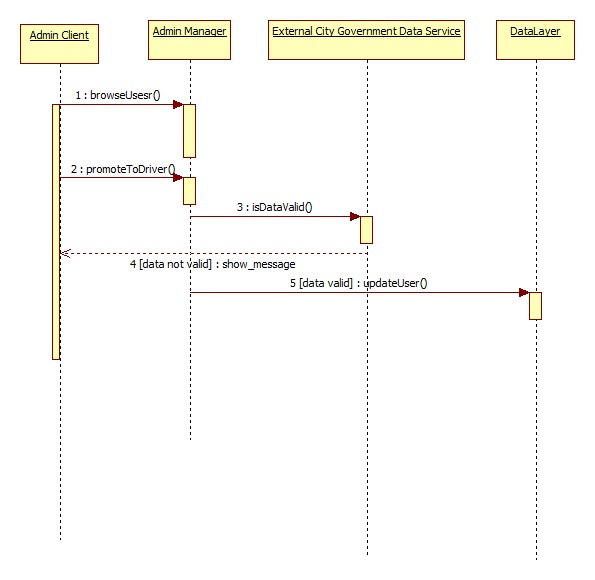
Report user

User can report another user typing in the reason and clicking the SubmitReport button. After that, RegisteredUserManager component interacts with DataLayer, executing query that inserts previously written report into database. It is also needeed to stop drive event, and taxi driver is available again and added to queue. In this case, RegisteredUser Manager interacts with Scheduler, while the scheduler itself changes driver availability and adds it to taxi queue.



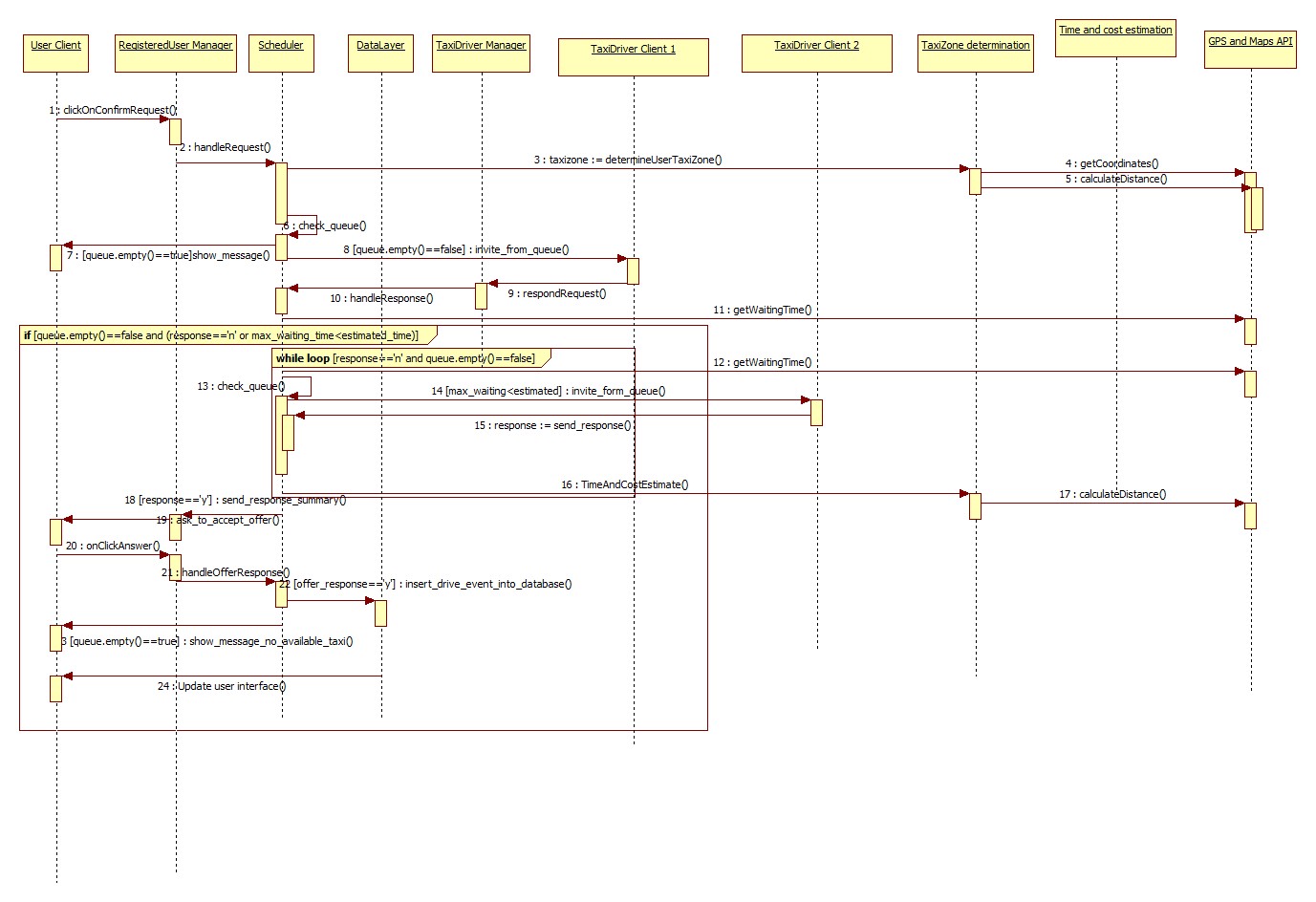
Promote to taxi driver

Administrator can promote users to taxi drivers. Administrator first finds the user he/she wants to promote to taxi driver, then enters the data. After submitting the data, car id number and driving license validity are checked using the external database. If data is not valid, administrator gets message. If data is valid, update query is performed in database (interaction between AdminManager and DataLayer).



Request a taxi

Crucial part of this software is related to taxi requests. Users can make taxi requests. They click location on a map and optionally enter maximum time they are able to wait. After that, UserManager consultes Scheduler to handle the request and dispatch a taxi if there is one available. Scheduler than looks up the queue corresponding taxi zone of the user that sent the request. Component dedicated to taxi zone determination does this part of job, and returns the value to the scheduler. Now, scheduler polls taxi from queue. Maximum waiting time is compared with the time needed for taxi to come to user’s starting point. External GPS Maps API is used for this. Taxi driver can either accept or reject drive request. If taxi driver rejects the request, than this request is forwarded to next taxi in corresponding queue. When driver accepts the request, user gets an offer which provides information like estimated cost and waiting time. In these calculations, external component responsible for GPS maps API is used, while cost estimation is part of the system. If users accepts the offer, then, new drive event is inserted into database. But, in situation where user rejects the offer, than Scheduler stops searching for taxi.



## Component interfaces

In this part of the document, component interfaces are going to be described. In section 2.3, interfaces between components are presented, but not in details. In what follows, each of the component interfaces is going to be broken into functions whose parameteres and retrun values are going to be described based on what is presented already in sections 2.2 and 2.3 inside this document.

Interaces:

1. IAdmin

This interface is used by admin users and contains functions that allow administrators to:

* 1. Browse users

void onClickBrowse(Event e)

Users[] browseUsers(string username, string firstName, string lastName, string fiscalCode)

* 1. View reports

void onSelectUser(Event e)

Report[] viewReports (string userid)

* 1. View drives

void onSelectUser(Event e)

Drive[] viewReports (string userid)

* 1. View drive and view report

void onSelectDrive(Event e)

void onSelectReport(Event e)

void viewDrive(string driveId)

void viewReport(string reportId)

Drive viewDrive (string userid)

Report viewReport (string userid)

* 1. Delete user

:

bool deleteUser(string userid)

void onUpdateUser(Event e)

bool updateUser(userid,carId,carModel,licenseNumber)

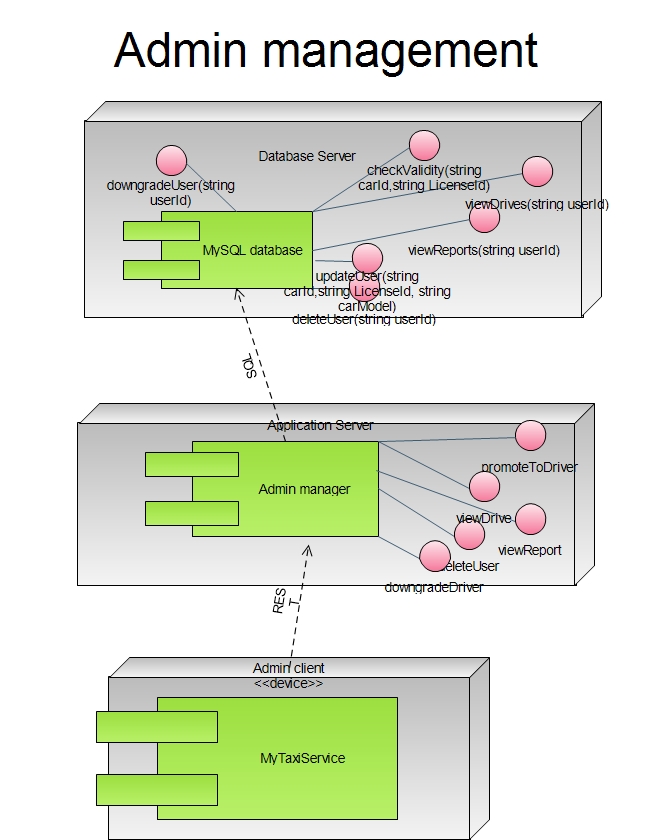
Bool isEligible(string carId, string licenseId)

bool downgradeUser(userid)

bool isValidFiscalCode(string fiscalCode)

bool isValidCarId(string carId)

bool isValidLicenseNumber(string licenseNumber)



2. IGuest

2.1 Register

void onClickRegister(Event e)

User registerUser(string username, string firstName, string lastName, string fiscalCode, string password, string confirmPass, string picturePath, character Gender, string mobilePhone)

void createSession(string username)

2.2 Log in

void onClickLogin(Event e)

User newUser=Login(username, password)

void createSession(string username)



3. IReg

This is interface related to registered users. They can edit profile, make requests for a taxi, log out, and respond the drive offer.

3.1 Edit profile

void onClickModify(Event e)

User ModifyUser(user.username, firstName, lastName, fiscalCode, password, confirmPass, picturePath, Gender, mobilePhoneNumber)

3.2 Report

void onClickReport(Event e)

Report newReport(string username1, string username2, string reason)

TaxiDriver getTaxiDriver(string username)

.

3.3 Log out

void onClickLogout(string username)

void destroySession(string username)

3.4 Request taxi

void onClickSubmitRequest(Event e)

Request makeRequest(User u, String maxWaiting, Location statpoint, Location endpoint)

3.5 Respond offer

Void onRespondOffer(Event e)

Drive handleResponse(Response response)

4. ITaxi

Now, let us consider Taxi Manager interface. It is very similar to IReg. In fact, it includes all the functionalities by IReg, but also includes some new interfaces related to request accept or reject.

4.1 Change availability

Each taxi driver could either be availble or not. After this action, availability state becomes its complemet.

When taxi driver pushes the switch, event handler is activated

void onAvailabilityChange(Event e)

After that, interface towards Scheduler is used

bool changeAvailability(string taxiId)

Scheduler changes availability of thet taxi and removes it from or adds it to corresponding queue.

4.2 Respond request

Taxi driver accepts or rejects the request for a drive. Taxi driver has interface to TaxiDriverManger to respond the request by accpeting or rejecting it.

bool onClickButton(Event e)

{

if (e.ClickedButton.text==’’Yes’’)

return true

else return false

}

TaxiDriverManager contacts the Scheduler by interface that can handle the response

TaxiDriver handleTaxiResponse(Response response)

Return value is null if request is rejected, otherwise, the response is Taxi Driver that accepted the request.

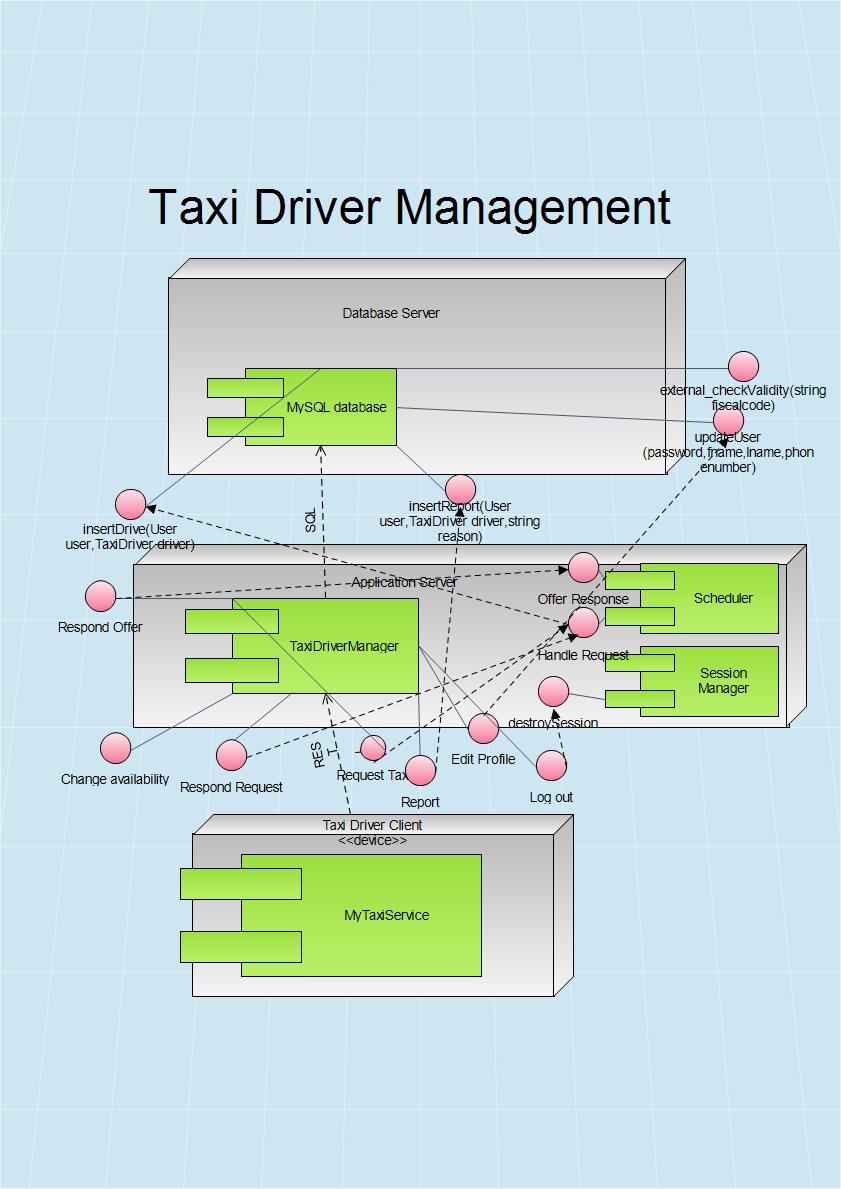
If answer is positive, then Scheduler will make an offer and forward it to user. Two functions will be used in order to estimate time and price:

float estimatePrice (startpoint, endpoint, timestamp)

Return value is price in euros. In section 3.3 (Cost estimation algorithm) this function is going to be defined completely.

time estimatedWaitingTime(startpoint, endpoint)

This interface is used to estimate how much user will have to wait until the taxi comes. After that, the Scheduler makes offer to the user. User can either accept or reject the offer.

Response makeOffer(TaxiDriver taxidriver, User user)

5. IDev

We shouldn’t forget that the application will have to offer API to developers. This API could give developers a chance to make their own applications that use MyTaxiService interfaces. In fact, they canuse any of the interfaces provided and make application that either extends or uses MyTaxiService. Of course, these developers should be registered by government in order to avoid misuse of this kind of freedom.

6. IGps

This is typical GPS API. We assume that it implements some basic operations, such as:

6.1.float distance(location startpoint, location endpoint)

This function returns distance between two GPS points.

6.2 Location getUserLocation(string id)

This function returns current GPS coordinates of the user, who is identified by device id number.

6.3 Time estimateTime(Location startpoint, Location endpoint)

This function estimates time required to travel by car from one point to another (Google Maps Api –alike function)

7. ICgds

7.1 bool isValidFiscalCode(string fiscalCode)

7.2 bool isValidCarId(string carId)

bool isValidLicenseNumber(string licenseNumber)

## Selected Architectural Styles and Patterns

2.7.1 Introduction

As it has been told already in the RASD document, the application will be released as a both web and mobile application.The application will provide API that could help other developers extend its functionality by developing additional services using the MyTaxiService functions as a primitives. Users can access the service both using mobile web application and browser-based web application. In order to use application as a taxi driver, taxi driver user has to login into mobile version of the application (because of the use of GPS sensor). Administrator’s application is targeted to be a web browser application, but will have also a mobile counterpart.

Considering the statements above, and according to the part 3.6.3.4 in RASD document version 1.4, the overall architectural design of the application will inherit the constraints of the programming languages, frameworks, technologies and communication interfaces used.

So, in what follows, the architectural design will be explained in terms of the technologies used – describing the architectural styles and patterns used and how their usage reflects on the design of this system.

2.7.2 Multitier Architectural Style

The selected architectural style is multitier based on Java Enterprise Edition implementation of this acrhitectural style.

In what follows, the advantages of this architecure and reasons for the selection aregoing to be explained.

The main benefits of the N-tier architectural style are:

Maintainability - Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.

Scalability - Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.

Flexibility - Because each tier can be managed or scaled independently, flexibility is increased.

Availability - Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Java Enterprise Edition is designed to develop large-scale and multi-tiered applications that are scalable and meet reliability conditions and are secure at the same time.

According to RASD, considering these and all the other functional and especially the non-functional requirements (RASD 1.4, part 3.6), it could be concluded that JEE is more than acceptable solutions in terms of these requirements.

MyTaxiService is going to be a multitier application that is also scalable (offer service to thousands of customers at the same time), but reliable and secure at the same time, maintaining high availability.

Server side needs to meet the special non-functional requirements in terms of security (RASD 1.4, part 3.6.4.3) – so the server side needs to be split into web, business logic and database part. Java EE offers the separate client, web, business and database tier, which is exactly what is needed. So, this will give ability to place firewalls between each two parts and make application secure and meet the security requirements.

So, in what follows, the core concept of JEE is going to be explained and the overall idea of the multitier implementation using JEE in terms of this application.

Multitier architectural model, in this case, consists of:

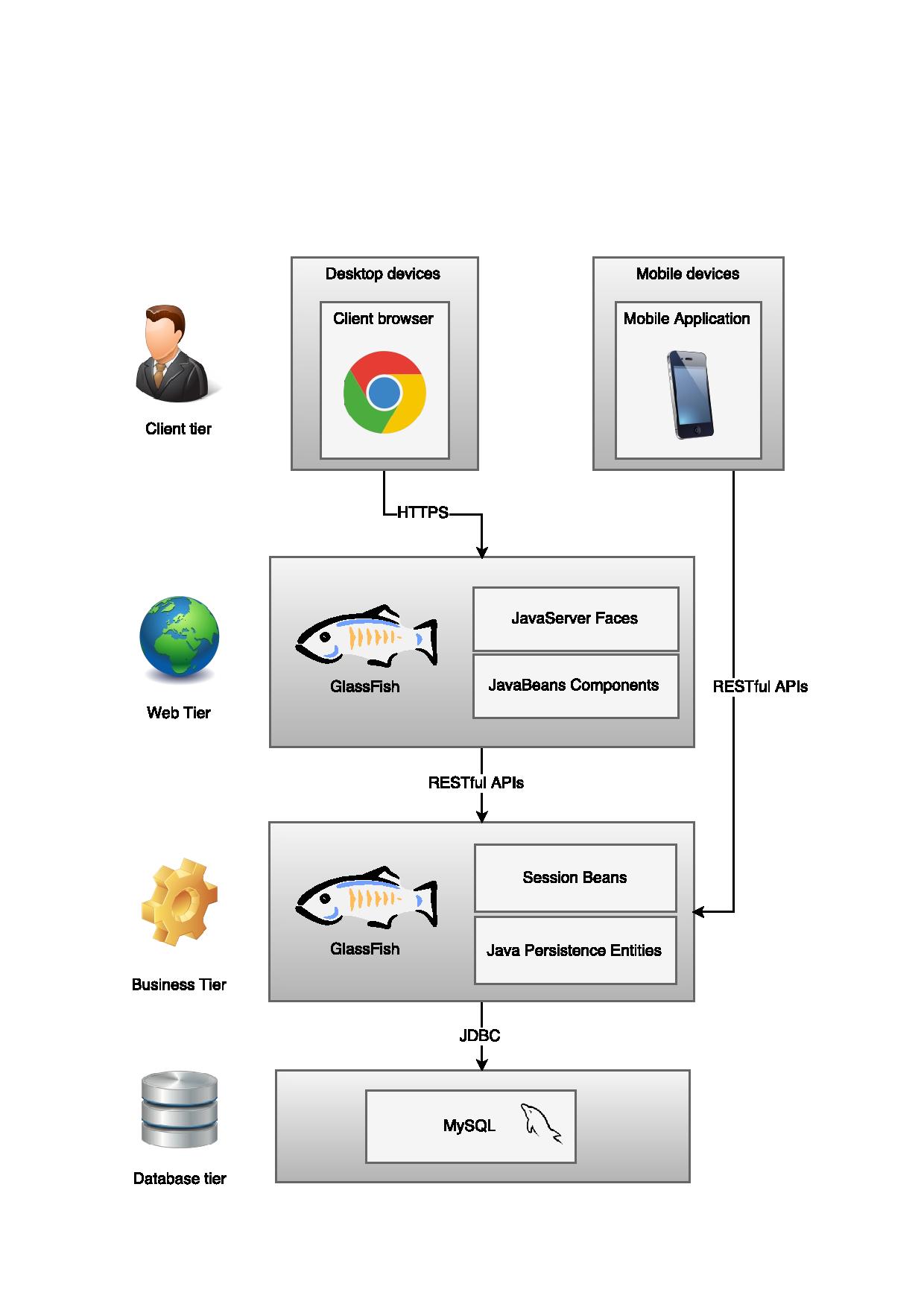
-Client tier that is running on the client machine. It contains Application Clients and Web Browsers and it is the layer that interacts directly with the actors. The client machine could be either mobile phone running application or web browser or personal computer running web browser in this case. Taxi drivers, according to RASD 1.4, part 2.1. must use mobile application.

-Web tier, running on the Java EE server. It contains the Java Server Faces. This tier receives the requests from the client tier and forwards the pieces of data collected to the business tier waiting for processed data to be sent to the client tier.

-Business tier, running on the Java EE server. It contains Java Beans, that contain the business logic of the application and Java Persistence Entities.

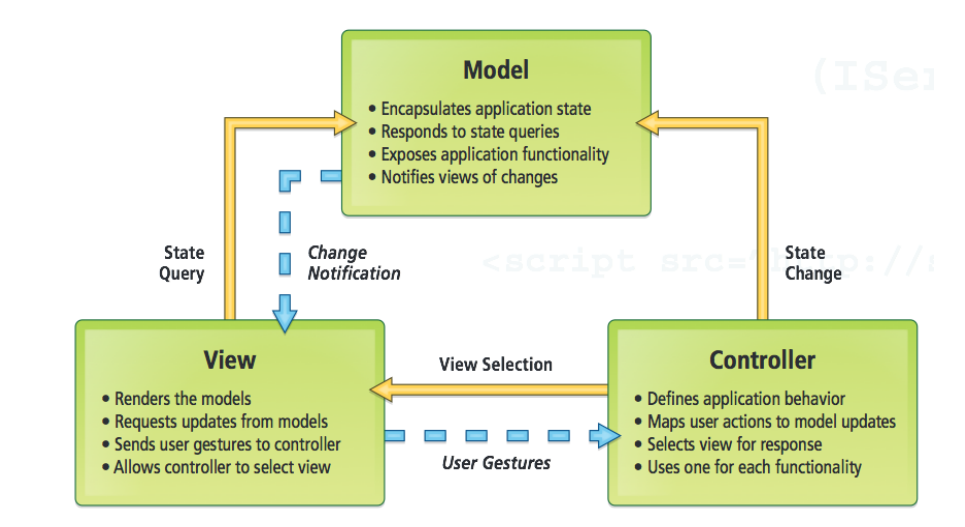
-Enterprise information system (EIS) running on the database server, consisting of data sources, to be more precise, databases and stores the data that needs to be retrieved and manipulated.

The server part is going to be run on a more powerful machine than a client – a high performance PC, according to RASD 1.4, part 3.6.2.1.

The illustration of the architectural style previously described, could be viewed below:

2.7.2 Model-View-Controller Pattern

MVC is a standard pattern that separates the user interface (View) and the business rules and data (Model) using a mediator (Controller) to connect model to the view.

The main benefit is the separation of concerns. Each part of the MVC takes care of its own work: the view takes care of the user interface, the model takes care of the data, and the controller sends messages between both of them.

Picture: Model-View-Controller pattern

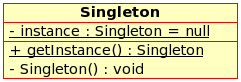
In MyTaxiService case, this pattern is going to be implemented on client side of the application – either a web or mobile app.

All the forms and pages that are used to interact with users (Register form, Login form, Edit profile form, Request taxi form etc.) belong to View. As users take actions – click on buttons – these forms send user gestures to controller. Controller maps user actions to model updates. The Controller classes are going to be defined for these forms that are previously mentioned. User actions trigger the state change. Model encapsulates application state and responds to state queries. When the model gets data, it changes according to the data received, so the model will notify the view about the changes, so the view could render the model.

2.7.3 Singleton pattern

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the singleton pattern is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that restricts the [instantiation](https://en.wikipedia.org/wiki/Instantiation_(computer_science)) of a class to one [object](https://en.wikipedia.org/wiki/Object_(computer_science)). This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the [mathematical concept of a singleton](https://en.wikipedia.org/wiki/Singleton_(mathematics)).

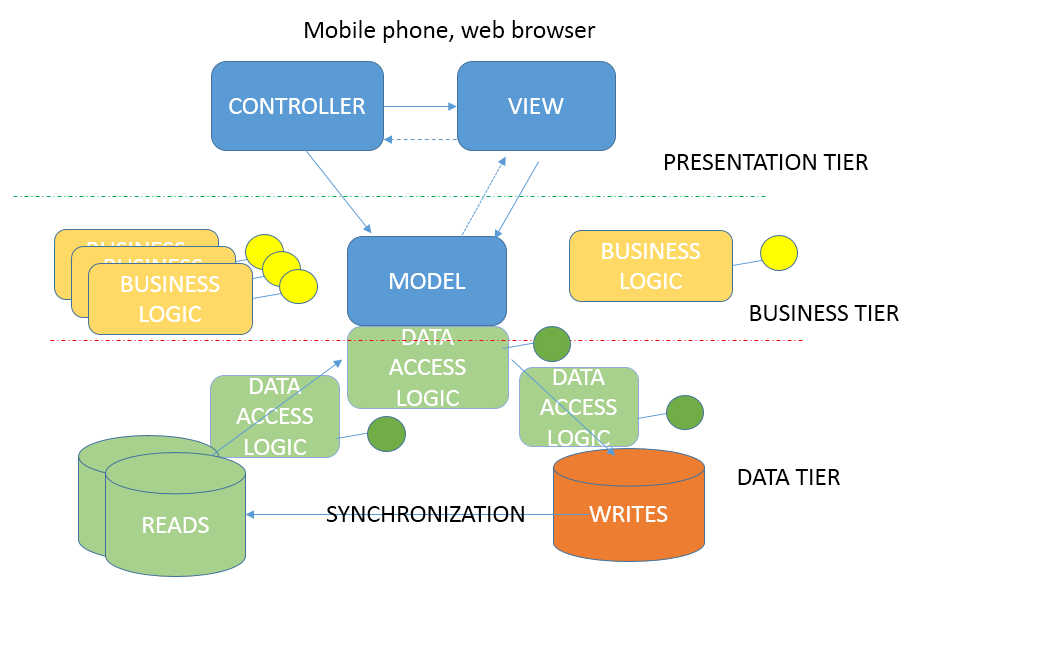
UML representation of singleton:



In MyTaxiService, the TaxiQueue manager as a part of Scheduler class is going to be implemented as singleton, because only one instance of this class is needed to take care of taxi queue and coordinate taxi management.

2.7.4 MyTaxiService architecture

Taking previous considerations into account, architecture of the whole MyTaxiService could be derived as a combination of n-tier client-server variant with MVC pattern implementation. Client and view are located on Client side as a part of Presentation tier, while Model is located on server side as a part of Business Logic tier. View would contain pages that contain input forms. Controller classes are classes that handle the user requests on these pages – send requests and responses,and according to that, update the user interface and are on the Server side in web application. Data access is located in Data Tier and deployed on database server. In what follows, the derived architecture is illustrated.

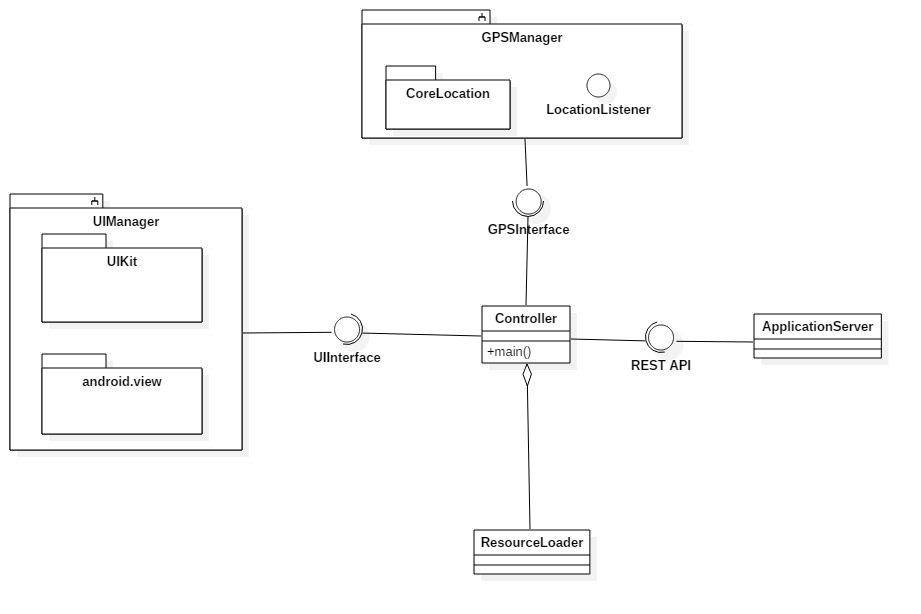


2.7.5 Mobile application

Users can use either mobile or web browser application, while taxi drivers must use mobile application. It is obvious that mobile application uses GPS sensor and services, so it makes more use to taxi driver in this form.

As it has been previously told in RASD, mobile application is going to be Android OS oriented, so android.view and CoreLocation are going to be used. This application implements a variantof MVC pattern.

User interacts with GUI. Controller transfers this actions to application vis REST API and returns results. Controller uses GPS manager in order to deal with operations that require GPS locations and transfers these values to application server via REST API.



# Algorithm desgin

This section explains the most important algorithms that are part of MyTaxiService application. There are three important algorithms that are unique to this application and they are going to be explained and illustrated, while others are not considered here, because most of them are trivial and well-known (register procedure, log in, etc.)

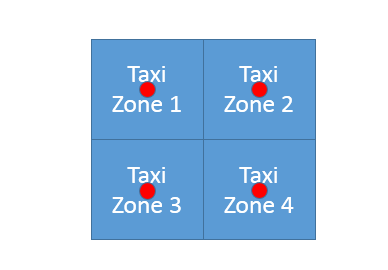
## Scheduling algorithm

This is the most important algorithm of the application, and represents the core of MyTaxiService. It represents the main idea of the MyTaxiService – fair taxi management, based on taxi queues assigned to taxi zones. The algorithm itself is performed by the Scheduler component, responsible for taxi dispatch. In what follows, high-level algorithm is going to be displayed – in a form of a diagram with high-level pseudo-code (see ALGORITHMS folder for a higher quality JPEG picture and file algorithm\_schedule.jpg or visio project file).

## D:\swe2\Software-Engineering-2-Project\Deliveries\DD\ALGORITHMS\algorithm_schedule.jpg

First, the user makes a request for a drive to the desired destination by selecting the desired location on a map and setting, optionally the maximum waiting time. It should be mentioned that maximum waiting time is used as the first criteria to forward request to next taxi. If taxi has to spend more time than user entered as maximum waiting time,in order to come to user’s location, then the request is forwarded to next taxi driver. After that, user submits the request, and after that, the taxi zone corresponding to user’s current location is determined (algorithm is going to be explained in 3.2). Scheduler checks the queue for the corresponding taxi zone if it is not empty. The Scheduler polls the first taxi from the queue and forwards the drive request by user. Taxi driver can either accept or reject. If driver rejects, the Scheduler continues to forward the request until it reaches the end of the queue for the corresponding taxi zone. If the driver accepts the request, then the user is notified about the drive offer that displays the estimated time and price (3.3). If the user accepts the offer, then the taxi is dispatched, and drive event is created. Otherwise, if user rejects the drive offer, algorithm will finish and user will be returned to previous screen.

## Taxi zone determination

As it is previously assumed in RASD document, the determination of the corresponding taxi zone is done by system. According to the definition of taxi zone in terms of this problem, each taxi zone is defined by its center point. It is also assumed that the divsion of the city region on taxi zones itself is done by the government manually (center points). But, it should be mentioned that it is required to have taxi zones that are approxiamtely 2km2 each. So, it would be necessary to find a model for this problem.

Let’s consider each taxi zone as a square of approximately 2km2 with center point defined in advance.

So knowing that:

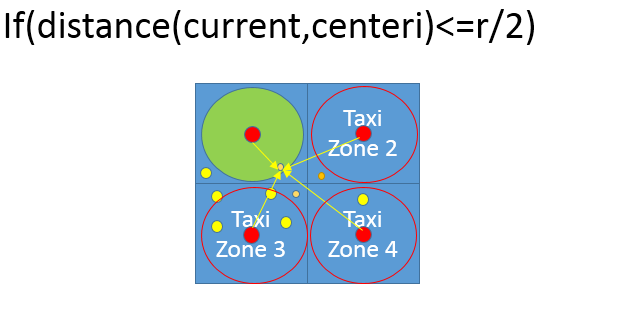
=2 [km2]

where a is length of the side, it could be concluded that maximum distance from the center in a particular taxi zone could be equal to half of the diagonal (outter circle radius) ,while we have:

[km]

=2 [km]

Now, let’s consider a possible situations.

1. We want to know what is the situation when we are completely sure that some point belongs to a certain taxi zone. In case that location is inside the inner circle of the square (r=a, so r/2=sqrt(2)/2 ), we can be sure that it belongs to a particular taxi zone.

Current- user’s currentlocation

Small ornage circles – users

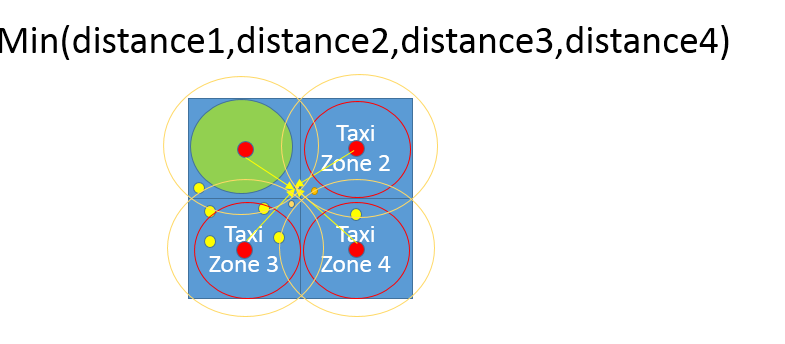
Yellow circles – taxi cars

Red circles – taxi zone center

Green circle – inner circle related to the corresponding taxi zone

r – radius of the inner circle

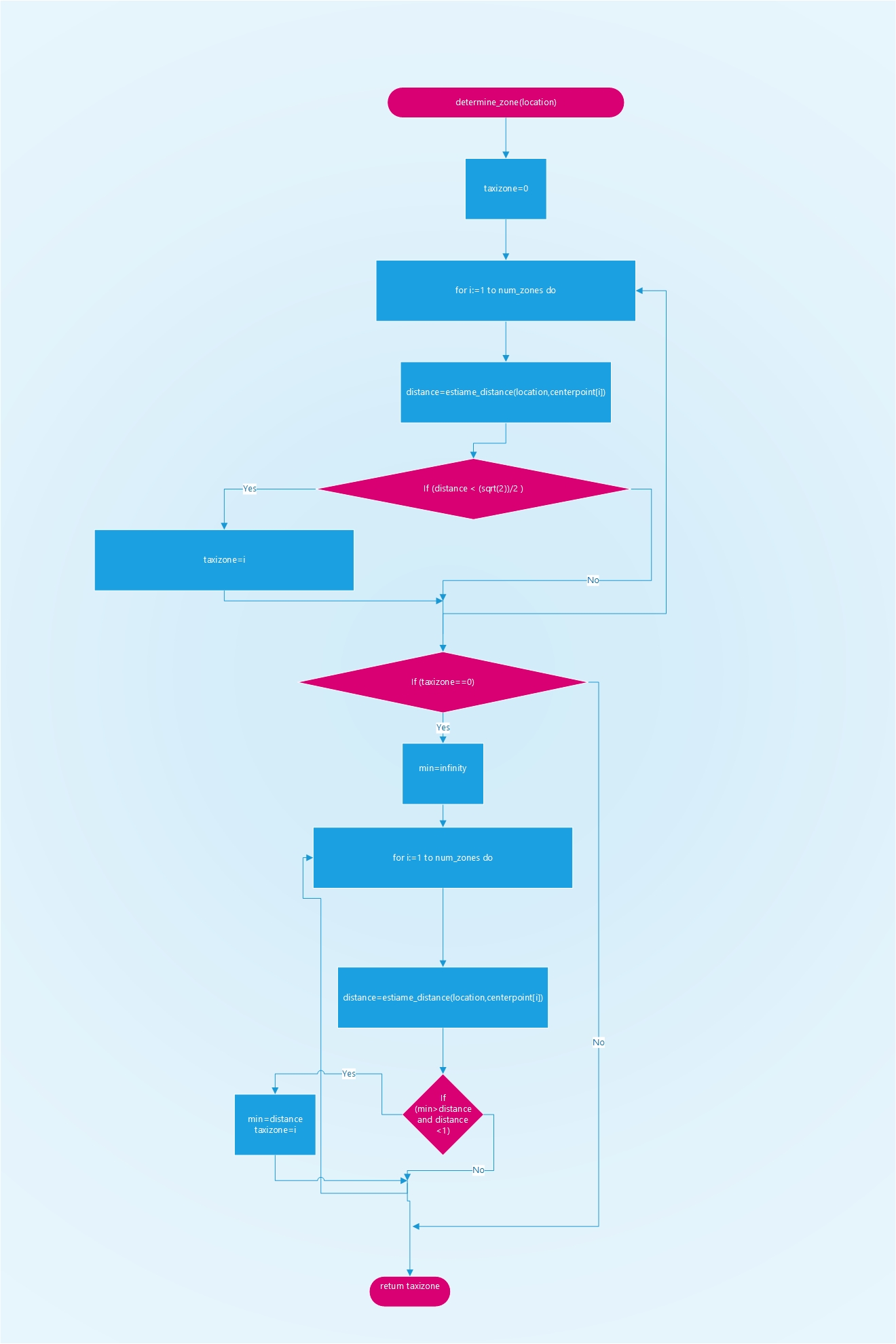
centeri – center corresponding to the ith taxi zone

1. Now, lets consider situatiion when the tested point doesn’t belong to inner circle. In this case, the outter circle is considered

After that, to find the corresaponding taxi zone, it is necessary to find the taxi center of a taxi zone which is the closest to the current location, because, as it can be seen, the current location can belong to intersection of taxi zones.

1. If the location is out of city boundaries, the taxi zone will stay 0 (see the algortithm), and it would be a sign that user has selected invalid location.

In what follows, the high-level algorithm interprettation is given in form of a diagram and high-level C++-alike pseudo-code. If you are having difficulties reading the algorithm, please see high-quality JPEG picture. ALGORITHMS/taxizone.jpg or corresponding visio project file.



## Cost estimation algorithm

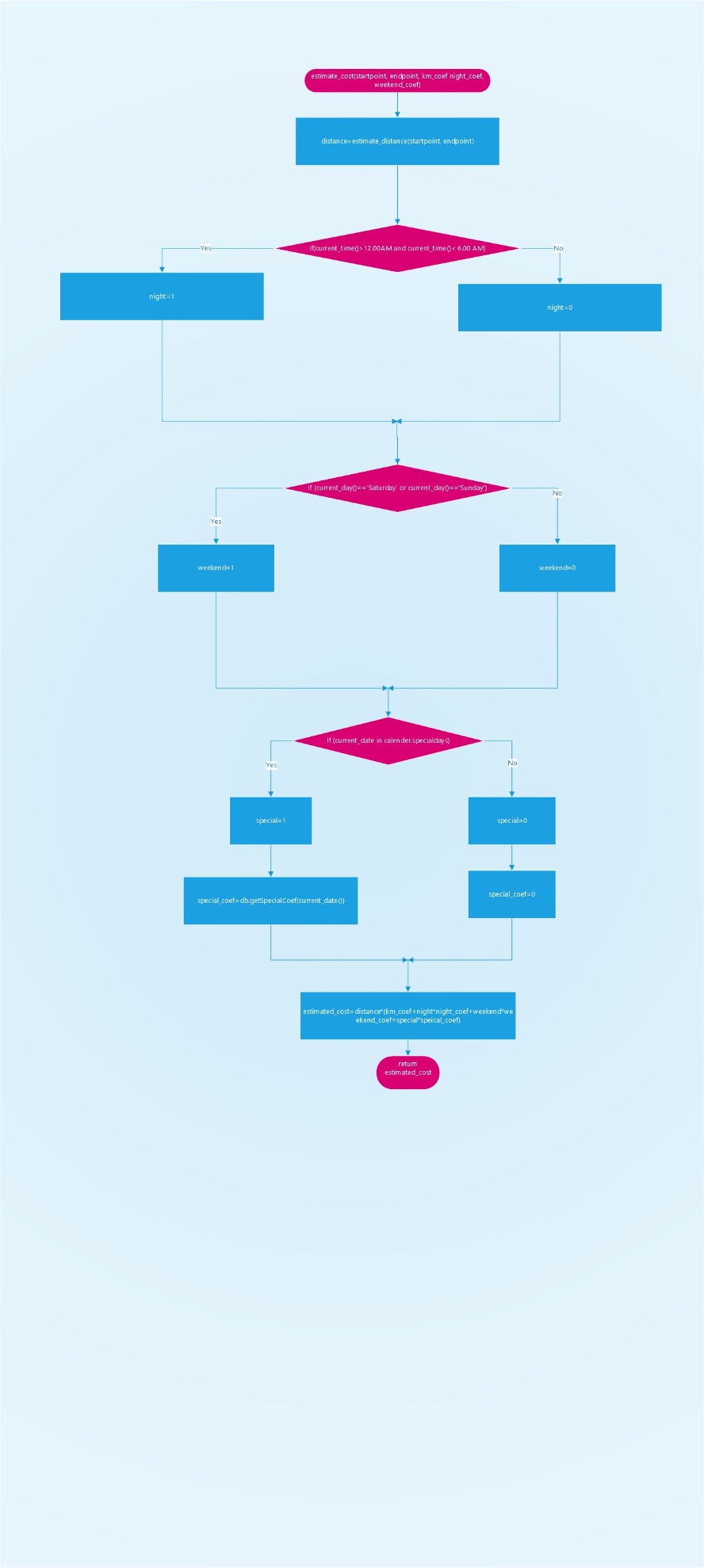
As a part of the drive negotitation protocol, cost estimation plays important role, as it gives overview to users what would be the price for a taxi service to the desired destination, so the user can either accept or reject the drive offer.

As the pricing policy could be modified and changed, the algorithm is going to be flexible. Several different factors are taken into account:

1. Distance – the government determines what would be the price per kilometer - rate
2. Time of the day – from 12.00 am to 6.00 am, price could be increased
3. Weekend price – on weekend, pricecould be lower or higher, depending on geovernment’s decision
4. Special days – it depends on current date. Calendar is checked. If it is a special day or holiday (Christmas, New Year, Thanksgiving etc.) a special price increase or decrease could be defined , so the database is checked for the coefficient of increase (positive) or decrease (negative coefficient).

Each of these coefficients could be changed according to the specific city government’s decisions, so this coefficients and rates are function parameters that could be easily changed.

In what follows, the high-level algorithm interprettation is given in form of a diagram and high-level C++-alike pseudo-code. If you are having difficulties reading the algorithm, please see high-quality JPEG picture. ALGORITHMS/cost\_estimate.jpg or corresponding visio project file.



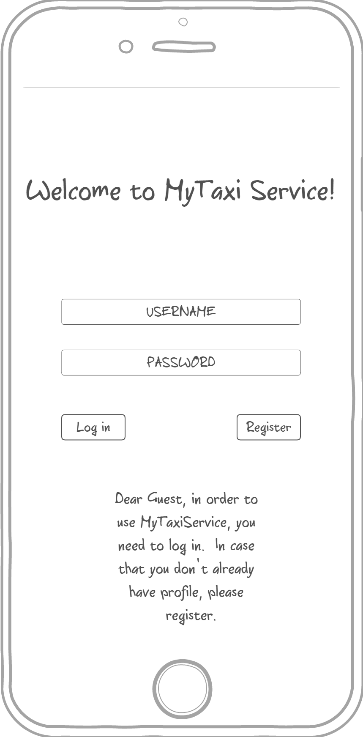
# User interface design

This part of the document references to RASD document, section 3.1.1.

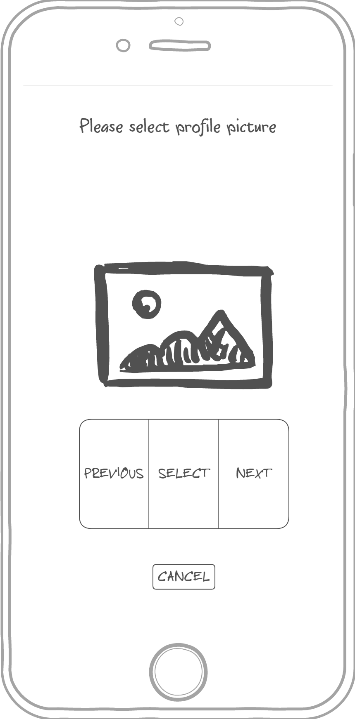
3.1.1 User Interfaces

In what follows, mockups that represent the idea and conceptual design of the application are presented – mobile and web application. The mockups presented may differ from the final product and they are placed in this document for mainly ilustrative purposes – to give you better idea how is the final product going to look. First, the user’s point of view is presented, and then, taxi driver’s and administrator’s, in this order.

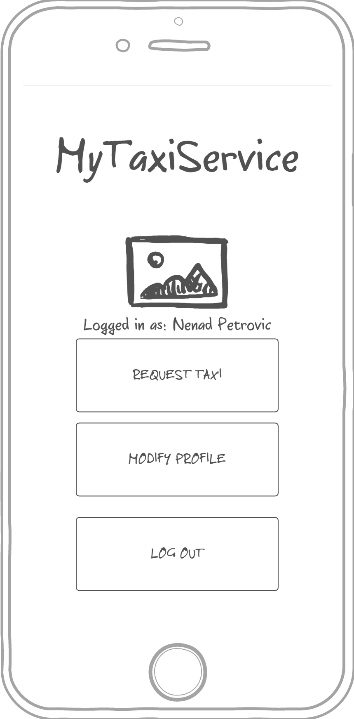
User’s point of view

3.1.1.1 Login - The mockup bellow shows the home page of MyTaxiService. Here users can log in to the application and guests can access to the regitration form in order to use the service. Here, the mobile application mockup is presented (the web version will look as similar as possible)

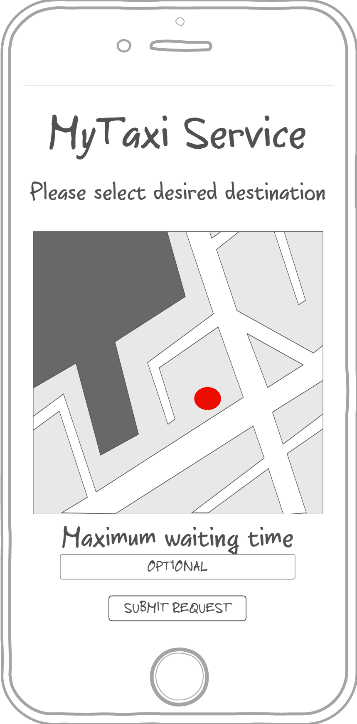
3.1.1.2 Registration form - This mock shows the registration form page on a mobile device. This is form that consists of user-filled textboxes. Profile picture is optional and helps in situations when user/driver needs to recognize another person that is involved into drive process. After pressing the button called „Confirm registration“, the guest becomes registered user and can benefit from usage of this system. Profile picture is optional, and user can select a picture from device’s storage.

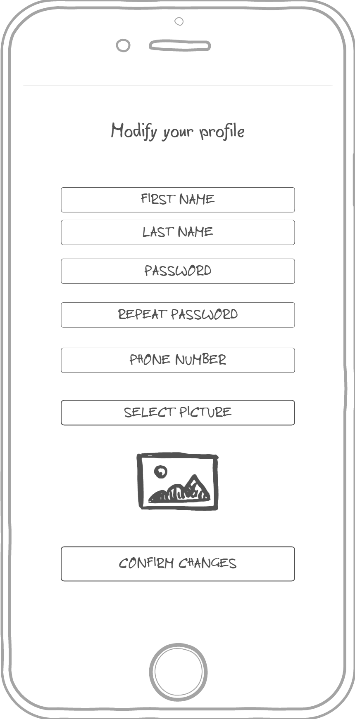


3.1.1.3 User menu – This mockup shows what user can do once logged in – request a taxi, edit profile or log out.

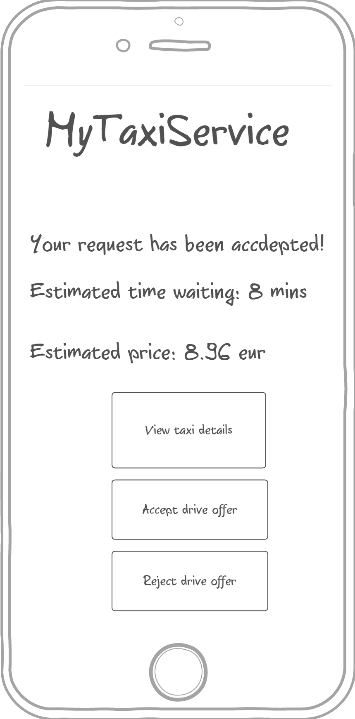
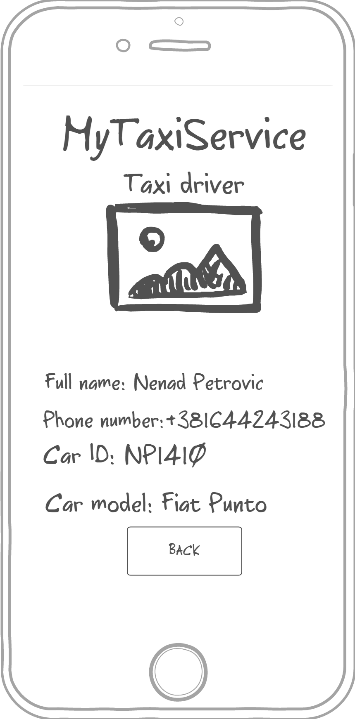


3.1.1.4 Request a taxi - This mock up shows what happens when user wants to send a request for a taxi drive. User selects desired location (left picture), optionally enters maximum waiting time and submits the request (right picture).



3.1.1.5 Profile modification - This mock up shows the form where user can edit his/her that or change picture, but can’t change fiscal code.

3.1.1.6 Waiting for response screen -This mock up shows what happens when the user submits the request for a drive. System responds within 180 seconds, but user can press quit waiting before that and cancel request and return to menu.

3.1.1.7 System response - This mock up shows what user sees when request is accepted by taxi. User can view taxi details (displayed on the right picture), accept or reject the offer. Estimated waiting time and estimated price are shown, so they can help user decide whether to accept the offer or not.

3.1.1.8 No taxi available – This mockup shows the screen displayed to user when there is no taxi available.

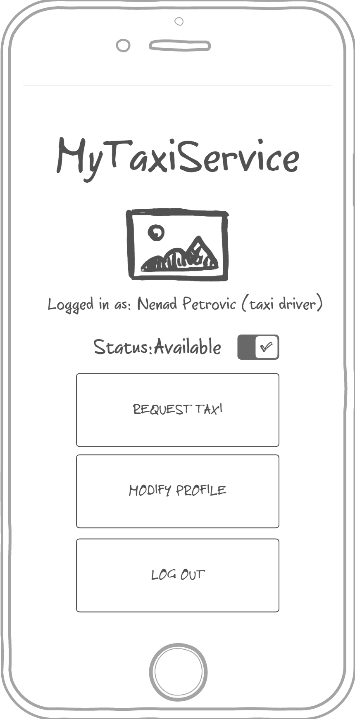


3.1.1.9 Drive progress and report form – This mockup shows the screen displayed to user after accepting the drive offer. User can see the drive progress, estimated time remaining, but can also report driver or send an S.O.S signal. After clicking the Report driver button, the screen on the right side shows up. User has to enter the reason of reporting the driver and confirm report in order to make it valid.

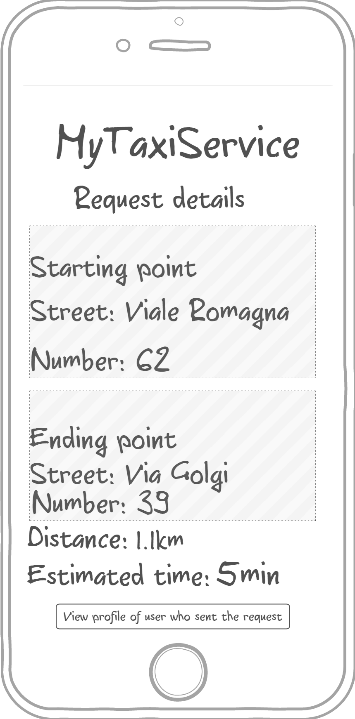
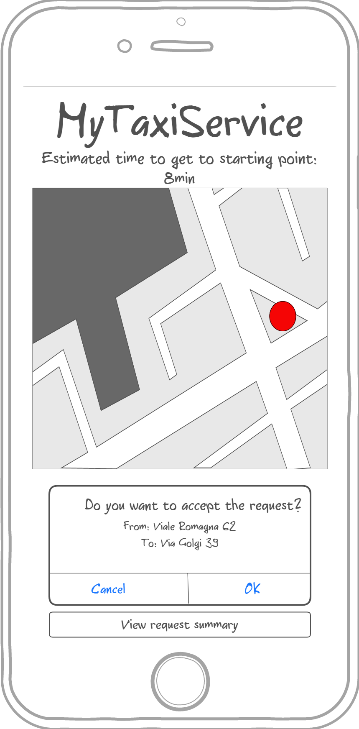


Driver’s point of view

3.1.1.10 Driver menu – this mockup is similar to user’s menu, but includes availability status button.



3.1.1.11 User request accept/reject page – Driver can accept or reject user’s request for a taxi drive. It is possible to optionally view request summary, as shown on the middle picture. After that, it is possible ro view user profile of the person who sent the request (right side).

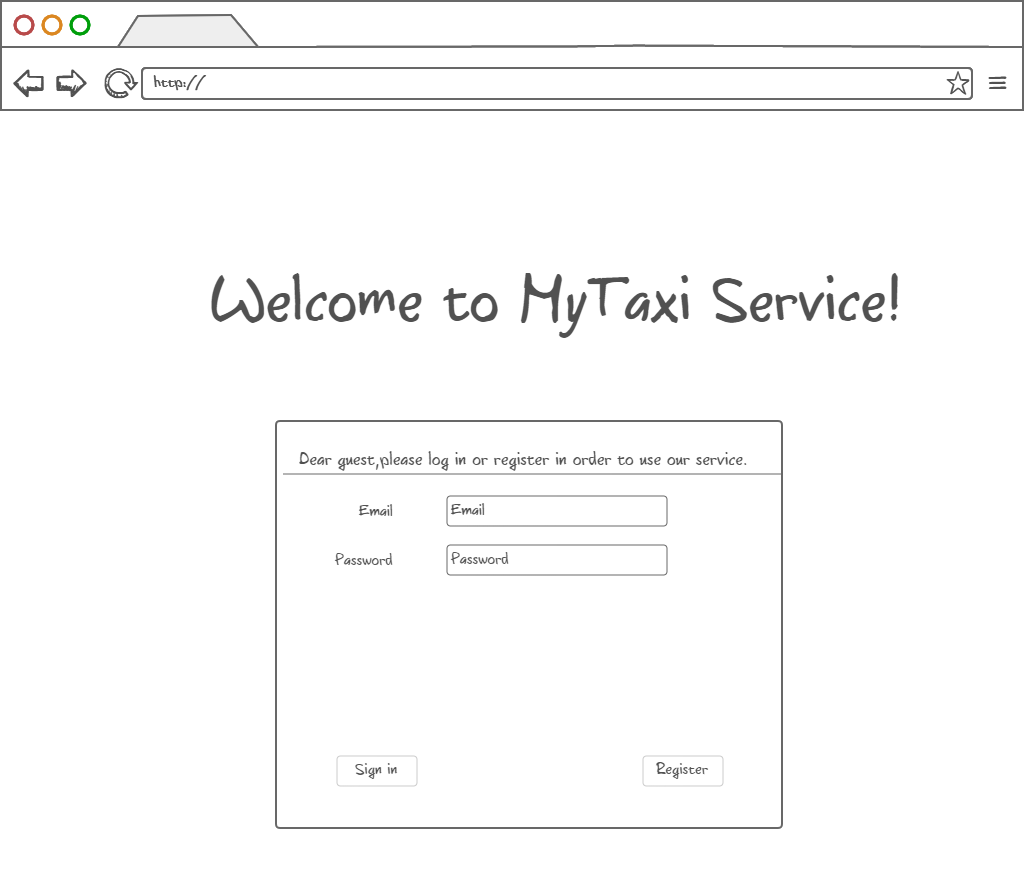


3.1.1.12 Waiting user response – Within 60 seconds, user who requested drive must accept or reject the drive offer and taxi driver has to wait (the left side). After accepting the offer, the similar screen is displayed to a taxi driver like in 3.1.1.9 (the right side). Driver can also report user the same way as in 3.1.1.9.

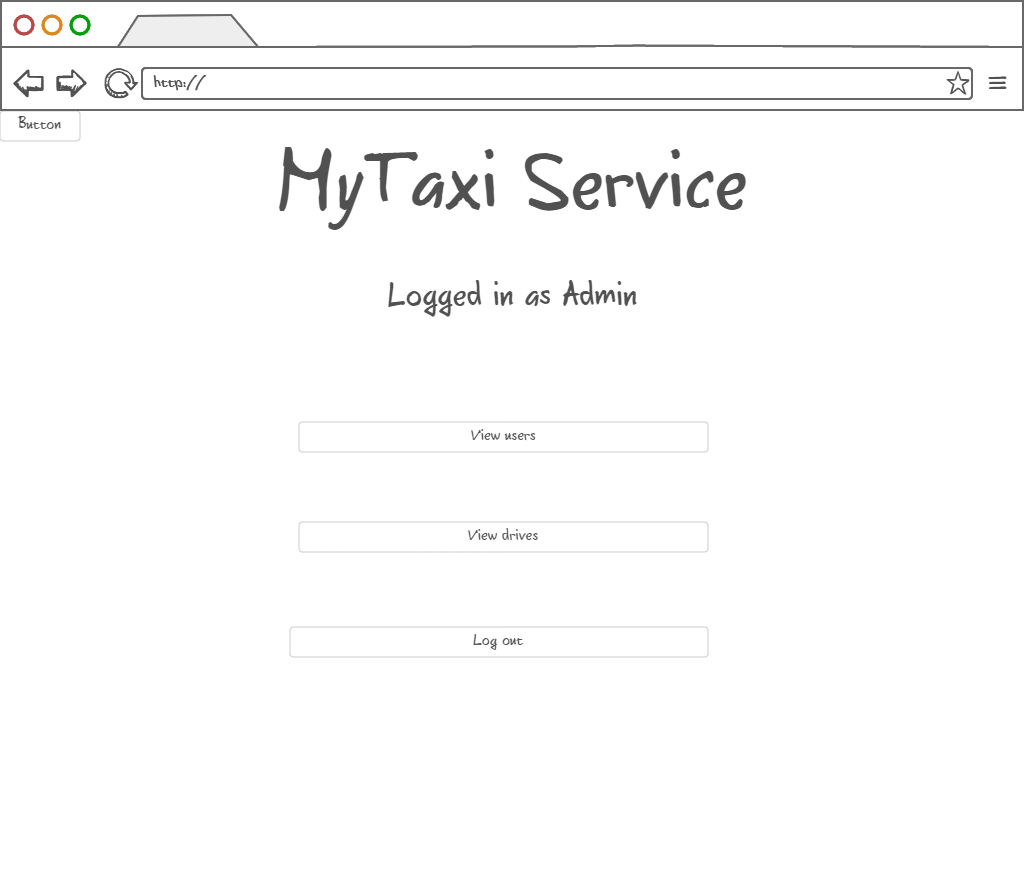


Administrator’s point of view

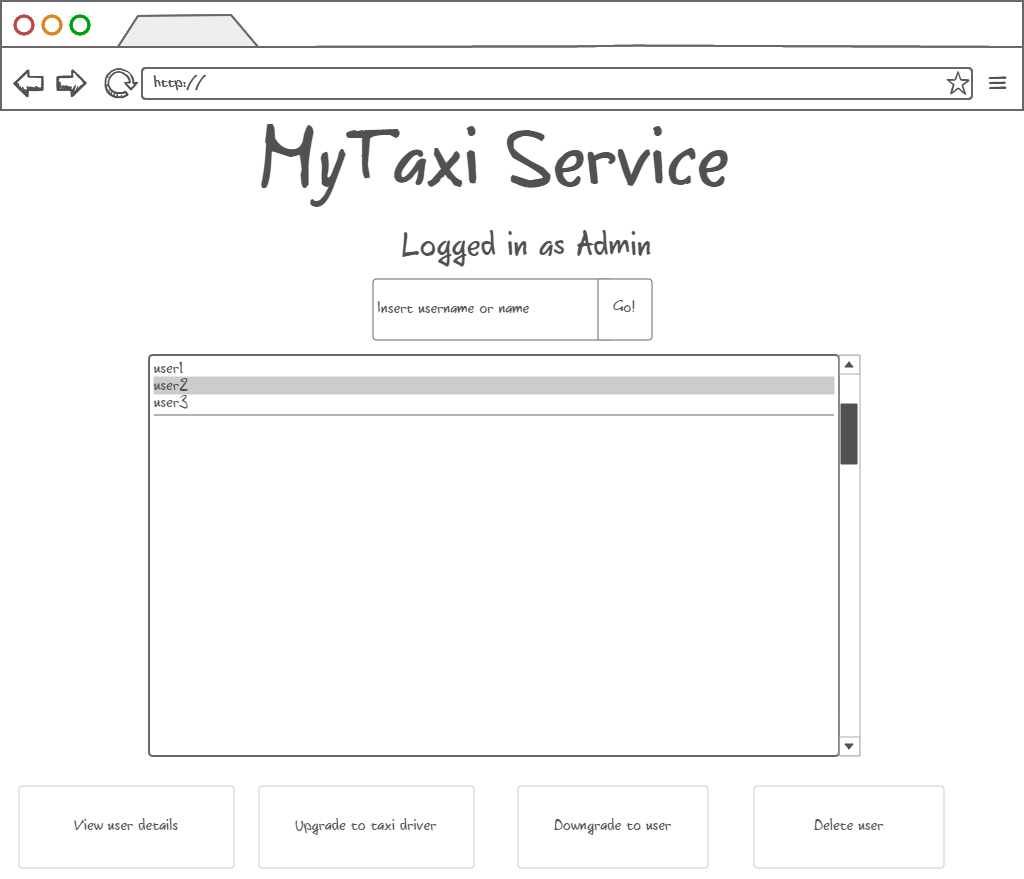
Administrator’s application is optimized for web browser, but will also have a mobile counterpart. In what follows, the web browser version is going to be presented.

3.1.1.13 Login page – In case of using the web browser version, user logs in by entering data in form like shown bellow.

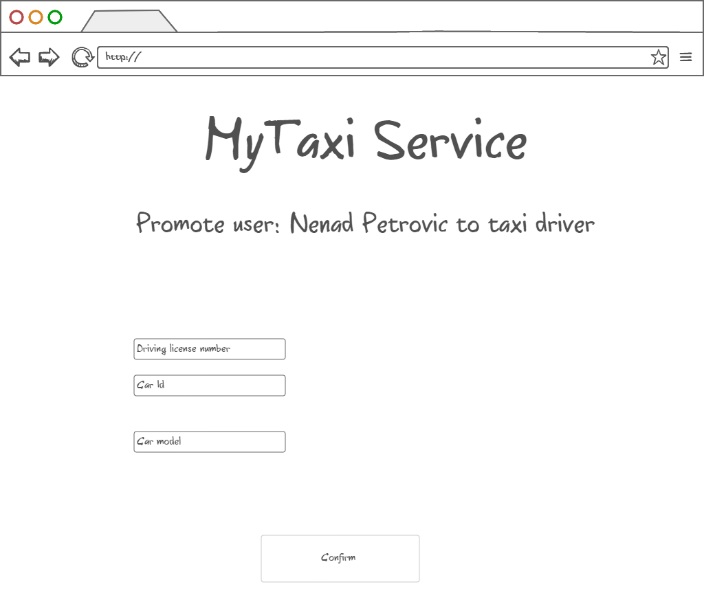
3.1.1.14 Administrator menu – Administrator can view users (both tzpical users and taxi drivers) and their reports in order to decide their removal from system or view drives. Of course, there is also a „Log out“ button.

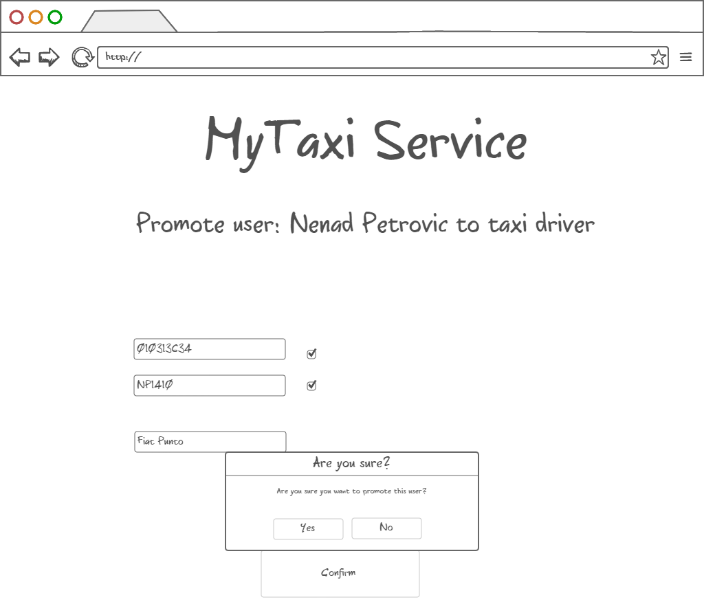


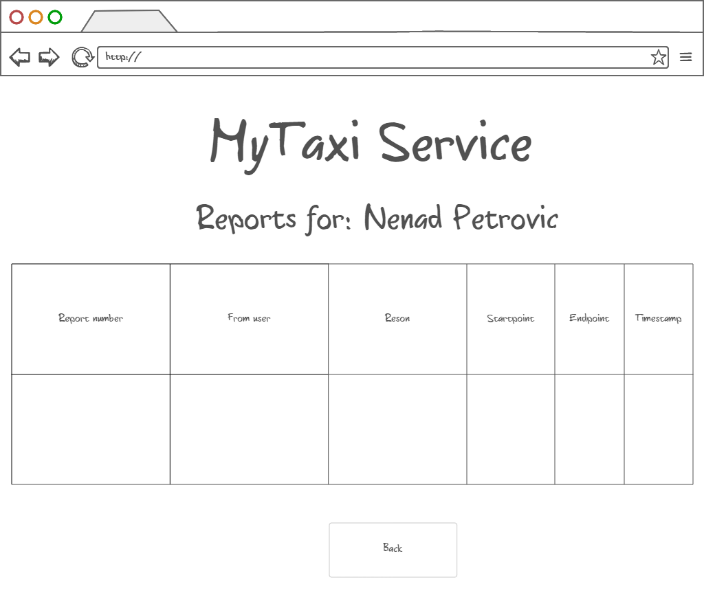
3.1.1.15 Browsing users – Administrator can view users and their reports in order to decide if they deserved removal from system, promote users to taxi drivers, downgrade taxi drivers to users and edit user’s data.

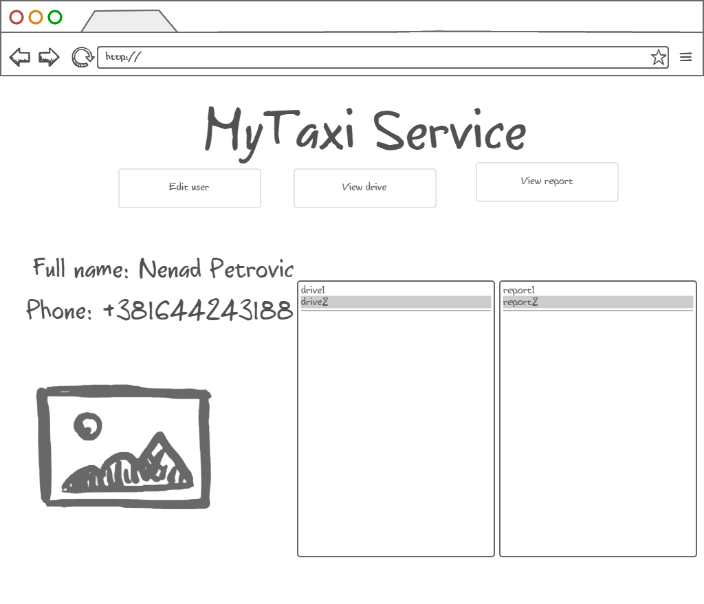


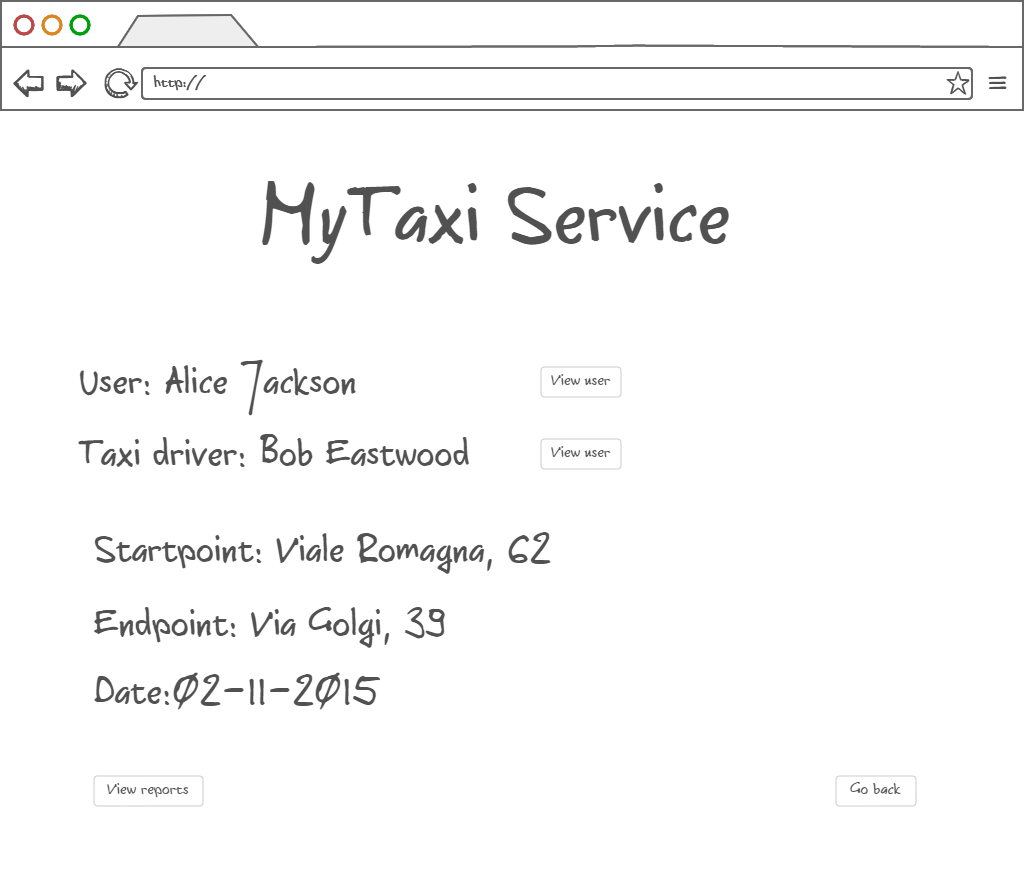
3.1.1.16 Promotion to taxi driver – After user selection, administrator can promote user to taxi driver by entering valid driving license number, car identification number and car model (optional). Data is checked by the system. If data is valid, promotion can be confirmed.



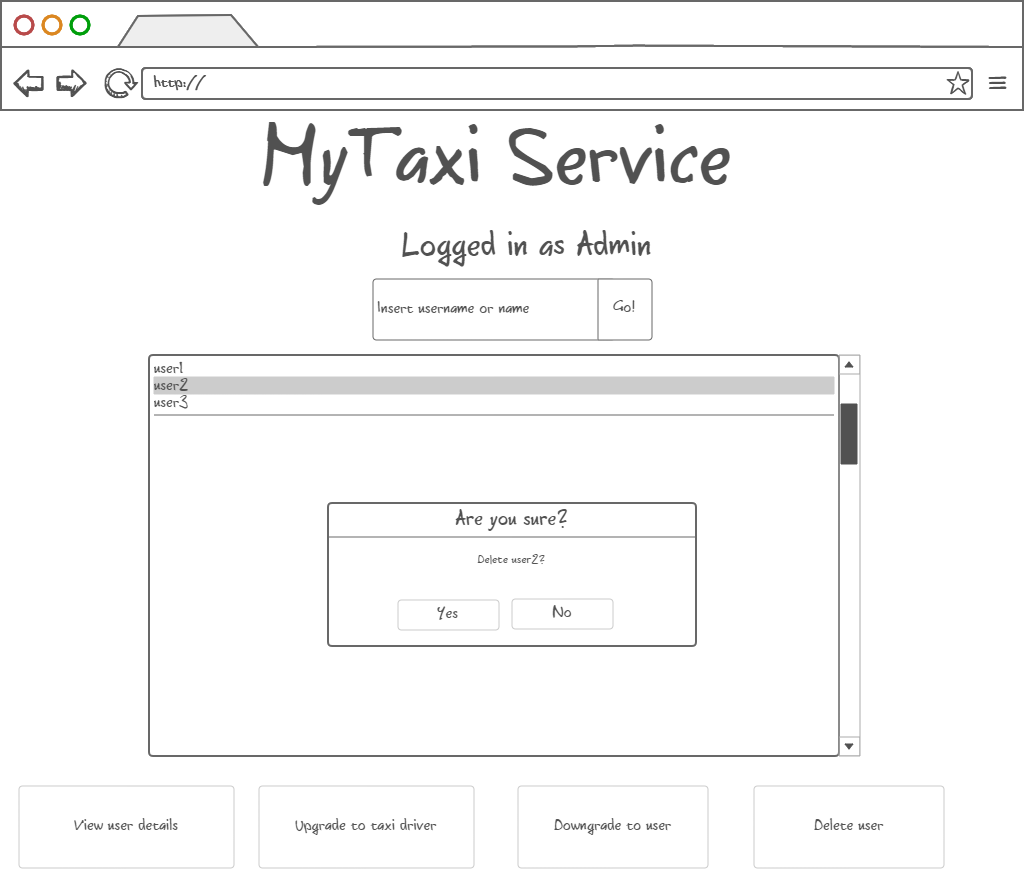


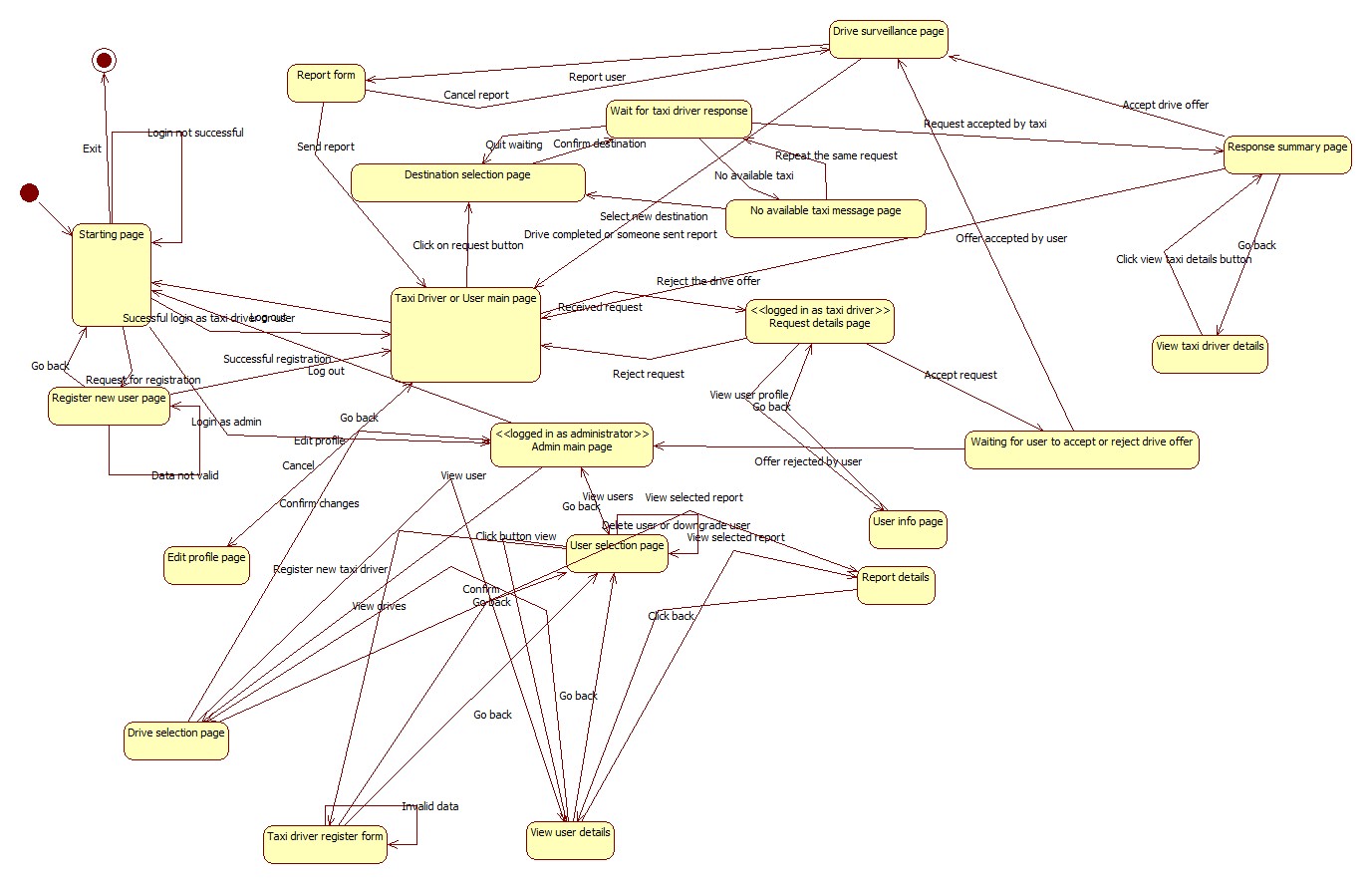
3.1.1.17 Viewing user info, reports and drive event – After user selection, administrator can view user details and all reports and drive events related to a particular user (left picture). It also possible that administrator wants to delete the user after reading the reports. The reports of selected user are shown in a form of table as on the right-side picture. Similar form appears when system administrator is notified about new report which needs an action.



Drive events related to selected user could also be viewed in a form of a short summary with option of viewing details of actors involved in the selected drive event.

3.1.1.17 Deleting selected user – After user selection, administrator can also delete the selected user if he/she deserved. The deleted users can’t login nor register anymore. Their fiscal code is blacklisted.



The following state machine diagram gives a simplified vision of the entire application. Together with mockups from 3.1.1, this diagram gives a complete overview of the user interface design.

3.1.2 API – Application Programming Interface

Software interface to other applications will be in a form of Application Programming Interface.

MyTaxiService will offer API that targets developers as possible users. Developers will have programmatic interface that will allow them to call MyTaxiService functions inside their application, so they can easily integrate with their apps and extend the extend the usage of MyTaxiService.

API would contain objects and functions that encapsulate MyTaxiService functionalities.

Developers will be able to connect their social network-alike applications and easily migrate data to register using already created profiles that have necessary information for registration. For example, in notation of Java-alike object – oriented pseudo-language, it would look like:

MyTaxiServiceObject.register(username: String, password:String, fiscalCode: String, phoneNumber: String, gender: Char, picturePath: String)

Developers would also have ability to request a taxi inside their application if they have a valid session (previously logged in):

MyTaxiSession session1=MyTaxiServiceObject.login(username: String, password:String)

Session1.RequestTaxi( )

Functions like this would give developers ability to make more complex systems using MyTaxiService functions as primitives.

# Requirements traceability

In this part, it is going to be considered how the requirements previously defined in RASD map to design decisions, in other words, how the requirements affect the software desgin.

Functional requirements

Registration/Taxi driver promotion data check – Considering the requirement that system must find out whether the provided fiscal code/car identification/driving license number is valid or not, and belongs to corresponding person, MyTaxiService needs to be able to commmunicate with some external database that could provide the necessary information. So, MyTaxiService communicates via REST service with external city government database which would check data validity related to citizens.

GPS coordinates determination and calculation –As it is necessary to use calculations based on GPS coordinates and current location, it is also needed to use external API here (outside of the application). When it comes to taxi zone determination – it is necessary to calculate distances between two locations, for example. When user sends a request, his/her current location is taken as a starting point.As it can be seen, it is necessary to have full API that deals with GPS coordinates and calculations between locations. As this is not implemented as a part of this application, extrnal API is used, such as Google Maps API.

Reports – One of the requirements is to give possibility that users can report each other in case of bad behvaiour. Administrator can later see the reports, read their reasons and decide to ban users or not. So, reports must be stored in database and there s REPORT table in database related to this system.

Blacklist – after ban, users are not able to log in, nor to register again using same fiscal code. So, it is necessary to keep track of blacklisted users. There are two ways, as it is going to be explained in database design part. They could be stored in a separate table (their fiscal code) or we can add additional attribute to each user which is set to 1 after ban („Blacklisted“ attribute).

Non-functional requirements

Maintainability – as it is necessary to estimate the cost of a drive from one place to another, it is necessary to use some kind of pricing policy. It is also needed to make algorithm as flexible as possible,because this policy could be related to only one city, and it could be changed during time. So, the algorithm implementation takes many factors into account – distance, time of the day, is it weekend or not, and is it a holiday or special day. These considerations are typical and commonj to any pricing policy. The algorithm has many parameters that could be easily changed, so it could be shaped according to pricing policy. Putting coefficient 0 means that the parameter is not taken into account, or putting negative parameter, it means that it decreases price (discounts). This is one of the non-functional requirements - maintainability requirement. It was needed to make easily changeable and flexible algorithm that could be customized according to pricing policy and changed in future. When this software is customized for another city, only algoirthm parameters are going to be changed, but not the algorihtm itself.

Security – as it has been told in RASD document, server side will use separated web, application and database part, with firewall between each of them to prevent unauthorized users from access. Taking this consideration into account, it is found obviuos that Java EE is solution that would be suitable for this kind of separation, as it offers the separate client, web, business and database tier, which is exactly what is needed. So, this will give ability to place firewalls between each two parts and make application secure and meet the security requirements.

Considering the other non-functional requirements, it could be said that decision to use Java EE reflects many of the non-functional requirements previously defined in RASD, because Java EE is, generally based on n-tier architectural style which brings many benefits when it comes to non-functional requirements.

Java Enterprise Edition is designed to develop large-scale and multi-tiered applications that are scalable and meet reliability conditions and are secure at the same time.

The main benefits of the N-tier architectural style are:

Maintainability - Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.

Scalability - Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.

Flexibility - Because each tier can be managed or scaled independently, flexibility is increased.

Availability - Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

So, the decision to use N-tier and Java EE implementation satisfies most of the non-functional requirements and is related to them.

Interface

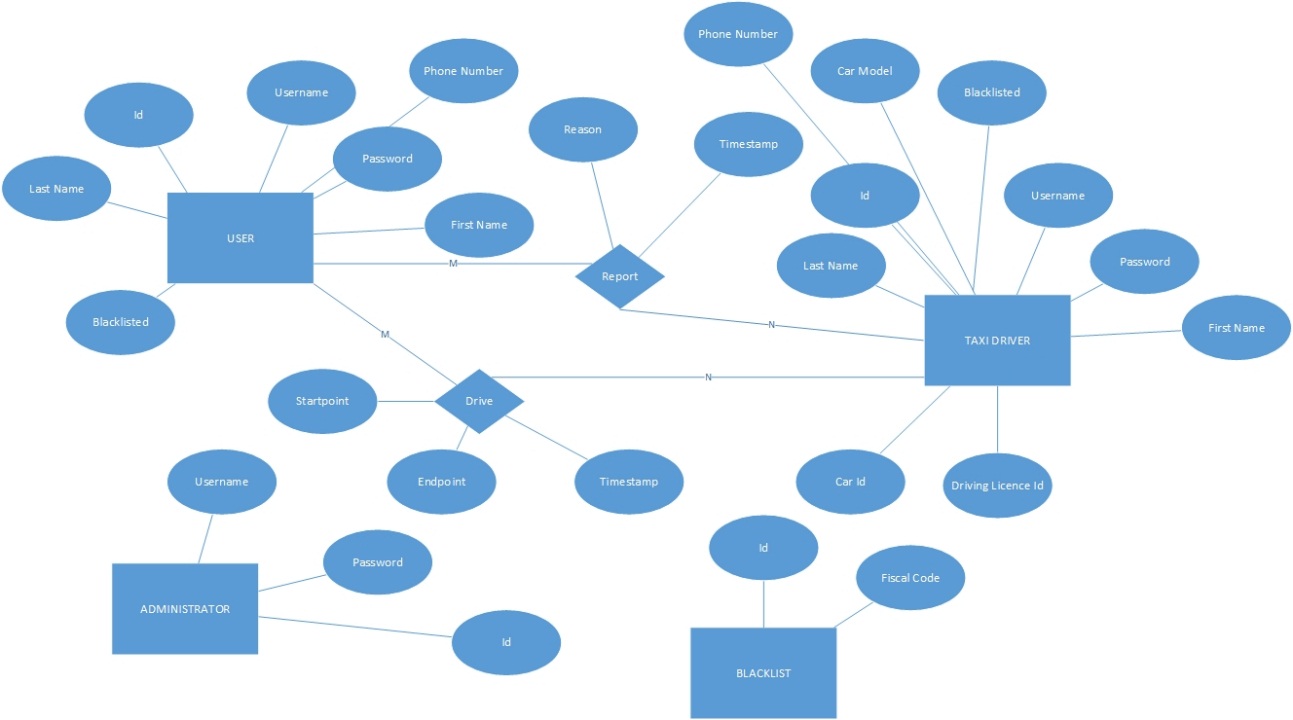
In the end, a few words about developer API and how it affected the design.

API for developers gives them ability to use any of the MyTaxiService functions in order to make their own application based on MyTaxiService operations as a primitives. So, it is decided to make full access to developers to MyTaxiService via REST service, but developers must be registered in city government office in order to get such privilegues.

# Database design

This system uses simple database that has several purposes:

* to store registered users and their information, which would be necessary in order to give them ability to log in and use the service
* to store drive events summary, which would give overview to the administrator
* to strore report, which would help administrators to bring decisions about deleteing users that show bad behaviour
* to keep track of users that are banned

In what follows, ER diagram of this simple database is presented:

We can see that it stores necessary information for user, taxi driver and administrator which would allow them to log in – username and password.

Taxi driver is different from user only in several attributes – it has all the attributes the users have, but also car model, car id and driving license id.

Reports are stored in database, so they can be later read by administrators. Report reson is necessary in bringing decsion to ban user or not, so it is stored as an attribute in this database.

Blacklist could be a separate table, where user fiscal codes are stored after their ban by admin. There is also alternative way – to store attribute „Blacklisted“ and set it to 1 after ban, and not delete the banned users from database.

Let’s take a look at a translation of this ER model into database:

User(Id,Username,Password,FirstName,LastName,PhoneNumber,Blacklisted,isTaxiDriver,CarModel,CarId,LicenseId)

Administrator(Id,Username,Password)

As it can be seen, taxi driver is not translated as a separate table. Attribute “isTaxiDriver” is added, instead. When this attribute is set to 1, attributes CarModel,CarId and LicenseId are considered valid. When Blacklist attribute is set to 1, user can’t log in or register again.

Report(Id,*Sender*,*Receiver*,Reason,Timestamp)

As it can be seen on ER diagram, we need M-N relationship, so Report is translated as a separate table with foreign keys corresponding to the sender of the report (the one who reports another user) and receiver of the report (user who is showing bad behaviour) . In fact, their Ids are used as a foreign keys.

Similarly, Dirive is translated:

Drive(Id,*User*,*Driver*,Startpoint,Endpoint,Timestamp)

Startpoint is the starting location, while endpoint is the desired destiantion. User is foreign key which corresponds to the user, while Driver is foreign key which corresponds to the Taxi Driver participating in this drive event.

# Software and tools used

* Microsoft Office Word 2013: To create and redact this document
* StarUML: to create component view, runtime view, deployment view, sequence diagrams in runtime view.
* Edraw Max 7.9: to draw component interfaces
* Microsoft Visio 2016: to draw high-level architecture, algorithms and database design.
* NinjaMock [(https://ninjamock.com/)](http://balsamiq.com/products/mockups/): to create mockups for both mobile and web variants of the application.
* Github (reporsitory: https://github.com/penenadpi/Software-Engineering-2-Project/Deliveries)

## Hours of works

The time spent for constructing this document and Alloy model:

* Nenad Petrovic: ~ 50 hours.

# References

1. Software Design and Software Architecture (Softwaree Engineering 2 course, Politecnico di Milano)-6. Design I –II.pdf
2. Jave EE7 introduction (Software Engineering 2 course, Politecnico di Milano)- Jave EE.pdf
3. Java API for restful services, <https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services>
4. Java Persistence API, <https://en.wikipedia.org/wiki/Java_Persistence_API>
5. Java Enterprise Beans, <http://docs.oracle.com/javaee/5/tutorial/doc/bnblt.html>
6. Java Server Faces, https://en.wikipedia.org/wiki/JavaServer\_Faces

1. http://docs.oracle.com/javaee/5/tutorial/doc/bnblt.html [↑](#footnote-ref-1)
2. I assume that Unit Test Plan is already done, because UTPD (Unit Test Plan Document) was not previously developed as an assignment or part of any assignment. [↑](#footnote-ref-2)
3. REST (REpresentational State Transfer) is an architectural style, and an approach to communications that is often used in the development of [Web services](http://searchsoa.techtarget.com/definition/Web-Services-Glossary).  [↑](#footnote-ref-3)
4. JAX-RS: Java API for RESTful Web Services (JAX-RS) is a [Java programming language](https://en.wikipedia.org/wiki/Java_(programming_language)) [API](https://en.wikipedia.org/wiki/Application_programming_interface) that provides support in creating [web services](https://en.wikipedia.org/wiki/Web_service) according to the [Representational State Transfer](https://en.wikipedia.org/wiki/Representational_State_Transfer) (REST) architectural pattern. [↑](#footnote-ref-4)
5. HTTPS ( HTTP over SSL, and HTTP Secure) is a [protocol](https://en.wikipedia.org/wiki/Communications_protocol) for [secure](https://en.wikipedia.org/wiki/Network_security) communication over a [computer network](https://en.wikipedia.org/wiki/Computer_network) which is widely used on the [Internet](https://en.wikipedia.org/wiki/Internet). HTTPS consists of communication over [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP) within a connection encrypted by [Transport Layer Security or its predecessor, Secure Sockets Layer](https://en.wikipedia.org/wiki/Transport_Layer_Security). The main motivation for HTTPS is [authentication](https://en.wikipedia.org/wiki/Authentication) of the visited [website](https://en.wikipedia.org/wiki/Website) and protection of the [privacy](https://en.wikipedia.org/wiki/Information_privacy) and [integrity](https://en.wikipedia.org/wiki/Data_integrity) of the exchanged data [↑](#footnote-ref-5)
6. JPA:The Java Persistence API (JPA) is a  [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) specification that describes the management of [relational data](https://en.wikipedia.org/wiki/Relational_data_model) in applications using [Java Platform, Standard Edition](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition) and [Java Platform, Enterprise Edition](https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition). [↑](#footnote-ref-6)