Politecnico di Milano

A.A. 2015-2016

Software Engineering 2: “MyTaxiService”

Design Document

Version 0.2

Ivana Salerno

Alexis Rougnant

Daniel Vacca

February 2nd 2016

Table of contents

List of tables

List of figures

1. Introduction

1.1. Purpose

This software design document describes the architecture and system design of MyTaxiService.

1.2. Scope

This document contains a complete description of the design of MyTaxiService model.
The basic architecture is a web server characterized by different architectural styles and patterns: Model-View-Controller, Multi-tier, Distributed Objects, Publish-Subscribe, Service Oriented Architecture.
As concerns the language or implementation frameworks that shall be used we assume that such choices will be taken in later iterations.

1.3. Definitions, acronyms and abbreviations

In this document we do not include additional terms that refer to the domain of the problem, with respect to those presented in the RASD. However we do use some technical language. The most relevant definitions are given here.

- Distributed objects style: is an architectural style which follows the object oriented paradigm to design the components and their connections; the difference is that the objects are distributed in different runtime environments. Additional components to deal with the remote invocation of methods (and communication outside the runtime environment) are needed.
- Façade pattern: is a design pattern which suggests the creation of a class to encapsulate the operations provided by a set of classes which conform a component. This helps to simplify the interface to access the component, to hide the coupling of the component, or to define an entry point to the component.
- Multi-tier style: is an architectural style that can be seen as an extension of the Client-server style. The system can be deployed among more than two nodes, so the components will have to communicate through the network.
- MVC style: is an architectural style which suggests to organize the system components into three main sections, each one of them concerning a specific function. The model section will deal with the business logic, the view will deal with the graphic representation of data, and the controller will communicate the view and the model when necessary (to invoke business operations or update graphic data from the model).
- MTS: stands for MyTaxiService. This prefix is recurrently used along the document.
- Proxy pattern: is a design pattern which suggests the creation of a class to act as another class for some parts of the system. It is used when it is not convenient (or possible) to have instances of the represented class accessed from certain classes. The most usual scenario is the remote method invocation.

- Publish-subscribe style: is an event-based architectural style. Some components (publishers) generate events that are of the interest of other components (subscribers). The *subscribers* (as suggested by the name) subscribe to specific events, so that they are notified when such events happen; *publishers* publish the event information whenever they occur.
- Service oriented architecture: is an architectural style in which the system works by using web services. Most of the components are external applications which provide operations through a web service; the system will use such services to perform its work, with some intermediate processing of the results.

1.4. Reference documents

The following documents present information which is relevant to understand this Design document:

- MyTaxiService project AA 2015-2016 description.
- RASD for MyTaxiService, by Ivana Salerno, Alexis Rougnant and Daniel Vacca.

1.5. Document structure

The document is composed by 5 sections:

- Section 2: Architectural design, style and patterns
  In this section is described the design of the document, high level components and how they interact between them (through UML diagrams). Then there is a deep description of the components through three points of view: component, deployment and runtime. In the end of this section are described architectural styles and patterns which guides the architecture and how they have been adopted within this context.
- Section 3: Algorithm design
  In this section are described the algorithms used to manage different parts of the system.
- Section 4: User interface design
  In this section are showed and described the interfaces the user (both passenger and taxi driver) effectively sees through mobile application and web site.
- Section 5: Requirements traceability
  In this section is showed how the requirements of the RASD are related to the architecture.
- Section 6: References

2. Architectural design

2.1. Overview

This section presents the architectural design of the MyTaxiService software. We first present a high level description of the components. These are followed by further explanations of the components from three different perspectives (component, deployment and runtime). Finally we expose the design decisions, some of which refer to the selected architectural and styles patterns.

It is worth to mention that this document refers to a logical architecture of the system, which is platform and specific technologies independent. This means that we have not made any decisions on the language or implementation frameworks (e. g. Java EE, PHP, Ruby) that shall be used; we assume that such choices will be taken in later iterations. However, some of the expected behaviors of the implementing platforms are provided.

2.2. High level components and their interaction

The following diagram presents the main components of the system at a high level view. It only shows their names and the other related components and actors (either users or systems). In the *Component view* we refine this model and add deeper explanations.

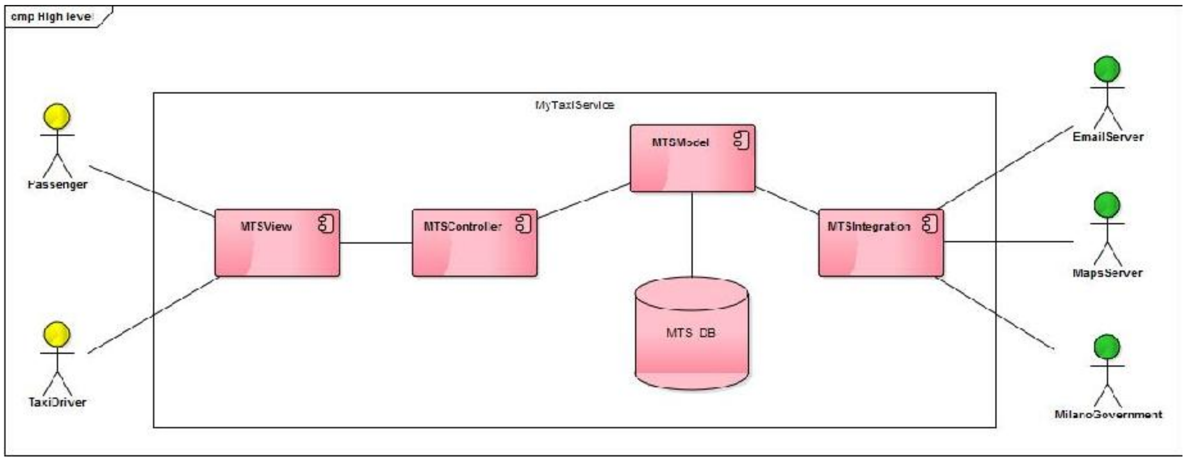


Figure 1: High level components

The yellow actors are the users of MyTaxiService:

- **Passenger:** makes use of the MyTaxiService to make a request for a taxi service. He can communicate with the system either through the web site or the mobile application by interacting with the appropriate subcomponents of the MTSView (this will be shown in the *Component view*).
- **TaxiDriver:** makes use of the MyTaxiService to attend requests for a taxi service. He can communicate with the system through the mobile application by interacting with the appropriate subcomponent of the MTSView.

The pink boxes represent software components that define MyTaxiService. Note that the use of the MVC architectural pattern is visible:

- **MTSView:** this component is in charge of displaying graphical components to allow both the Passenger and the TaxiDriver to interact with the system. It will communicate such interactions to the MTSController. In the *Component view* we will see that this component includes subcomponents that will support the access of the users from the different devices (computer and mobile phones).
- **MTSController:** this component is in charge of receiving the events produced by the interactions between the users and the software, and performing the corresponding invocations on the MTSModel to execute the related tasks.
- **MTSModel:** this component is in charge of receiving invocations from the MTSController and execute the appropriate operations. It implements the business logic. Some operations might involve accessing to the MTS_DB to retrieve persistent data, or to the MTSIntegration to communicate with external systems.
- **MTS_DB:** represents the Database Management System that will store the persistent data. This document does not include a Persistent view, but we expect this component to store the following information:
    - The data of the passengers, such as name, email address, password, and whether their email address has been confirmed.
    - The data of the taxi drivers, such as name, email address, username, password, taxi code, and taxi capacity.
- **MTSIntegration:** this component is in charge of communicating with the external systems of the domain (EmailServer, MapsServer, MilanoGovernment), as consequence of requests of the MTSModel.

The green actors are the external systems that MTS will interact with (these are already implemented):

- **EmailServer:** this system will be used by the MTSModel (through the MTSIntegration) to send messages to the users as described in the *External system interfaces* section of the RASD (section 2.1.2).
- **MapsServer:** this system will be used by the MTSModel (though the MTSIntegration) to perform location analysis tasks as described in the *External system interfaces* section of the RASD (section 2.1.2).
- **MilanoGovernment:** this system will be used by the MTSModel (though the MTSIntegration) to perform data validation tasks as described in the *External system interfaces* section of the RASD (section 2.1.2).

2.3. Component view

This section provides further details on the components defined in the previous section. We have identified subcomponents aiming to the *Deployment view* and the two different users, which will also be more detailed as they are explained in this subsection (by means of class diagrams).

The data types used to describe some classes should be understood as language independent; using names or notations which are recurrent in some languages does not mean that we have decided to use them. This is also valid for collections of objects; these do not necessarily have to be Lists. More appropriate data structures can be derived in more detailed design documents.

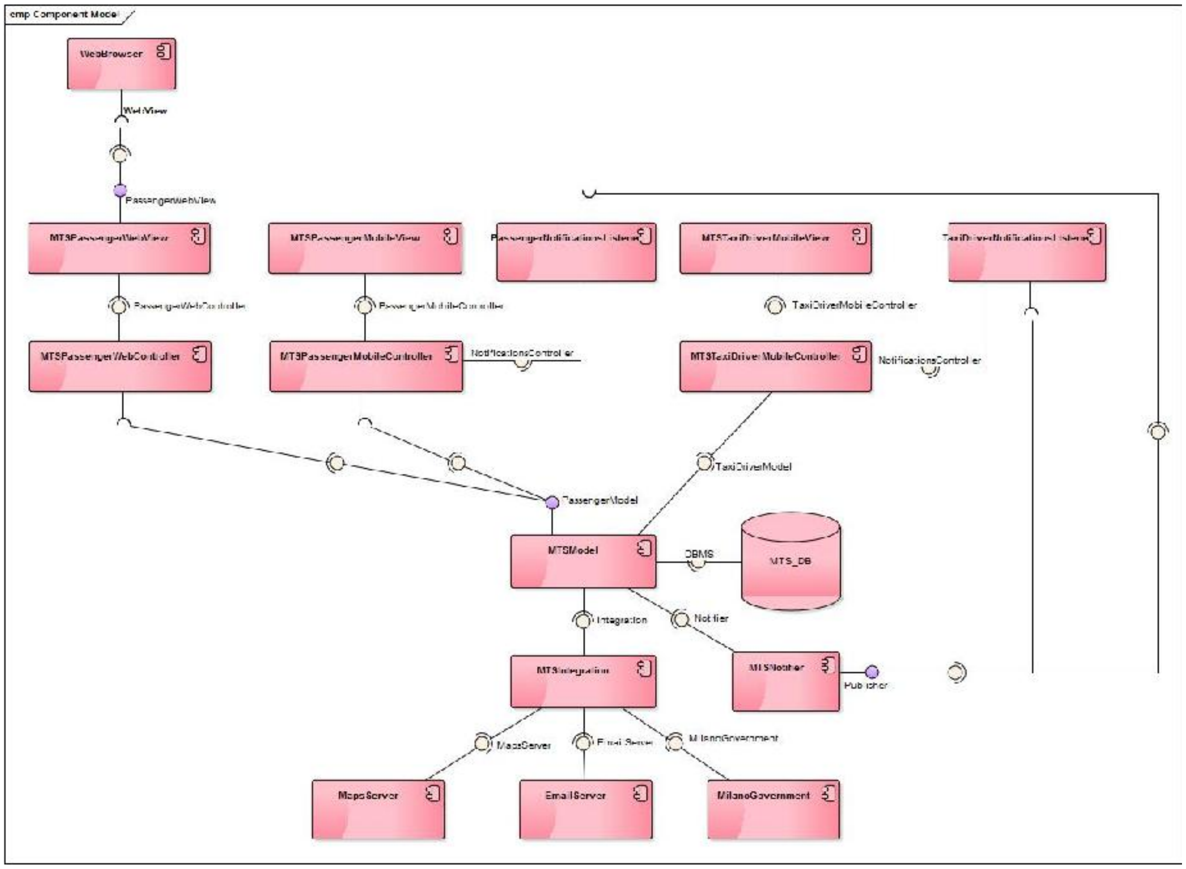


Figure 2: Components view

In the following subsections we expose the descriptions of the components in the diagram above. We only give their definition since the interfaces are defined later in the document. The components are grouped by the high level component that they refine.

2.3.1. MTSView and MTSController

Because of the adoption of a MVC architecture, we do not include class diagrams for the description of this subcomponents.

For the MTSView, such classes would only represent implementation views of the information in the *User interface design*. We list instead the events that are produced by the interactions of the users with the graphical components. The sequence diagrams in the *Runtime view* put together this events with the other components.

For the MTSController we list instead the methods that are invoked as a result of the interpretation of the interactions of the users with the graphical components. The sequence diagrams in the *Runtime view* put together this interpretations with the other components.

Users can interact with the system through the views. In particular MTSPassengerWebView (using website) and MTSPassengerMobileView (using mobile application) permit the passenger to have an interaction with the system and MTSTaxiDriverMobileView permits taxi driver to have an interaction with the system. In both cases this interaction takes place through the objects contained in User Interface Design.

So the views manage the actions the user makes in particular through buttons, links and icons which refer to methods.

All the information acquired from the interfaces is sent to the respective controller (MTSPassengerWebController, MTSPassengerMobileController or MTSTaxiDriverMobileView) that transmits it to MTSModel.

Hereafter is showed to which method of the Model each event of the interfaces refer

Image 6.1

loginButtonPushed() $\rightarrow$ login(String, String): Passenger, login(String, String): TaxiDriver

createPassengerLinkClicked()

createDriverLinkClicked()

Image 6.2

subscribeButtonPushed() $\rightarrow$ createAccount(Passenger): Passenger

Image 6.3

makeRequestButtonPushed()

pendingRequestsButtonPushed()

editProfileLinkClicked()

notificationIconPushed()

Image 6.4

saveButtonPushed() $\rightarrow$ editAccount(Passenger): Passenger cancelButtonPushed()

editPictureLinkClicked()

notificationIconPushed()

Image 6.5

cancelButtonPushed()

submitButtonPushed() $\longrightarrow$ receiveRequest(RequestForService): boolean

Image 6.6

cancelRequestButtonPushed() $\longrightarrow$ cancelRequest(String): boolean

mapButtonPushed() $\longrightarrow$ getMap(GPSPosition): Map

arrowIconPushed()

Image 6.7

arrowIconPushed()

Image 6.8

subscribeButtonPushed() $\longrightarrow$ createAccount(TaxiDriver): TaxiDriver

Image 6.9

changeAvailabilityButtonPushed()

acceptedRequestsButtonPushed()

pendingRequestsButtonPushed()

mapButtonPushed() $\longrightarrow$ getMap(GPSPosition): Map

editProfileLinkClicked()

notificationIconPushed()

Image 6.10

arrowIconPushed()

Image 6.11

cancelButtonPushed()

saveButtonPushed() $\longrightarrow$ setAvailability(boolean, String): void

Image 6.12

cancelRequestButtonPushed() $\rightarrow$ cancelRequest(String): void

arrowIconPushed()

Image 6.13

acceptRequestButtonPushed() $\rightarrow$ answerRequest(String, boolean): void

ignoreRequestButtonPushed() $\rightarrow$ answerRequest(String, boolean): void

arrowIconPushed()

Image 6.14

cancelButtonPushed()

saveButtonPushed() $\rightarrow$ editAccount(int, String, String, String): void

editPictureLinkClicked()

notificationIconPushed()

2.3.2. MTSModel

This component has only one subcomponent with both the same name and function as its parent.

The following is the diagram of the classes encapsulated by this subcomponent.

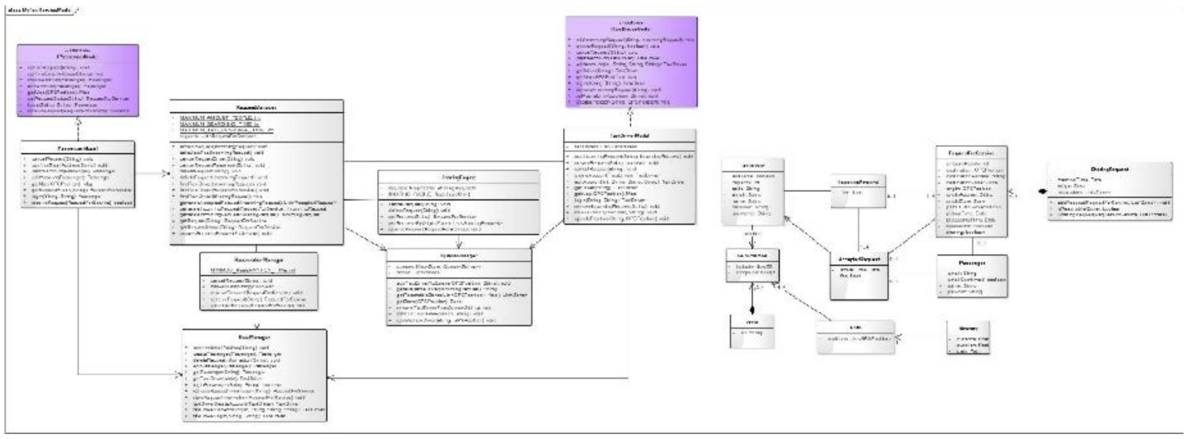


Figure 3: MTSModel

- **IPassengerModel:** this interface represents the operations that the MTSModel provides to the Passenger:
    - o Cancel a request, given the email address of the passenger (which uniquely identifies the request). If necessary, the involved TaxiDriver and Passengers are notified. The corresponding requests are deleted.

  - Confirm the email address, given the email address of the passenger. By doing this, the Passenger will be able to successfully login to the system.
  - Create an account, given the information of the Passenger. Returns the created Passenger object if the creation is successful.
  - Edit an account, given the new information of the Passenger. Returns the edited Passenger object if the edition is successful.
  - Get a map, given a GPSLocation. Returns the Map that the MapsServer provides for that location.
  - Get the status of a request, given the email address of the passenger. Returns the RequestForService object, associated to its corresponding AcceptedRequest if any.
  - Login to the system, given the email address and the password of the passenger. Returns the existing Passenger object if the logins is successful.
  - Receive a request for service, given the corresponding information. Returns true it is successfully received and false otherwise.
- **PassengerModel:** is a concrete implementation of the IPassengerModel interface. Thus, it will implement the inherited methods.
- **ItaxiDriverModel:** this interface represents the operations that the MTSModel provides to the TaxiDriver
  - Add an incoming request: given the code of a taxi and an incoming request, it add this incoming request to the taxi driver. Then the taxi driver is notified.
  - Answer a request: Assume a taxi driver as received an incoming request. Given the code of this taxi driver and a boolean that represents the answer of the request it will start the process of answering that is depends on the value of the answer.
  - Cancel a request: Assume a taxi driver as already accepted a request. Given the code of this taxi driver, the related requests (incoming request, accepted requests, requests for service) will be delete and the taxi driver and the passengers notified
  - Create an account: Given the code of a taxi driver, it will ask the Milano government and return an instance of taxi driver if the code is correct.
  - Edit an account: Given the code of a taxi driver and some information such as a new password, a new capacity, a new email address and in the case the provided email address is valid, it will modify the information in the data base and return the updated instance of the taxi driver
  - Get a map: Given a GPS position, return the map the map server provide for that location
  - Log in to the system: Given a taxi driver code and a password, return the created instance of taxi driver if the login information are validation
  - Remove an incoming request: given the code of a taxi, it removes this incoming request from the taxi driver.

  - Set the availability status: Given a taxi driver code and a boolean representing his new availability, it will modify this attribute and update the taxi driver position in the zone queues.
  - Update the GPS position : Given a taxi driver code and a GPSPosition provided by the hardware, it will modify this attribute and update the taxi driver position in the zone queues
- **TaxiDriverModel:** is a concrete implementation of the ItaxiDriverModel interface. Thus, it will implement the inherited methods.
- **RequestManager:** this class is supposed to receive valid RequestForService and process them. It keeps the collection only of the RequestForService that have been sent to a taxi driver (either the response has or not been received). In order to do so, it offers the following operations:
  - Receive and accepting response from a TaxiDriver, given the IncomingRequest that was sent and accepted to him, and which is associated to the corresponding RequestForService. The involved Passengers are notified.
  - Calculate the fee of a request, given the IncomingRequest that has the information of the ride in its RequesForService. The specific procedure of this calculation is shown in the *Algorithm design* section.
  - Cancel a request when done by a Passenger, given his email address. If necessary, the involved TaxiDriver and Passengers are notified. The corresponding requests are deleted.
  - Receive a declining response from a TaxiDriver, given the declined IncomingRequest.
  - Delete a request: given an incoming request, it will delete it and all the request and accepted request related as well.
  - Delete a request, given the email address of the passenger who owns it.
  - Find a taxi driver, given the IncomingRequest. Once found, the IncomingRequest will be sent.
  - Find a taxi driver, given the RequestForService. It will also generate the IncomingRequest and send it to the driver.
  - Find a taxi driver, given the SharingRequest. It will also generate the IncomingRequest and send it to the driver.
  - Generate an IncomingRequest, given the RequestForService of the ride. Returns the generated IncomingRequest, ready to be sent to a TaxiDriver.
  - Generate an IncomingRequest, given the SharingRequest of the ride. Returns the generated IncomingRequest, ready to be sent to a TaxiDriver.
  - Get the information of a request, given the email address of the related Passenger. Returns the corresponding RequestForService.
  - Process a request, given the RequestForService information. This means to generate the appropriate IncomingRequest and to try to find a TaxiDriver to whom send the request.

   - Process a request, given the SharingRequest information. This will dispatch the request to the pertinent class (SharingEngine or ReservationManager) or will process it if it is ready to be processed.
- **ReservationManager:** this class is supposed to receive the reservation requests for later processing. In order to do so, it provides the following operations:
   - Cancel a request, given the email address of the Passenger. This means to delete the request (since it has not been processed yet).
   - Check if a request is the reservation list, given the email address of the Passenger. Returns true if it is in there, and false otherwise.
   - Receive a request to book, given the RequestForService. This means to schedule the processing and store the request in the MTS_DB.
   - Retrieve a request, given the email address of the Passenger. It returns the corresponding RequestForService.
   - Schedule a request for later processing, given the RequestForService.
- **SharingEngine:** this class is supposed to receive the sharing requests and try to find compatible ones. This class keeps a list of SharingRequests that have not been completed yet. In order to do so, it provides the following operations:
   - Cancel a request, given the email address of the Passenger. This will delete the request from the associated SharingRequest.
   - Delete a request, given the email address of the Passenger.
   - Get a request, given the email address of the Passenger. This returns the instance of the RequestForService.
   - Get the list of the SharingRequest whose origin is the given Zone.
   - Receive a request, given the RequestForService. This will try to associate the request to a compatible of SharingRequest, or create a new one for it if none is found.
- **QueueManager:** this class is supposed to manage the zone queues of the TaxiDrivers. It keeps a collection of the zones and of the queues of the drivers' codes for each one of them. In order to do so, it provides the following operations:
   - Add a taxi driver to a queue, given the GPSPosition and the code of the TaxiDriver. This means to look for the appropriate zone for the driver and add it to the associated queue.
   - Get an available driver for a request, given the IncomingRequest. This is done based on the origin zone and the amount of people.
   - Get reachable zones from a path, given the path (as a collection of GPSPositions) and the radius. This returns the zones that can be reached from the path at a distance equals to the radius. This is used by the SharingEngine, as it will be described in the *Algorithm design.* It returns a collection of Zones.
   - Get a zone, given the GPSPosition. This returns the Zone for that Position.
   - Remove a driver from a queue, given the code of the TaxiDriver.
   - Send a driver to the top of the queue, given his code and GPSPosition.
   - Update a driver's position, given his code and GPSPosition.

- **DataManager:** this class is supposed to decouple the access to the MTS_DB from the rest of the components. This class will provide operations on the data that will be translated into queries and sent to the MTS_DB. In order to do so, it provides the following operations
  - Confirm the email address of the passenger, given the email address. It will change the corresponding attributes in the MTS_DB.
  - Add a passenger to the system, given the Passenger. It returns the created instance of the Passenger
  - Delete a request, given the email address of the Passenger.
  - Edit a passenger in the system, given the Passenger. It returns the edited instance of the Passenger.
  - Get the information of a passenger, given his email address. It returns the instance of the Passenger.
  - Check if the login information of a passenger is correct, given the email address and password.
  - Retrieve the information of a request, given the email address of the Passenger. It returns the corresponding RequestForService.
  - Store the information of a request, given the RequestForService.
  - Create an account for the taxi drive. This method stores the information concerning the instance of a taxi driver provided as argument (it is supposed to be send by the Milano government) then return this instance.
  - Edit the account of the taxi driver. This method changes the information given in argument concerning the taxi driver related to the code given: the capacity, the email address if it is valid and the password. Then it returns the updated instance of this TaxiDriver.
  - Login a taxi driver. Given a taxi code and a password, if they are valid, it returns the instance of the related TaxiDriver.
- **Passenger:** it holds the information of a passenger: name, email address, password, and whether its email has been confirmed or not.
- **RequestForService:** it holds the information of a request sent by a passenger: amount of people, destination and origin (in the form of GPSPosition, address and zone), estimated path from the origin to the destination, pick-up time, start of processing time (in order to apply the timeouts), whether it is a reservation or not, and whether it is a sharing or not. Note that not all of this data is provided by the Passenger. The origin and destination zones are obtained through the QueueManager, the origin and destination positions and the estimated path through the MapsServer (when not indicated by the passenger), the processing time is written by the RequestManager.
- **IncomingRequest:** holds the information of an incoming request that can be sent to a taxi driver. It includes the fee but has also access to the origin and destination through the RequestForService.

- **AcceptedRequest:** holds the information of an accepted request for a passenger: the estimated arrival time (calculated with the MapsServer), the fee of that passenger, and the relevant information of the TaxiDriver (name, taxi code, position).
- **SharingRequest:** holds the information of a sharing request: time of the creation of the instance (for applying the timeouts), origin zone, and the collection of reachable zones (calculated by the QueueManager). It puts together a set of compatible RequestForService.
- **TaxiDriver:** It is the class that represents the taxi drivers, it owns different attribute: available, capacity, code, email, name, password, username, GPS position, incoming request. Instances of this class are created through a method belonging to TaxiDriverModel if it is called by the taxi driver interface. It is possible to edit some attributes, add and remove an incoming request and update the GPS position threw the interface.
- **GPSPosition:** It represents the information of a spatial position with two attributes: the latitude and the longitude.
- **Zone:** It represents an area of the city. It is defined by a set of three or more GPS positions that are the angles of a polygon. Each zone is identified by its ID.
- **Path:** It is a set of GPS position that define a path through the city roads.
- **Itinerary**: It owns a path, a distance as a float and a duration as a float as well.

2.3.3. MTS_DB

This component has only one subcomponent with both the same name and function as its parent.

2.3.4. MTSIntegration

This component has only one subcomponent with both the same name and function as its parent.

The following is the diagram of the classes encapsulated by this subcomponent.

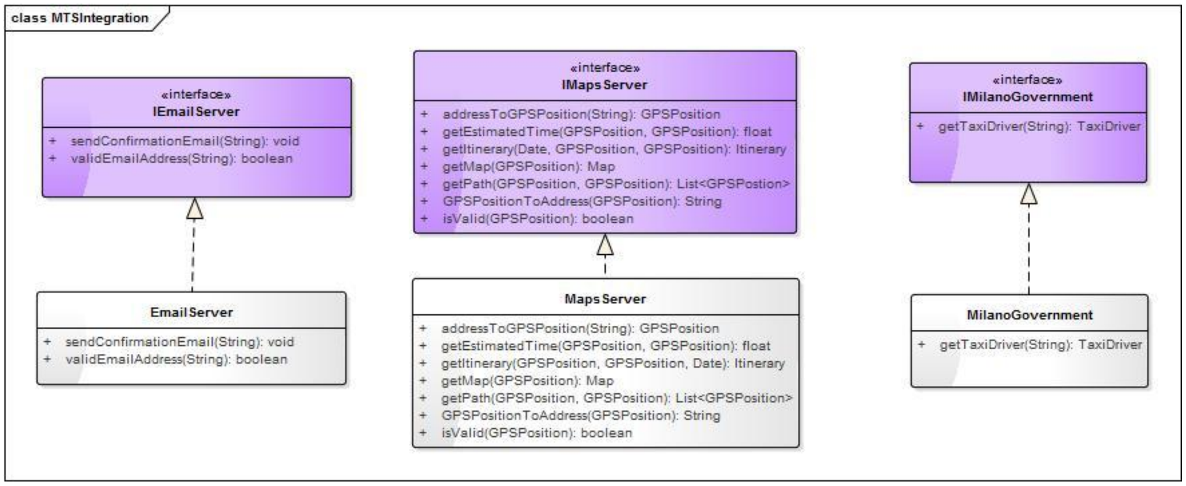

class MTSIntegration
«interface»
IEmailServer
+ sendConfirmationEmail(String): void
+ validEmailAddress(String): boolean
EmailServer
+ sendConfirmationEmail(String): void
+ validEmailAddress(String): boolean
«interface»
IMapsServer
+ addressToGPSPosition(String): GPSPosition
+ getEstimatedTime(GPSPosition, GPSPosition): float
+ getItinerary(Date, GPSPosition, GPSPosition): Itinerary
+ getMap(GPSPosition): Map
+ getPath(GPSPosition, GPSPosition): List<GPSPostion>
+ GPSPositionToAddress(GPSPosition): String
+ isValid(GPSPosition): boolean
MapsServer
+ addressToGPSPosition(String): GPSPosition
+ getEstimatedTime(GPSPosition, GPSPosition): float
+ getItinerary(GPSPosition, GPSPosition, Date): Itinerary
+ getMap(GPSPosition): Map
+ getPath(GPSPosition, GPSPosition): List<GPSPosition>
+ GPSPositionToAddress(GPSPosition): String
+ isValid(GPSPosition): boolean
«interface»
IMilanoGovernment
+ getTaxiDriver(String): TaxiDriver
MilanoGovernment
+ getTaxiDriver(String): TaxiDriver

Figure 4: MTSIntegration

- **IEmailServer:** this interface represents a proxy for the EmailServer external system. The communication to that server is going to be managed here. The provided operations by this class are translated into messages that are sent to the EmailServer. In order to do so, I provides the following operations:
  - Send a confirmation email, given the email address of the passenger. This includes the appropriate creation of the message.
  - Check whether the email address is a valid one.
- **EmailServer:** is a concrete implementation of the IEmailServer interface. Thus, it will implement the inherited methods.
- **IMapsServer:** this interface represents a proxy for the MapsServer external system. The communication to that server is going to be managed here. The provided operations by this class are translated into messages that are sent to the MapsServer. In order to do so, I provides the following operations:
  - Convert an address into a GPSPosition, given the address.
  - Get the Map related to one location, given the GPSPosition of the location.
  - Get a shortest possible path between two locations, given the GPSPositions. It returns a list of GPSPosition that represent the Map.
  - Convert a position into an address, given the GPSPosition.
  - Check whether a position is valid, given the GPSPosition.
- **MapsServer:** is a concrete implementation of the IMapsServer interface. Thus, it will implement the inherited methods.
- **IMilanoGovernment:** this interface represents a proxy for the MilanoGovernment external system. The communication to that server is going to be managed here. The provided operations by this class are translated into messages that are sent to the MilanoGovernment. In order to do so, I provides the following operations:
  - Get the information of a taxi driver, given the taxi code. Returns the instance of the TaxiDriver.

- **MilanoGovernment:** is a concrete implementation of the IMilanoGovernment interface. Thus, it will implement the inherited methods.

2.3.5. MTSNotifier

This component is in charge of sending the notifications to the passenger and the taxi driver when they are logged in the mobile application.

The following is the diagram of the classes encapsulated by this subcomponent.

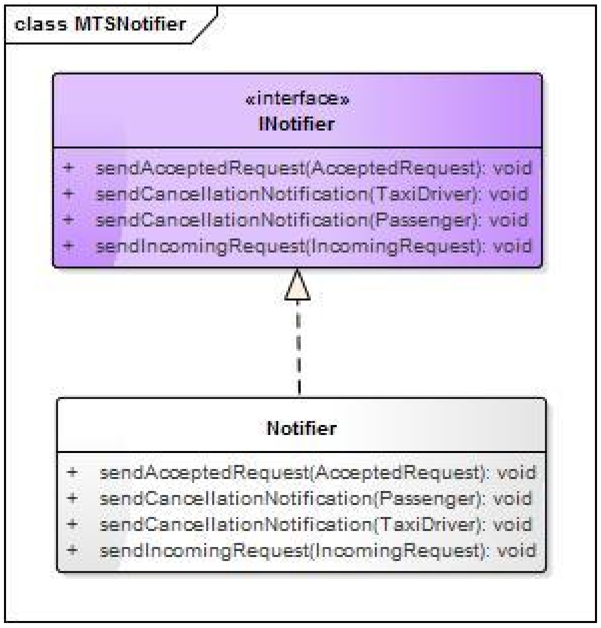

class MTSNotifier
«interface»
INotifier
+ sendAcceptedRequest(AcceptedRequest): void
+ sendCancellationNotification(TaxiDriver): void
+ sendCancellationNotification(Passenger): void
+ sendIncomingRequest(IncomingRequest): void
Notifier
+ sendAcceptedRequest(AcceptedRequest): void
+ sendCancellationNotification(Passenger): void
+ sendCancellationNotification(TaxiDriver): void
+ sendIncomingRequest(IncomingRequest): void

Figure 5: MTSNotifier

- **INotifier:** this interface represents the operations that the class is going to provide to the MTSModel:
  - Send an accepted request, given the AcceptedRequest. This will notify the passenger about the acceptance of a request.
  - Send a cancellation notification to a taxi driver, given the TaxiDriver. This will notify the driver about the respective cancellation.
  - Send a cancellation notification to a passenger, given the Passeger. This will notify the passenger about the respective cancellation.
  - Send an incoming request to a taxi driver, given the IncomingRequest. This will notify the driver about the new request.
- **Notifier:** is a concrete implementation of the INotifier interface. Thus, it will implement the inherited methods

As we will explain later, this component corresponds to the Publisher in a Publish-subscribe architecture. The Subscribers are the Listener components (PassengerNotificationListener and TaxiDriverNotificationListener), which will receive the notifications and will make the Controller to update the View.

2.4. Deployment view

In this subsection we present the physical distribution of the components in the *Component view*. Given that this document gives a logical architecture of the system, we do not specify communication protocols or deployment units; we will only refer to some attributes that the corresponding implementations in later iterations should satisfy in order to get the expected system behavior.

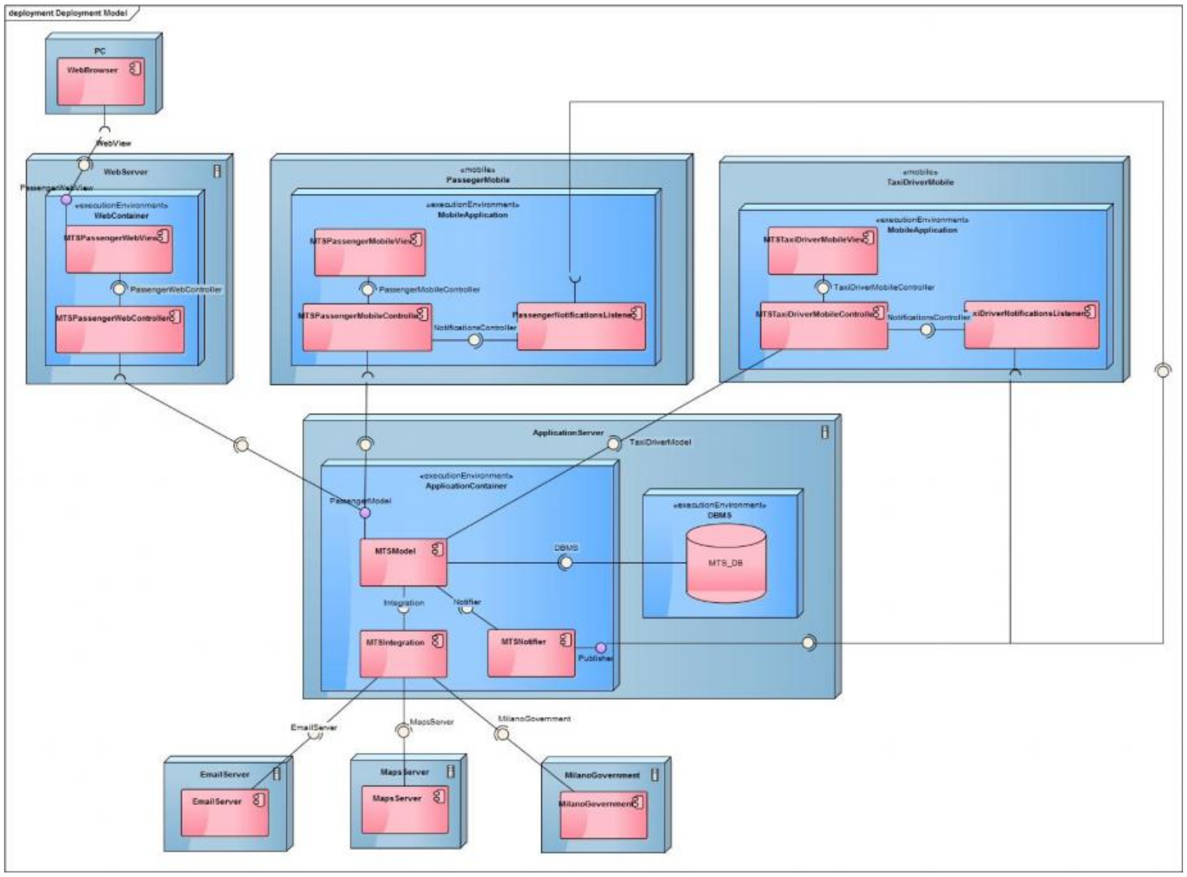


Figure 6: Deployment view

The components have already been described so we only give explanations on the nodes:

- **PC:** it represents the computer from where the Passenger will access the web site of the MyTaxiService system; that is why it contains the WebBrowser component.
  It will connect to the WebServer to get the web pages that will be displayed to the passenger, and to communicate the passenger interactions on those pages.
- **WebServer:** is the machine that will contain the web view, so here we deploy the MTSPassengerWebView and MTSPassengerWebController components. These components are deployed inside a WebContainer, which is a server application capable of receiving requests for pages and the interactions of the users on those pages, and sending the response to the clients. Thus it is capable of holding sessions with the clients. Non concurrency is needed here since the instances are unique for each session

It will connect the AppliationServer to communicate the requests for actions on the logic (which are the result of the MTSController interpreting the interaction on the pages). This implies that the WebContainer should also be capable of sending remote invocations on the objects in the ApplicationServer and receive the responses.

- **ApplicationServer:** is the machine that will contain the business logic, so here we deploy the MTSModel, MTSIntegration and MTSNotifier components. They will be deployed inside an ApplicationContainer, which is a server application capable of receiving requests for actions on the model. This it is capable managing the concurrent access to the components. No sessions are needed to be held since the instances are the same for very client (e. g. all the users connect to the same RequestManager). This is also capable of receiving and handling remote invocations on the objects in the components.
  This machine will also have a DBMS to deploy the MTS_DB. This is a server application capable of receiving requests to the MTS_DB. This is capable of managing the database under the usual conditions (concurrency, reliability, etc.).
  The machine will connect to the external nodes EmailServer, MapsServer, and MilanoGovernment. It will also connect to the mobile phones of the passenger and the taxi driver to send the notifications.
- **PassengerMobile:** is the mobile phone that the passenger will use launch the application of MyTaxiService, so here we deploy the MTSPassengerMobileView, MTSPassengerMobileController and the PassengerNotificationsListener. They will be deployed inside a MobileApplication, a software that runs in a mobile phone that displays the graphic components to the passenger and handles the interactions on them. It is also capable of receiving the notifications from the ApplicationServer.
  It will connect the AppliationServer to communicate the requests for actions on the logic. This implies that the MobileApplication should also be capable of sending remote invocations on the objects in the ApplicationServer and receive the responses.
- **TaxiDriverMobile:** is the mobile phone that the passenger will use launch the application of MyTaxiService, so here we deploy the MTSPassengerMobileView, MTSPassengerMobileController and the PassengerNotificationsListener. They will be deployed inside a MobileApplication, a software that runs in a mobile phone that displays the graphic components to the passenger and handles the interactions on them. It is also capable of receiving the notifications from the ApplicationServer.
  It will connect the ApplicationServer to communicate the requests for actions on the logic. This implies that the MobileApplication should also be capable of sending remote invocations on the objects in the ApplicationServer and receive the responses.
- **EmailServer:** is the machine in which the EmailServer is deployed.
- **MapsServer:** is the machine in which the MapsServer is deployed.
- **MilanoGovernment:** is the machine in which the MilanoGovernment is deployed.

2.5. Runtime view

Along this subsection it is exposed a dynamic view of the system by means of sequence diagrams. These are supposed to show the interactions between the components (and their internal items) that are performed to respond to the user actions on the software.

When the classes of the Notifier components are used, we omit the part on how the message gets to the corresponding Listener in the mobile phones, for the sake of simplicity. This process only includes the invocation of the Listener from the Notifier and the subsequent handling of the notifications by the Controller, which will be an update of the View.

2.5.1. A taxi driver answers a request

This diagram shows how the system reacts to the response of a taxi driver to a request. We assume here that the corresponding request has already been received.

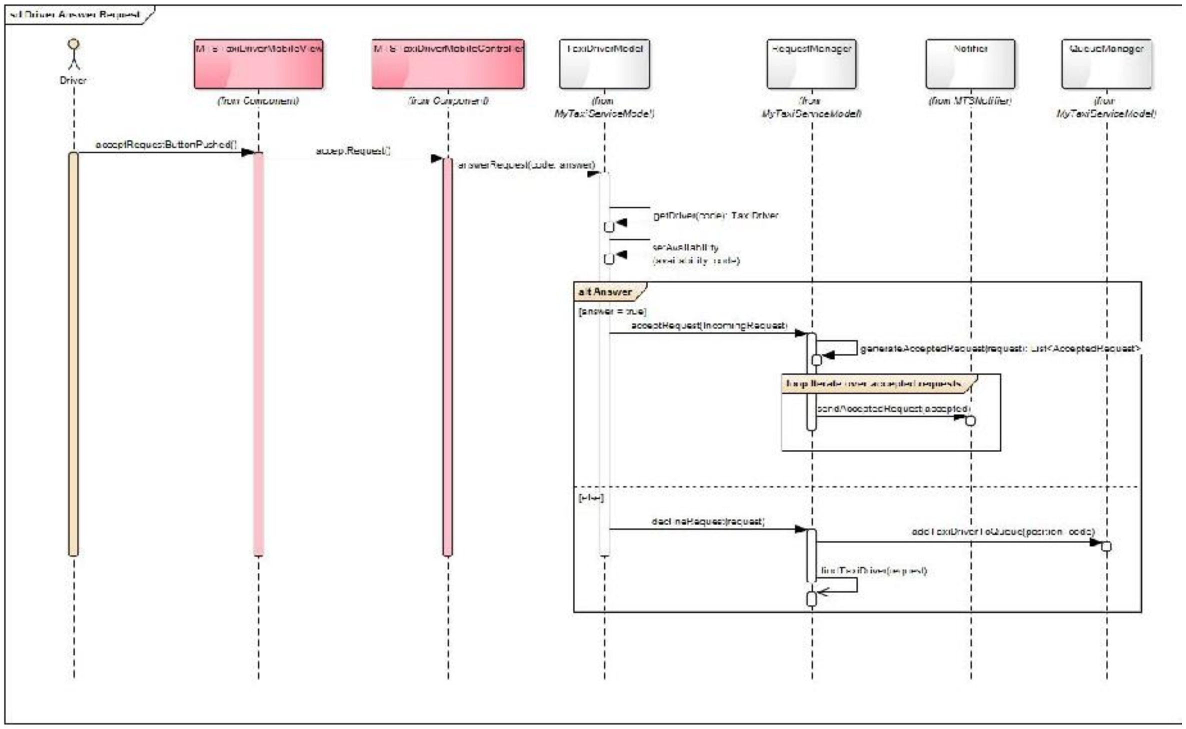

sd Driver Answer Request
Driver
MTS::TaxiDriverMobileView
(from Component)
MTS::TaxiDriverMobileController
(from Component)
TaxiDriverModel
(from MyTaxiServiceModel)
RequestManager
(from MyTaxiServiceModel)
Notifier
(from MTSNotifier)
QueueManager
(from MyTaxiServiceModel)
acceptRequestButtonPushed()
acceptRequest()
answerRequest(code, answer)
getDriver(code): TaxiDriver
setAvailability
(availability, code)
alt Answer
[answer = true]
acceptRequest(IncomingRequest)
generateAcceptedRequest(request): List<AcceptedRequest>
loop Iterate over accepted requests
sendAcceptedRequest(accepted)
[else]
declineRequest(request)
addTaxiDriverToQueue(position, code)
findTaxiDriver(request)

Figure 7: Sequence diagram. Taxi driver answers a request

2.5.2. A taxi driver cancels a request

This diagram shows how the system reacts to the cancellation of a request by a taxi driver.

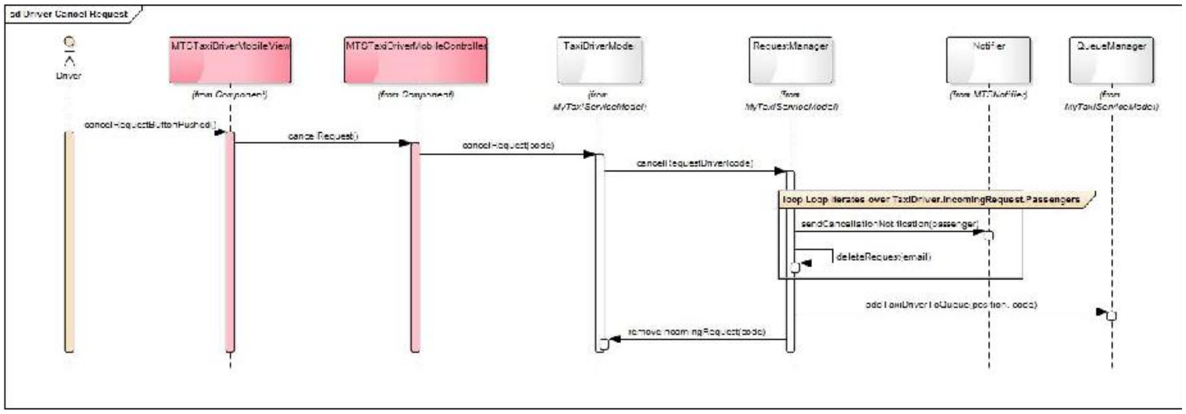

sd Driver Cancel Request
Driver
MTSTaxiDriverMobileView
(from Component)
MTSTaxiDriverMobileController
(from Component)
TaxiDriverModel
(from MyTaxiServiceModel)
RequestManager
(from MyTaxiServiceModel)
Notifier
(from MTSNotifier)
QueueManager
(from MyTaxiServiceModel)
cancelRequestButtonPushed()
cancelRequest()
cancelRequest(code)
cancelRequestDriver(code)
loop Loop iterates over TaxiDriver.IncomingRequest.Passengers
sendCancellationNotification(passenger)
deleteRequest(email)
addTaxiDriverToQueue(position, code)
removeIncomingRequest(code)

Figure 8: Sequence diagram. Taxi driver cancels a request

2.5.3. A taxi driver creates an account

This diagram shows how the system reacts when a taxi driver wants to create an account.

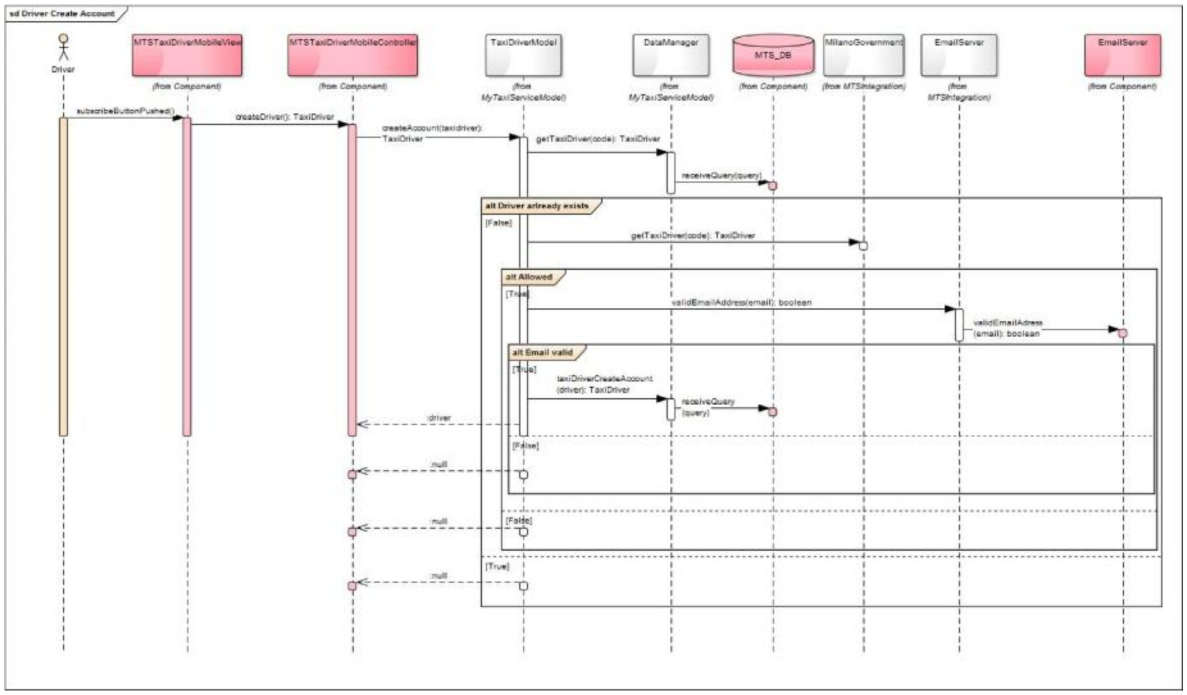


Figure 9: Sequence diagram. Taxi driver creates an account

2.5.4. A taxi driver edits his account

This diagram shows how the system reacts when a taxi driver edits the information of his account.

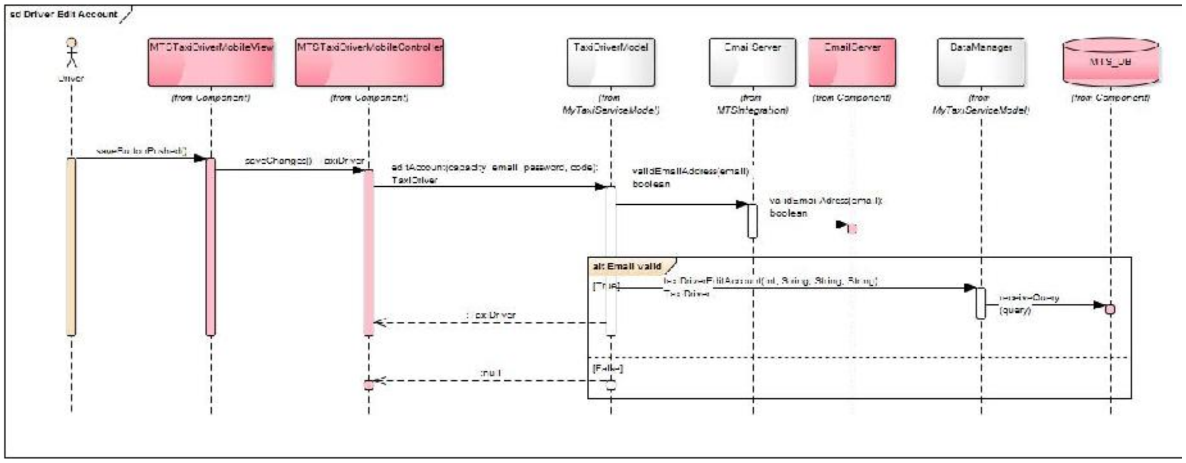

sd Driver Edit Account
Driver
MTSTaxiDriverMobileView
(from Component)
MTSTaxiDriverMobileController
(from Component)
TaxiDriverModel
(from MyTaxiServiceModel)
Email Server
(from MTSIntegration)
EmailServer
(from Component)
DataManager
(from MyTaxiServiceModel)
MTS_DB
(from Component)
saveButtonPushed()
saveChanges(): TaxiDriver
editAccount(capacity, email, password, code): TaxiDriver
validEmailAddress(email): boolean
validEmailAdress(email): boolean
alt Email valid
[True]
taxiDriverEditAccount(int, String, String, String): TaxiDriver
receiveQuery (query)
:TaxiDriver
[False]
:null

Figure 10: Sequence diagram. Taxi driver edits his account

2.5.5. A taxi driver logs in

This diagram shows how the system reacts when a taxi driver wants to login.

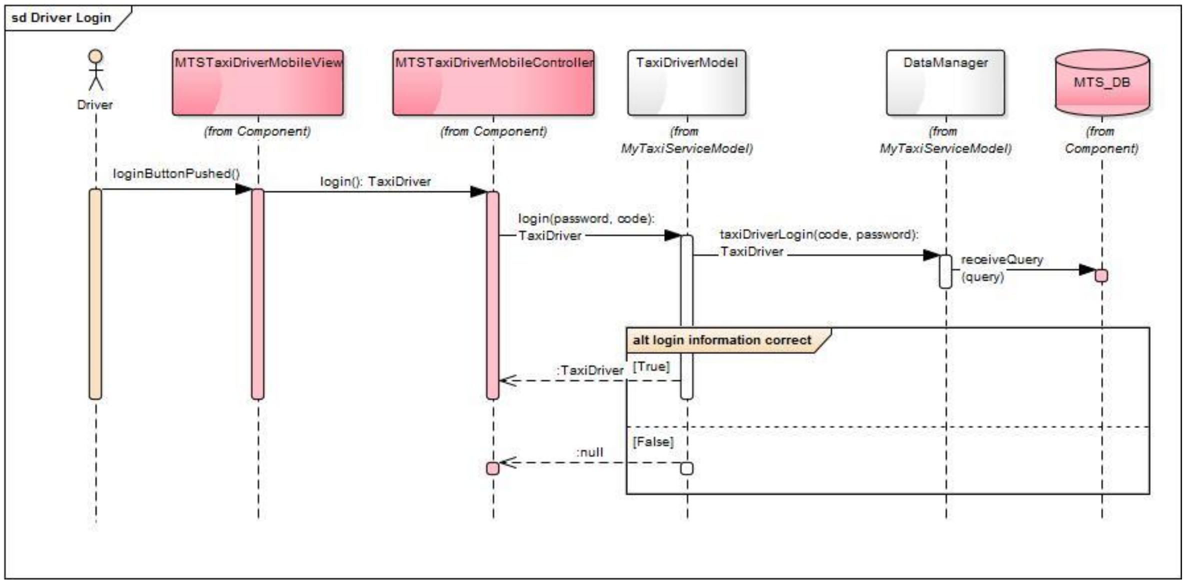


Figure 11: Sequence diagram. Taxi driver logs in

2.5.6. A taxi driver sets his availability

This diagram shows how the system reacts to the setting of the availability by a taxi driver.

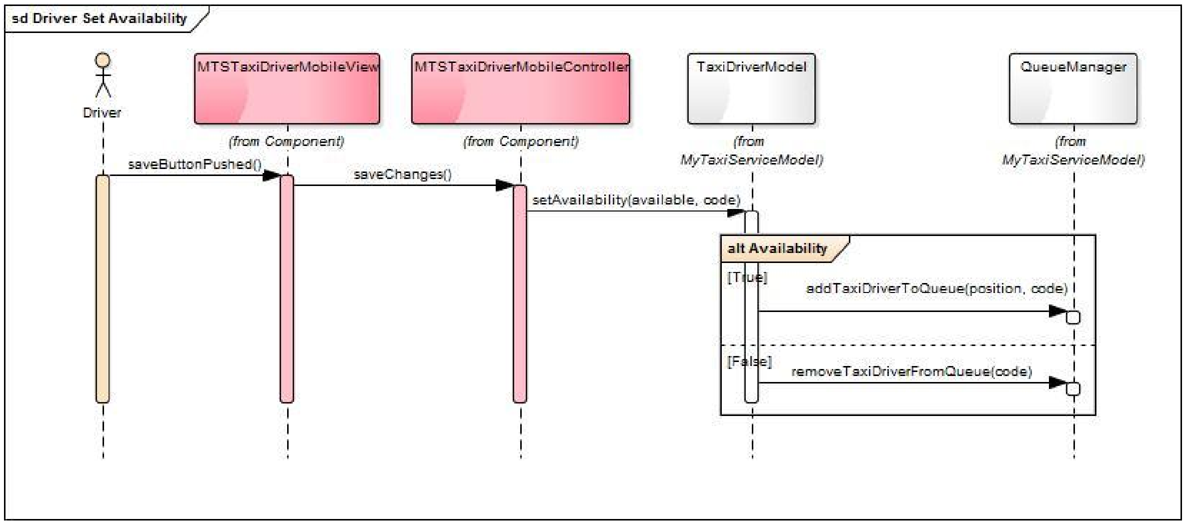

sd Driver Set Availability
Driver
MTSTaxiDriverMobileView
(from Component)
MTSTaxiDriverMobileController
(from Component)
TaxiDriverModel
(from MyTaxiServiceModel)
QueueManager
(from MyTaxiServiceModel)
saveButtonPushed()
saveChanges()
setAvailability(available, code)
alt Availability
[True]
addTaxiDriverToQueue(position, code)
[False]
removeTaxiDriverFromQueue(code)

Figure 12: Sequence diagram. Taxi driver sets his availability

2.5.7. A taxi driver updates his position

This diagram shows how the system reacts to the updating of the position of the driver by the mobile application of the driver.

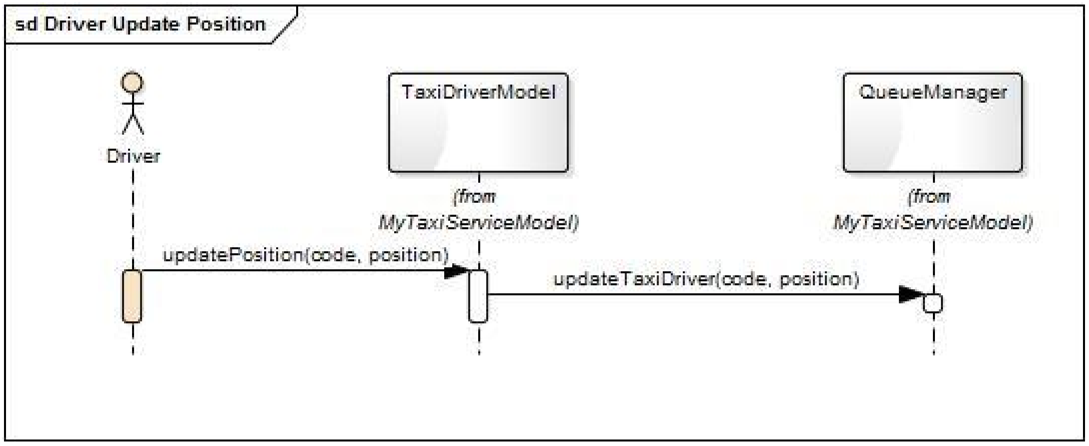

sd Driver Update Position
Driver
TaxiDriverModel
(from MyTaxiServiceModel)
QueueManager
(from MyTaxiServiceModel)
updatePosition(code, position)
updateTaxiDriver(code, position)

Figure 13: Sequence diagram. Taxi driver updates his position

2.5.8. A passenger cancels a request

This diagram shows how the system reacts to the cancellation of a request by a passenger.

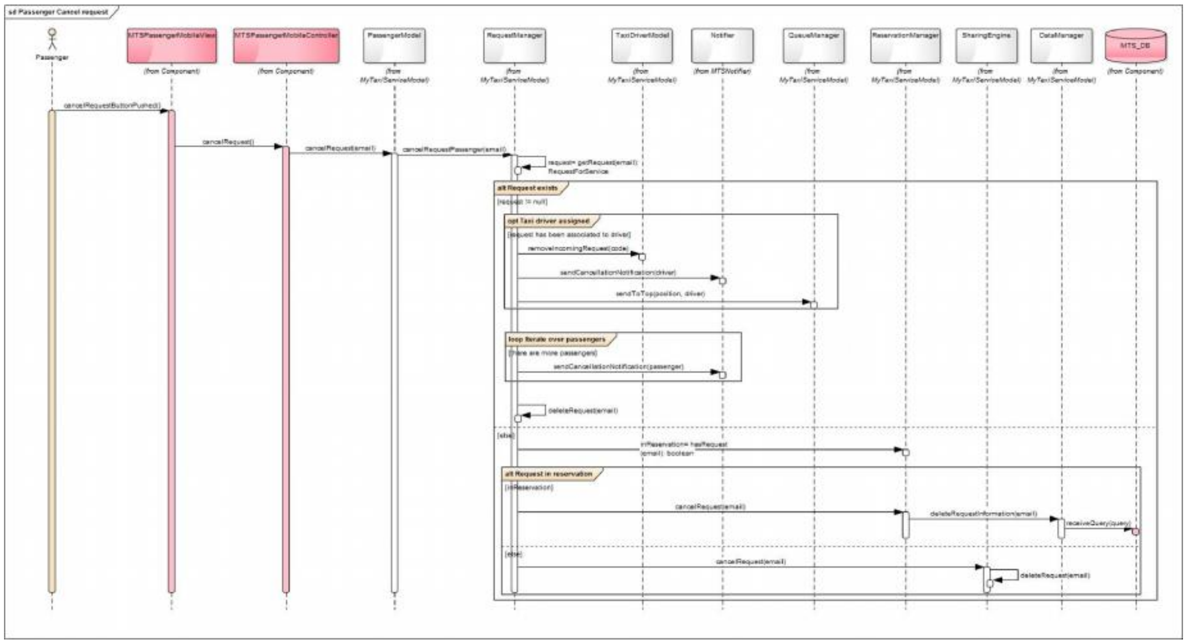

sd Passenger Cancel request
Passenger
MTSPassengerMobileView
(from Component)
MTSPassengerMobileController
(from Component)
PassengerModel
(from MyTaxiServiceModel)
RequestManager
(from MyTaxiServiceModel)
TaxiDriverModel
(from MyTaxiServiceModel)
Notifier
(from MTSNotifier)
QueueManager
(from MyTaxiServiceModel)
ReservationManager
(from MyTaxiServiceModel)
SharingEngine
(from MyTaxiServiceModel)
DataManager
(from MyTaxiServiceModel)
MTS_DB
(from Component)
cancelRequestButtonPushed()
cancelRequest()
cancelRequest(email)
cancelRequestPassenger(email)
request= getRequest(email): RequestForService
alt Request exists
[request != null]
opt Taxi driver assigned
[request has been associated to driver]
removeIncomingRequest(code)
sendCancellationNotification(driver)
sendToTop(position, driver)
loop Iterate over passengers
[there are more passengers]
sendCancellationNotification(passenger)
deleteRequest(email)
[else]
inReservation= hasRequest (email): boolean
alt Request in reservation
[inReservation]
cancelRequest(email)
deleteRequestInformation(email)
receiveQuery(query)
[else]
cancelRequest(email)
deleteRequest(email)

Figure 14: Sequence diagram. Passenger cancels a request

2.5.9. A passenger confirms his email address

This diagram shows how the system reacts when a passenger accesses the link that confirms his email address.

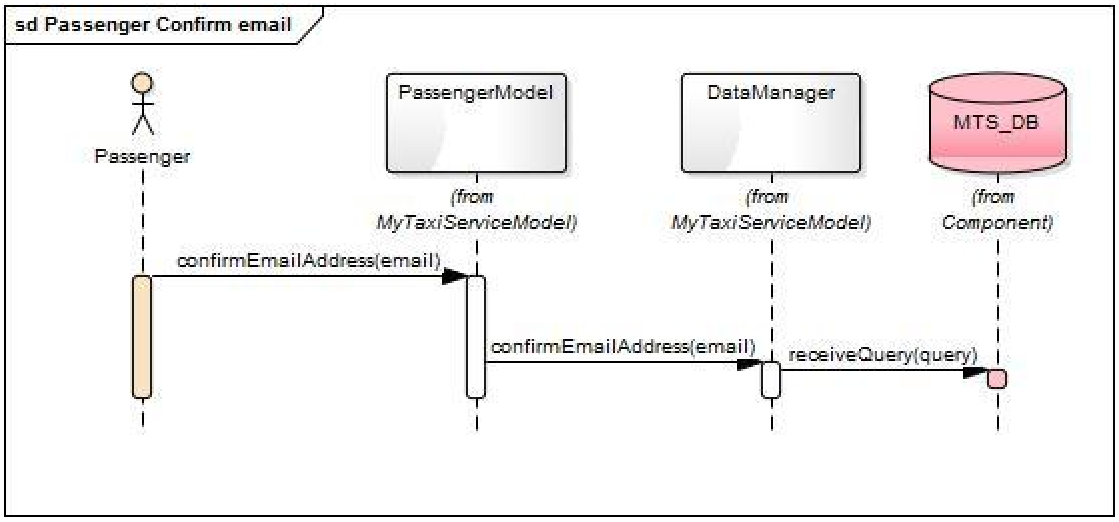

sd Passenger Confirm email
Passenger
PassengerModel
(from MyTaxiServiceModel)
DataManager
(from MyTaxiServiceModel)
MTS_DB
(from Component)
confirmEmailAddress(email)
confirmEmailAddress(email)
receiveQuery(query)

Figure 15: Sequence diagram. Passenger confirms his email address

2.5.10. A passenger creates an account

This diagram shows how the system reacts when a passenger creates an account.

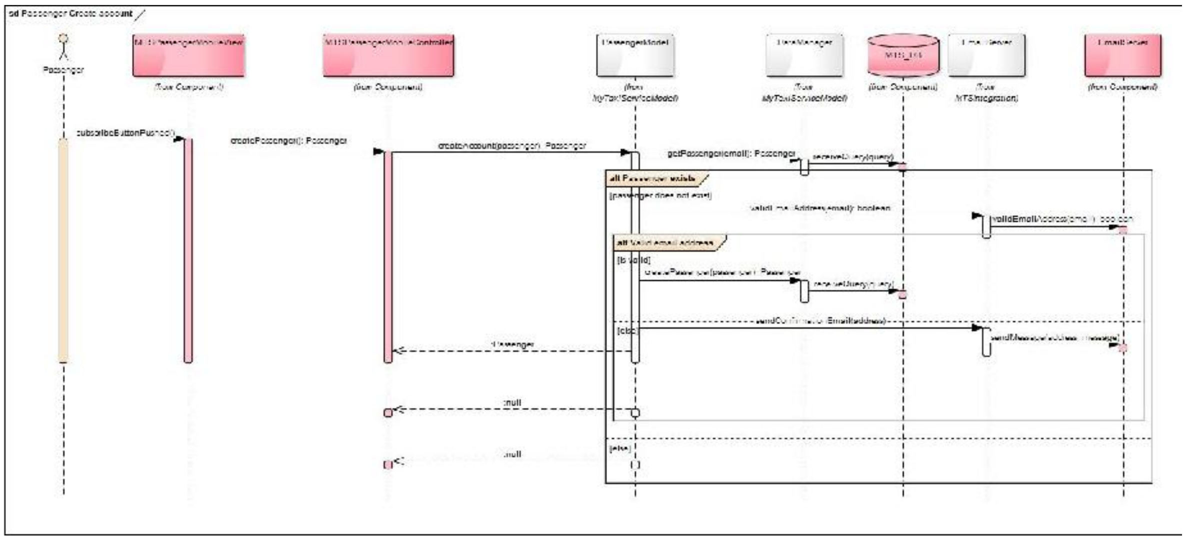

sd Passenger Create account
Passenger
MTSPassengerMobileView
MTSPassengerMobileController
PassengerModel
DataManager
MTS_DB
EmailServer
subscribeButtonPushed()
createPassenger(): Passenger
createAccount(passenger): Passenger
getPassenger(email): Passenger
alt Passenger exists
[passenger does not exist]
validEmailAddress(email): boolean
alt Valid email address
[is valid]
[else]
:null

Figure 16: Sequence diagram. Passenger creates an account.

2.5.11. A passenger edits an account

This diagram shows how the system reacts when a passenger edits his account.

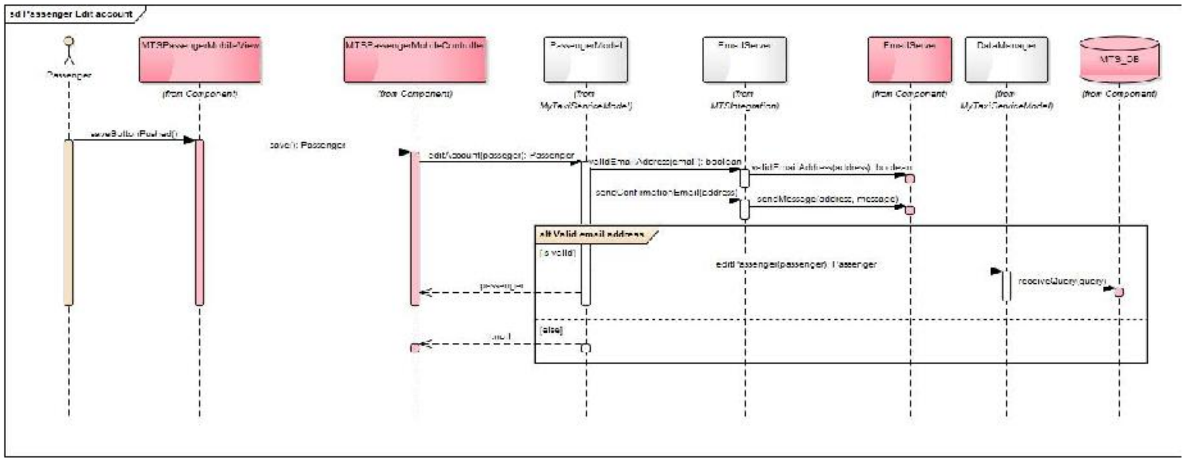
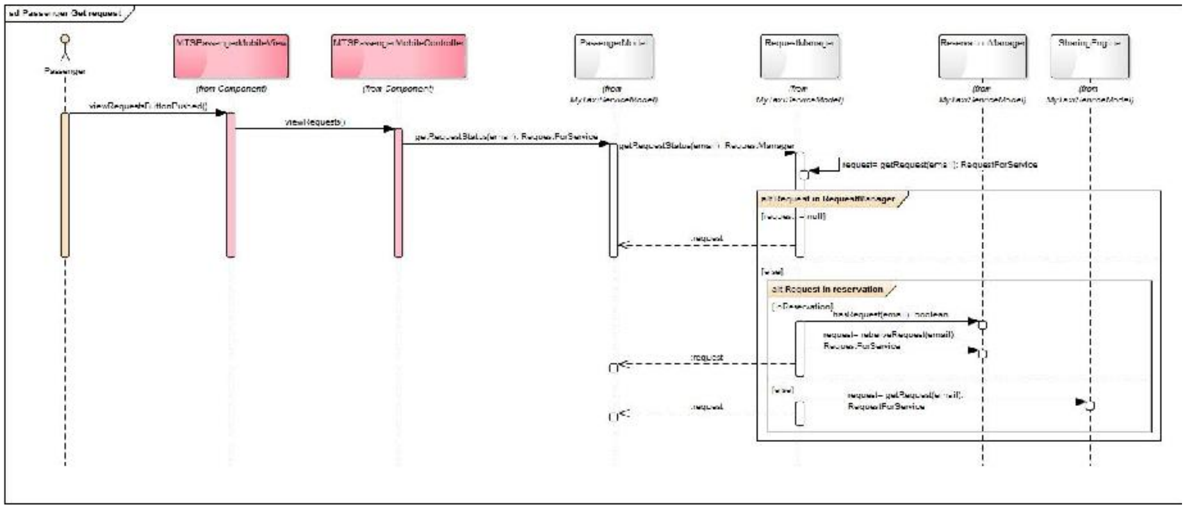

sd Passenger Edit account
Passenger
MTSPassengerMobileView
MTSPassengerMobileController
PassengerModel
EmailServer
DataManager
MTS_DB
(from Component)
saveButtonPushed()
save(): Passenger
editAccount(passenger): Passenger
validEmailAddress(email): boolean
sendConfirmationEmail(address)
sendMessage(address, message)
alt Valid email address
[is valid]
editPassenger(passenger): Passenger
receiveQuery(query)
[else]
:null

Figure 17: Sequence diagram. Passenger edits his account.

2.5.12. A passenger wants to get the status of his request

This diagram shows how the system reacts when a passenger wants to see the status of his request.



sd Passenger Get request
Passenger
MTSPassengerMobileView
(from Component)
MTSPassengerMobileController
(from Component)
PassengerModel
(from MyTaxiServiceModel)
RequestManager
(from MyTaxiServiceModel)
ReservationManager
(from MyTaxiServiceModel)
SharingEngine
(from MyTaxiServiceModel)
viewRequestsButtonPushed()
viewRequests()
getRequestStatus(email): RequestForService
getRequestStatus(email): RequestManager
request= getRequest(email): RequestForService
alt Request in RequestManager
[request != null]
:request
[else]
alt Request in reservation
[inReservation]
hasRequest(email): boolean
request= retrieveRequest(email): RequestForService
:request
[else]
request= getRequest(email): RequestForService
:request

Figure 18: Sequence diagram. Passenger gets the status of his request.

2.5.13. A passenger logs in

The diagram shows how the system reacts when a passenger wants to log into the system.

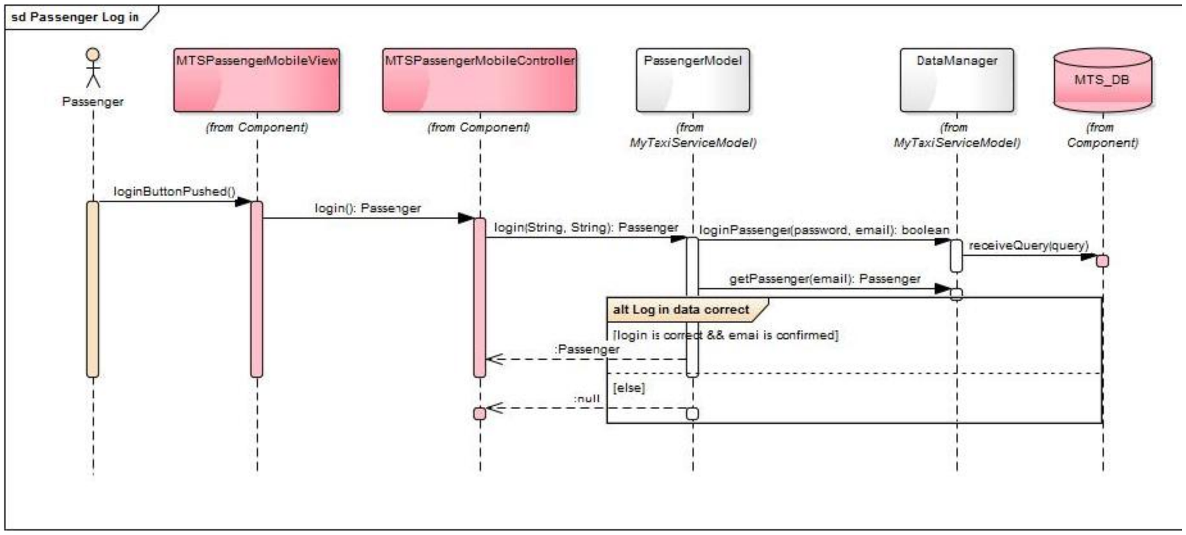


Figure 19: Sequence diagram. Passenger logs in.

2.5.14. A passenger sends a request

This diagram shows how the system reacts when a passenger sends a request. This is one of the most complex processes of the software, so we have separated it in various diagrams.

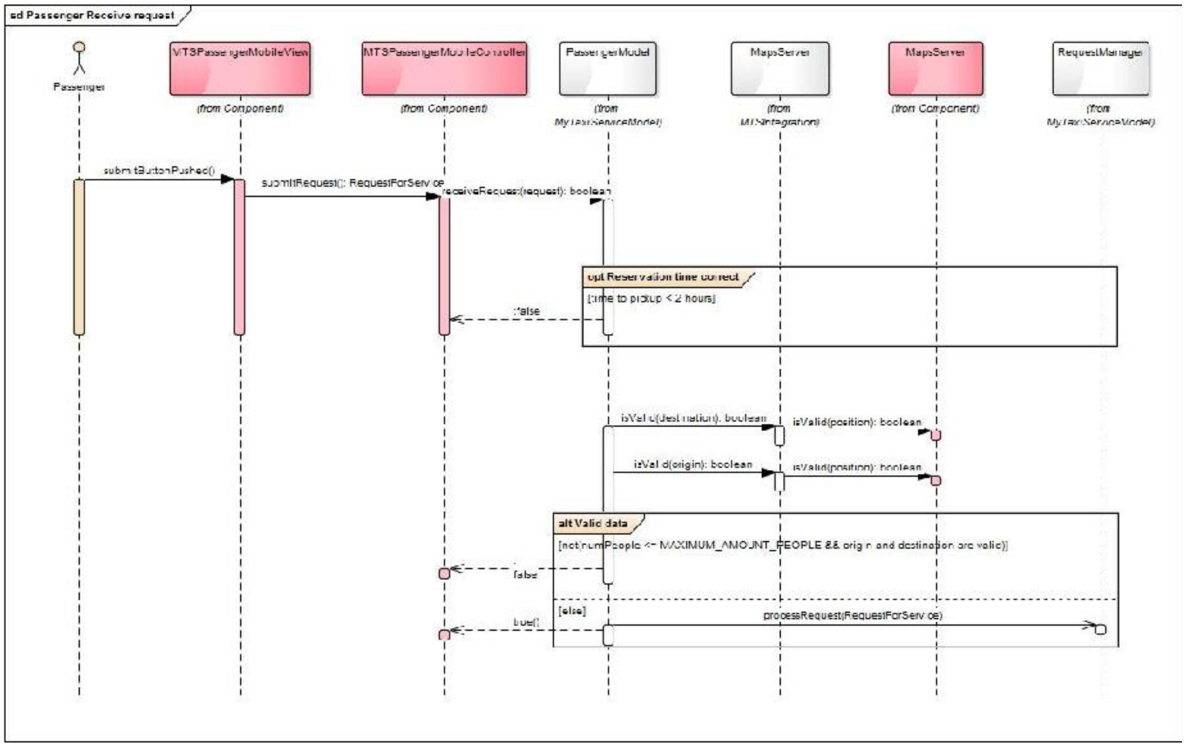

sd Passenger Receive request
Passenger
MTSPassengerMobileView
(from Component)
MTSPassengerMobileController
(from Component)
PassengerModel
(from MyTaxiServiceModel)
MapsServer
(from MTSIntegration)
MapsServer
(from Component)
RequestManager
(from MyTaxiServiceModel)
submitButtonPushed()
submitRequest(): RequestForService
receiveRequest(request): boolean
opt Reservation time correct
[time to pickup < 2 hours]
false
isValid(destination): boolean
isValid(position): boolean
isValid(origin): boolean
isValid(position): boolean
alt Valid data
[not(numPeople <= MAXIMUM_AMOUNT_PEOPLE && origin and destination are valid)]
false
[else]
processRequest(RequestForService)
true()

Figure 20: Sequence diagram. Passenger sends a request.

Here we only show what happens until the RequestManager begins to process the request in an asynchronous way, which allows the system to send the confirmation message to the passenger and process the request later.

The processing is explained with detail in the *Algorithm view* section.

2.6. Component interfaces

We will present in this part of the document the interfaces relevant for each component identified in the *Component view*. To do so, we describe the required and provided interfaces, and the corresponding operations that can be executed on them. Note that the *Deployment view* induces remote and local interfaces on some of the components, but we assume that both of them will provide the same operations.

- **WebBrowser**: It allows the displaying and the use of the MyTaxiService website, thus it encapsulates some display functions and some income functions as well such as login, create account, create request. It needs to communicate with the MTSPassengerWebView that will provide it all the displaying command. The WebBrowser interface is on the one hand responsible of the transmission of the displaying command from the MTSPassengerWebView to the WebBroswer and on the second hand is responsible of the way back transmission of the input events received by the web browser.
- **MTSPassengerWebView**: It is an intermediary between the web browser and the controller. The PassengerWebController interface is in charge to give displaying

command from the controller to the MTSPassengerWebView and to transmit input events in the opposite direction, thus encapsulates some display and income functions.

- **MTSPassengerWebController:** It provides displaying orders to the MTSPassengerWebView that are based on the information received from the MTSModel. It also receives notifications about input event from MTSPassengerView then sends the appropriate requests to the MTSModel. It owns some result interpretation functions that will create a displaying command from the result of a Model method and an input event interpretation that will from an input event call a model method. So it needs a PassengerModel interface that allows it to communicate with the MTSModel. This interface is used to call the PassengerModel methods and to receive results of those calls.
- **MTSPassengerMobileView:** It has the same behavior than the MTSPassengerWebView unless that it doesn't command a WebBrowser but owns some display and income functions to command the hardware directly inner the mobile. The interface between it and the MTSPassengerMobileController has the same purpose than the one between the WebView and the WebController.
- **MTSPassengerMobileController:** It has the same behavior than the MTSPassengerWebController. It communicates with the MTSPassengerMobileView for sending displaying commands and receiving input event notifications. It also communicates with the MTSModel through a PassengerModel interface to call the PassengerModel methods and receive results of those calls.
- **MTSDriverMobileView:** It has the same behavior than the MTSPassengerMobileView unless that it communicates with the MTSDriverMobile Controller.
- **MTSDriverMobileController:** It has the same behavior than the MTSPassengerMobileController unless it communicates with the MTSModel through a TaxiDriverModel interface.
- **MTSModel:** Its methods are called by the controllers and it sends back to them the results. It includes all the methods described in the document such as editing an account, answering a request. It has an interface with the MTSDataBase to send some write and read requests and receive the results. It has also an interface with the MTSIntegration component to call some methods that belong to external components and receive the results of those.
- **MTSDataBase:** It communicates only with the MTSModel to receive from it some read or write requests and send back some results. It includes all the data access methods such as creating an account, editing an account, loging in, getting a passenger instance..
- **MTSIntegration**: This component share an interface with the MTSModel that sends some external methods calls that will be transferred trough others interfaces either to the MapServer, the EmailServer or the MilanoGovernment component. These one will send back the results of those methods that will be transferred to the MTSModel. Thus it includes the external functions : getting a map, get information from the Milano government, sending email

- **MapServer, EmailServer, MilanoGovernment**: It exists an interface between each of them and the MTSIntegration. It allows the MTSIntegration to call some external methods and get the result of those. They each include some external functions : getting a map, get information from the Milano government, sending email

2.7. Selected architectural styles and patterns

The previously presented architecture was guided by some architectural styles and patterns. Now we present them and we explain how they have been adopted within this specific context.

- Model-View-Controller:
  It was used to logically decouple the graphic logic and the business logic. This allows to perform parallel design and implementation of those components. It also enables reuse of the business logic component from any graphic component (e. g. the web view and the mobile view use the same model).
  The implementing components are:
  - Model: MTSModel.
  - View: MTSPassengerWebView, MTSPassengerMobileView and MTSTaxiDriverView.
  - Controller: MTSPassengerWebController, MTSPassengerMobileController and MTSTaxiDriverController.
- Multi-tier:
  It was used to physically decouple the graphic logic and the business logic. This allows to perform independent maintenance of the nodes and decreases the impact of the modifications. Also allows to easily introduce redundancy and load balancing of the servers, without others noticing it.
  The tiers are:
  - Client: PC
  - GUI: WebServer
  - Logic: ApplicationServer
  - Data: MTS_DB
- Distributed Objects:
  The system is designed with an object oriented approach, so the physical separation that induces the Multi-tier pattern makes such objects communicate each other in a remote way.
  For the remote invocation of a method, this pattern requires the creation of additional objects in each machine that will work as proxies for the remote target objects. We omitted it and placed instead simple connections among objects deployed in different physical machines, in order to keep simpler diagrams. However, pertinent comments were done in the *Deployment view*.
- Publish-Subscribe:
  It was used to asynchronously communicate with the passenger and the taxi driver, by means of notifications.

The implementing components are:

- Publisher: MTSNotifier. When some message muste be sent to the users, it creates a notification and publishes it for the interested Listeners.
- Subsciber: PassengerNotificationsListener and TaxiDriverNotificationsListener. When the users log in, this components subscribe to the notifications of the specific user in the MTSNotifier.

- Service Oriented Architecture:
  It was used to model the communication with the external systems. Using this approach allows us to build the architecture without depending on how those systems are actually implemented. We get reduction of coupling and obtain modifiability.
  We assume that the communication with the external systems is made through a web service that they expose. In order to do that, the objects in the MTSIntegration component represent proxies of such systems and allow the communication with them.

Some more low level design patterns were also used:

- Façade:
  The interfaces IPassengerModel and ITaxiDriverModel encapsulate the access to the classes in te MTSModel component. This allows us to provide a specific set of operations among all the avaiblable ones.
- Proxy:
  For the Distributed Objects and the Service Oriented Architecture, the local objects work as proxies of the remote ones.

2.8. Other design decisions

Those design decisions which are not based on the selection of an architectural pattern are presented and justified here.

As previously stated, this document represents a logical architecture. By doing so, we pretended to focus on the behavior of the system independently of specific communication and implementation issues. This adds portability to the architecture, so that it might be implemented on a broad variety of platforms.

The reservations requests are stored in the MTS_DB in order to free resources and space in the system. Otherwise, the information of such requests should be kept in main memory, for example, and this would unnecessarily consume resources.

3. Algorithm design

Along this section we give deeper details on the most complex algorithmic parts of the system. These processes were already referred to in the *Runtime view*.

3.1. Process a request

In the *Runtime view* we described the steps that are performed by the system when a passenger sends a request, up to the point when the processing begins. The steps between this point and the creation and sending of the IncomingRequest are described here.

When the request is received, the RequestManager checks if it is a reservation. If that is the case, its processing is scheduled for a latter moment and the request is stored into the MTS_DB by the ReservationManager.

If the request was not a reservation but a sharing, the system tries to find a compatible SharingRequest: among the SharingRequest that have the same origin Zone, the SharingEngine looks for one from which the destination of the just received request is reachable. If such a SharingRequest is found, we add the new request. If no compatible SharingRequest exists, we create a new SharingRequest for the request.

If the request was neither a reservation nor a sharing, the RequestManager will try to find a taxi driver for it. This part is explained later.

This sequence diagram illustrates the just described scenario.

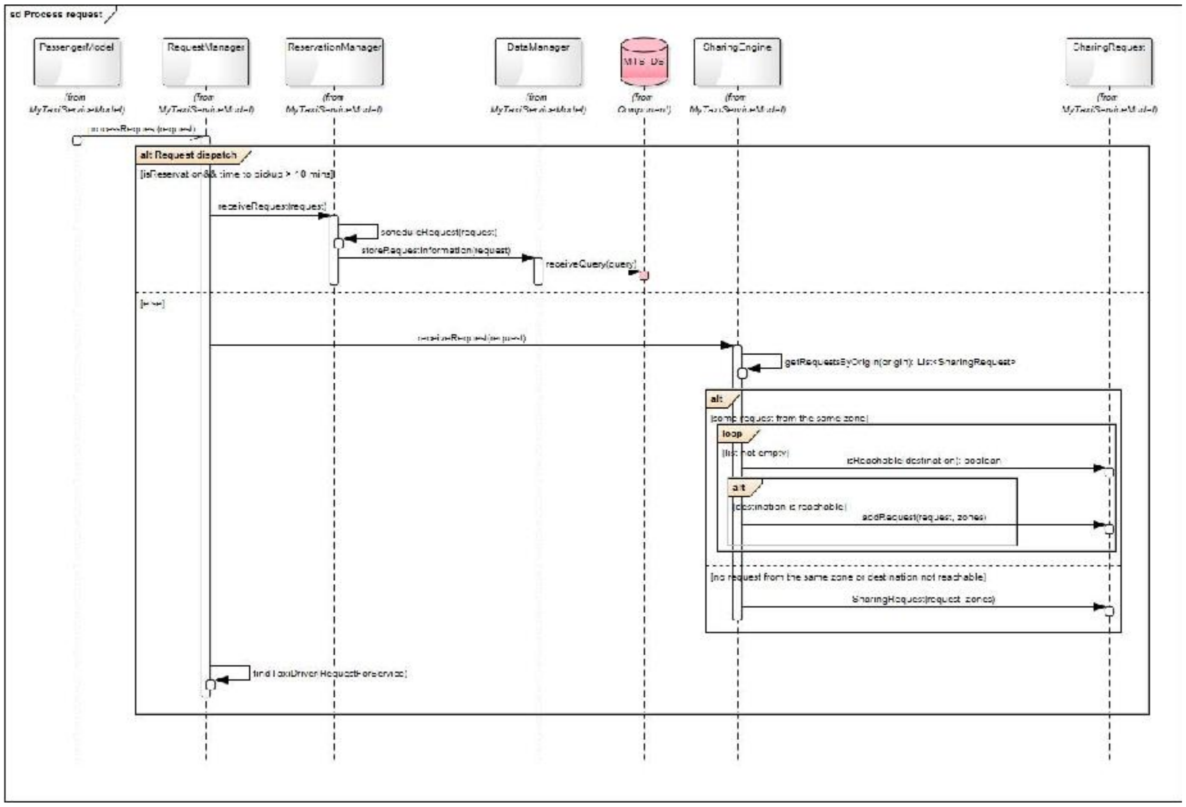

sd Process request
PassengerModel
RequestManager
ReservationManager
DataManager
MTS DB
SharingEngine
SharingRequest
(from MyTaxiServiceModel)
(from Component)
processRequest(request)
alt Request dispatch
[isReservation && time to pickup > 10 mins]
receiveRequest(request)
scheduleRequest(request)
storeRequestInformation(request)
receiveQuery(query)
[else]
receiveRequest(request)
getRequestsByOrigin(origin): List<SharingRequest>
alt
[some request from the same zone]
loop
[list not empty]
isReachable(destination): boolean
alt
[destination is reachable]
addRequest(request, zones)
[no request from the same zone or destination not reachable]
SharingRequest(request, zones)
findTaxiDriver(RequestForService)

Figure 21: Process request. Sequence diagram.

The activity diagram below describes what happens after each class receives the request.

The ReservationManager will wait for the time for the processing is reached to retrieve the request and send it back to the RequestManager. This is done through the invocation of the method *processRequest* for a RequestForService.

The SharingEngine will wait for the completion of the SharingRequest or the maximum searching time to be reached, and after that it takes the SharingRequest to the RequestManager. This is done through the invocation of the method *findTaxiDriver* for a SharingRequest.

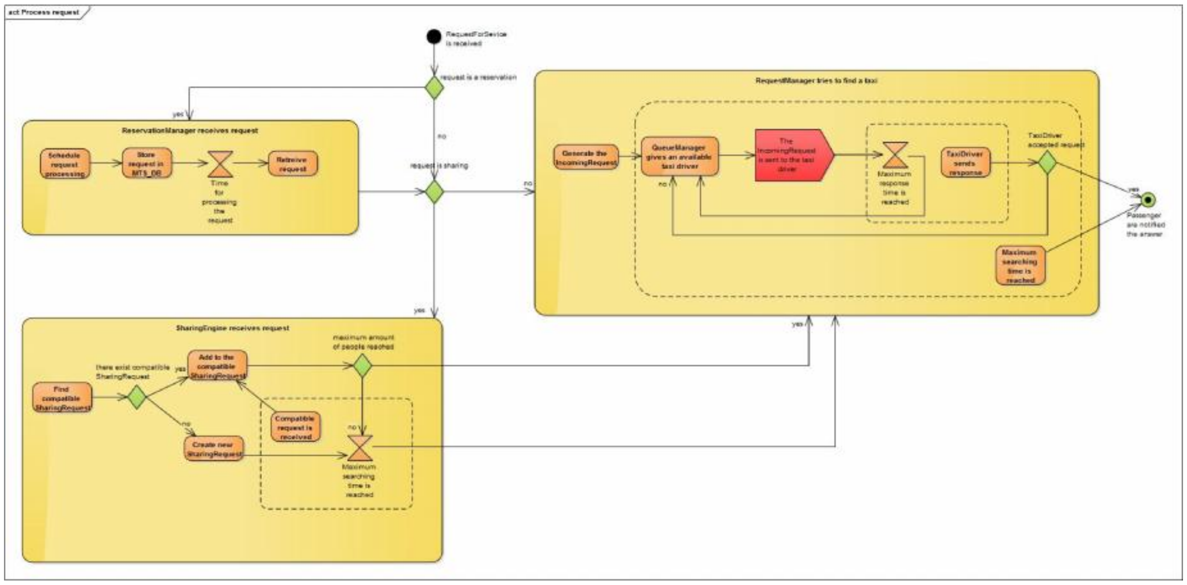

act Process request
RequestForSevice is received
request is a reservation
yes
no
ReservationManager receives request
Schedule request processing
Store request in MTS_DB
Time for processing the request
Retreive request
request is sharing
no
yes
RequestManager tries to find a taxi
Generate the IncomingRequest
QueueManager gives an available taxi driver
The IncomingRequest is sent to the taxi driver
Maximum response time is reached
TaxiDriver sends response
TaxiDriver accepted request
no
yes
Passenger are notified the answer
Maximum searching time is reached
SharingEngine receives request
Find compatible SharingRequest
there exist compatible SharingRequest
yes
Add to the compatible SharingRequest
maximum amount of people reached
no
Create new SharingRequest
Compatible request is received
Maximum searching time is reached
yes

Figure 22: Process request. Activity diagram.

Note that, if the request is not a sharing, we go to the method *findTaxiDriver* for a RequestForService. This and the one of the SharingRequest are similar, so we only illustrate the first.

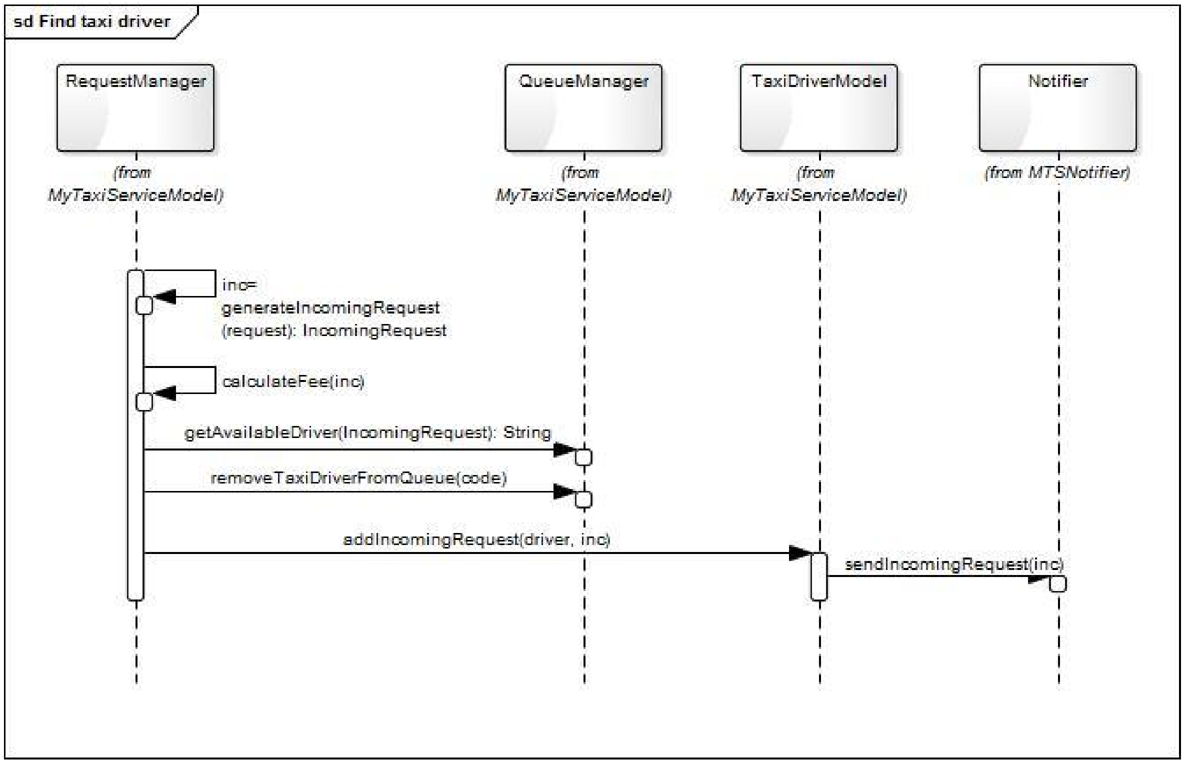

sd Find taxi driver
RequestManager
(from MyTaxiServiceModel)
QueueManager
(from MyTaxiServiceModel)
TaxiDriverModel
(from MyTaxiServiceModel)
Notifier
(from MTSNotifier)
inc= generateIncomingRequest (request): IncomingRequest
calculateFee(inc)
getAvailableDriver(IncomingRequest): String
removeTaxiDriverFromQueue(code)
addIncomingRequest(driver, inc)
sendIncomingRequest(inc)

Figure 23: Find taxi driver

A sequence diagram that illustrates the steps after the taxi driver response is in the

Some additional comments:

- For the creation of a SharingRequest, it is needed the RequestForService and the list of the zones that can be reachable from path associated to this. Remember that such path was calculated by the MapsServer, and the list of reachable zone was derived by the QueueManager, based on that path.
- The taxi drivers have a maximum response time when they are sent an InomingRequest. Even though it is not shown in the diagram, both no response and no acceptance will make the driver go to the bottom of the queue.
- At any moment of this process, the request can be canceled either by the driver or the passenger.

3.2. Calculate a fee

According the official website of Milan, the different fees for taxis are the following:

Use	**Period**	**Initial Fee**		**Additional Fee**	
		Normal	From Airport	Per kilometer	Per Hour
Normal	Day	3,30	13,10	1,09	28,32
	Day during holidays	5,40			
	Night (10PM - 6AM)	6,50			

Table 1: Calculation of the fee

The choice made for a ride is to split the price of the share parts of the ride between all the participants. We didn't consider the sharing option offered by the taxi service of Milan which has some specific price for shared ride but need at least 3 passengers which we didn't have assumed.

Example

Assume it is 3:30 PM, Alessandro want to go from Loreto to Lodi. He ask for a sharing ride. The application found Elisa who want to go from Loreto to Porta Vittoria to share the ride. The itinerary will be Loreto – Porta Vittoria – Lodi.

Since Loreto – Porta Vittoria takes 12 min and 3,8 km and Porta Vittoria – Lodi takes 8 min and 2,5 km, the fees for Alessandro and Elisa will be approximately :

Alessandro: (3,3 + 1,09*3,8 + 28,32*12/60)/2 = 6,55 €

Elisa: (3,3 + 1,09*3,8+ 28,32*12/60)/2 + 1,09*2,5+ 28,32*8/60 = 13,05 €

Pseudo Code of the fee calculation method

The main steps of the algorithm are the following:

0) Declaration of the fee constants

1) Determination of the fee constants we will use depending of the date

2) Sorting the requests according to their distance origin-destination

3) Calculation of the itineraries of every part of the ride

4) Calculation of the final fee for every request based on the distance and the duration of every part of the ride

5) Adding the fees to the related accepted request

```
//0)

//Declaration of all the fee constants

durationfee=28,32

distancefee=1,09

dayfee=3,3

holidayfee=5,4

nightfee=6,5


calculateFee(IncomingRequest) : void


//1)

//We set the constants which are related to the date and the hour of the request

//We take the information from the first request of the list


  date =  IncomingRequest.request[0].pickupTime()

  hour = date.hour()

  if 22<=hour<24 or 0<=hour<6 :

            fixedfee=nightfee

  elif date in holidays :

            fixedfee = holidayfee

  else:
        fixedfee = dayfee


  requestlist = []
  origin=IncomingRequest.requests[0].origin //common origin for all the requests


// 2)
//Fill the requestlist with couples (request, distance origin-destination)
  for request in  IncomingRequest.requests :
        destination= request.destination
        requestlist.add((request,origin-destination) // calculate the distance between origin and destination : |origin.latitude-destination.latitude|+|origin.longitude-destination.longitude|
  requestlist.sort(1) // sort the requestlist increasingly according to the distance origin-destination


// 3)
//Create a list of itinerary between the destination of the previous request and the destination of the current request
  itineraries = []
  for e in requestlist :
        request=e[0] // get the first component of e which is the request
        destination= request.destination
        itineraries.add(MapServer.getItinerary(origin,destination,hour))
        origin=destination


//4)
// Calculate the fees or every request and store it in a list. The calculation is based on the time and distance provided by the Map server threw the itineraries. The fee is divided according to the amount of passengers who are sharing the taxi for a given itinerary
//The requests are sorted in the list in the same order that they will leave the taxi. We calculate
```

then the fee for the ride between the last leave point to the following and add to it the fixedfee and the fee of the previous request without the fixedfee

```
  fees=[]
  n = itineraries.length() //maximum amount of passengers for this ride
  f=0 //distance and duration fee of the previous request
  m=n //current mount of passengers
  for itinerary in itineraries :

        fees.add( fixedfee/n + itinerary.distance/m*distancefee + itinerary.duration/m*durationfee + f )
        f=f+ itinerary.distance/m*distancefee + itinerary.duration/m*durationfee
        m=m-1
```

//5)

//Add the fee of all request to the AcceptedRequest related in doing the link between the requestlist and the fees list

```
  for i in range(0,n):
requestlist[i][0].acceptedRequest.fee=fees[i]
```

4. User interface design

The graphic user interface for both the Passenger and the TaxiDriver are presented in this section. For the first one, we only present the mobile application perspective since the view from the WebBrowser is supposed to be analogous (this assumption is valid because we are presenting a logical architecture, even though we know that the specific implementations will be different in terms of code and platform).

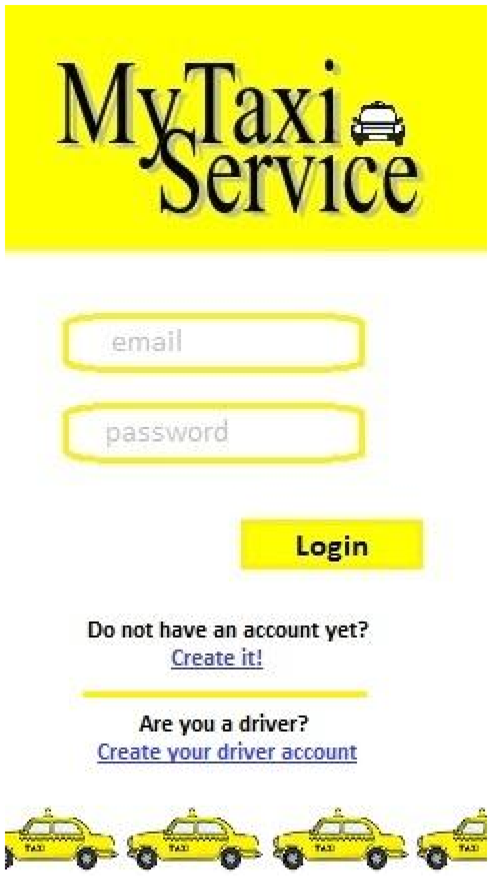


Image 6.1: Homepage/Login

When the user (driver or passenger) open his/her mobile application or the website, he/she sees this page. Then if he/she already has an account he/she fills in the form and click on the button "Login" (then he/she will be send to the page showed in Image 6.3 if he/she is a passenger and to the one showed in Image 6.9 if he/she is a driver). Otherwise the user can create a new account clicking on the link "Create it!" (Image 6.2) if he/she is a passenger or "Create your driver account" (Image 6.8) if he/he is a driver.

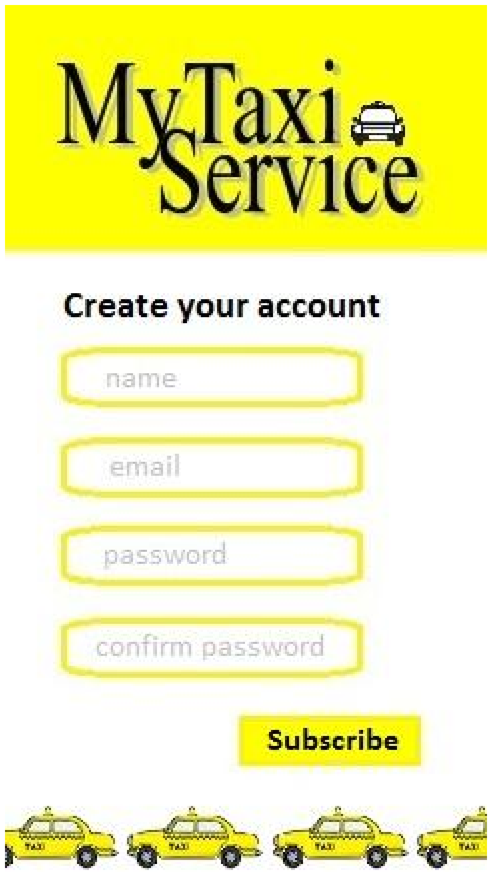


Image 6.2: Passenger creation account

The passenger fills in the form with all the information requested and clicking on the button “Subscribe” the system will check if email is valid and the two password inserted are the same. Then the passenger will see a page with the new profile created (Image 6.3).

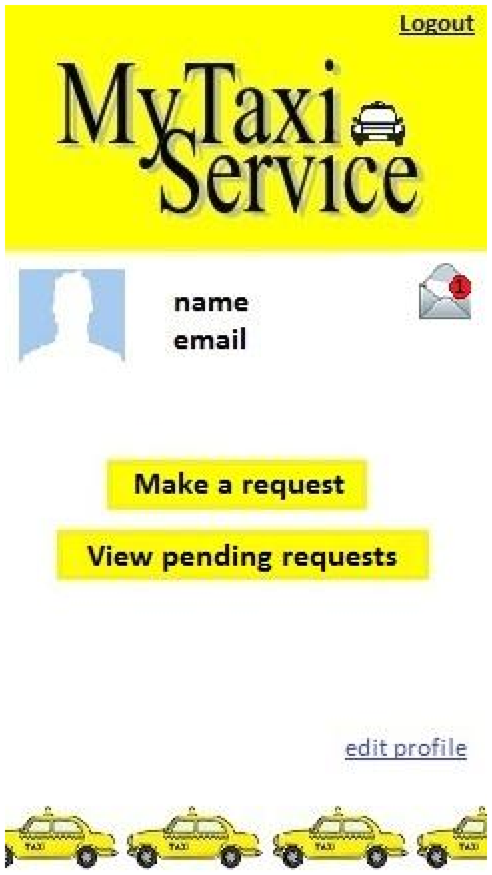


Image 6.3: Passenger profile

Through his/her own profile, the passenger can see his/her name, email and notification through the envelope on the right top of the page, make a request clicking on the related button (then the passenger will see the page showed in Images 6.5), see his/her incoming requests accepted or waiting to be accepted by a driver clicking on the related button (then the passenger will see the page showed in Image 6.6), edit the profile clicking on the related link, make the logout from the system.

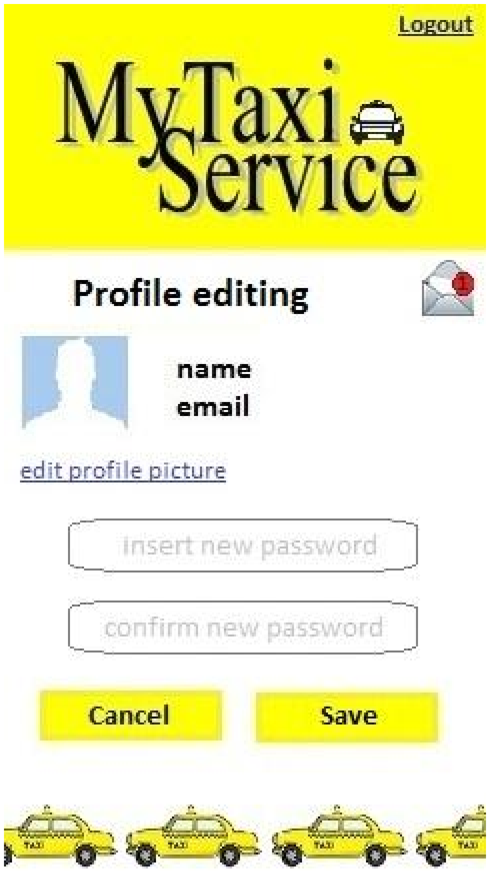
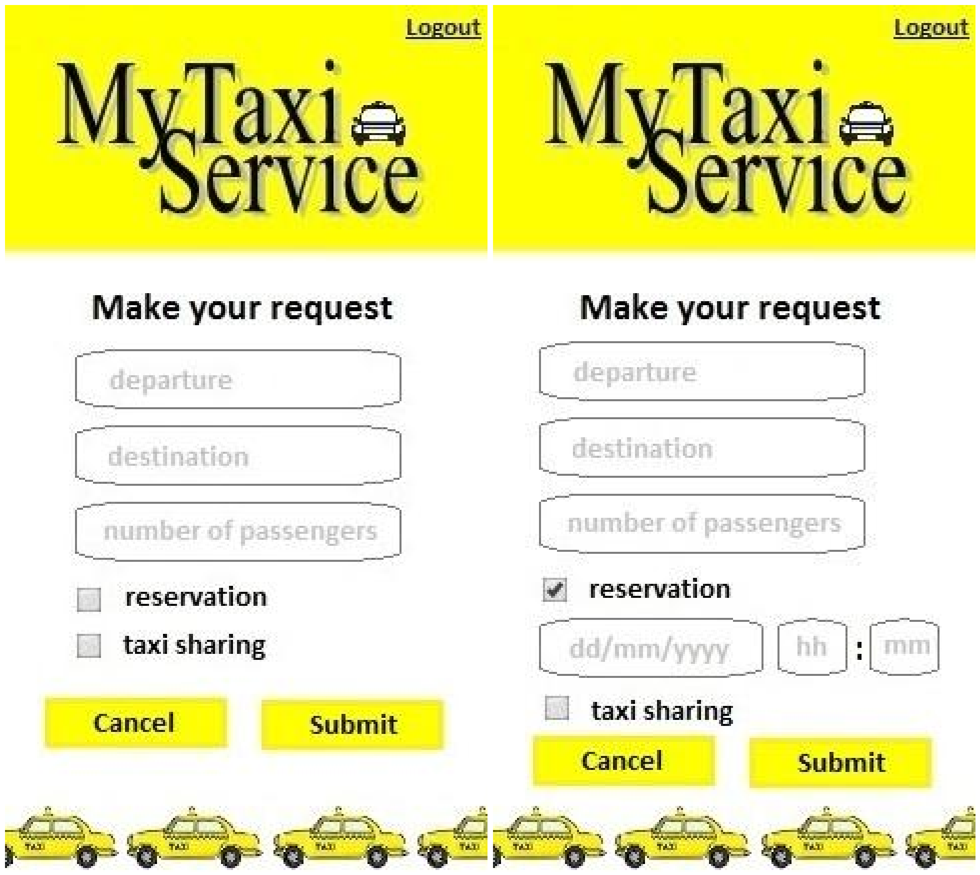


Image 6.4: Passenger profile editing

The passenger can only change his/her profile picture and change the password with a new one. The clicking on the button “Cancel” the system will return the passenger on the profile page (Image 6.3) without producing any changes otherwise clicking on the button “Save” the system will save all changes and will return on the profile page (Image 6.3). From this page the passenger can see the notifications and can logout from the system.



Images 6.5: Passenger request form

Through this form the passenger can make a request filling departure, destination and specifying if he/she want to make a simple request or a reservation (in this case is possible to choose day and time) and if he/she want to allow taxi sharing. Clicking on the button “Cancel” the system will show again to the passenger his/her profile page (Image 6.3) and clicking on “Submit” the system will add the request to the passenger’s pending requests (Image 6.6) and show the passenger his/her profile page (Image 6.3).

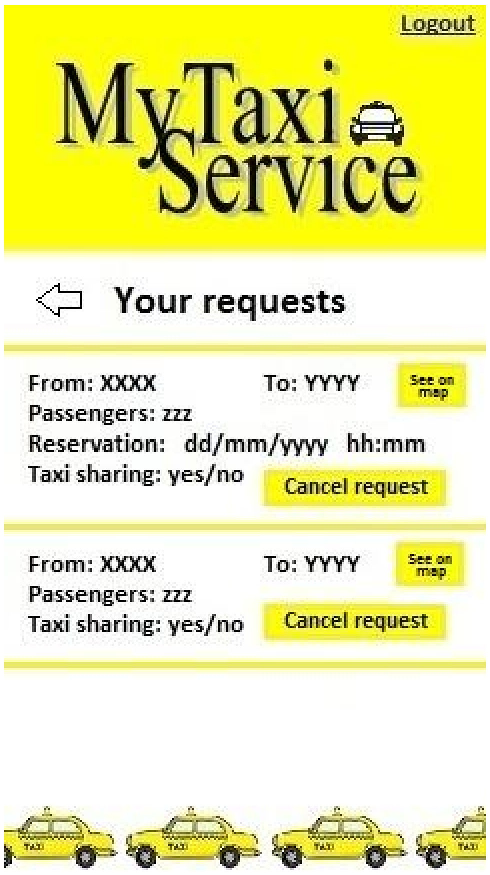


Image 6.6: Passenger's pending requests

In this page the passenger can see all the requests he/she has done and which are waiting to be evaluated or have already been accepted/ignored by the driver. Clicking on the button "Cancel request" the passenger can cancel the request and clicking on the button "See on map" he/she can see the position of the taxi driver in the map. Clicking on the arrow the system will show the passenger his/her profile page (Image 6.3).

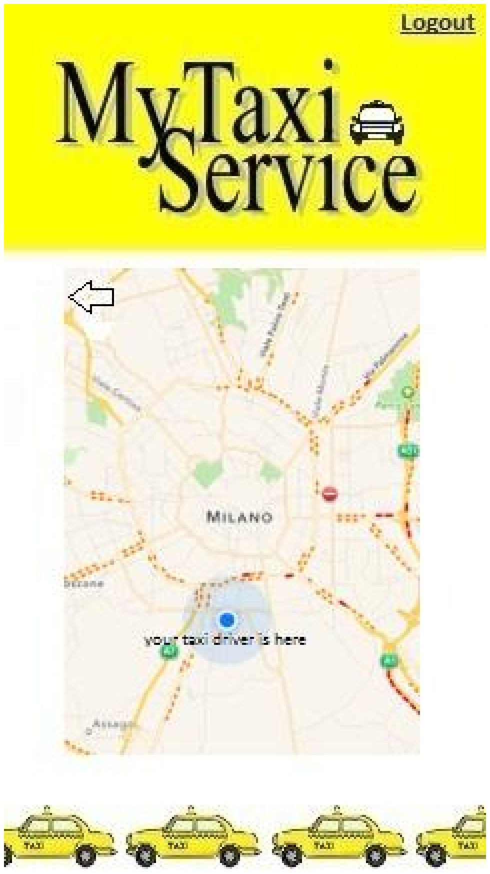


Image 6.7: Passenger map

Through this page the passenger can see where the driver is assigned to the specific request. Then he/she can return to the previous page (Image 6.6) clicking on the icon with the arrow or logout clicking on the link.

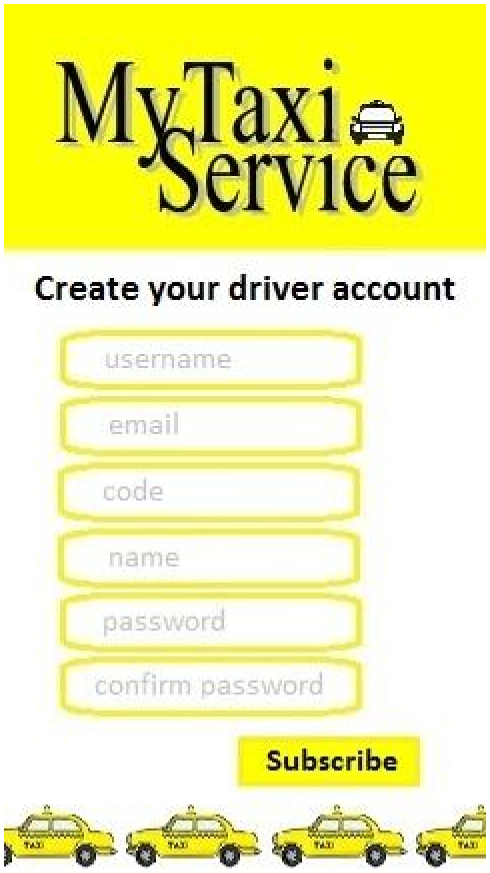


Image 6.8: Driver creation account

The driver fills in the form with all the information requested and clicking on the button “Subscribe” the system will check if username, email and code correspond to data stored in Milano’s database and the two password inserted are the same. Then the driver will see his/her new profile (Image 6.9).

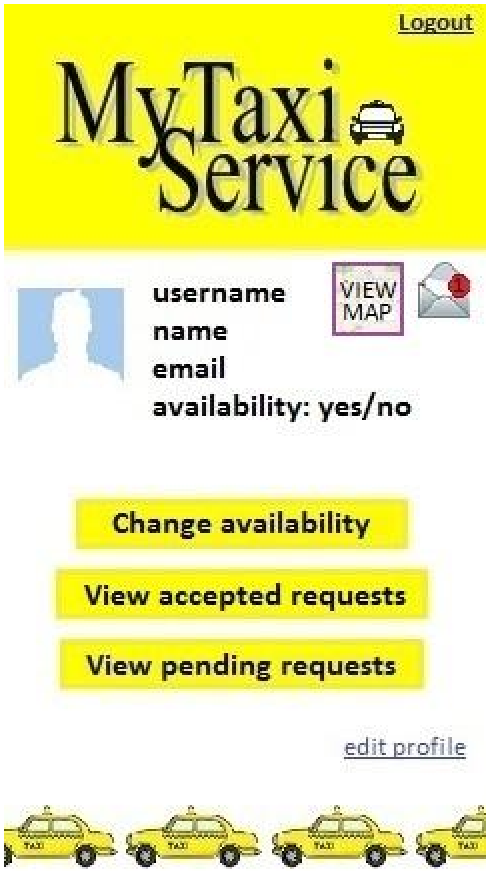


Image 6.9: Driver profile

Through his/her own profile, the driver can see his/her username, name, email, availability, notification through the envelope on the right top of the page and the map with his/her position and the destinations of the accepted requests (Image 6.10), change availability clicking on the related button (then the driver will see the page showed in Image 6.11), see accepted requests clicking on the related button (then the driver will see the page showed in Image 6.12), see pending requests waiting to be accepted by him/her clicking on the related button (then the passenger will see the page showed in Image 6.13), edit the profile clicking on the related link (Image 6.14), make the logout from the system (he/she will return on the page showed in Image 6.1).

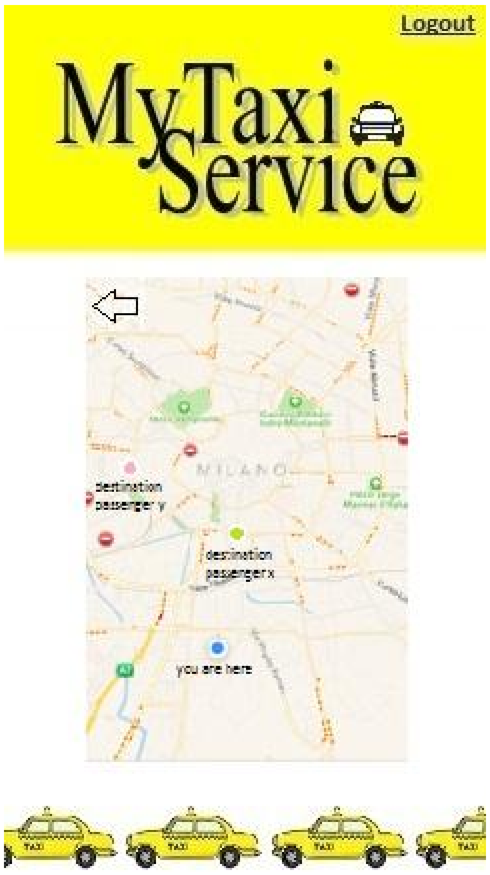


Image 6.10: Driver map

Through this page the driver can see his/her position and all the destinations of the requests already accepted. Then he/she can return to the previous page (Image 6.9) clicking on the icon with the arrow or logout clicking on the link (he/she will return on the page showed in Image 6.1).

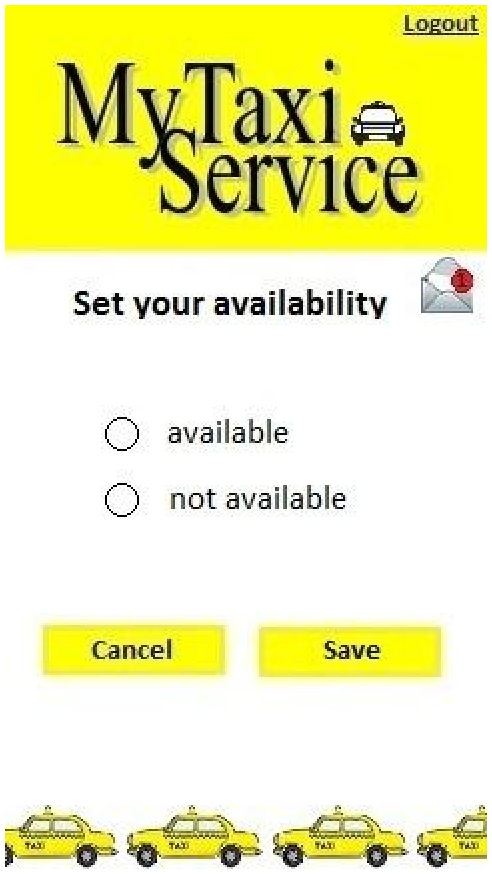
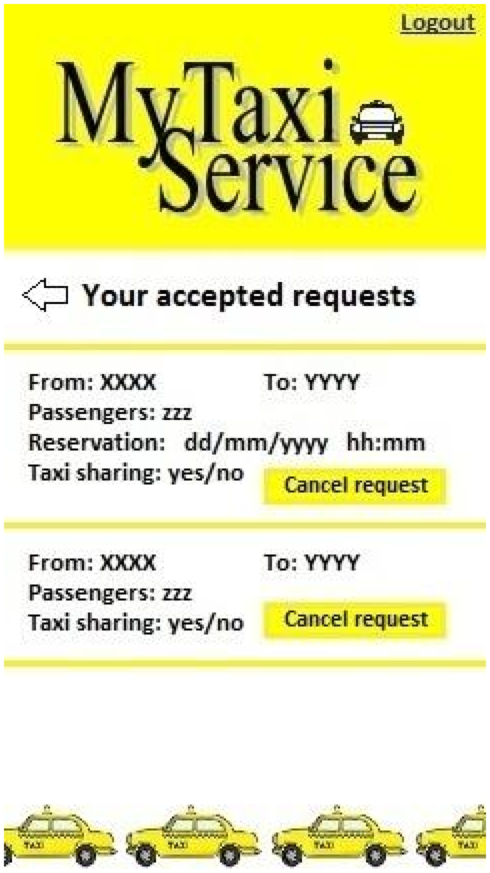


Image 6.11: Driver changing availability

Through this page the driver can change his/her availability selecting the option he/she wants to choose. Clicking on the button "Save" the system will save the changes otherwise clicking on "Cancel" the system will cancel changes. In both cases the system will return the driver to the profile page (Image 6.9). From this page the driver can make logout (then he/she will see the homepage of Image 6.1).

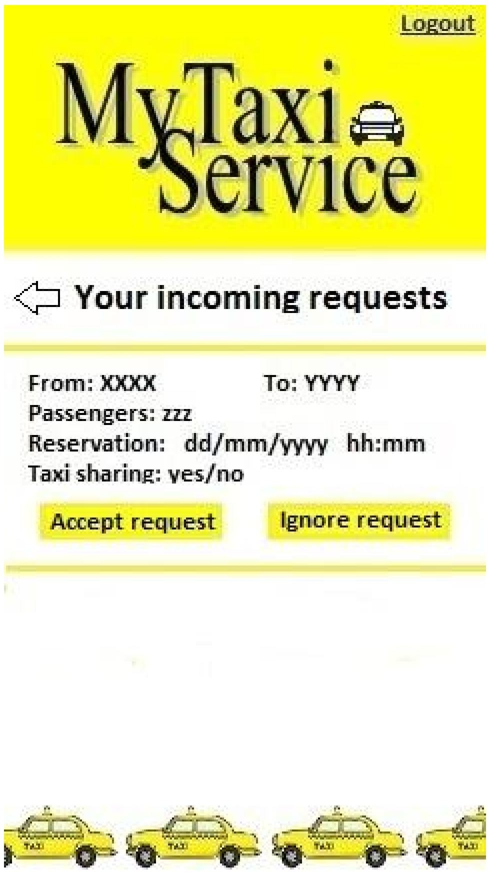


From: XXXX To: YYYY
Passengers: zzz
Reservation: dd/mm/yyyy hh:mm
Taxi sharing: yes/no

From: XXXX To: YYYY
Passengers: zzz
Taxi sharing: yes/no

Image 6.12: Driver accepted requests

In this page the driver can see all the requests he/she has already accepted. He/she can cancel each of them by clicking on the button “Cancel request”. After deleting the request the passenger will see the same page (Image 6.12). Otherwise clicking on the arrow the system will show the driver his/her profile page (Image 6.9).



From: XXXX To: YYYY
Passengers: zzz
Reservation: dd/mm/yyyy hh:mm
Taxi sharing: yes/no

Image 6.13: Driver pending requests

In this page the driver can see all his/her incoming requests. For each request the driver can choose to accept or ignore it by clicking on the related button. Clicking on one of any of the previous buttons the system will return the driver on the same page (Image 6.13). Otherwise clicking on the arrow the system will show the driver his/her profile page (Image 6.9). Driver can make logout the he will see the homepage (Image 6.1).

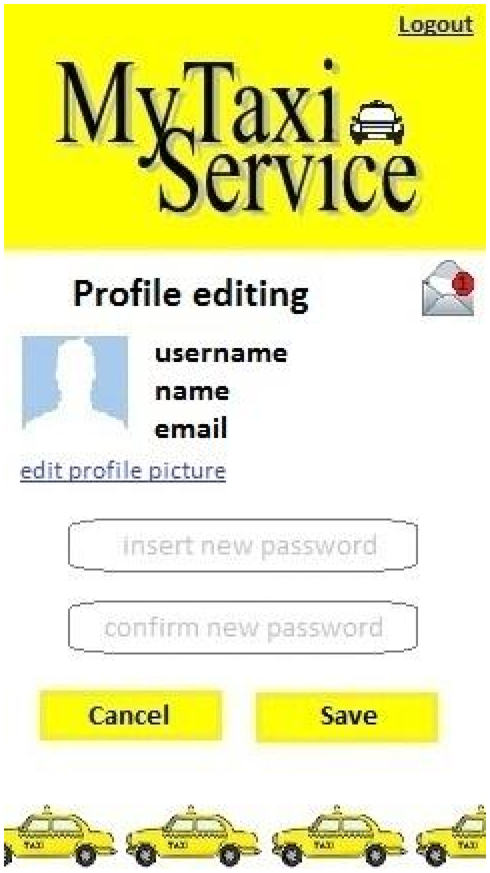


Image 6.14: Driver profile editing

The driver can only change his/her profile picture and change the password with a new one. The clicking on the button “Cancel” the system will return the driver on the profile page (Image 6.9) without producing any changes otherwise clicking on the button “Save” the system will save all changes and will return on the profile page (Image 6.9). From this page the driver can see the notifications and can logout from the system.

5. Requirements traceability

This final part of the document is intended to relate the presented architecture with the requirements in the RASD.

1.1: The system must be accessible by the passengers through the website and the mobile application

This requirement can be satisfied thanks to MTSPassengerWebController and its higher components WebBrowser and MTSPassengerWebView (in case of website) and MTSPassengerMobileController and its higher component MTSPassengerMobileView (in case of mobile application).

1.2: The system must allow passengers to create an account

This requirement can be satisfied thanks to MTSModel, MTS_DB and all their higher components on the side of the passenger.

1.3: The system must allow passengers to log in

This requirement can be satisfied thanks to MTSModel, MTS_DB and all their higher components on the side of the passenger.

1.4: The system must allow passengers to edit their account information

This requirement can be satisfied thanks to MTSModel, MTS_DB and all their higher components on the side of the passenger.

1.5: The system must provide a form in order to allow passengers to make a request

This requirement can be satisfied thanks to MapsServer and all its higher components on the side of the passenger.

1.6: The system must allow the passenger to check the current state of his request (phase of processing, driver's location, accepted request information)

This requirement can be satisfied thanks to MapsServer and all their higher components on the side of the passenger.

1.7: The system must allow passengers to cancel a request

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the passenger.

1.8: The system must inform the taxi driver (and the other passengers if necessary) when he receives a cancelation

This requirement can be satisfied thanks to EmailServer (in case is not possible to take contact with the passengers) and all its higher components on both sides (the passenger's one and the taxi driver's one).

1.9: The system must delete the request for service information, incoming request information and accepted request information ten minutes after the estimated arrival time

This requirement can be satisfied thanks to MTSModel and all its higher components on both sides (the passenger's one and the taxi driver's one).

1.10: The system must not allow any passenger to make more than one request

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the passenger.

1.11: The system will send notifications to the passenger wherever he is logged in (mobile application or web site). If he is logged in nowhere, the notification is sent via email

This requirement can be satisfied thanks to EmailServer (if he is not logged in) or MTSModel (if he is logged in) and all its higher components on the sides of the passenger.

2.1: The system must be accessible by the taxi drivers through a mobile application only

This requirement can be satisfied thanks to MTSTaxiDriverMobileController and its higher component MTSTaxiDriverMobileView.

2.2: Taxi drivers must be able to register to the application

This requirement can be satisfied thanks to MilanoGovernment and all its higher components on the side of the taxi driver.

2.3: The system must allow registered taxi drivers to login

This requirement can be satisfied thanks to MTSModel, MTS_DB and all their higher components on the side of the taxi driver.

2.4: The system must allow taxi drivers to edit their account information

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the taxi driver.

2.5: The system must provide an availability section allowing drivers to change their status

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the passenger.

3.1: When a taxi driver receives an incoming request, the system must allow him to see the incoming request information

This requirement can be satisfied thanks to MapsServer and all its higher components on the side of the taxi driver.

3.2: The system must allow taxi drivers to accept an incoming request

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the taxi driver.

3.3: When a taxi driver accepts an incoming request, the system must change is status to not available

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the taxi driver.

3.4: When a taxi driver accepts an incoming request, the system must send to the passenger the accepted request information

This requirement can be satisfied thanks to EmailServer (if passenger is not reachable otherwise) or MTSModel and all its higher components on the side of the passenger.

3.5: When a taxi driver has accepted an incoming request, the system must allow him to see in a map his current location and the passenger(s) origin(s) and destination(s)

This requirement can be satisfied thanks to MapsServer and all its higher components on the side of the taxi driver.

3.6: The system must allow taxi drivers to decline an incoming request

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the taxi driver.

3.7: When a taxi driver declines an incoming request, the system must resend it to the next available passenger

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the taxi driver.

3.8: When no available taxi drivers are found, the system must cancel the request and inform it to the passenger(s)

This requirement can be satisfied thanks to EmailServer (if passengers are not reachable otherwise) or MTSModel and all its higher components on the side of the passenger.

3.9: The system must allow taxi drivers to cancel an already accepted request

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the taxi driver.

3.10: When a taxi driver cancels an already accepted request, the system must inform it to the passenger

This requirement can be satisfied thanks to EmailServer (if passenger is not reachable otherwise) or MTSModel and all its higher components on the side of the passenger.

4.1: The system must be able to recognize the requests with correct request for service information

This requirement can be satisfied thanks to MTSModel.

4.2: The system must be able to process the recognized requests

This requirement can be satisfied thanks to MTSModel.

4.3: The system must use zone queues to manage available taxi drivers

This requirement can be satisfied thanks to MTSModel.

4.4: The system must send a processed request only to the first available taxi driver in the zone queue that corresponds to the origin of the request

This requirement can be satisfied thanks to MTSModel.

4.5: The system must use the taxi's GPS to know the position of the driver

This requirement can be satisfied thanks to MapsServer, MTSIntegration and MTSModel.

4.6: The system must use the Map service to assign a zone queue to the driver according to its position

This requirement can be satisfied thanks to MapsServer, MTSIntegration and MTSModel.

4.7: When the system has identified the zone of a just available taxi driver, it must send him to the bottom of the zone queue

This requirement can be satisfied thanks to MTSModel.

4.8: The system must refresh the zone of every available taxi driver every 5 minutes

This requirement can be satisfied thanks to MapsServer, MTSIntegration and MTSModel.

4.9: When an available taxi driver changes his zone, the system must send him to the bottom of the zone queue

This requirement can be satisfied thanks to MTSModel.

4.10: The system must send to the bottom of the zone queue those taxi drivers who decline a received request

This requirement can be satisfied thanks to MTSModel.

4.11: The system must send to the bottom of the zone queue those taxi drivers who cancel an already accepted request

This requirement can be satisfied thanks to MTSModel.

4.12: The system must send to the top of the zone queue those taxi drivers that have received a cancelation for an already accepted request

This requirement can be satisfied thanks to MTSModel.

4.13: The system must remove from of the zone queue those taxi drivers that change their status to not available

This requirement can be satisfied thanks to MTSModel.

5.1: The system must provide a form in order to allow passengers to make a request

This requirement can be satisfied thanks to MapsServer and all its higher components on the side of the passenger.

5.2: The option "Taxi sharing" will be effectively activated only if there will be other people who have requested it in the same area at the same time

This requirement can be satisfied thanks to MapsServer and all its higher components on the side of the passenger.

5.3: Passenger will receive a notification by the system in which there will also be specified if the taxi sharing option has been effectively accepted

This requirement can be satisfied thanks to EmailServer (if passenger is not reachable otherwise) or MTSModel and all its higher components on the side of the passenger.

6.1: The system must provide a form in order to allow passengers to make a request

This requirement can be satisfied thanks to MapsServer and all its higher components on the side of the passenger.

6.2: The reservation must be made by the passenger at least 2 hours before time of departure

This requirement can be satisfied thanks to MTSModel and all its higher components on the side of the passenger.

6.3: The system has to confirm the reservation to the passenger

This requirement can be satisfied thanks to EmailServer (if passenger is not reachable otherwise) or MTSModel and all its higher components on the side of the passenger.

6.4: The system allocates a taxi 10 minutes before the departure time in the requested place

This requirement can be satisfied thanks to MTSModel.

6. References

The following documents were used as source of technical definitions and general information:

- Lecture slides on Architectural styles.
- Lecture slides on Design.
- Official website of Milano Government for the taxi services fees.

7. Appendix

This section includes additional information concerning the development of the project itself.

The amount of hours that each one of the team members devoted to the elaboration of this document is:

- Ivana Salerno: 34 hours
- Alexis Rougnant: 28 hours
- Daniel Vacca: 29 hours

From the version 0.1 presented on December the 6th to this version presented on February the 2nd, we have included:

- Sections 1.3 and 1.4 concerning the Definitions and Reference documents.
- Section 6 concerning additional references.
- Section 7 concerning the worked hours.