POLITECNICO
MILANO 1863

DD

Design Document

FEDERICO DI DIO - 893730
FLAVIO DI PALO - 898541
MATTEO BELLUSCI - 898380

INDEX

1. Introduction

1.A Purpose

The aim of this Design Document is to describe software design and architecture of the “**Travlendar+**” system. The main architecture and the relationships between the modules is described using UML standards. The reader to which this document refers, is a team developer or a project manager: the document provides a solid structural overview of the system-to-be.

1.B Scope

The system aims to offer a simple and reliable activity scheduling service, which also allow

The software system is divided into three layers, which will be presented in the document. The architecture has to be easily extensible and maintainable in order to provide new functionalities.

Every component must be conveniently thin and must encapsulate a single functionality (high cohesion). The dependency between components has to be unidirectional and coupling must be avoided in order to increase the reusability of the modules.

Design patterns and architectural styles will be used for solving architectural problems in order to simplify the system comprehension and avoid misunderstanding during the implementation phase

1.C Definitions, Acronyms, Abbreviations

1.C.1 Definitions

● **Activity:** it is a scheduled appointment it could be either a Meeting (or Fixed Activity) or a Flexible Activity

● **Flexible Activity:** a particular event characterized by a fixed duration that can be scheduled in a given time slice by the system.

● **Meeting (or Fixed Activity):** a scheduled appointment characterized by a precise hour, day and destination.

● **Route:** is the path from the user's current position to the position of his Scheduled Activity

● **Sub-Route:** is a portion of the Route that is performed with a single mean of transport

● **Back-end:** term used to identify the Application server.

● **Front-end:** the components which use the application server services,namely the mobile application.

●**Web server:** the component that implements the web-based front-end. It interacts with the application server and with the users' browsers.

1.C.2 Acronyms

- DD: Design Document
- MVC: Model View Controller Pattern
- RASD: Requirement Analysis and Specification Document.
- API: Application Programming Interface
- S2B: System-to-be
- UX: User Experience
- DBMS: Database Management System.
- EJB: Enterprise JavaBean.

1.C.3 Abbreviations

- **[Gn]**: identifier of n-th Goal.
- **[Rn]:** identifier of n-th Functional Requirement.
- **[DDn]**: identifier of n-th Requirement mapped into the DD.

1.D Revision history

- **Version 1.0,** 26th november 2017

1.E Reference Documents

- *Specification Document:* Mandatory Project Assignments.pdf
  IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications.
- "Verification and Validation,part I", Michele Guerriero
- "Verification and Validation,part II", Michele Guerriero
- "Verification Tools", Michele Guerriero "Integration Testing Example Document"
- http://www.keynote.com/resources/white-papers/testing-strategies-tactics-for-mobile-applications
- *"Fundamentals of Software Engineering"*, Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli

1.F Document Structure

This document is structured in six parts:

Chapter 1: Introduction. This section provides general information about the DD document and the system to be developed.

Chapter 2: Architectural Design. This section shows the main components of the systems with their sub-components and their relationships. This section will also focus on design choices, styles, patterns and paradigms.

Chapter 3: Algorithm Design. This section will present and discuss in detail the main algorithms designed for the system.

Chapter 4: User Interface Design. This section shows how the user interface will look like and behave. The means used are mock-up and UX diagrams.

Chapter 5: Requirements Traceability. This section shows how the requirements in the RASD are satisfied by the design choices of the DD.

Chapter 6: Implementation, Integration and Test Plan. This section shows how we plan to implement the subcomponents of our system and the order in which we plan to integrate such subcomponents and test the integration.

In the last pages we presented the team effort and references.

2. Architectural Design

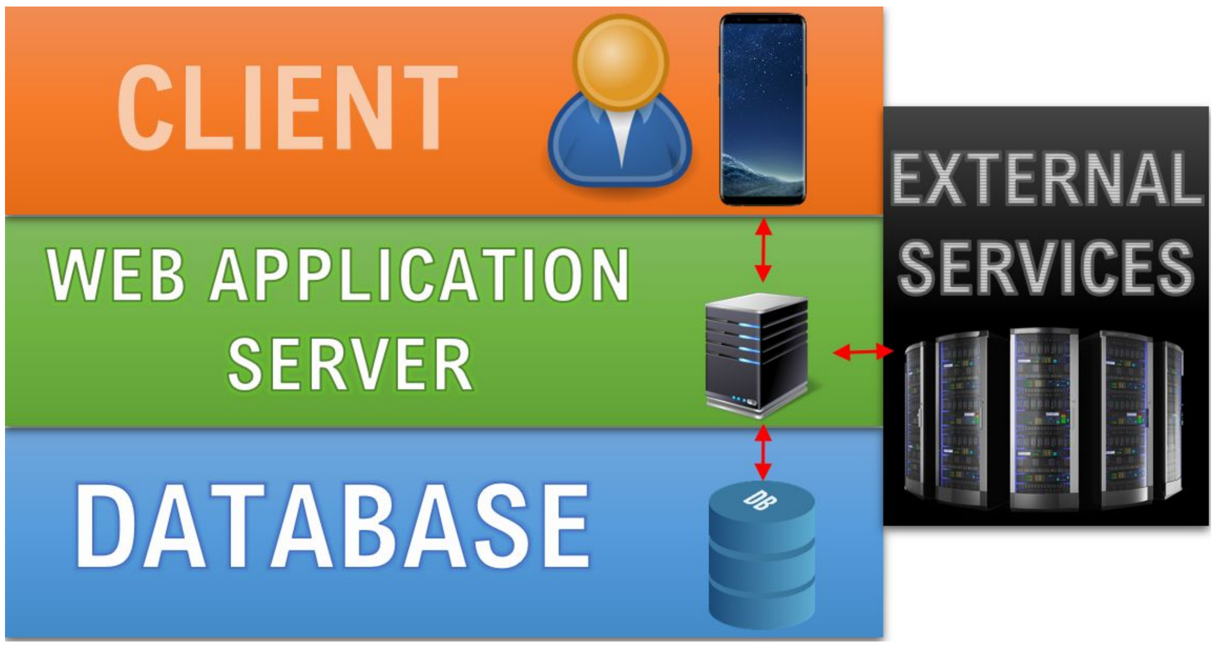
2.A. Overview

Our "Travlendar+" system presents a multi-tiered architecture structured as follows:

- **Database:** the data layer is responsible for the data storage and retrieval. It does not implement any application logic. This layer must guarantee ACID properties.

- **Web Application Server:** this layer includes business and application logic of the system. Here are processed the policies and the algorithms. This layer offers a *service-oriented* interface.

- **Mobile Application:** this presentation layer regards the mobile client, which communicates directly with the web server.

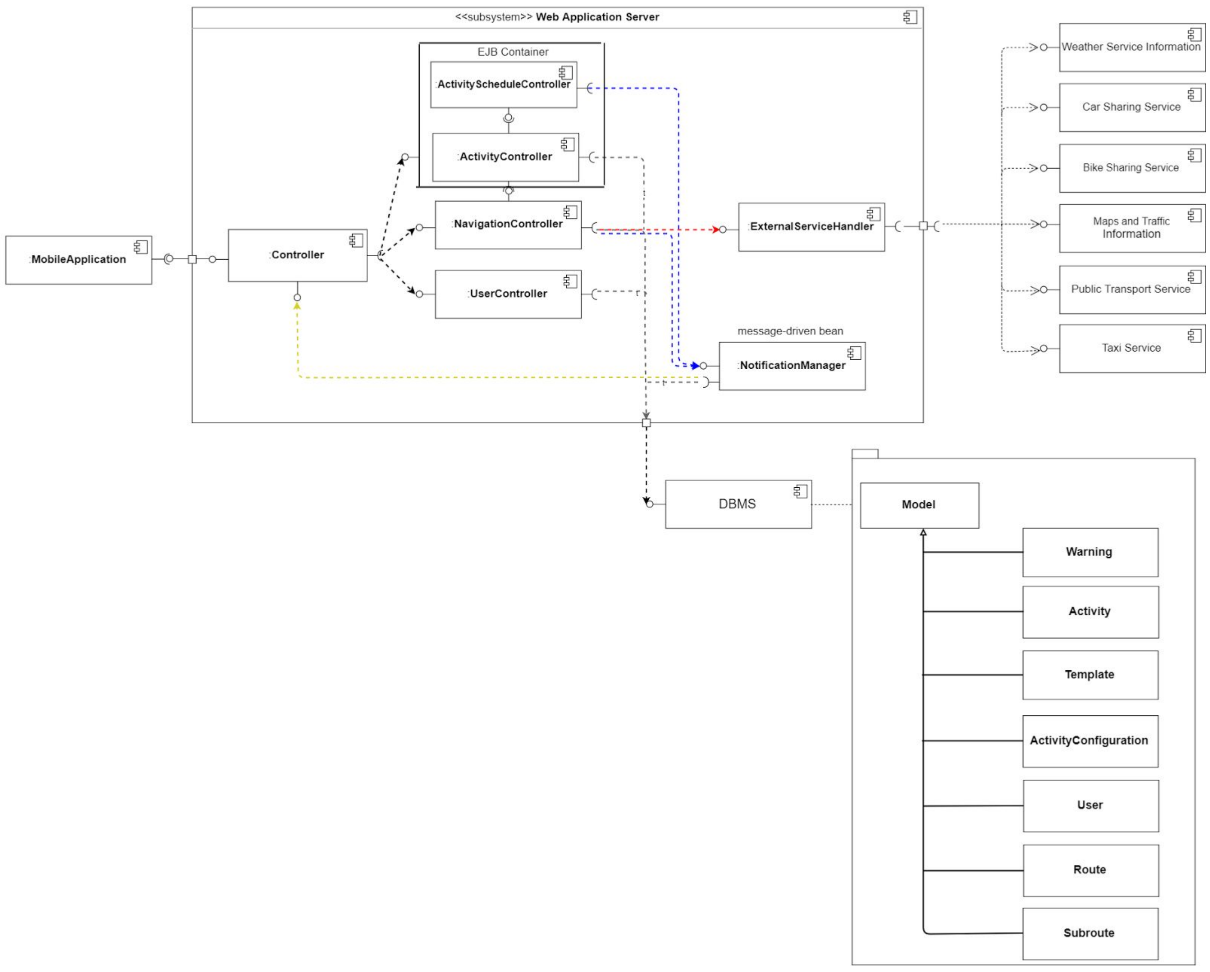


The architecture is structured in three principal layers; the diagram shows also the interaction with external services (Traffic Service, Car/Bike Sharing Service, Taxi Service, Public Transport Service, Weather Information Service).
This application is native, to take advantage from device features such as GPS and Bluetooth.

2.B. Component view

2.B.1. High Level Component Diagram

The following diagrams show the main component of the system and the interfaces through which they interact.



Database: its function is to store all datas such as credentials of the User, the informations about the activities (start time, end time ecc), the route (coordinates, subroutes ecc.), the subroutes and the templates.

Controller: it manages the incoming request from the **MobileApplication** service and addresses the request to more specific controllers when the task of the specific controller (**xController**) is finished the **Controller** component sends the response to **MobileApplication** in JSON format

UserController: it verifies all the User's informations during the access, check the request of all user related Services, Login/SignUp Functionality and the **GeneralSettings** modification are managed by this controller

NavigationController: computes the best route to reach the destination and manages the user's navigation at the destination; in order to do this it it collects informations about traffic and strikes, rental cars and bikes position and weather informations and suggesting the best means of transport for each subroute. Whenever it is needed, the user can buy public transportation tickets and rent bikes or cars.

ActivityScheduleController: it deals with scheduling the user's activities in the best possible way, eventually rescheduling them when there are substantial variations (e.g. one activity is deleted, or a new flexible activity is created).

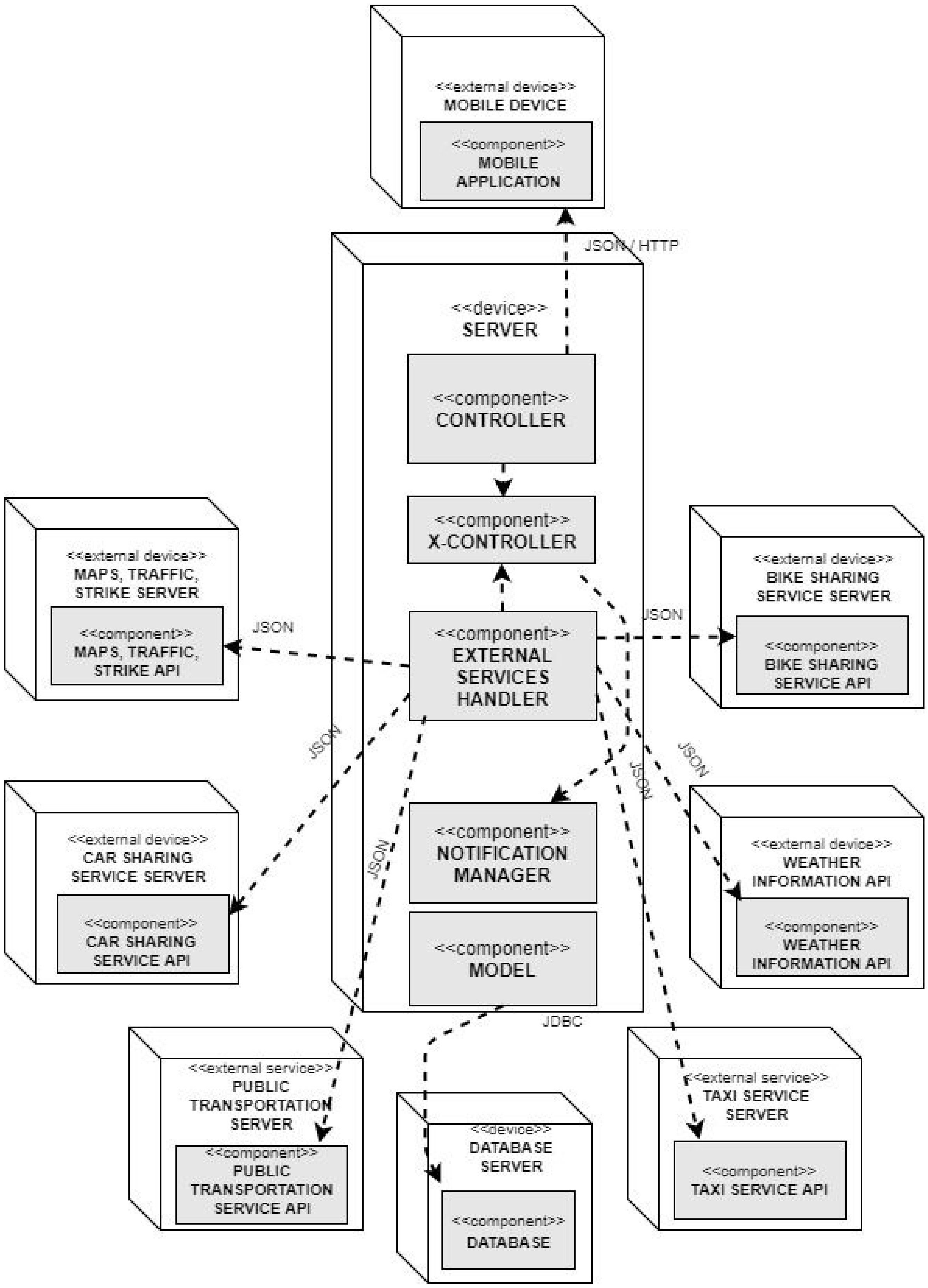
ActivityController: allow the User to create a new Activity, to edit it, to delete it and communicates with **ActivityScheduleController** and **NavigationController** in order to Manage the activity Scheduling or Show Warnings concerning an Activity. It also allow the User to create a template, edit or delete it.

NotificationManager: it manages the notifications to the User, its main function is to show a Warning to the User in particular conditions.

ExternalServicesHandler: it manages the interactions with the External Services API

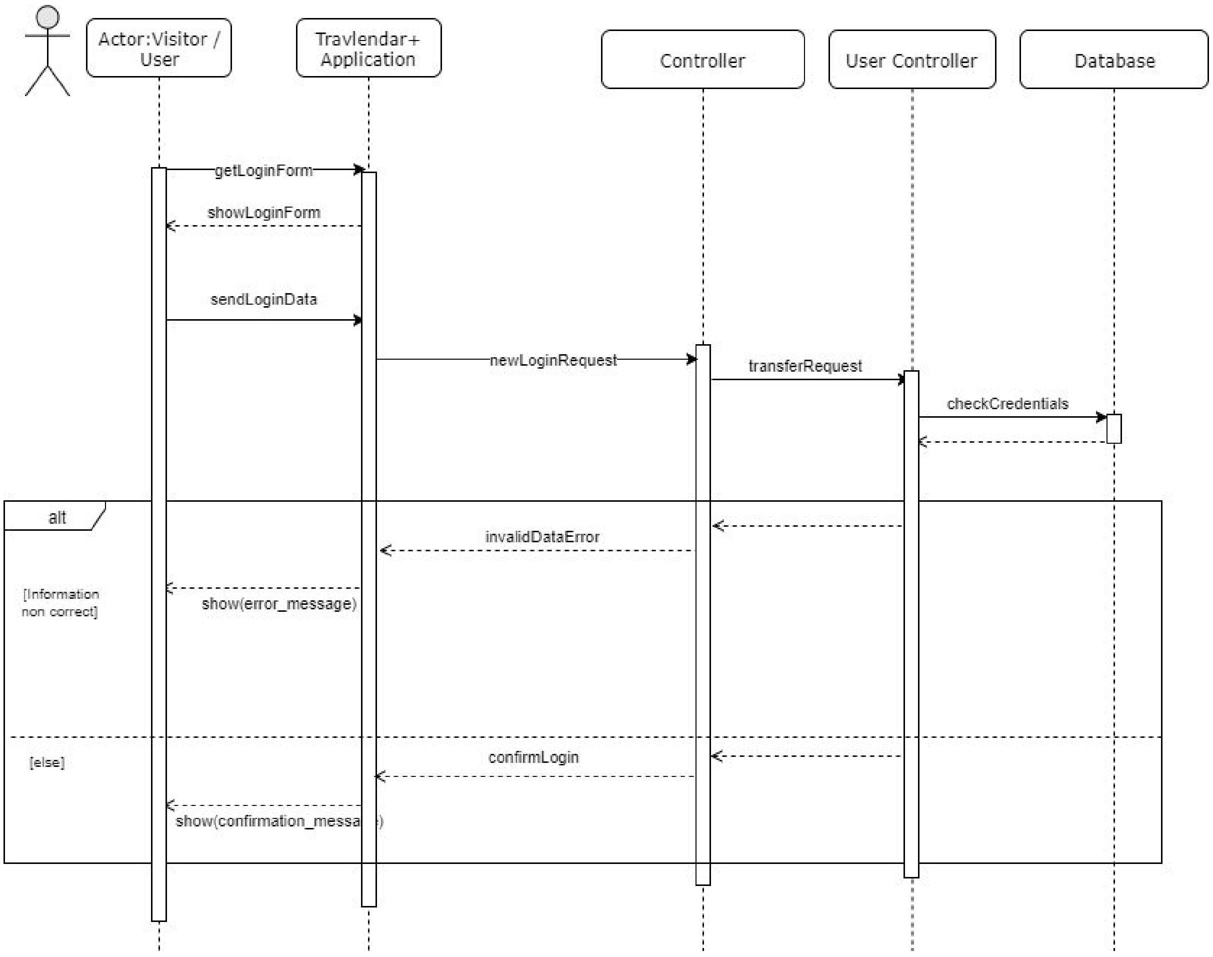
MobileApplication: is the Component which Allow the interaction with the User, it will be developed for the three principal mobile OS present on the market: iOS, Android and WindowsPhone. In order to provide the User with native-app experience and to contain cost of development and development time the Application will be written using the Xamarin Framework in order to develop a single project that can run on any mobile OS.
The request from **MobileApplication** to **Travlendar+** WebServer will be encoded in JSON format

2.C. Deployment view



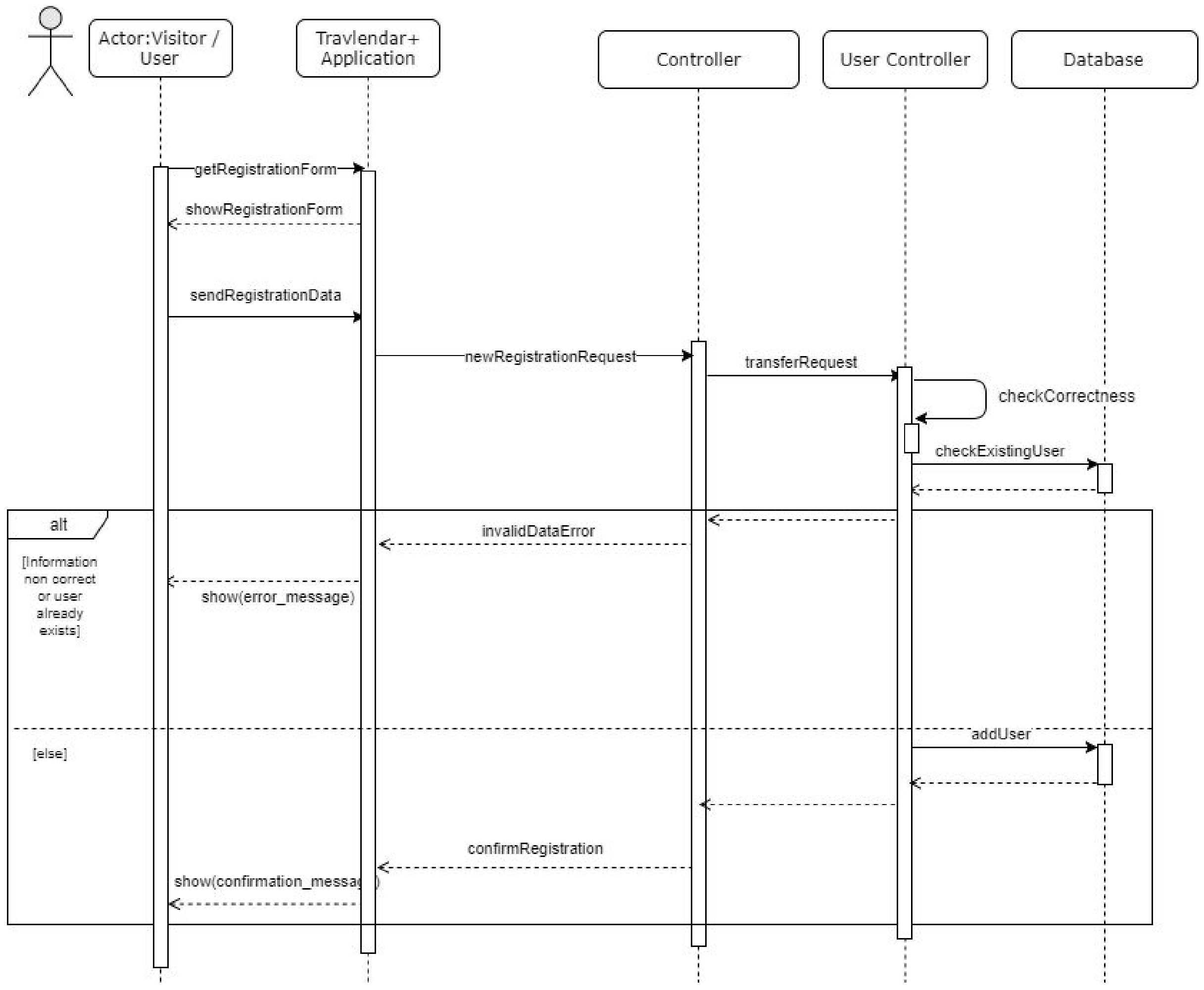
2.D. Runtime view

2.D.1. Sign In



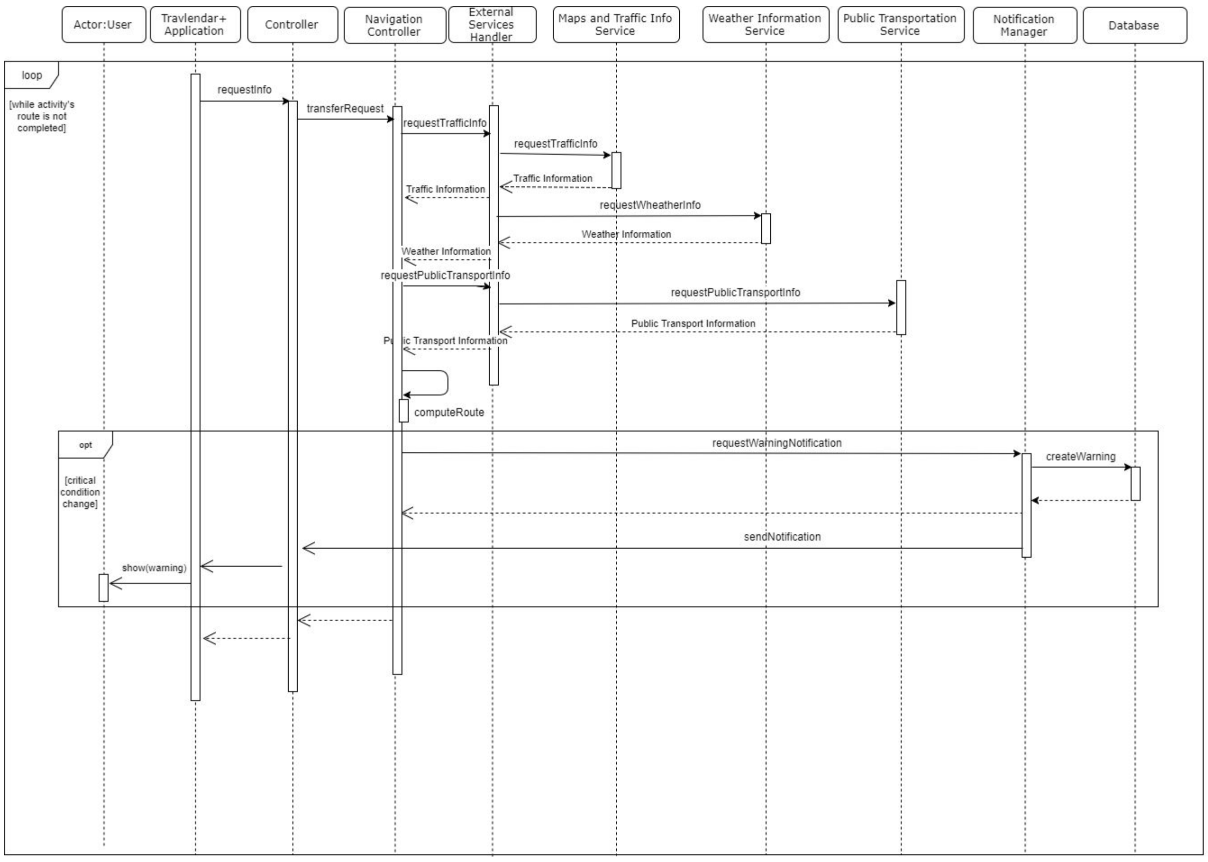
This sequence diagram describes the Log In operation. The User inserts its credentials which are sent by the Travlendar+ application to the Controller, which redirects the request to the **UserController**. This one interacts directly with the database, verifying the correctness of credentials: if these are correct, a confirmation message is displayed to the user, which is then authenticated, otherwise an error message is displayed and the User can insert and send again the data.

2.D.2. Sign Up



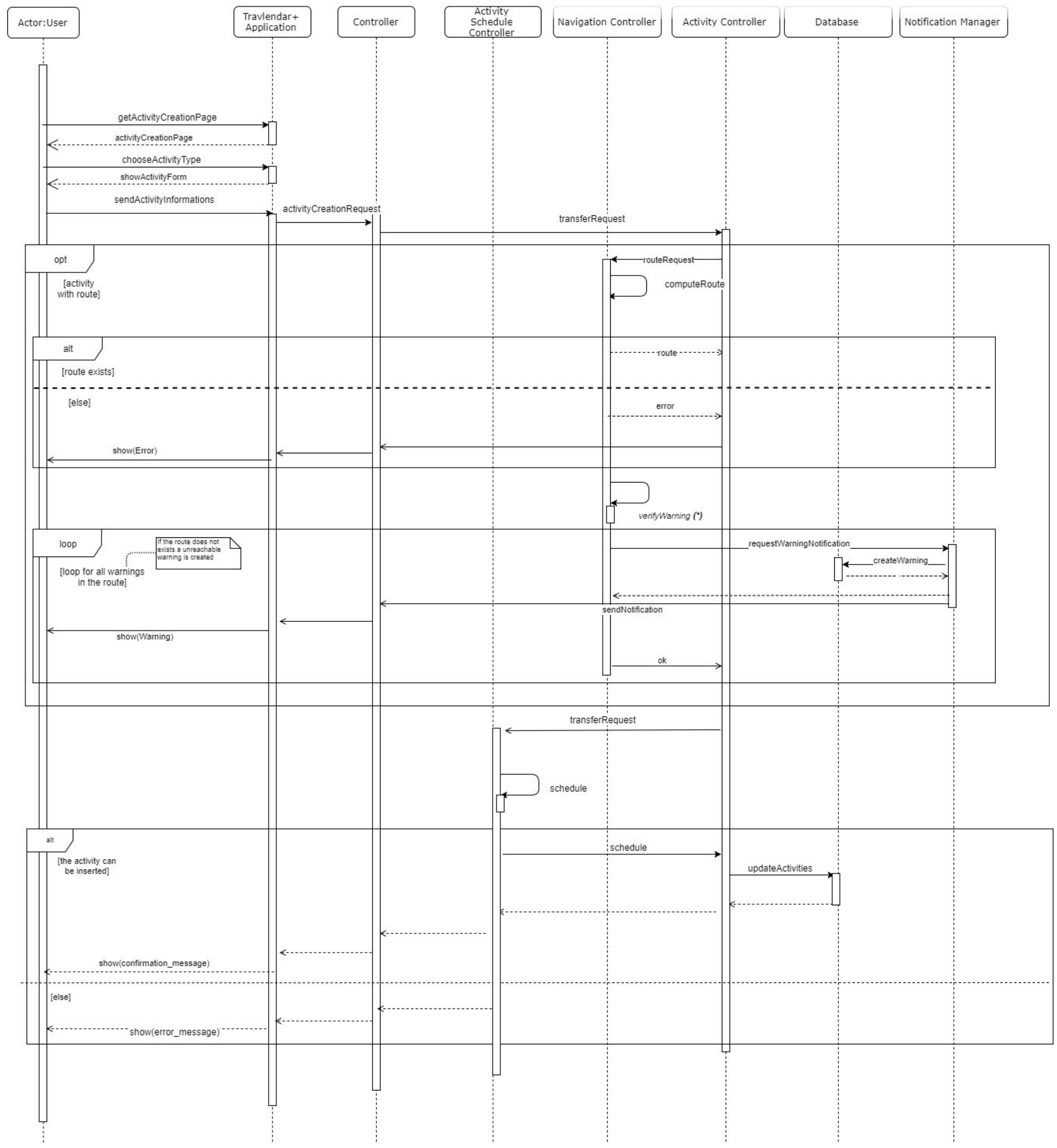
This diagram describes the SignUp operation. The User inserts its data which are sent by the Travlendar+ application to the Controller, which directs the request to the User Controller. This one checks the correctness of inserted data and verifies if there are some existing user the database. If all the data are correct, the user is added on the Database, otherwise an error message is displayed and the User can insert and send again the data.

2.D.3. Collecting External Informations and Warning


Actor:User
Travlendar+ Application
Controller
Navigation Controller
External Services Handler
Maps and Traffic Info Service
Weather Information Service
Public Transportation Service
Notification Manager
Database
loop
[while activity's route is not completed]
requestInfo
transferRequest
requestTrafficInfo
requestTrafficInfo
Traffic Information
Traffic Information
requestWheatherInfo
Weather Information
Weather Information
requestPublicTransportInfo
requestPublicTransportInfo
Public Transport Information
Public Transport Information
computeRoute
opt
[critical condition change]
requestWarningNotification
createWarning
sendNotification
show(warning)

This sequence diagram describes the operation of collecting information from outside the system and create a warning to the user. All operations are within a *loop*, representing the fact that it is a periodically repeated operation in the background as long as the activity's route is not completed. Travlendar+ mobile application, passing through the main Controller, redirects any request to the **NavigationController** which sends them to the **ExternalServicesHandler**: this one redirects to the respective external service (Weather Information Service, Public Transportation Service and Maps, Traffic Info Service).
In case a critical condition is detected, a warning is created through the Notification Manager, which sends the notification to the Navigation Controller, which redirects it to the Controller and Travlendar+ application; the latter shows the warning to the user.
A critical condition may be represent an increase in traffic, rain foreseen for the time of transport, a strike etc. the **NavigationController** detects them from the outside and allow the **NotificationManager** to create a warning for that Activity.

2.D.4. Activity Creation



(*) The procedure *verifyWarning() is used in the sequence diagram for semplicity. It includes all the operations related to collect informations from the external services as described in 2.D.3. sequence diagram (Collecting External Informations and Warning).*

This diagram describes the task of creating an Activity. The User interacts with the application by choosing the type of the Activity and entering the informations that will be sent to the Controller, which directs it to the ActivityController. If is specified a precise location, the request is forwarded to the Navigation Controller with the Activity configuration data and the best route is computed, but only if it's possible to reach the position before the Activity is over; otherwise the Route is not created. Then the Navigation

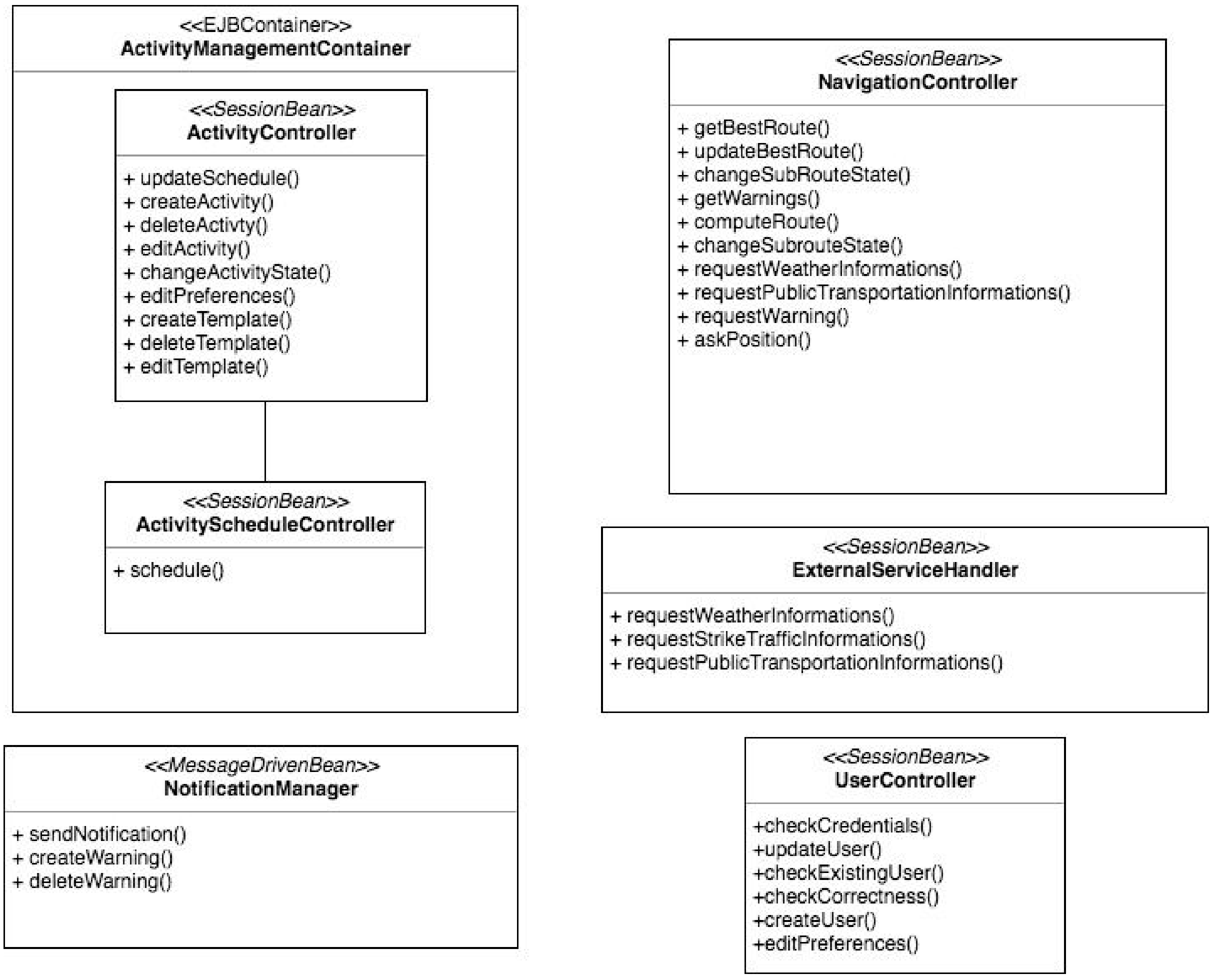
Controller checks through external services if there are any warnings, which are updated on the database. The request is forwarded to the **ActivityScheduleController** to check if it is possible to insert the Activity; the schedule procedure, if necessary, reschedules if the insertion of the Activity entails a better schedule to perform the other activities, otherwise it simply add it. If the insertion succeeds, the Activity Controller's UpdateActivities updates the Activities informations (also the ones who eventually has been edited from the rescheduling) added to the Database (in addition to any moves of other activities) and a confirmation message is sent to the User, otherwise (if the Activity start hour is before the actual time) it is rejected and an error message is displayed.

2.E. Component interfaces

As shown in the deployment view, the server has only an application part. The Web Server has to interpret the HTML requests from the client and to address them to the Application Server. This one that is the core of the server and elaborates the requests. Here we describe the Web Application interfaces and their functions, with parameters and return values.

Web Application Server

Some JEE modules are used to implement the components of Web Application Server. ActivityController and **ActivityScheduleController** are included in an *EJB Container*. ActivityController, **ActivityScheduleController**, UserController and **ExternalServiceHandler** are *Session Beans* to handle concurrent executions in different threads. The Notification Manager is a *Message-Driven Bean*, to process Notification messages asynchronously acting as a message listener.



Controller

transferRequest

This procedure plays the main role of the Controller, ie submitting requests to the most specific xControllers

User Controller

checkCredentials

This procedure controls the User credentials by comparing them with those stored in the database, to allow the User to sign in to the system. The *parameters* are the credentials, the *return value* is a boolean.

updateUser

This procedure allow the User to edit his informations, which will be stored in the database. The *parameters* are the credentials, the *return value* is a boolean.

checkExistingUser

This procedure checks that the User that is trying to be created is not already present. The *parameter* is the User to check, the *return value* is a boolean.

checkCorrectness
This procedure checks the correctness of the entered data. The *parameters* are the credentials to check, the *return value* is a boolean.

createUser
This procedure permits to add a new User, storing it to the database. The *parameter* is the User to add, the *return value* is a boolean.

editPreferences
This procedure allow the User to edit his preferences, which will be stored in the database. The *parameters* are the new preferences, the *return value* is a boolean.

Navigation Controller
computeRoute
This procedure takes care of calculating the best path by analyzing all possible ones, following the User's preferences to divide the route into subroutes, each one with the best means of transport. The parameters are the coordinates of the destination, the return value is the route.

changeSubrouteState
This procedure update Subroute’s state in real time. The *parameter* is the Subroute to update, the return value is the edited Subroute.

External Services Handler
requestWeatherInformations
This procedure requests to the external Weather Information Service informations about the weather that could influence the computation of the route. There are no parameters, the return value is composed of weather informations.

requestTrafficInformations
This procedure requests to the external Traffic aService some informations about the traffic that could influence the computation of the route. There are no parameters, the return value is composed of traffic informations.

requestPublicTransportationInformations
This procedure requests to the Public Transportation Service informations about the public transportation services (trains and metro timetables etc.) useful to compute the route.

There are no parameters, the return value is composed of public transportation informations.

Activity Controller

updateSchedule

This procedure updates the activities in the database according to a new schedule, due to tasks addition or deletion. The parameter is a list of Activity, the return value is an updated list of Activity.

createActivity

This procedure permits to add a new Activity, storing it to the database. The *parameter* is the Activity to add, the *return value* is a boolean.

deleteActivity

This procedure permits to remove an Activity, removing it from the database. The *parameter* is the Activity to remove, the *return value* is a boolean.

editActivity

This procedure permits to edit an Activity. The *parameter* is the Activity to edit, the return value is the edited Activity.

createTemplate

This procedure permits to add a new Template storing it to the database. The parameter is the Template to add, the return value is a boolean.

removeTemplate

This procedure permits to remove a Template, removing it from the database. The parameter is the Template to remove, the return value is a boolean.

editTemplate

This procedure permits to edit a Template, removing it from the database. The parameter is the Template to remove, the return value is a boolean.

changeActivityState

This procedure update Activity’s state in real time. The *parameter* is the Activity to update, the return value is the edited Activity.

editPreferences

This procedure allow the User to edit all preferences of an individual Activity. The *parameters* are the new preferences and the Activity, the *return value* is a boolean.

Activity Schedule Controller

schedule

This procedure reorganizes activities when creating or removing one of them. The parameter is a list of Activities (the one to be reorganized), the *return value* is also a list of Activities (the one that has been reorganized).

Notification Manager

sendNotification

This procedure sends a Notification to the Controller, that directs it to the application which shows it to the User. There are no parameters in input, the return value is a Warning.

createWarning

This procedure creates a Warning and stores it on the Database to the respective Activities. There are no parameters in input, the return value is a Warning.

2.F. Selected architectural styles and patterns

2.F.1 RESTful API

Our system is characterized by three-tier architecture, a client-server software architecture pattern in which the three levels are:

- *Data Tier*, the layer where the informations are stored.
- *Presentation Tier*, the user interface characterized by the mobile application (client side).
- *Business Logic Tier* that is related to application coordination, processes and logic decisions, calculations and commands.

The communication between Presentation Tier and Business Logic is handled by devoloping **RESTful APIs**, which represents an architectural style for distributed systems that uses HTTP protocol to operate on informations, data and resources from client to server and returns content through JSON.
JSON is used to represent the simple structure of response and HTTP included operations such as GET, POST, PUT, DELETE and so on.

An example of HTTP commands:

/activities
GET: list of all Activities
POST: create a new Activity
PUT: update all Activities
DELETE: delete all activities update

/activities/00001
GET: show the specific Activity
POST: error if exists
PUT: update the Activity, else error
DELETE: delete the corresponding Activity

/activities/00001/01
GET: show the specific Route if exists, else error
POST: error if exists
PUT: error
DELETE: delete the corresponding Route

2.F.2. Client Server

The client & server style is used in our design, in particular the MobileApplication act as a ThinClient of our WebServer, the MobileApplication only manages the presentation layer while all the Business Logic is processed on the WebServer.
The complete separation of the application server with the MobileApplication is useful because if the application has a critic failure or the User device is destroyed, lost or stolen the User can always retrieve his data logging from any other device with his username and password.

2.F.3 Thin Client

The thin client paradigm is implemented with relation to the interaction between user's machine and the system, in developing the MobileApplication we have chosen to design it as Thin Client. Having a thin client in our case is an advantage because all the application logic is on the application server, which has sufficient computing power and is able to manage concurrency issue efficiently.

2.F.4 Model-View-Controller

The WebServer and the MobileApplication are built following the Model-View-Controller design pattern. MVC is the design pattern of choice because it is the most common and the most convenient pattern used with object-oriented languages (including Java) dealing with complex application, and allows to design software with the separation of concerns principle in mind.

2.H. Other Design Decisions

2.H.1 Maps

The system uses an external service, Google Maps, to offload all the geolocalization, distance calculation and map visualization processes. The reasons of this choice are the following:
• manually developing maps for each city is not a viable solution due to the tremendous effort of coding and data collection required;
• Google Maps is a well-established, tested and reliable software component already used by millions of people around the world;
• Google Maps offers APIs, enabling programmatic access to its features;
• Google Maps can be used both on the server side (calculation, shortest paths, traffic, incident reporting) and on the client side (map visualization);
• the users feel comfortable with a software component they know and use everyday.

2.H.2 Availability and redundancy

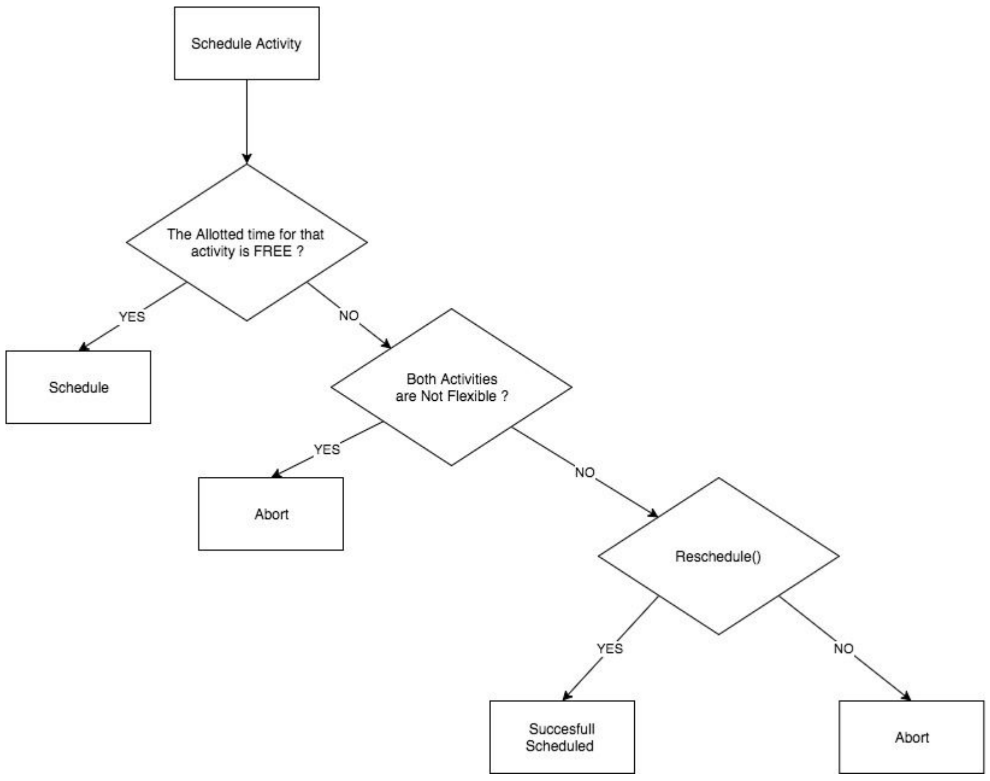
In the RASD (section 3.E.2) we stated that our system has an availability of 99.4%, This means approximately 2 days of downtime per year.
We did not specify a greater availability because the architecture we designed is not redundant: the tiers are not replicated at any level. In future implementations, replication can be added (for example, at the database and web tiers), improving the availability.

3. Algorithm Design

In the following part we will describe the most important algorithms in **Travlendar+** system.

3.A Activity Scheduling

In the following part we describe the algorithm used to schedule an Activity.



The core of the algorithm is the Reschedule Method that is called when a Flexible activity and a non flexible activity are in conflict.

in order to reschedule the activities **Travlendar+** will use the following algorithm we will see if a new schedule is possible in order to schedule all the activity or if the insertion of the activity has to be aborted.

If the operation is aborted a Warning is to the User.

Reschedule():

1. Schedule all Non Flexible activities (Meetings).
2. From the flexible activity list we define a priority:
   for each activity in the list we compute:
   $flexibility = startingTravelingDate + travelDuration + activity.durationInMinutes - startingDateOfNextScheduledActivity$
3. we insert the in our scheduling the activities in our list that have the smallest flexibility starting from their starting date.
4. if two activities have the same flexibility we schedule first the activity with the lowest starting date

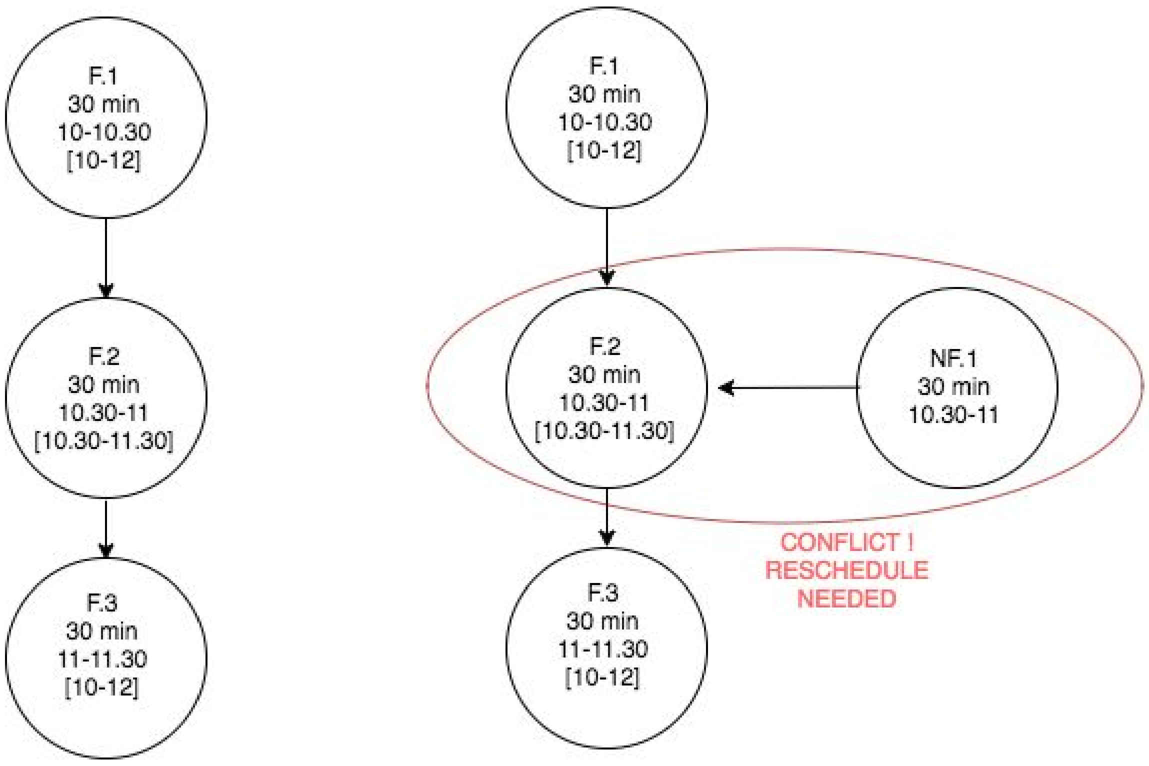
if one is not possible to schedule an Activity in the activity list then the system aborts the scheduling operation and a Warning is signaled to the User.

if the new Scheduling is possible the Reschedule algorithm returns a list of the activity following the scheduling that has been found.

EXAMPLE

Current Scheduling:

TIME	ACTIVITY NAME
10-10.30	F.1
10.30-11	F.2
11-11.30	F.3
11.30-12	



In the first step of the rescheduling we place the non Flexible Activity first.

Rescheduling Step 1

TIME	ACTIVITY NAME
10-10.30	

10.30-11	
11-11.30	NF.1
11.30-12	

Then we consider that F.2 is the less Flexible Activity and so we schedule it first.

Rescheduling Step 2

TIME	ACTIVITY NAME
10-10.30	
10.30-11	F.2
11-11.30	NF.1
11.30-12	

the other two activities have the same scheduling interval and the same scheduling interval, so they can be placed in each order, (it have to take account of F.3)

Rescheduling Step 3

TIME	ACTIVITY NAME
10-10.30	F.3
10.30-11	F.2
11-11.30	NF.1
11.30-12	

Rescheduling Step 4

TIME	ACTIVITY NAME
10-10.30	F.3
10.30-11	F.2
11-11.30	NF.1
11.30-12	F.1

3.B Compute Best Route

The system has to compute the best route considering Public Transport Strikes and Traffic Info, we will develop our algorithm on the Google Maps Directions API. The Google API takes in account Public Transport Strikes and Traffic Info and give us the possibility to take in account four means of transport:

1. Walk
2. Public transport
3. Car
4. Bike

This API suggests the User a destination using only **one** of this mean of transport for the whole path.
We want to enrich this algorithm considering that the user can save time performing a track that mixes up various means of transport.

Our algorithm will also take in account the possibility of using a shared car or bike also considering the time user needs to reach the shared mean.

The idea of the following algorithm is to check at each sub-path if there is another option, among the user preferred means to reach that activity, that can decrease the time of travel.

```
$track \leftarrow []$;
**Function** $computeRoute\ (coordinates_a, coordinates_b)$
    $path \leftarrow getBestOption(coordinates_a, coordinates_b)$
    $track.append(path[0]);$
    **if** $path[0].destination == coordinate_b$ **then**
        $return;$
    **else**
        $computeRoute(path[0].destination, coordinates_b)$
    **end**
```

track: is a list of subpath ad it is the track that the user will effectively follow
path: is a list of **subpath** that is returned by getBestOption.

The **subpath** contained in the list has a destination field that contains the coordinates of the subpath.

```
Function $getBestOption\ (coordinates_a, coordinates_b)$
    foreach $preference \in activity.preferences$ do
        $newPossibility =$
          $apiWrapper((coordinates_a, coordinates_b), preference)$
        $possibilities.append(newPossibility)$
    end
    $possibilities \leftarrow possibilities.sortByDuration()$
    $return\ possibilities[0];$
```

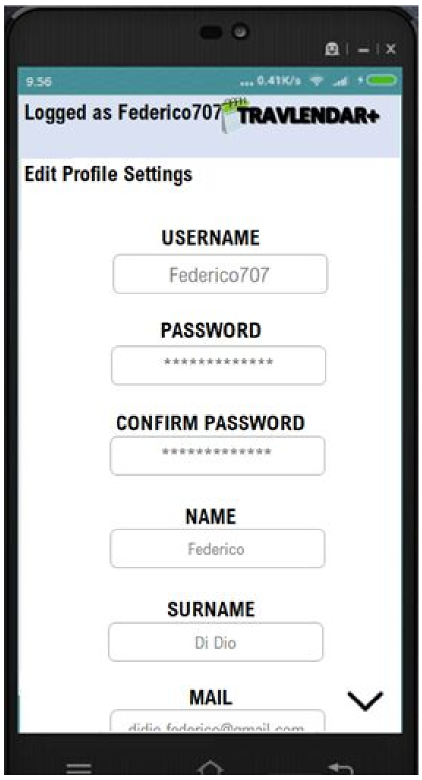
activity.preferences is a list of preferences for that activity that are expressed by the user **possibilities** is a list of **path.**

APIWrapper is a method used to interact with API when user preferences includes 'car_sharing' or 'bike_sharing' a first call to APIs of the sharing provider is made in order to locate nearest mean of transport and then the duration of the subroute is computed taking in account the time to reach the shared mean location (using Google APIs) and the time for reaching our destination with that mean of transport(using Google APIs).

4.User Interface Design

4.A Mock-up

The main mock-ups have already been presented in the Requirements Analysis and Specification Document. Below we have included some extensions.



USERNAME: Federico707

PASSWORD: *************

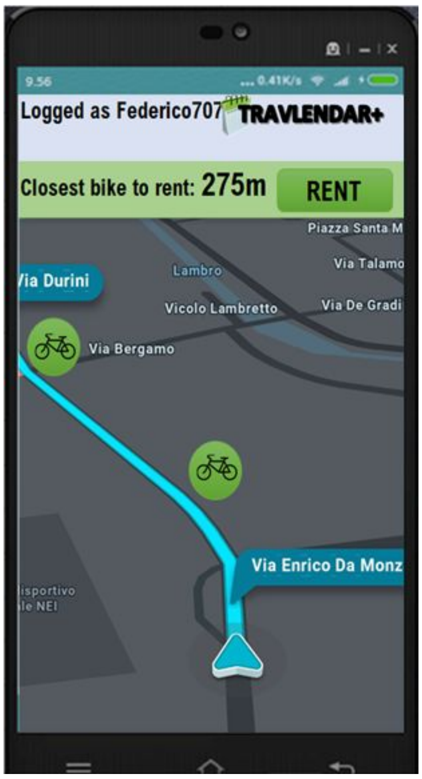
CONFIRM PASSWORD: *************

NAME: Federico

SURNAME: Di Dio

MAIL

In this mock-up is shown the functionality of editing profile settings.

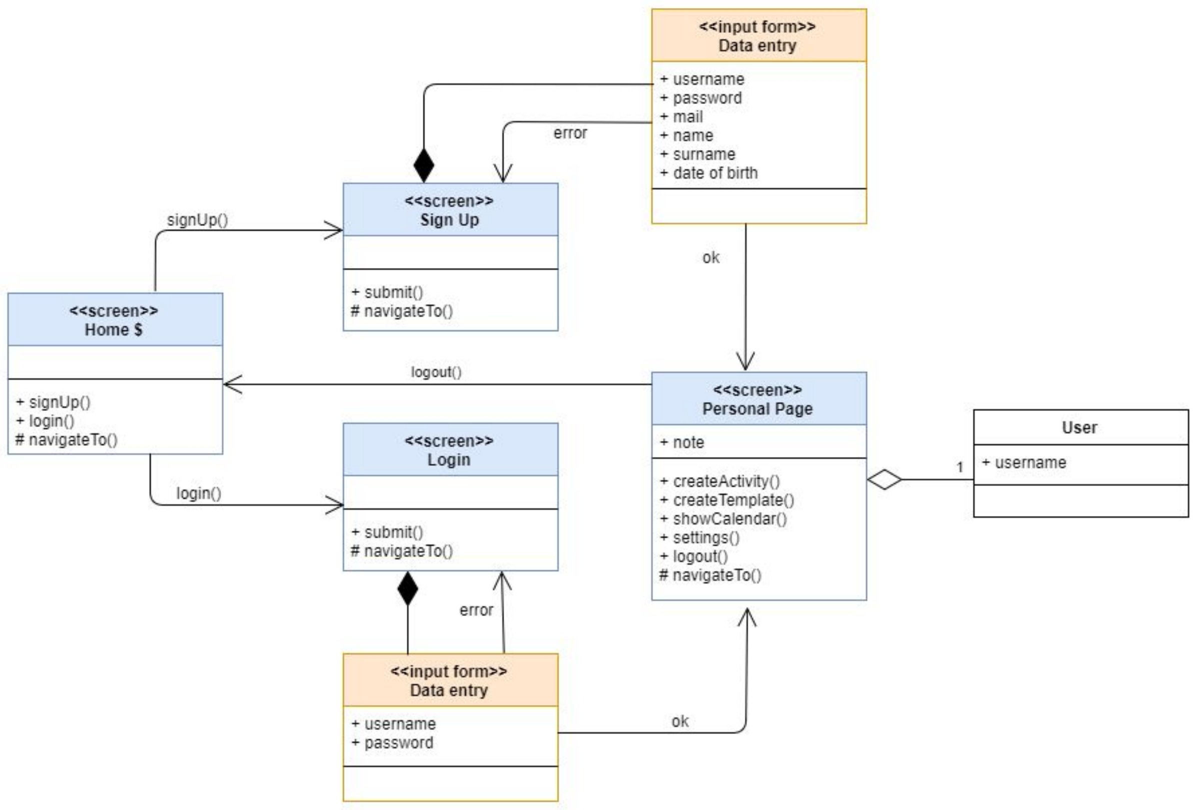


In this mock-up is shown the possibility to rent a bike during the Navigation.

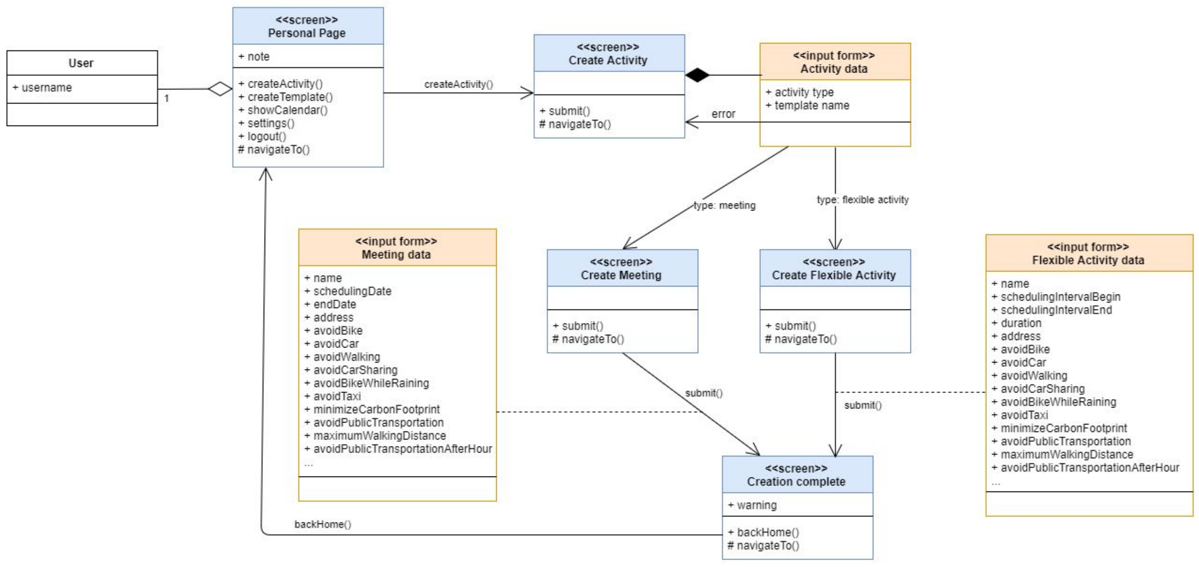
4.B UX Diagrams

The following User Experience (UX) diagrams show how the User (or Visitor for Sign Up and Login) can interact with the application. The diagrams show a view of the various pages and screens and how the user can access them to perform their actions. The various diagrams are consistent with the mock-ups just shown. Some pages contain information about the user, often just the username. As usual pages that contain information stored in databases have been highlighted as pages with attributes.

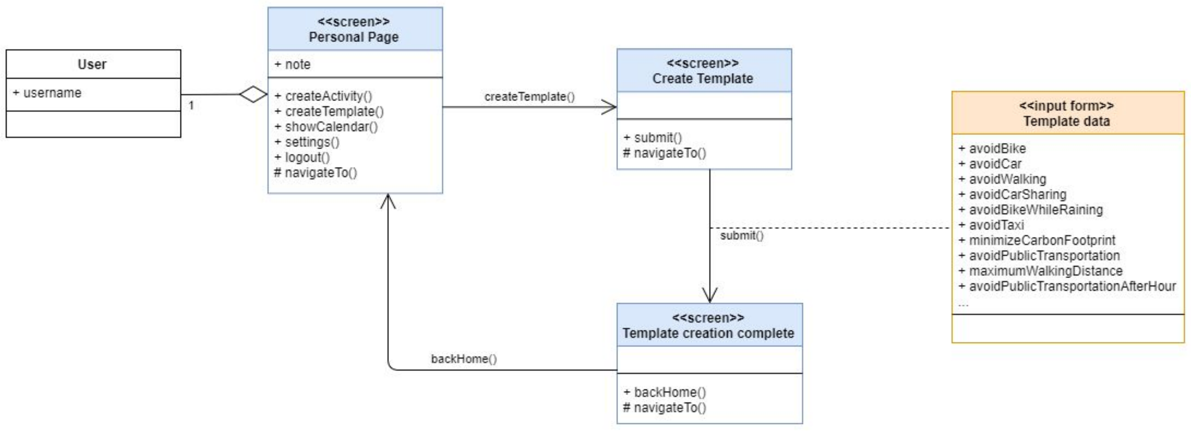
4.B.1 Sign up, Login and Logout UX Diagram



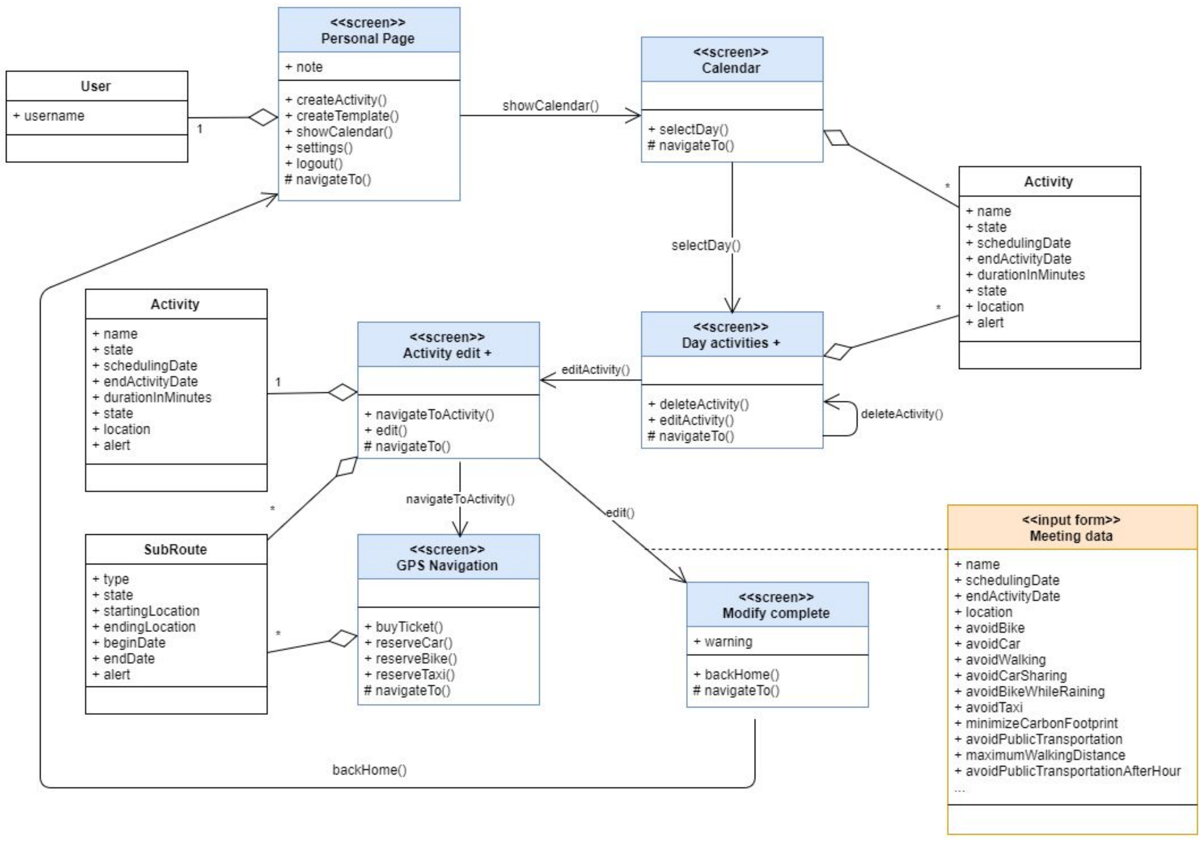
4.B.2 Create activity UX Diagram



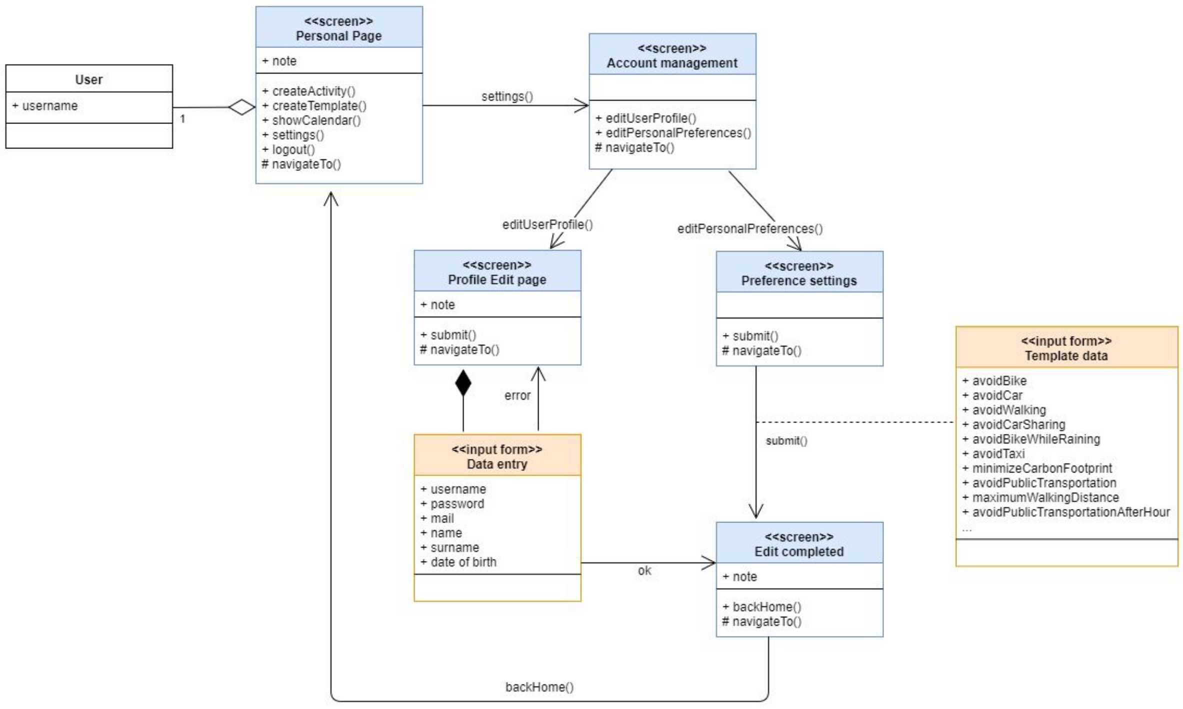
4.B.3 Create template UX Diagram



4.B.4 Navigate to activity and modify activity UX Diagram



4.B.5 Settings edit UX Diagram



5. Requirements Traceability

Here are described how the main goals and functional requirements are achieved in practice by Travlerndar+

[G1] Allow a Visitor to become a User with a personal account

[R 1.1] The User must be able to Sign Up to Travlendar+ by inserting their personal information and email address.

[DD.1.1.1] The **UserController** component inserts User stores data into **database**.

[R 1.2] During the Sign Up phase the System must verify that email address is not yet registered on the DBMS

[DD.1.2.1] The **UserController** component and Interface verifies through the **database** that the email address is not yet registered.

[R 1.3] The Visitor must be able to Log In from Travlendar+.

[DD.1.1.3] The **UserController** component and Interface verifies through the **database** that the email address is stored and that the password is correct.

[R 1.4] The User must be able to Log Out from Travlendar+.

[DD.1.1.3] The **UserController** component and Interface allows the user to log out from Travlendar+.

[R 1.5] During Log In phase the system must verify that the entered username and password are already registered into the DBMS and correct to allow the user to access.

[DD.1.3.1] The **UserController** component and Interface verifies through the **database** that the email address is stored and that the password is correct.

[R 1.6] The System must store User default preferences and general settings on his account

[DD.1.6.1] The **UserController** component and Interface stores in the **database** preferences and general settings for each User.

[R 1.7] The User must be able to change his default preferences and general settings in any time

[DD.1.7.1] The **UserController** component and Interface allows the user to store his default preferences and general settings in the **database**.

[G2] Allow a User to manage his Activities

[R 2.1] The system must allow the User to create Activities only if the scheduled date and time are after the creation time

[DD 2.1.1] The **ActivityController** component and Interface checks if the scheduling of the activity happens after the creation date

[R 2.2] The system must verify that the scheduled Activity is reachable on time

[DD 2.2.1] The **ActivityScheduleController** component (**Activity Scheduling Algorithm**) and Interface and the Navigation Controller (**Compute Route Algorithm**) check if the scheduling of the Activity is reachable on time considering also other present activities.

[R 2.3] The system must show a warning if a Scheduled Activity is not reachable on time

[DD 2.3.1] The **ActivityScheduleController** component and Interface using the **Activity Scheduling Algorithm** throws a warning if an Activity is unreachable through the Notification Manager.

[R 2.4] The system must allow the User to delete Activity in any moment

[DD.2.4.1] The **ActivityController** component and Interfaces delete activity from the **Database** basing on User input.
[DD.2.4.1] The **ActivityController** component and Interfaces after the activity deletion signals to the **ActivityScheduleController** to reschedule the activities in order to take advantage of the free time created by the deletion

[R 2.5] The system must allow the User to edit Activity preferences and settings in any moment

[DD.2.5.1] The **ActivityController** component and Interfaces allow the user to edit the activitySettings and stores them into the Database
[DD.2.5.2] The **ActivityController** component and Interfaces after the activity setting modification signals to the **ActivityScheduleController** to reschedule the activities in order to take advantage of the possible free time created by the modification of an Activity

[G3] The User has to be warned if a scheduled Activity becomes unreachable in the accounted time.

[R 3.1] The System must verify that the scheduled Activity is reachable on time considering both Public Transportation Strikes and Weather Conditions

[DD 3.1.1] The **ActivityController** component and Interfaces interact with the **NavigationController** in order to know if a given activity is reachable in time or not.
[DD.3.1.2] NavigationController uses **Weather Information Service** APIs, **Public Transportation** (for eventual strikes) APIs and **Map, Traffic Information Service** APIs in order to compute the time user will use to reach an activity.
[DD.3.2.1] NavigationController interacts with **Public Transportation** APIs.

[R 3.2] The System must correctly handle Public Transportation APIs

[DD.3.2.1] NavigationController interacts with **Public Transportation** APIs.

[R 3.3] The System must correctly handle Weather Information APIs

[DD.3.2.1] NavigationController interacts with **Weather Information Service** APIs.

[R 3.4] The System must show a warning to the User If the scheduled Activity is not reachable on time.

[DD 3.1.1] The **ActivityController** component and Interfaces interact with the **NavigationController** in order to know if a given activity is reachable in time or not .
[DD 3.4.1] The **ActivityController** component throws a warning to the User if the a scheduled activity is not reachable on time.

[G4] Allow a user to choose his Preferences to reach the Activity

[R 4.1] The User must be able to select which means of transport avoid for reaching that particular activity

[DD.4.1.1] The **ActivityController** component and Interfaces stores into the **Database** a **ActivityConfiguration** object for each **Activity**.

[DD.4.1.2] the **ActivityConfiguration** object has boolean field that represent what means of transport the User have chosen to reach that activity.

[DD.4.1.3] The **NavigationController** component using the **ComputeBestRoute** algorithm takes in account **ActivityConfiguration** when computing the best route that the User have to follow

[R 4.2] The User could select if the wants to avoid bike while raining

[DD.4.1.1] The **ActivityController** component and Interfaces stores into the **Database** a **ActivityConfiguration** object for each **Activity**

[DD.4.2.1] the **ActivityConfiguration** object has boolean field that represent the choice of the user about avoiding bike while raining.

[DD.4.1.3] The **NavigationController** component using the **ComputeBestRoute** algorithm takes in account **ActivityConfiguration** when computing the best route that the User have to follow

[R 4.3] The User could select if the wants to choose the route that minimizes the Carbon Footprint

[DD.4.1.1] The **ActivityController** component and Interfaces stores into the **Database** a **ActivityConfiguration** object for each **Activity**

[DD.4.3.1] the **ActivityConfiguration** object has boolean field that represent if the User have chosen to minimize the carbon footprint for that Activity.

[DD.4.1.3] The **NavigationController** component, using the **ComputeRoute** algorithm takes in account **ActivityConfiguration** when computing the best route that the User have to follow.

[R 4.4] The User can select the maximum walking distance he wants to perform for each route

[DD.4.1.1] The **ActivityController** component and Interfaces stores into the **Database** a **ActivityConfiguration** object for each **Activity**.

[DD.4.4.1] The **ActivityConfiguration** object has an integer field that represents the maximum Walking distance the wants to perform for each route.

[DD.4.1.3] The **NavigationController** component using the **ComputeRoute** algorithm takes in account **ActivityConfiguration** when computing the best route that the User have to follow.

[R 4.5] The User can choose to avoid the Public Transport after a given hour

[DD.4.1.1] The **ActivityController** component and Interfaces stores into the **Database** a **UserPreferences** object for each **Activity.**

[DD.4.5.1] the **ActivityConfiguration** object has an integer field that represents the maximum Walking distance the wants to perform for each route.

[DD.4.1.3] The **NavigationController** component using the **ComputeRoute** algorithm takes in account **ActivityConfiguration** when computing the best route that the User have to follow.

[R 4.6] The preferences has to be correctly stored into the System and available in any moment.

[DD.4.1.1] The **ActivityController** component and Interfaces stores into the **Database** a **UserPreferences** object for each **Activity.**

[G5] The application should suggest travel means depending on the nature of appointment.

[R 5.1] The User must be able to create Activity Templates in which he can choose what travel mean he want to use for each different kind of Activity.

[DD.5.1.1] The **ActivityController** component and Interfaces handles the Template creation and stores into the **Database** a **Template** object.

[R 5.2] The system must suggest to the User the travel mean basing on created Templates.

[DD.5.2.1] The **ActivityController** component and Interfaces allow the User to choose a **Template** for Each User Activity.

[DD.5.2.2] The **Template** used for an **Activity** contains the **ActivityConfiguration** object that stores all the User preferences.

[G6] Allow a User to reach an activity using the best mobility option and considering user Preferences.

[R 6.1] The System must compute the best route by analyzing all possible paths considering traffic conditions, eventual accidents, weather conditions or Public transportation strikes.

[DD.6.1.1] The **ComputeBestRoute** algorithm basing on **External APIs** takes in account the traffic conditions, accidents, public transport strikes and weather conditions.

[R 6.2] The System must take in account preferences expressed by the User to reach the Activity.

[DD.6.2.1] The **ComputeBestRoute** algorithm takes in account **ActivityConfiguration** object for each Activity in order to suggest the mean of transport preferred by the User for each Activity.

[R 6.3] The System must correctly handle Maps and Traffic and APIs.

[DD.6.3.1] The **NavigationController** components and interfaces are configured to handle **Map and Traffic APIs.**

[R 6.4] The System must correctly handle Weather Information APIs.

[DD.6.4.1] The **NavigationController** components and interfaces are configured to handle **Weather Information APIs.**

[G7] Allow a User to buy public transportation tickets/passes.

[R 7.1] The System must correctly handle the Public Transportation Company APIs.

[D.7.1.1] The **NavigationController** signals the User the need for using an external means of transport and thow the **ExternalServicesHandler** gives to the User the possibility to purchase the needed service on the provider Website

[G8] Allow a User to use bike-sharing and car-sharing services.

[R 8.1] The System must correctly handle the Bike Sharing Service APIs.

[DD.8.1.1] The **NavigationController** components and interfaces are configured to handle **Bike Sharing APIs** throw the **Bike Sharing Interface.**

[R 8.2] The System must show the position of the rentable bikes on the map.

[DD.8.2.1] The **NavigationController** components and interfaces are configured to handle **Map Information Service API** in order to Show the coordinates on the map

[R 8.3] The System must correctly handle the Car Sharing Service APIs.

[DD.8.3.1] The **NavigationController** components and interfaces are configured to handle **Car Sharing APIs** throw the **Car Sharing Interface.**

[R 8.4] The System must show the position of the rentable cars on the map.

[DD.8.4.1] The **NavigationController** components and interfaces are configured to handle **Map Information Service API** in order to Show the coordinates on the map

[G9] Allow a User to schedule an Activity in a flexible way according to his other meetings.

[R 9.1] The User must be able to specify a time interval in which will be done a Flexible Activity.

[DD.9.1.1] The **ActivityController** components and interfaces give the User the possibility to create a **FlexibleActivity** Object specifying a **SchedulingIntervalBegin** date and **SchedulingIntervalEnd** date.

[R 9.2] The system must compute the best time in interval specified by the User to place the Flexible Activity, according to the other meetings during the day.

[DD.9.2.1] The **ActivitySchedulingController** components and interfaces using the **ActivityScheduling** algorithm place the Flexible Activity considering other Activities during the day **.**

[G10] Allow the User to view a calendar with all his scheduled activities

[R 10.1] The System must show the User a Calendar from which he can visualize and manage his daily meetings.

[DD 10.1.1] The **ActivityController** components and interfaces reads from the **Database** all the activities for a given user

[DD 10.1.2] The **MobileApplication** receives data from **WebServer** about user's Activities and show them on a Calendar

[G11] Allow a User to take advantage of the taxi service.

[R 11.1] The System must correctly handle the Taxi Service APIs.

[DD 11.1.1] The **NavigationController** components and Interfaces are configured to handle **Taxi Service APIs** throw the **Taxi Service Interface.**

6. Implementation, Integration and Test Plan

6.1 Implementation Strategy

Starting from details developed in this document and in the RASD we can divide the components to develop into three categories:

• **Front-end components**: Mobile Application
• **Back-end components**: Web Server with all the sub-component described in the Component View and the DBMS.
• **External components**: all the components which refer to functionalities provided by external systems and their APIs.

Some components belonging to Front-end, Back-end and External categories are independent one from each other and we assume that external Systems API are already developed and stable.
We will follow a bottom-up implementation Strategy.

We start with developing core components of Our Back-end:

- DBMS in order to map our "Model" in the Database

- Controller
- ActivityController
- UserController
- NavigationController
- ActivityScheduleController
- NotificationManager

After the development of 60% of these component and unit testing of developed single component functionality we can to develop:

- the **ExternalServiceHandler** component that handles the interactions with External APIs and allows our component to interact with External Systems.
- The **MobileApplication**

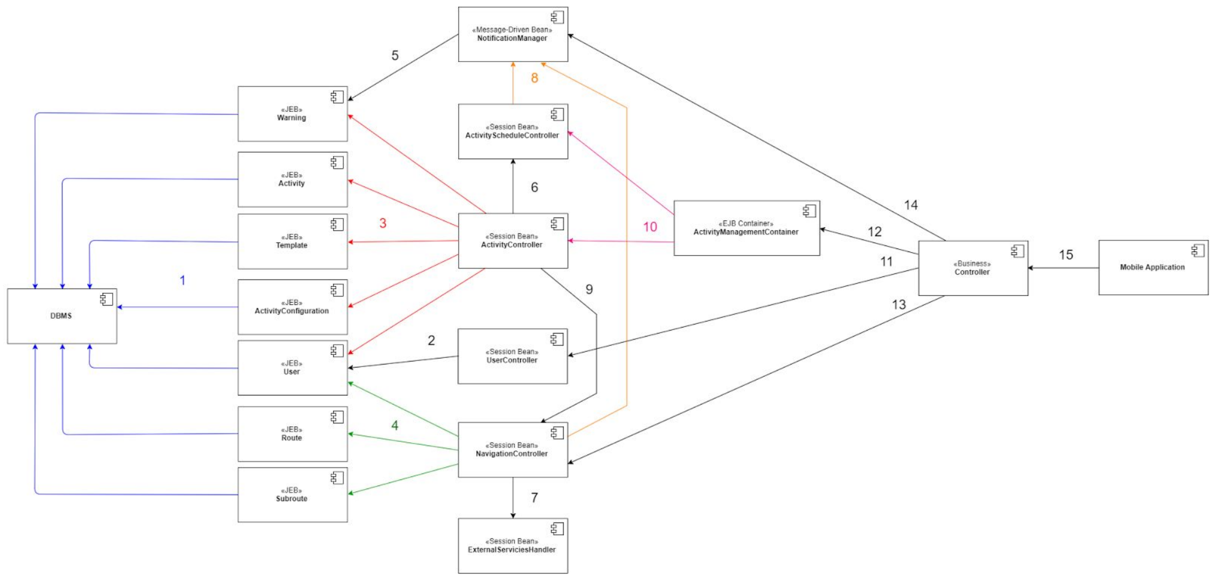
After the development of these components we can proceed to unit testing and then to the integration with all components

6.2 Integration Strategy

In order to integrate the various components a **bottom-up approach** will be used, which consists in integrating a low hierarchy component with its parent. One of the main reasons for using this approach is that the components are easy to reuse.

In the following diagram we have represented a general view of the various components and the order in which they will be integrated: in the diagram there are numbers next to the arrows, they represent the order of integration of the various components. The table below contains information about all of the various steps of integration represented in the chart.

It is worth point out that the **ExternalServicesHandler** is integrated with all services indicated in the component view diagram of 2.B paragraph. This integration occurs indicatively before the Step 7 highlighted in the table.


«Message-Driven Bean» NotificationManager
«JEB» Warning
«JEB» Activity
«JEB» Template
«JEB» ActivityConfiguration
«JEB» User
«JEB» Route
«JEB» Subroute
DBMS
«Session Bean» ActivityScheduleController
«Session Bean» ActivityController
«Session Bean» UserController
«Session Bean» NavigationController
«Session Bean» ExternalServiciesHandler
«EJB Container» ActivityManagementContainer
«Business» Controller
Mobile Application
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Integration Step in the diagram	
1	The first components to be integrated are those relative to the data access. So, we will integrate first all JEBs with the DMBS.
2	Integration continues by integrating the components from the one with the minimum number of dependencies. In particular the UserController will be integrated with the JEB User.
3	The ActivityController must be integrated with more than one JEB: in addition to normal activities management, the ActivityController takes care of managing user preferences and then acts also as a template controller.
4	The NavigationController manages the route (and then the subroutes) of all activities. It detects, mostly by means of external services, possible warnings, which will then be notified to the user by means of the NotificationManager.
5	Naturally the message-driven bean NotificationManager must have access to all the warnings in the database. We will integrate the NotificationManager with the Warning JEB.
6	The ActivityController will be integrated with the ActivityScheduleController. The ActivityController delegates the activity scheduling to the ActivityScheduleController, which is able to detect warning caused by the impossibility of scheduling all the

	activities consistently, for example following the rules imposed described in the alloy model of the RASD.
7	The NavigationController must be integrate with the ExternalServicesHandler, without this integration it would be impossible to interact with external services related to the single route or detects warning stemming from external services related to the single route.
8	• The ActivityScheduleController must be integrate with the message-driven bean NotificationManager in order to signals warnings stemming from an impossibility to schedule all fixed activities (meetings) and all flexible activities. • The NavigationController must be integrate with the message-driven bean NotificationManager in order to signals warnings stemming from subroute problems: rain, strike...
9	The ActivityController will be integrated with the NavigationController. This aggregation is important because the NavigationController provides also Warnings, position of the user and several further information, such the activity state after checking all subroutes states.
10	The EBJ ActivityManagementContainer groups up the ActivityController and the ActivityScheduleController in a single container.
11, 12, 13, 14	The main (business) controller will be integrated with the EBJ ActivityManagementContainer, the two session beans UserController and NavigationController and the message driven bean NotificationManager.
15	Since the system is based on a mobile application, a final integration is required. We will integrate then the Mobile application with the business controller.

6.3 Integration Testing

All classes and methods will be tested with the framework JUnit.

The **documentation** is also relevant: all classes and functions will be sufficiently explained using JavaDoc. This practice is important in order to have a solid reference when the integration testing phase is in progress. Those who works to the integration test must have the RASD and DD available as a landmark.

The basic principle to follow for the integration testing is to test the integration in same order as the integration has been done. Following this criterion we will test first the integrations between DBMS and Controllers. However, it’s not mandatory to strictly follow the same order and some tests could be performed in parallel with others.

In order to test the integration of two components, the main features of both of them should have been developed and the related unit tests should have been performed. It is necessary to test at least 65% of each controller.

It is also important to test the integration between the ExternalServicesHandler and all the other external services modules. Tests do not only test if interaction between modules works, but test a proper functioning of all features and therefore all external services should be active at the time of testing. In fact, before proceeding to the single test it is necessary to wait first that all the preconditions have been verified.

This is an example of one integration test between UserController and DBMS:

Test ID	S2T3
Components	UserController, DBMS
Input Specification	Login of a user
Output Specification	Check if login is completed successfully
Exception	Login is not allowed if the user does not exist or the password is incorrect
Test Description	The UserController interacts with the User JEB in order to verify user credentials through the use of queries
Testing Method	Automated with the framework JUnit

(S2T3: Login test case)

7. Effort Spent

Hours of work per day:

6/11/2017 (Together): 3hrs

8/11/2017 (Together): 2hrs
14/11/2017 (Together): 5hrs
15/11/2017 (Together): 3hrs
18/11/2017 (Together): 3hrs
20/11/2017 (Together): 2hrs
21/11/2017 (Together): 4hrs
23/11/2017 (Together): 3hrs
24/11/2017 (Together): 2hrs

Hours do not include individual work, about approximately 7-8 hours of work per member of the group.

8. References

- “RASD 1.1”,2017, Matteo Bellusci, Federico Di Dio, Flavio Di Palo
- Google Maps Directions API Documentation: **https://developers.google.com/maps/documentation/directions/**
- Xamarin Platform Documentation: **https://www.xamarin.com/platform**