POLITECNICO
MILANO 1863

Travlendar+

Design Document

Alessandro Saverio Paticchio – 894092
Andrea Tricarico – 898406
Davide Santambrogio – 900204

Table of content

1. Introduction 3
    a. Purpose 3
    b. Scope 3
    c. Definitions, Acronyms, Abbreviations 3
    d. Reference Documents 4
2. Architectural Design 5
    a. Overview: High level components and their interaction 5
    b. Component View 8
    c. Deployment view 13
    d. Runtime view 15
    e. Component interfaces 18
3.Alghoritm design: 23
4.User interface design: 26
5.Requirements traceability: 30
6. Implementation, Integration and Test Plan 32
7.Effort Spent 47
8.Document History 47

1. Introduction

a. Purpose

This document has been written in order to explain how the system behind Travlendar+ is projected and how it will be composed.

In particular, we will focus on the architectural approach of the application, unveiling how logical and physical component work together to let the application support the user.

Furthermore, here you will find the reasons of some design choices, that hopefully will drive the application toward the best possible configuration.

b. Scope

Travlendar+ is an application that will be developed in order to help people in arranging their day. It will offer the functionality of inserting all your daily schedules, suggesting the smartest way to move among them.

This will be done by suggesting you the most suitable path with respect to your preferences, in terms of transportation services, walking distance and many others.

To computer itineraries, the system will also retrieve information about weather forecast, strikes and everything that can be useful to allow the user to be in time to its appointment.

Therefore, here you can find how the request of the user are managed by the system, where they pass and how they are elaborated.

In particular, we make a distinction in how the application will be deployed:

Installed application: it is a software application running on user's device (personal computer or mobile phone).

Web-based application: it is an application accessible via web browser.

c. Definitions, Acronyms, Abbreviations

RASD: Requirement Analysis and Specifications Document

JAX - RS: Java API for RESTful Web Services

URL: uniform resource locator

REST: REpresentational State Transfer

API: application programming interface

MOP: mean of transportation

Inconsistency: A schedule is inconsistent if two appointments overlap or other variables make the appointment difficult or even impossible to reach (weather conditions, traffic, strikes, ...)

Appointment: an entity that defines a period of time devoted to a user's activity.

Itinerary/Ride: Travel between two appointments, it is defined by starting time, ETA, vehicle, total cost.

Schedule: entity that represents the daily schedule of the user, each instance is composed by all the inserted appointments of that day.

d. Reference Documents

- IEEE DD standard document;
- Mandatory Project Assignment for Software Engineering 2

2. Architectural Design

a. Overview: High level components and their interaction

This application will be developed and implemented following the wide spread client-server pattern. In the designing phase, we privileged a two-tiered architecture (as shown in the figures below), that eventually becomes a three-tiered architecture, with the additional tier interpreted by the Web Server, in case of web-based interaction.

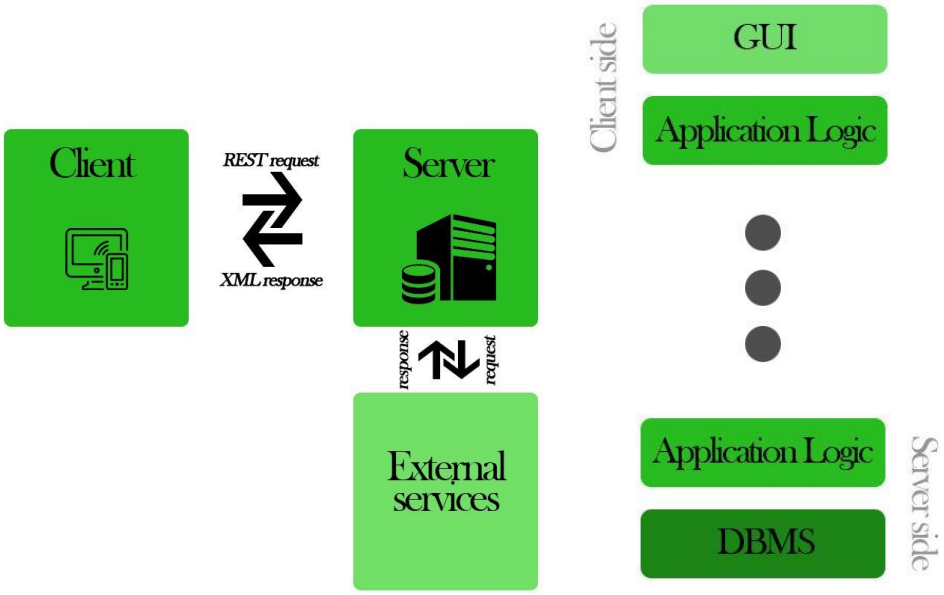
The reasons of this choice rely on the fact that the application must be as reactive as possible, therefore we instantiated the DBMS and the application-layer on the same tier, to reduce latency.

In fact, we decided to delegate to the central server both the handling of the application logic and the managing of the database, since the latter is neither heavy nor complex and it is required to be as fast as possible, allowing a high efficiency.

Furthermore, keeping the database as near as possible to the application-server leads to a strong security level, since the amount of data crossing the network decreases.

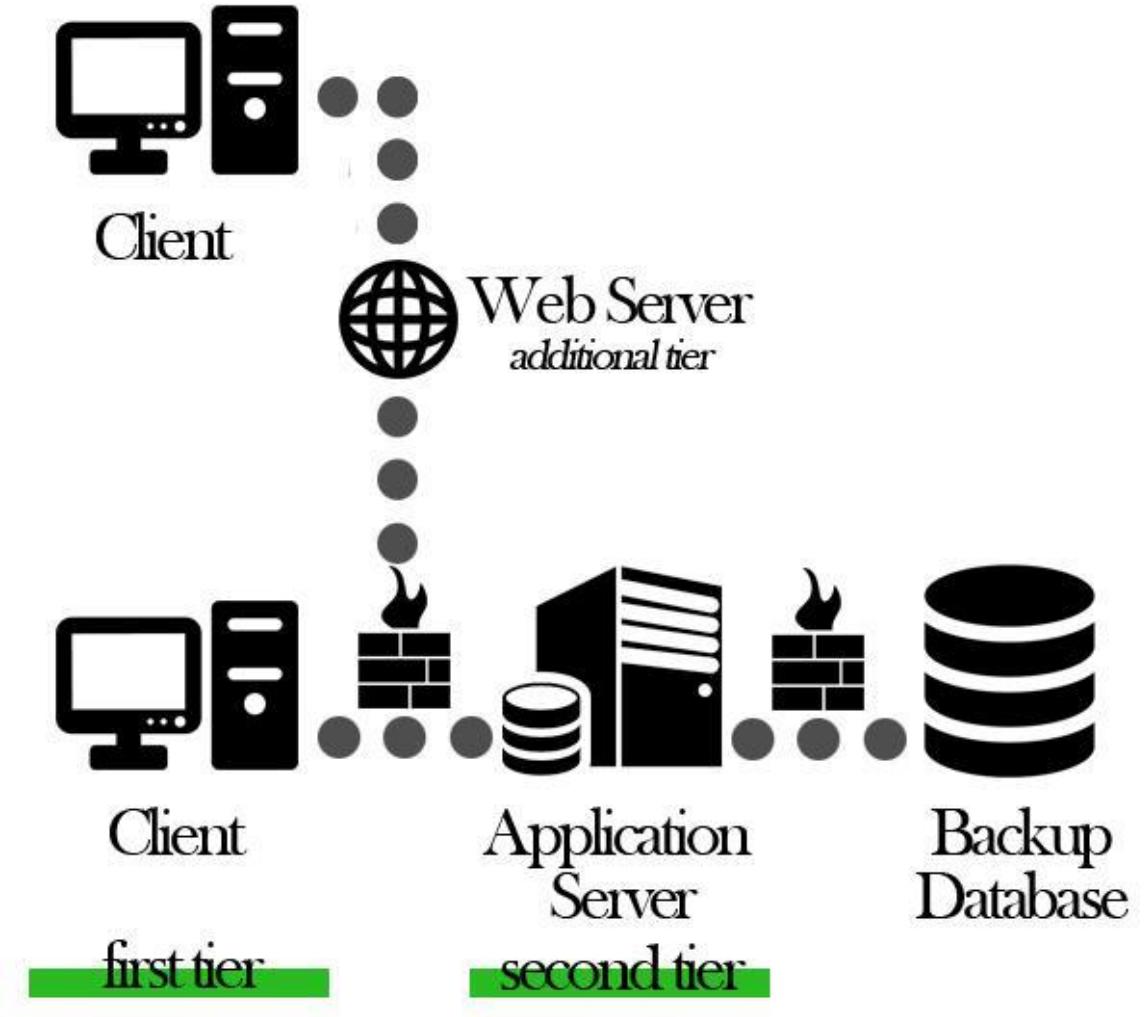
As far as the distribution of the logic is concerned, we observed that it would be useful and smart to leave a little part of the logic client-side, to lower the load of the server, reduce useless interactions and minimizing the delay.

Due to the notification functionality offered by the system (related to unpredictable delays, weather changes, strikes...), a publish-subscribe approach will be adopted.



Lefthand side: How our system communicates

Righthand side:two-tiered architeture



Architectural overview

As shown, the architecture foresees some interactions with a backup database, just to be fault tolerant.

The backup interaction is carried out periodically (every day at 3.00 GMT+1, when the server is supposed to be underloaded) and it will involve the whole database. This interaction crosses the network; therefore a firewall is necessary.

The messages of the registration procedure, and those that concern the recovery of the lost password, are exchanged in an asynchronous way, since the client sends to the server the form filled with all the data of the user, and the server replies with a confirmation email to the address indicated in the form (asynchronously).

The communication messages between the client and the server are exchanged synchronously, since the client waits an acknowledge for each message sent to the server.

Notification for incoming appointments, itinerary variations and possibility to buy a public transportation ticket are sent from the server to the client in an asynchronous way through a push notification service.

All the messages are made private through cryptography, since they contain personal information of the user.

The client-server interactions are handled through REST paradigm, to make it as flexible as possible, whereas external services interactions are carried out through proper APIs.

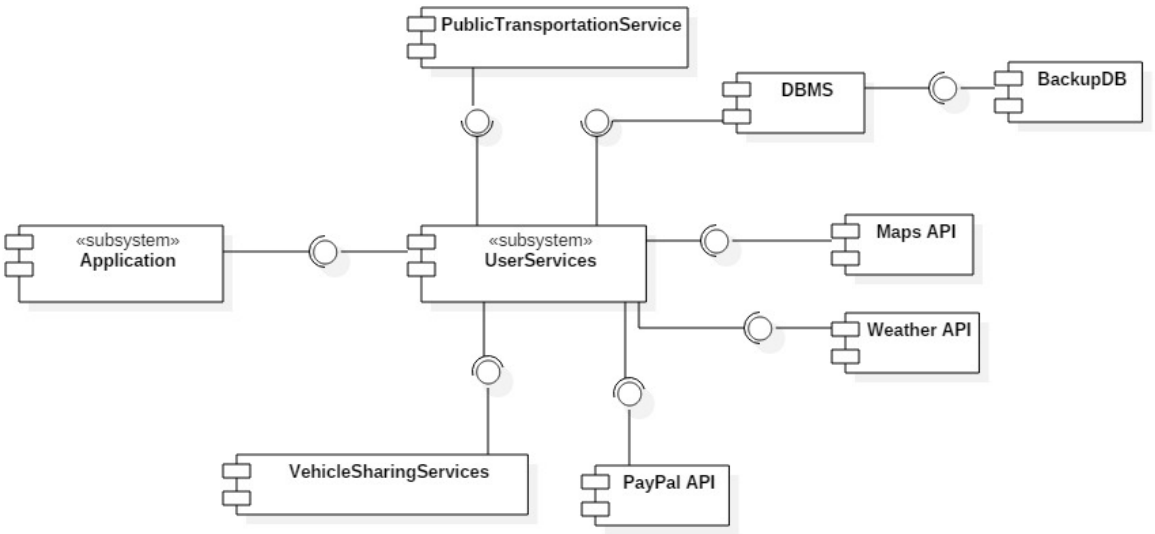
b. Component View

In this section we provide an overview of the components of our applications, by exploring how they interact with each other and how they handle the connections with outer entities.

High-level component view

This diagram aims to explain how the different components of the system interact.

On the left we find the client-side (Application), intended both as installed application and as web accessible. In fact, this component exposes one interface to the Web Server and one to the mobile clients via specific APIs in order to decouple the different layers with respect to their individual implementation.
On the right we find the server-side (UserServices), which interacts with our DBMS and all the external services necessary to accomplish the tasks.

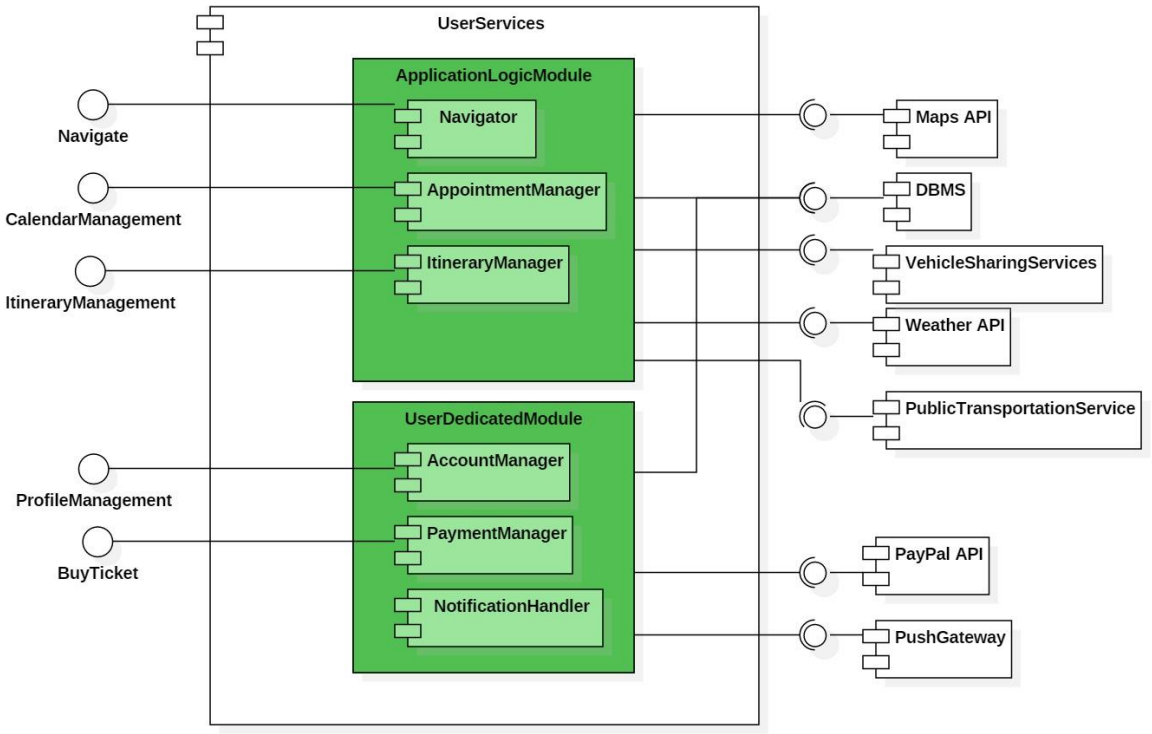


Component overview

The database layer must include a DBMS component, in order to manage the most useful data for the application. The system is expected to be a relational database, capable of guaranteeing the correctness of concurrent transactions and ACID properties. The data layer must only be accessible through the Application Server via a persistence unit to handle the dynamic behaviour of all of the persistent application data.

UserServices component view

The following diagram explodes the UserServices subsystem, to clarify the two fundamental modules and the exposed interfaces.
This layer is expected to manage the access to the data layer and the multiple ways of accessing the application from different clients and to retrieve information from external systems.
Furthermore, this component adapts itself to given interfaces by external systems.



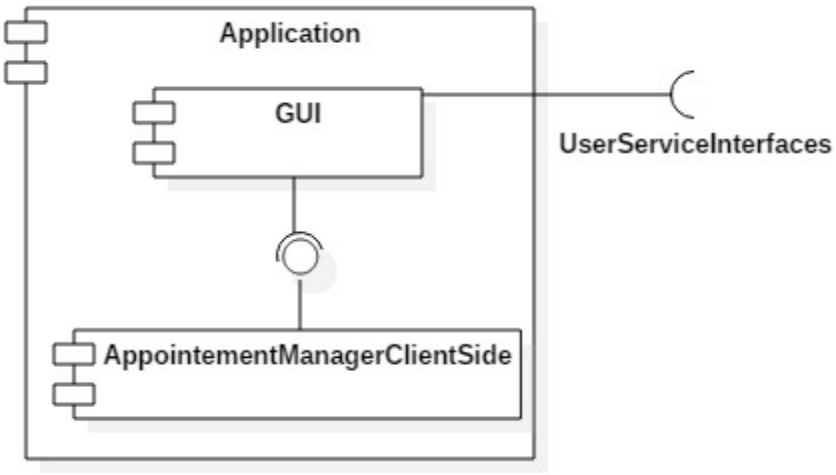
UserServices component diagram

This is composed by 6 main modules:

- Navigator: it guides the user toward its destination, following the indications already computed. This is done by retrieving user's GPS location and by analyzing the surrounding map.
- AccountManager: it handles all the user's preferences. It is responsible of the login/registration of users and to collect all the related preferences. It interacts with database by storing the most sensible data.
- AppointmentManager: it manages all the schedules of the users, checking the general consistency. It interacts with the maps provider in order to locate where the appointment will be held. Once the appointment has been scheduled, it is also stored in the database.

- ItineraryManager: it organizes all the trips among appointments, computing paths using both information provided by external services and users' preferences. Once the itinerary has been selected by the user, it is also stored in the database. This module also helps the AppointmentManager module in the consistency check operation.
- PaymentManager: it manages the payments to buy public transportation tickets. It is expected to interact through PayPal API to let the payment be easy and fast.
- NotificationHandler (Event Dispatcher): its aim is to warn user of incoming appointments, of itinerary variations and of the possibility to buy a public transportation ticket, through push notifications. It is the main component of our publish-subscribe pattern, better described below.

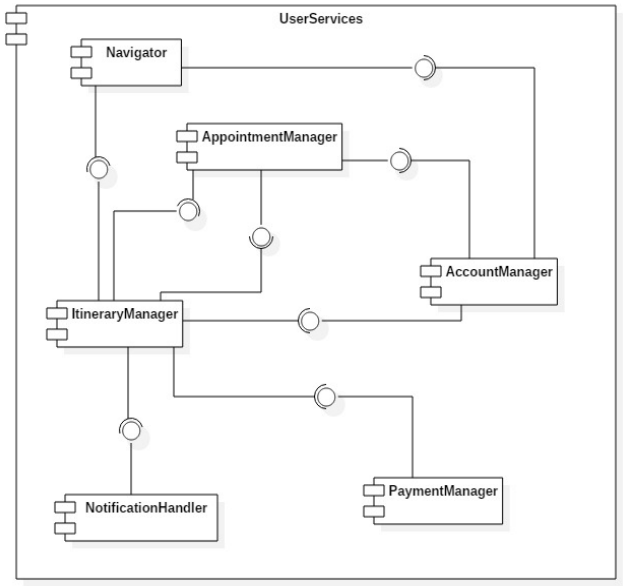
Installed application component view



Installed application component diagram

UserServices internal interfaces

The internal components of the UserServices subsystem expose interfaces to the others in order to perform actions that involve different components.



UserServices internal interfaces

AccountManager exposes an interface to:

- ItineraryManager, that uses it to retrieve the user preferences.

AppointmentManager exposes interfaces to:

- AccountManager, to give the possibility to an user that has logged in to manage its calendar.
- ItineraryManager, which uses it when an unexpected event occurs and it needs information about the daily schedule to reorganize the trips of the day.

ItineraryManager exposes interfaces to:

- AppointmentManager, that calls the ItineraryManager when the user creates/edits an appointment to compute itineraries and to check if it is feasible to reach all the appointments with the new information.
- Navigator, to be queried about paths.

Navigator exposes an interface to:

- AccountManager, to let an authenticated client start the navigation toward an appointment.

PaymentManager exposes an interface to:

- ItineraryManager, to be called in case of ticket purchase.

NotificationManager exposes an interface to:

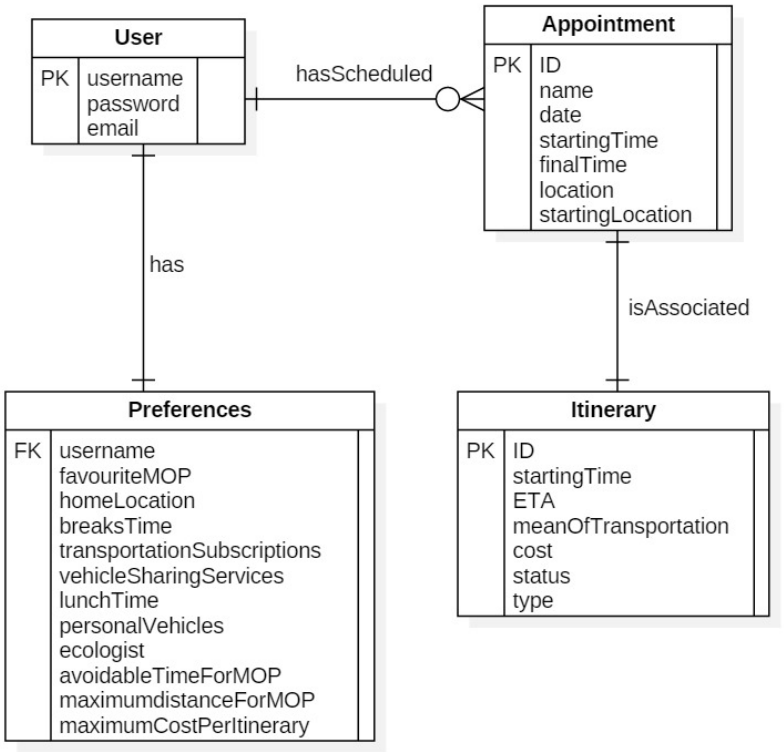
- ItineraryManager, to send notification to the client in case of unexpected events affecting itinerary.

Entity-Relation Diagram

The following is the model to follow in the implementation of the database.

As mentioned above, we aim to a system as flexible and light as possible, therefore we adopted a skinny approach for our database, in order to let the interactions be fast:
for instance, entities like DailySchedule (see UML Class Diagram) are not stored, but they are computed by the system by querying about all the appointments on the same date.

This approach also prevents from update anomalies, since redundancy is avoided.



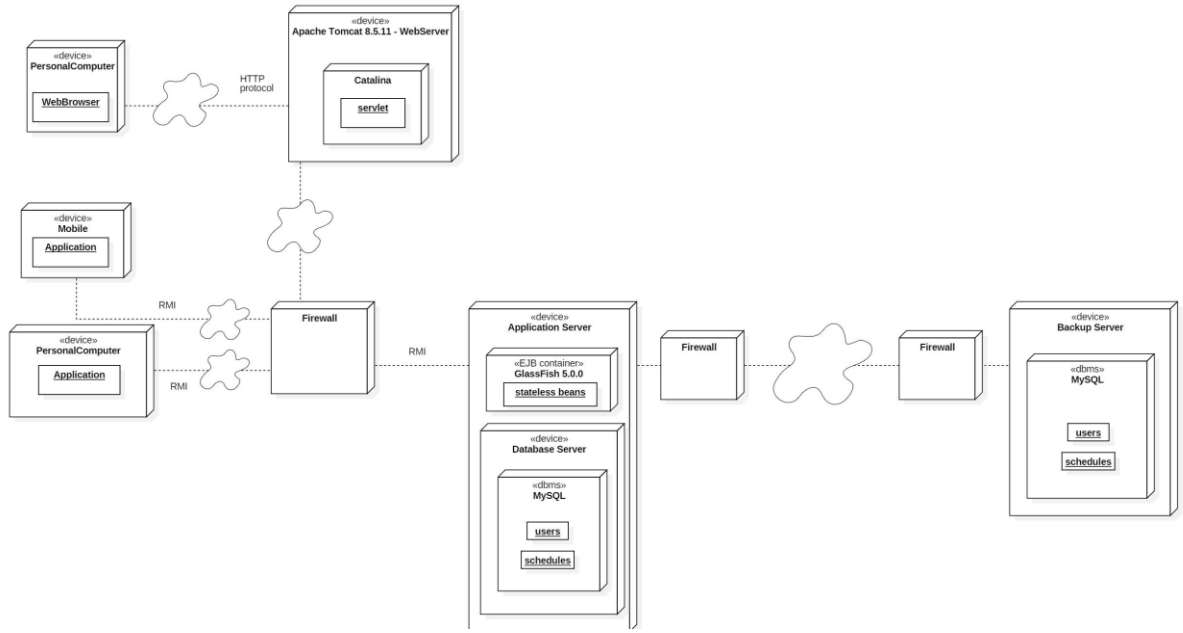
E-R Diagram

c. Deployment view

The following diagram is a deeper sight of the architectural and physical structure of our system.

As pictured, there are two kinds of interaction:

- through WebBrowser: the requests cross a Web Server and elaborated (in part) by a servlet and eventually passed to the Application Server.
- through installed Application: the request are directly forwarded to the Application Server, which serves the client using stateless beans.



Web Server design

This layer will be implemented using an open-source Java Servlet Container developed by the Apache Software Foundation (ASF), that is *Apache Tomcat*. Tomcat implements several Java EE specifications including **Java Servlet**, that will be used to implement a little part of the logic in the web server, reducing the load of the application server.

The component to be mentioned is *Catalina*: it is the servlet-container of this environment, responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access-rights.

Application Server design

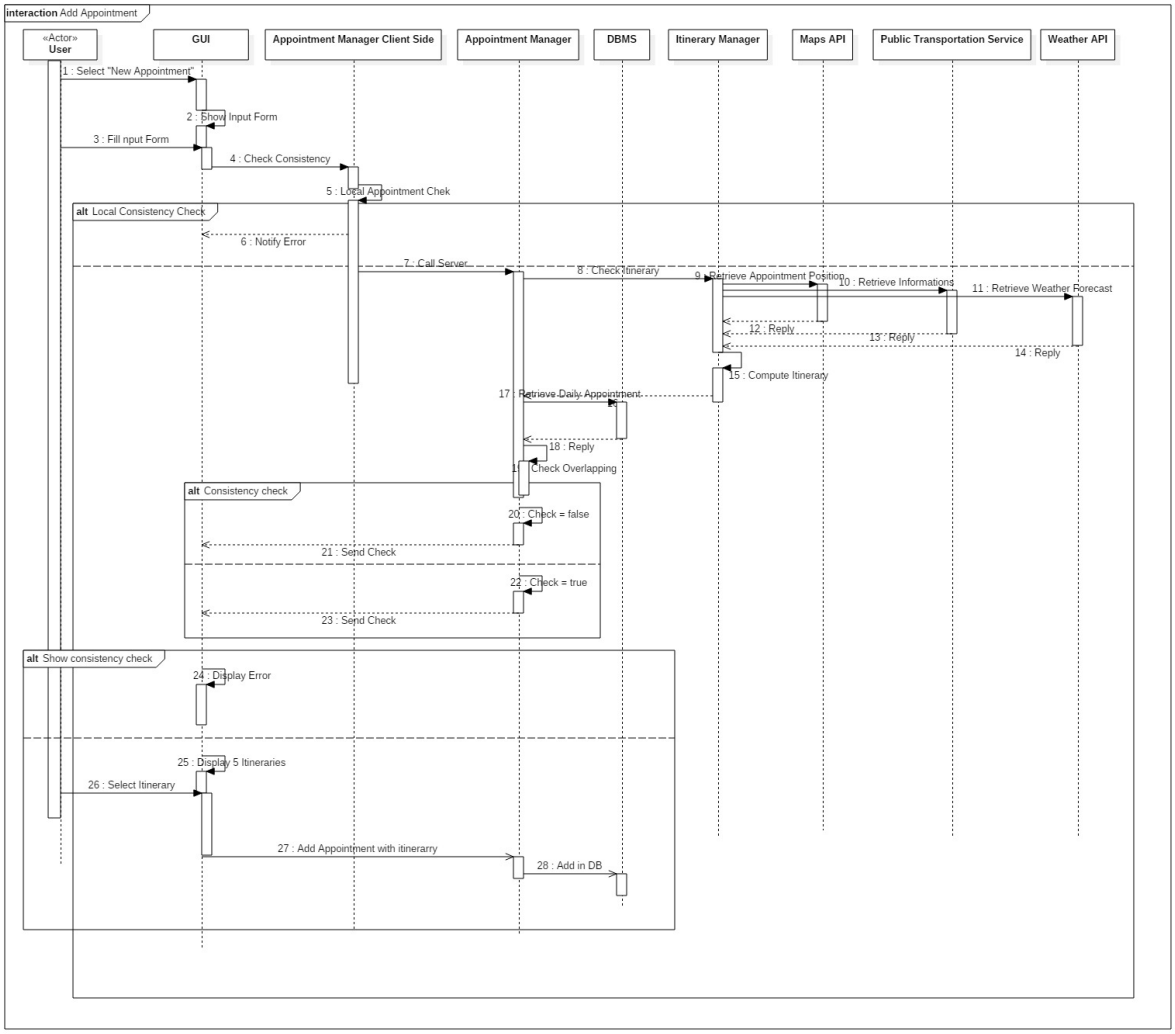
In order to support the several and different interactions among server and external systems, such as clients, web layer and database, the system will adopt another open-source platform to realize its application server, that is *GlassFish*.
GlassFish supports many types of interactions, in particular we will use it to develop the following features of our system:

- *Remote procedure calls* (RMI): used by the installed application to request services.
- *Enterprise JavaBeans* (EJB) to implement each business logic module, by using stateless beans.
- *JAX-RS* to implement RESTful APIs, that will be used to interact with clients.
- *Java Persistence API* (JPA), a programming interface that make the object representation of the database entities easier.

DBMS

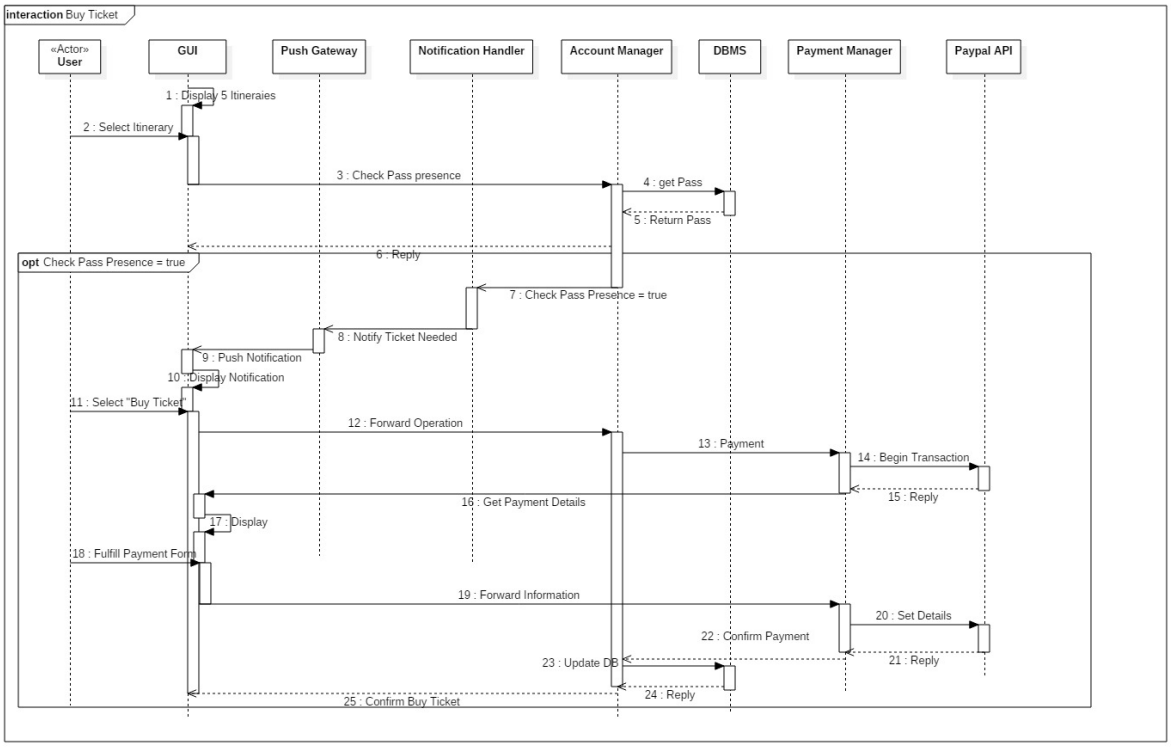
We will adopt *MySQL* as Database Management System, since it is one of the most solid, supported and used.

d. Runtime view



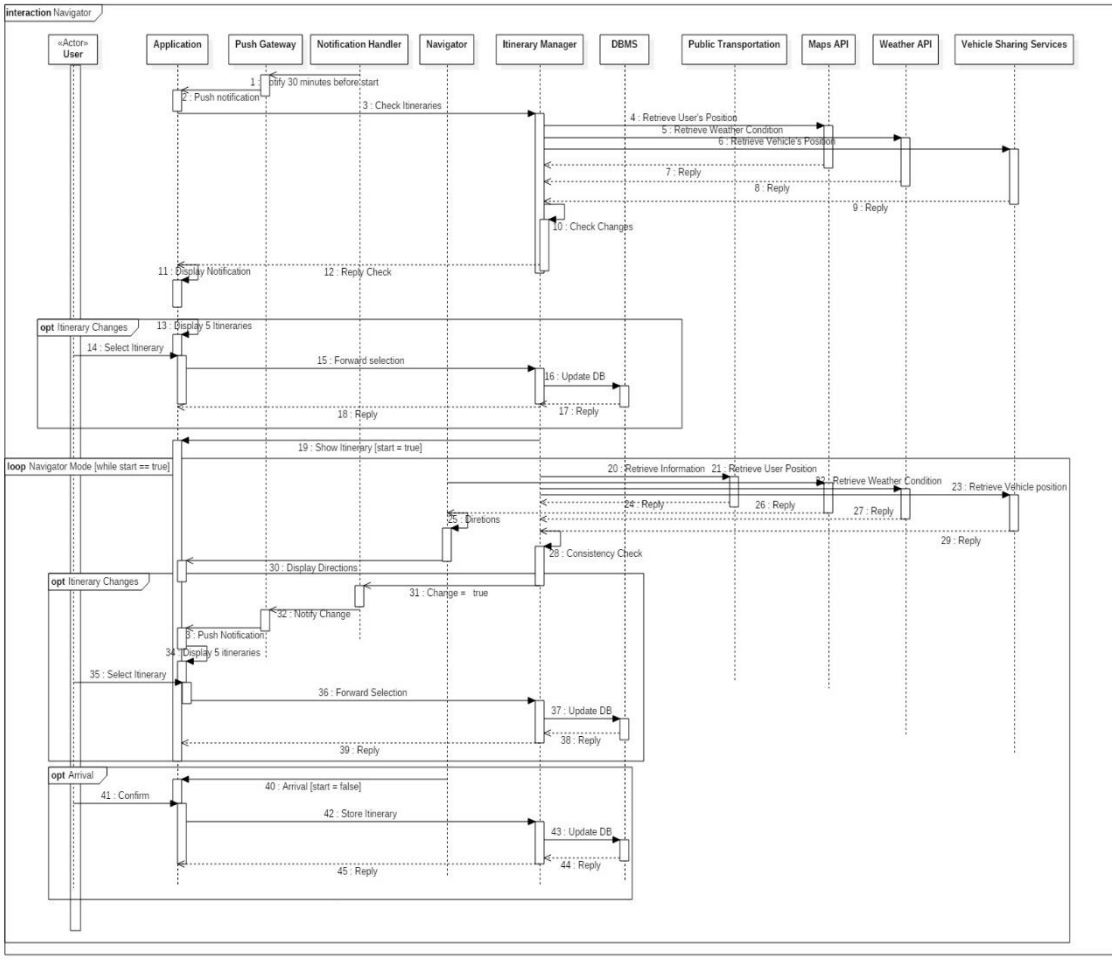
1addAppointment feature runtime view

This is one of the main application's functionality: adding appointment. Notice the role of the local Appointment manager which checks only the overlapping between the various appointments (thanks to local memory) while the server side appointment Manager calls the itinerary manger in order to complete the consistency check with all the needed information.



buyTicket feature runtime view

This diagram describe interactions between the account manager and the payment handler in order to allow the user to contact Paypal directly through the application and to pay the ticket if it is possible.



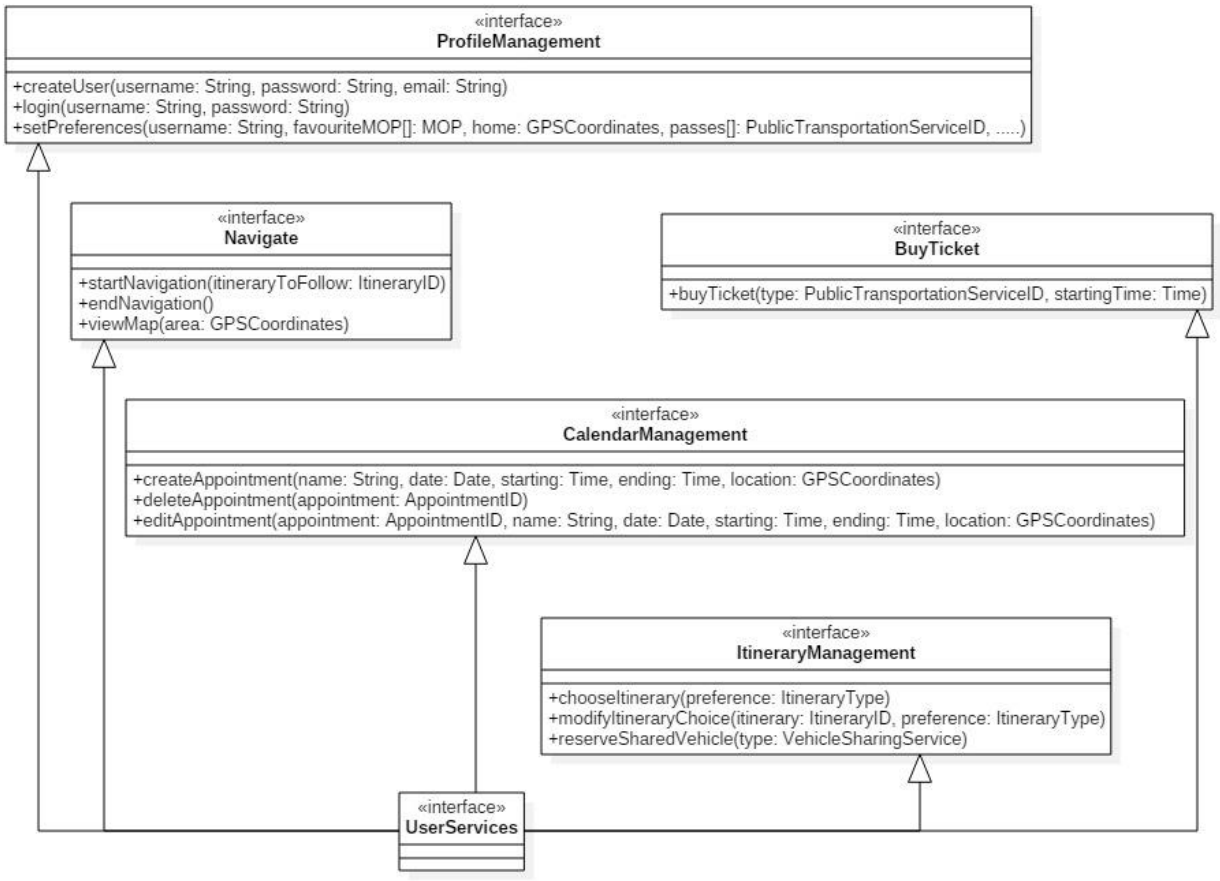
Navigator feature runtime view

This Diagram describe the interactions within the system in order to allow the Navigator functionality. As we can see the GUI continue to receive the operations of the user forwarding them to the right component and to display the various application response. The most important thing to notice is the role of the itinerary manager that, while the navigator sends directions thanks to the use of Maps API, it continuously check the track conditions and its consistency in order to notify the user if a change is necessary.

e. Component interfaces

The UML diagram shows the high-level interface exposed from the server and used by the client in the client-server paradigm:

- ProfileManagement is used by the application to register a new user, to check the credential in the logging phase and to manage the user preference.
- CalendarManagement and ItineraryManagement expose the methods to fill and edit the calendar with the appointments and the related itineraries.
- Navigate is necessary to have access to the commands of the navigator during the trips.
- BuyTicket behaves as a broker between the client which wants to buy a public transportation ticket and the PayPal service that is in charge of manage the trade.



Component interfaces

Further interfaces

Application Server & External Systems

The application server is expected to connect with other external systems:

- PayPal: it provides API to which the server itself must adapt in order to perform payments.
- Weather Forecast, Public Transportation Services, Maps Provider and Vehicle Sharing Services: these services are supposed to expose some interfaces in order to be queried. The information collected will be used to arrange itinerary and appointments.

Application Server & Web Server

The communication between Application Server and clients, both direct and via the Web Server, will be performed via RESTful APIs, provided by the Application Server itself and implemented using JAX-RS.

Database & Application Server

Access to the database is carried out through the Java Persistence API, mapping objects and actual relations.

Web Server & Web Browsers

As shown above, this interaction is completely based on HTTP protocols.

f. Selected architectural styles and patterns

Client-server multi-tiered architecture

Our application will be divided into 4 tiers:

1. Thick Client (GUI and a base logic) for installed application / Thin Client for web application
2. Web-Server (to manage clients which use the web-interface to run the application)
3. Application Server
4. Database

The client-server model is used at different levels in the system.

The mobile application is a client directly with respect to the Application Server, which exposes its services. Whereas, the user's browser communicates with the Web Server and this will forward the requests to the Application layer;

The Web Server, as a client, communicates with the Application Server in order to process a user's requests;

The Application Server takes the role of the client when it queries the Database that is responsible, as the server, of fetching query results.

We use a thick client as far as installed application is concerned, therefore we have a base of the application's logic directly in the user's device, in order to speed up some consistency check processes. In particular, the client directly looks for possible overlaps within the daily schedule while inserting (or editing) an appointment, without contacting the server.

For the browser application, this logical behaviour is performed by moving this part of computation on the web server, letting the web-client be thin.

Publish-Subscribe paradigm

This application is supposed to be a very reactive environment, whose evolution is strictly related to the events collected by the server and forwarded to the interested nodes of the network.

This behaviour will be realized with the *publish-subscribe* pattern, that allows all the users to be instantaneously informed about variations in the itineraries.

The component in charge of handling this process is the NotificationHandler (see Component View), who acts as an Event Dispatcher, collecting subscriptions to events related to given itineraries and notifying users if something interesting occurs.

This pattern will be implemented through Java Message Service API, that is a messaging standard that allows application components based on the Java Enterprise Edition to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be reliable and asynchronous.

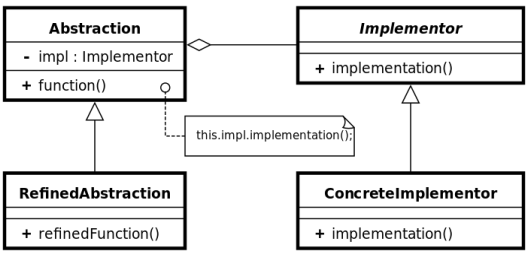
Model-View-Controller pattern

The application will be implemented using the Model-View-Controller architectural pattern in order to reach an high flexibility and make easier to implement new feature in the future: this pattern's aim is to separate the application's data model, user interface, and control logic into three distinct components, such that modifications to one component can be made with minimal impact to the others.

Singleton pattern

The server-side uses an event logger to keep track of the sequence of all activities made by the users and to make possible recovery operations. This logger will be implemented following the singleton pattern so all the parts of the serve should use the same instance of the logging system.

Bridge Pattern



In order to support various different systems for the client part of the application, we will use the Bridge pattern in order to decouple an abstraction from its implementation (by putting them in separate class hierarchies) so that the two can vary independently. It is useful to develop the application for different operative systems and to minimize the effort to add support for a new one in the future.

Object Pool Pattern

In order to optimize the creation of new itinerary and boost the performance of the system, we will use the Object Pool Pattern. It is useful for the itineraries because they are often destroyed and recreated due to unexpected events or delays. Object pools are used to manage the object caching. A client with access to an Object pool can avoid creating a new Itinerary by simply asking the pool for one that has already been instantiated instead. It is desirable to keep all Reusable objects that are not currently in use in the same object pool so the Reusable Pool class is designed to be a singleton class.

3.Alghoritm design:

Now we will see the central part of the application's functionality; in fact through this algorithm (divided in small pieces in order to allow a more precise comprehension of each part of it) it is possible to notice in details how it is expected to work an Appointment Scheduling that de facto touches all the main components and application logic's part.

At the beginning we have the various Feasible Vehicles which contains all the possible vehicles considered in the application. The first problem is to reduce them checking weather conditions, appointment type and ecologic mode (after adding to them the personal vehicles). Single checks are very simple and, if we have to specify very precisely, the Weather one needs its particular API in order to retrieve the right information about the appointment (but in this section this is not showed because it is too specific for the predisposed purpose).

Then, thanks to the Maps API, the track is computed for every single feasible vehicle left considering the shortest possibility. After that, the algorithm takes into account the coherency of the computed itinerary, checking if the maximum Distance and the avoided time for the Vehicle and the Maximum Cost in general are respected. Finally, there is the most important check for the appointment, the consistency. It allows the system to notice if there are any overlapping between a stored appointment and the itinerary which permits to reach the new appointment (actually the consistency check on the appointments overlapping is considered to be done before all this algorithm on the client side with the same form of the consistency check function written below.

Then we have all the feasible itineraries which the system could propose to the user (if there are not any feasible itinerary, it is expected an AppointmentInsertion Exception, which notifies the user of the impossibility to compute the track). Now the application has to propose *5* itineraries (not compulsorily different): the shortest, the cheapest, the one with minimum changes, the one with minimum walk distance and the ecologic one.

In conclusion, the Algorithm takes into account a Personal vehicle check and a preferred vehicle check in which, if the vehicle is in the feasible once and there are no proposed itinerary with at least one of them, the min cost and the min change alternatives are exchanged with one that uses a Personal or a Preferred Vehicle. So, the fact is that at least one proposed itinerary contains as vehicle attribute a Personal Vehicle, if it is possible. The same for the Preferred Vehicle selected in account customization.

We chose to write the algorithm in a Java like language only in order to easily help the reader to understand the various functionality (in fact being object oriented,

Java allows, through the usage of attributes, to describe in a more precise and visibly understandable way the structures and the virtual objects used in the application, such as vehicles, itineraries, appointments,...)

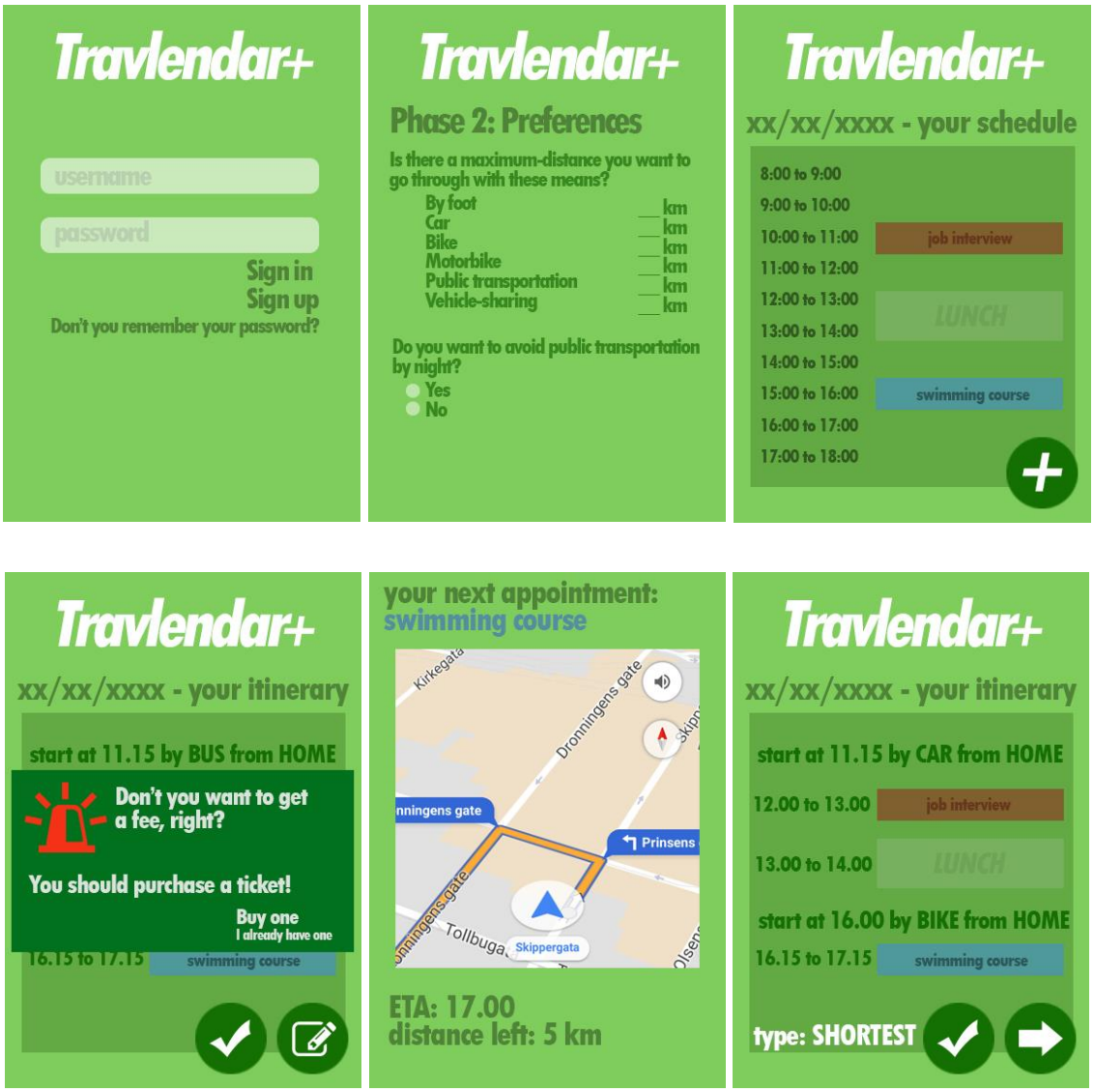
```
AppointmentScheduling (Appointment insertedAppointment){
ArrayList<Itinerary> feasibleVehicles = new
ArrayList<Itinerary>;
feasibleVehicles = PermittedVehicles(feasibleVehicles);
for (Vehicle vehicle : feasibleVehicles){
      Itinerary permittedItinerary =
computeShortTrack(vehicle,
insertedAppointment.predecessor.location,
insertedAppointment); //Maps API
      if(isCoherent(vehicle, avoidableTimeForMOP,
maximumDistanceForMOP, maximumCostForItineray,
permittedItinerary) &&
      ConsistencyCheck (permittedItinerary,
insertedAppointment.priority)){
            feasibleItineraries.add(permittedItinerary);
            }
      }
}
ProposeItinerary (feasibleItineraries);
}
ProposeItinerary(ArrayList<Itineray>
feasibleItineraries){
Itineray proposedItineary = new Array[5];
proposedItineary[0] = minTime(feasibleItineraries);
proposedItineary[1] = eco(feasibleItineraries);
proposedItineary[2] = minCost(feasibleItineraries);
proposedItineary[3] = minChange(feasibleItineraries);
proposedItineary[4] = minWalkDist(feasibleItineraries);
PersonalVehicleCheck(proposedItineary);
PreferredVehicleCheck(proposedItinerary);
}

PersonalVehicleCheck(Array
proposedItineary){//PreferredVehicle works in the same
way
for (Itinerary itinerary : proposedItineary){
      if(personalVehicles.contains(itinerary.vehicle)
                  return proposedItineary;
}
else{
      for (Vehicles vehicle : PersonalVehicles){
```

```
            if (feasibleVehicles.contains(vehicle){
                  proposedItineary[0] = computeShortTrack
                  (vehicle,
                  insertedAppointment.predecessor.location,
                  insertedAppointment);
                  return proposedItinerary;
            }
      }
}
return proposedItinerary;
}
PermittedVehicles (Vehicle feasibleVehicles){
for (Vehicle vehicle : personalVehicles){//in the
feasible vehicles we consider all the personal vehicle
      feasibleVehicles.add(personalVehicles);
      }
EcologicCondition(ecologist, feasibleVehicles);
WeatherCondition(feasibleVehicles, weatherForecast);
AppointmentTypeCondition(feasibleVehicles,
insertedAppointmentType);
}
//others minimization functions work in the same way
minTime (ArryList<Itinerary> feasibleItineraries){
Itinerary choice = feasibleItineraries(0);
for (Itinerary itinerary : feasibleItineraries){
      if (itinerary.time < choice.time)
            choice = itinerary;
}
return choice;
}


WeatherCondition (ArrayList<Vehicle> feasibleVehicles,
Weather weatherForecast) {
if (weatherForecast.POP >= 40 ||
weatherForecast.temperature <= 18){
            feasibleVehicles.remove(bike);
            feasibleVehicles.remove(foot);
}
}

AppointmentTypeCondition (ArrayList<Vehicle>
feasibleVehicles, AppointmentType
insertedAppointmentType){//Avoid vehicles becuase of
appointment type
for (Vehicle vehicle : feasibleVehicles){
```

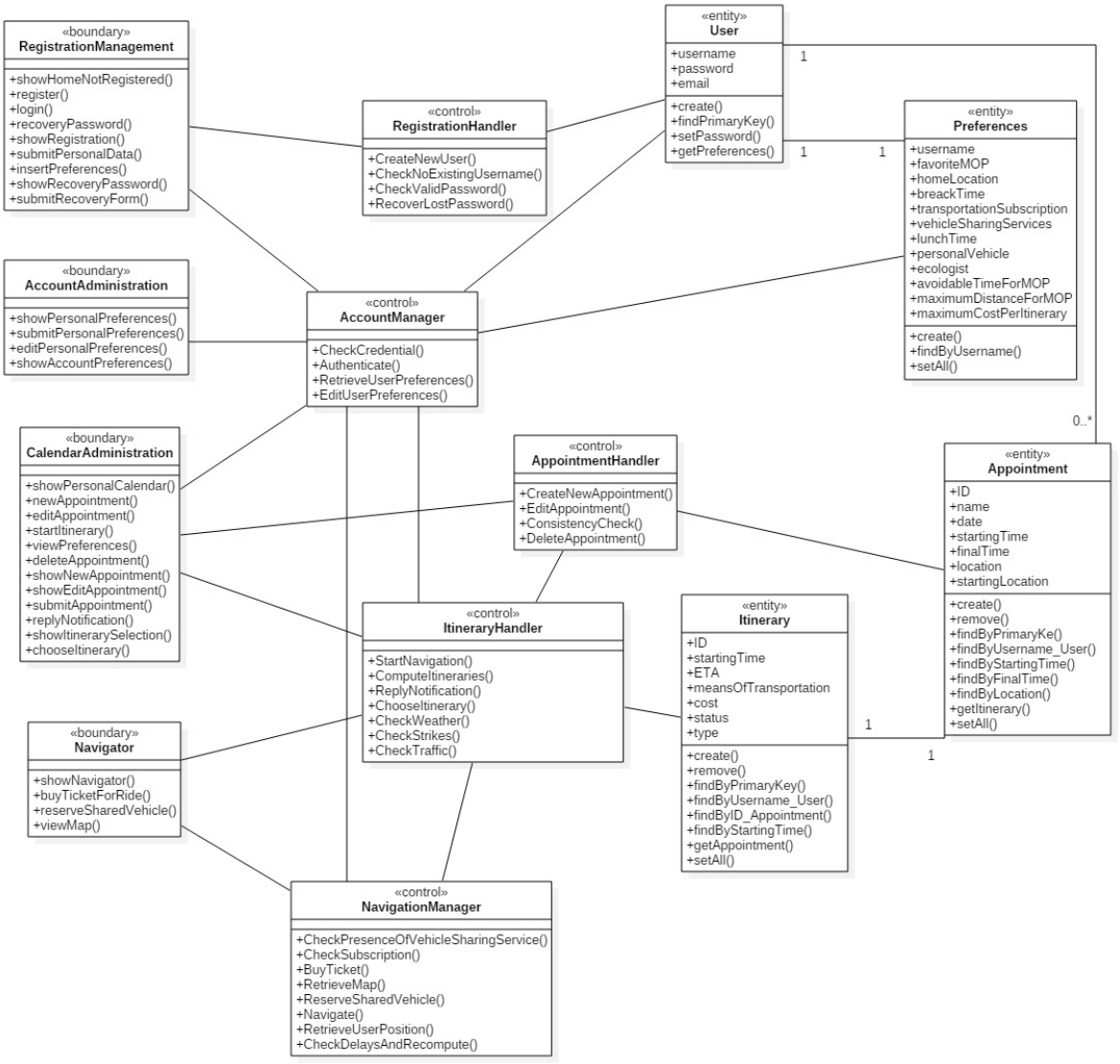
```
      if
(insertedAppointmentType.avoidedVehicles.contains(vehicle
)){
            feasibleVehicles.remove(vehicle);

EcologicCondition (boolean ecologist,
feasibleVehicles)//only bike and foot are permitted
If (ecologist){
feasibleVehicle.clean();
feasibleVehicle.add(bike);
feasibleVehicle.add(foot);
 }
}

isCoherent (int maxWalkDist, int maxCost, Itinerary
itinerary, int avoidableTimeForMOPinit,
avoidableTimeForMOPend){//no exceed maxWalkDist and
maxCost
      if (itinerary.walkDist < maxWalkDist &&
itinerary.cost < maxCost &&  itinerary.start
>avoidableTimeForMOPend && itinerary.finish <
avoidableTimeForMOPinit)
            return true;
      return false;

ConsistencyCheck (Itinerary permittedItinerary, int
priority, Appointment insertedAppointment){
for (DailySchedule dailySchedule : dailySchedules){
      if (dailySchedule.time == insertedAppointment.time){
            Array appointments =
dailySchedule.appointments;
            for (Appointment appointment : appointments){
                  if (appointment.start <=
insertedAppointment.start <= appointment.end ||
                  appointment.start <= insertedAppointment
<= appointment.end)
                        return false;
            }
      }
}
return true;
}
```

4.User interface design:

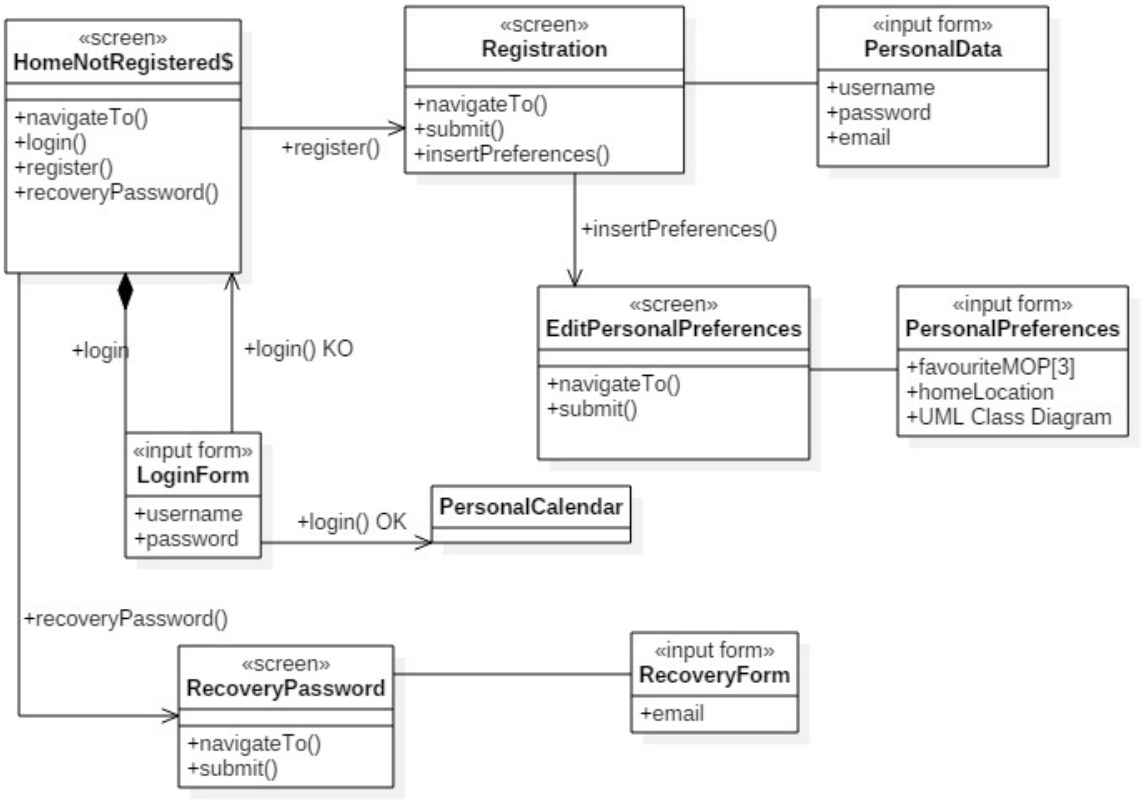
We repropose the screen already seen in the RASD.



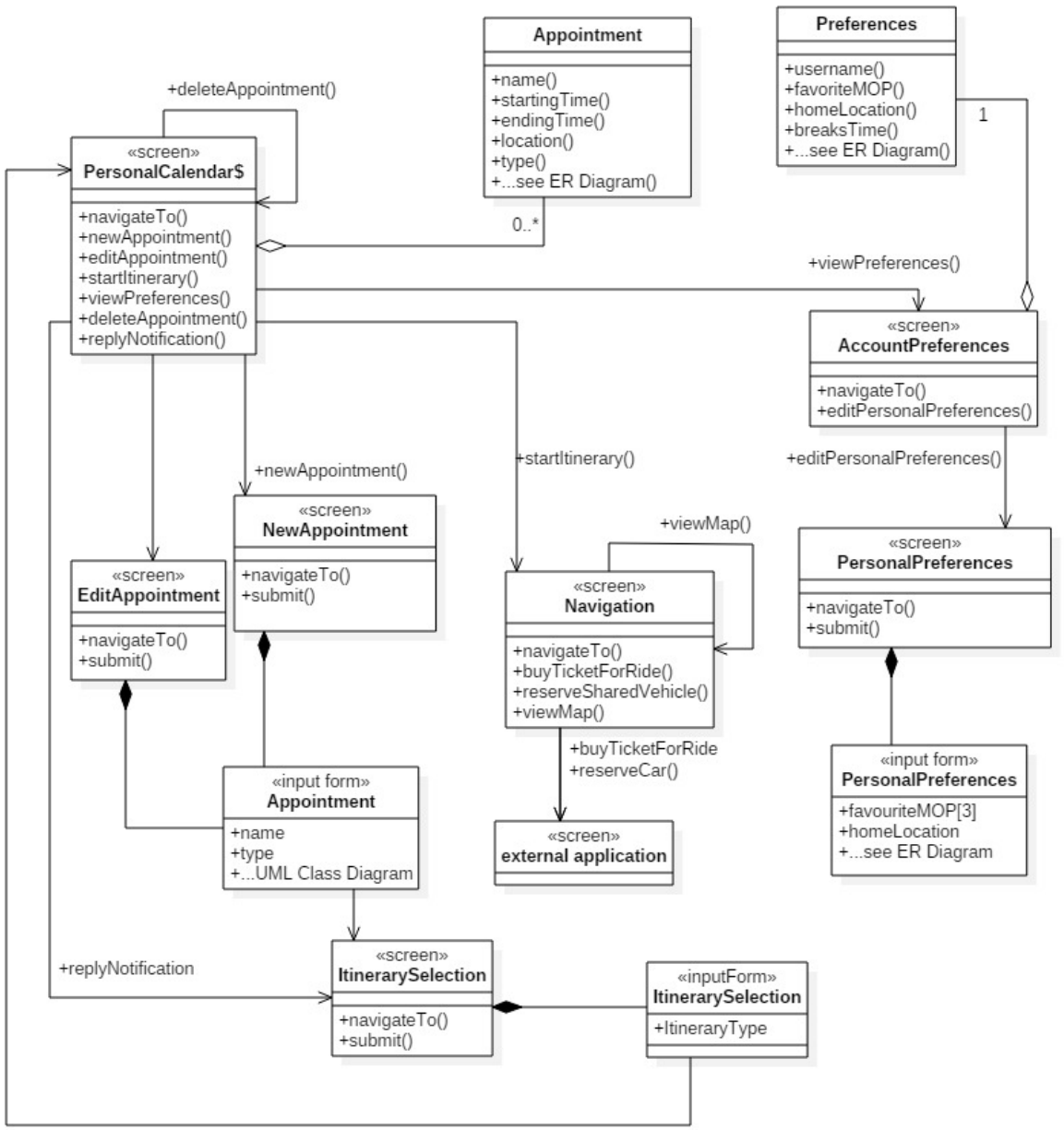
In-app screenshots


«boundary»
RegistrationManagement
+showHomeNotRegistered()
+register()
+login()
+recoveryPassword()
+showRegistration()
+submitPersonalData()
+insertPreferences()
+showRecoveryPassword()
+submitRecoveryForm()
«control»
RegistrationHandler
+CreateNewUser()
+CheckNoExistingUsername()
+CheckValidPassword()
+RecoverLostPassword()
«entity»
User
+username
+password
+email
+create()
+findPrimaryKey()
+setPassword()
+getPreferences()
1
1
1
«entity»
Preferences
+username
+favoriteMOP
+homeLocation
+breackTime
+transportationSubscription
+vehicleSharingServices
+lunchTime
+personalVehicle
+ecologist
+avoidableTimeForMOP
+maximumDistanceForMOP
+maximumCostPerItinerary
+create()
+findByUsername()
+setAll()
«boundary»
AccountAdministration
+showPersonalPreferences()
+submitPersonalPreferences()
+editPersonalPreferences()
+showAccountPreferences()
«control»
AccountManager
+CheckCredential()
+Authenticate()
+RetrieveUserPreferences()
+EditUserPreferences()
0..*
«boundary»
CalendarAdministration
+showPersonalCalendar()
+newAppointment()
+editAppointment()
+startItinerary()
+viewPreferences()
+deleteAppointment()
+showNewAppointment()
+showEditAppointment()
+submitAppointment()
+replyNotification()
+showItinerarySelection()
+chooseItinerary()
«control»
AppointmentHandler
+CreateNewAppointment()
+EditAppointment()
+ConsistencyCheck()
+DeleteAppointment()
«entity»
Appointment
+ID
+name
+date
+startingTime
+finalTime
+location
+startingLocation
+create()
+remove()
+findByPrimaryKe()
+findByUsername_User()
+findByStartingTime()
+findByFinalTime()
+findByLocation()
+getItinerary()
+setAll()
«control»
ItineraryHandler
+StartNavigation()
+ComputeItineraries()
+ReplyNotification()
+ChooseItinerary()
+CheckWeather()
+CheckStrikes()
+CheckTraffic()
«entity»
Itinerary
+ID
+startingTime
+ETA
+meansOfTransportation
+cost
+status
+type
+create()
+remove()
+findByPrimaryKey()
+findByUsername_User()
+findByID_Appointment()
+findByStartingTime()
+getAppointment()
+setAll()
1
1
«boundary»
Navigator
+showNavigator()
+buyTicketForRide()
+reserveSharedVehicle()
+viewMap()
«control»
NavigationManager
+CheckPresenceOfVehicleSharingService()
+CheckSubscription()
+BuyTicket()
+RetrieveMap()
+ReserveSharedVehicle()
+Navigate()
+RetrieveUserPosition()
+CheckDelaysAndRecompute()

BCE diagram



UX for NonRegisteredUser



UX RegisteredUser

5.Requirements traceability:

Here, we mean to map the main requirements (and the related goals), mentioned in the RASD document, on the components, presented in this document, which are involved in accomplishing them.

- [G1] Schedule user's appointments along the days:
  - [R.1 and R.2] Login/Registration: AccountManager;
  - [R.3 to 13] Appointments insertion/editing, consistency checking, breaks: AppointmentManager, ItineraryManager;
- [G2] Notify the user of incoming appointments, with details about starting time and mean of transportation.
  - [R.1] Login: AccountManager;
  - [R.2] Notify the user of coming appointments: NotificationHandler;
  - [R.3] Check of shared-vehicle in the neighborhood: Navigator;
- [G3] Propose a suitable itinerary throughout the appointments' locations, according to user's preferences, appointment description and external information about public transportation, weather forecast, traffic conditions.
  - [R.1 to 10] Login/ Customization of account preferences for the trips and information about personal vehicles: AccountManager;
  - [R.11 to 12] Customization of appointment priority: AppointmentManager;
  - [R.13 to 33] Contacting the external services and computing the itineraries taking into account the appointment's constraints and the external constraints (weather, strikes, etc..): ItineraryManager;
- [G4] Give the possibility to buy a transportation ticket.
  - [R.1] Login: AccountManager;
  - [R.2 and 3] Check of subscription / inform of possibility to buy a ticket: AccountManager, ItineraryManager, NotificationHandler;
  - [R.4] PayPal transaction for the ticket: PaymentManager
- [G5] Give the possibility to reserve a vehicle-sharing service.
  - [R.1 and 2] Login / Inserting public transportation subscription: AccountManager;
  - [R.3 and 4] Vehicle-Sharing-Service: ItineraryManager, Navigator;
- [G6] Guide the user through all the appointments' locations as a navigator.
  - [R.1] Login: AccountManager;
  - [R.2 and from 6 to 12] Consistency of trip/ check weather forecast/ recompute in case of unexpected events: ItineraryManager;

- [R.3 and 4] check vehicle-sharing-services, compute the correspondent itinerary and reserve the vehicle: ItineraryManager, Navigator;
- [R.5] Guide the user during the trip: Navigator;

6. Implementation, Integration and Test Plan

Implementation plan

The approach that will be adopted is bottom-up: this will allow us to test few components at a time and then integrate them at higher steps. The strategy, as we can see below, is the same for the testing strategy, in order to a more coherent project and to help testers and implementers to understand each other working in the same way and in the same order. For this reason, describing the interactions and so the integrations of the various component, we are talking about both from implementation and testing point of view.

Actually, the test strategy we will use is a Reactive approach, in which the testing is not started until after design and coding are completed.

Integration test strategy

Before beginning the test plan here we are some preconditions to be got in order to could have the possibility to successfully check the real operating application. First of all the RASD has to be completed,, having a general idea of the whole project. Then the percentage of the entire implemented application must be at least 90% with these details:

100% of the external services and interfaces with their API (Maps, Weather, ...).

90% of the service system with particular attention to the appointment manager and the itinerary manger which are the most important components in terms of functionality.

65% client side: it is important especially the local appointment manager in order to allow a correct consistency check of the appointment (the most critical part of the app's operation).

The integration test strategy is based on the bottom-up concept, testing first of all single components (Unit tests) and taking into account the various dependences between each other. Then we will test the groups of component integrated together and finally the entire module, the subsystem of every component. In fact there two main modules in the server side: the application logic module and the user dedicated module thanks to which we could separate the most critical logic part by the components that are more simple to implement and test. In conclusion we have final system tests, integrating all the modules together, followed by the usual acceptance test and the performance test, in order to verify that the application satisfies all the requirements described in the RASD and in the other previous parts. Thanks to this technique, first we could work on a specific functionality of the application and then, steps by steps, we could have the global view of the system rather than using a top-down strategy. Obviously, there are some modules that are more important than the other and this force us even to have an order in considering and testing components.

The components to be integrated are:

Application Logic Module

1. Step:

- Itinerary Manager
- Maps API
- Weather API
- DBMS
- Public Transportation Services

2. Step:

- Appointment Manager
- Itinerary Manager
- DBMS
- Maps API

3. Step:

- Navigator
- Appointment Manager
- Maps API
- Itinerary Manager

User Dedicated Module

4. Step:

- Payment Manager
- PayPal API
- DBMS

5. Step:

- Account Manager
- Payment Manager

6. Step:

- Notification Handler
- Itinerary Manager
- Account Manager

Here we show all the dependences between the main components of the application (on the server side) in order to highlight which component is necessary to develop and test before.

Integration phases

This section will provide information about how the integration testing will be carried out. For this part of the development, we will select a group of 5 persons in charge of giving inputs to the application and checking the output. The team will not be provided with the code, therefore they will not be influenced by looking at it.

We want to concentrate on enhancing end user experience, so it is important that every use case is tested and checked to work properly. The main goals of the application (which are all focused on the user) are mapped in requirements. Verification of all of these requirements is the aim of this testing process.

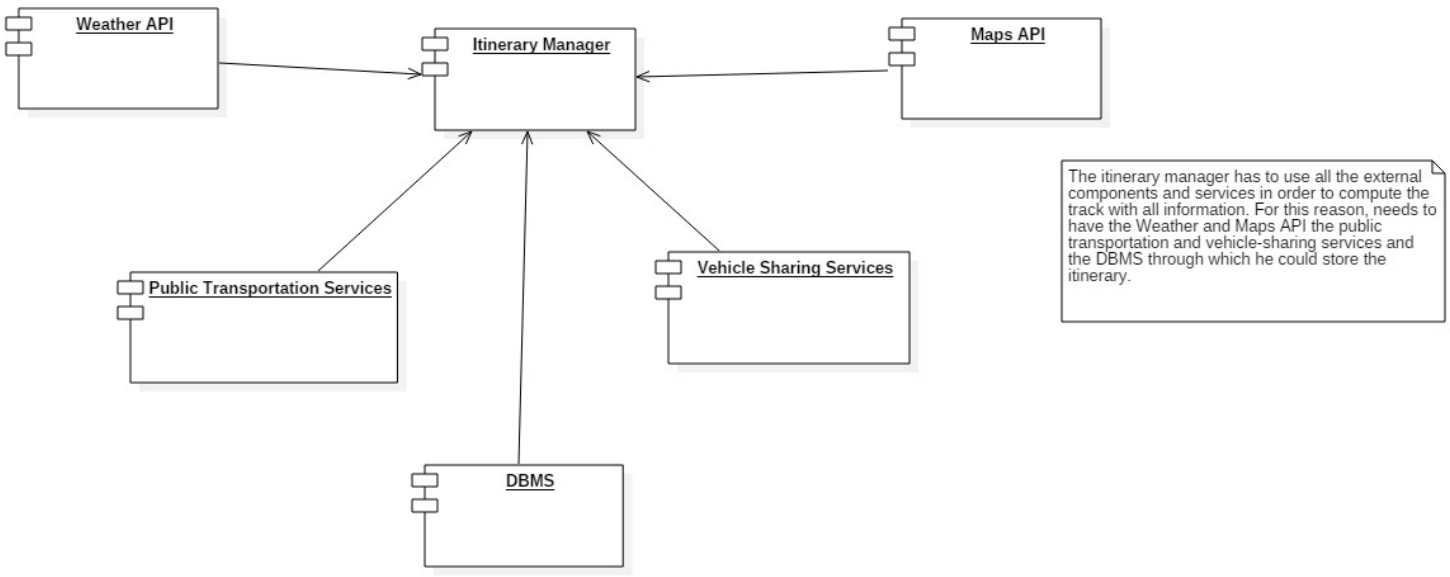
The approach is to test the requirements as soon as the relative components are implemented and integrated. In this way (following a bottom-up approach), we can discover bugs as soon as possible and we can fix them avoiding a cascade effect. This will also allow us to test few components at a time.

The following is the description of how we mean to proceed in testing phase: for each phase, the fundamental function to be tested is described.

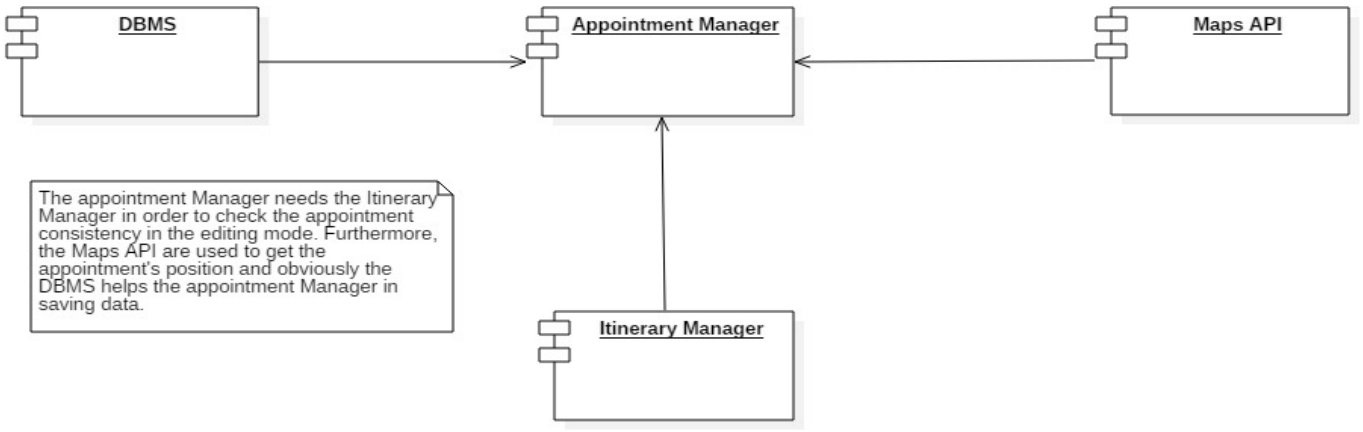
Every further step relies on previous ones and we assume that all the external services with their APIs and the DBMS perfectly work.

INTEGRATING UNITS

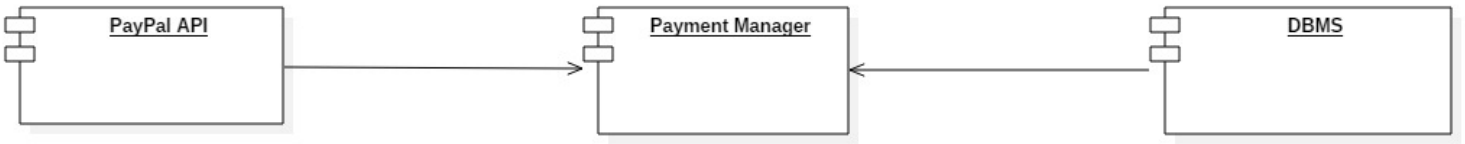
1. Step



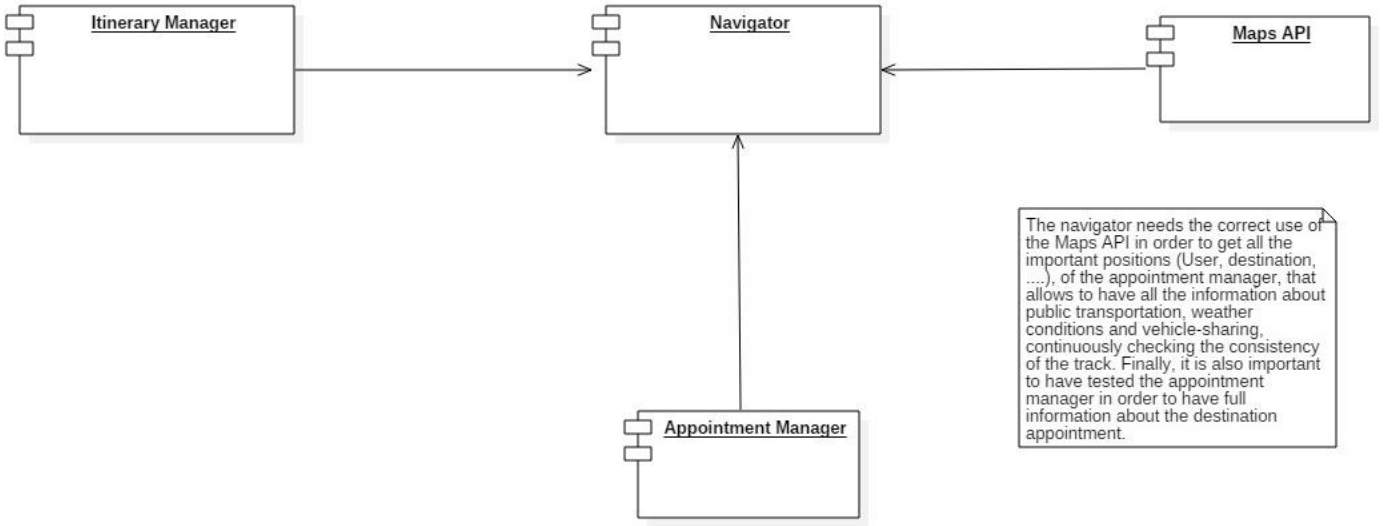
2. Step



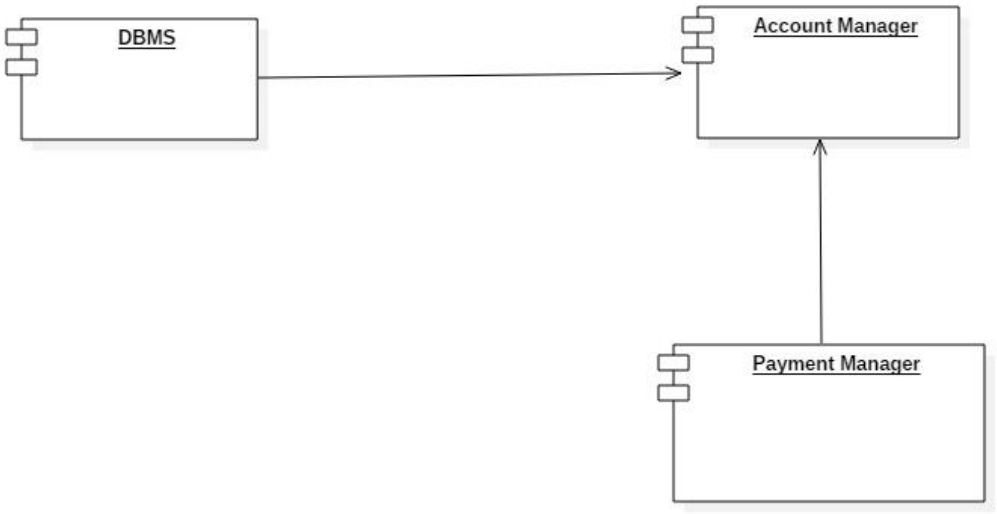
3. Step



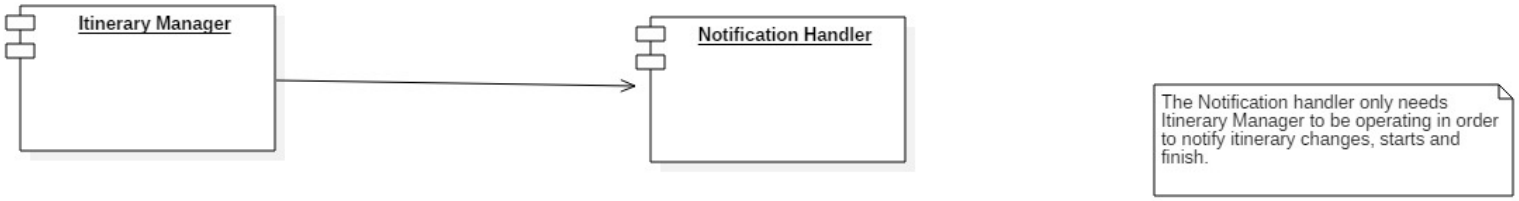
4. Step



5. Step

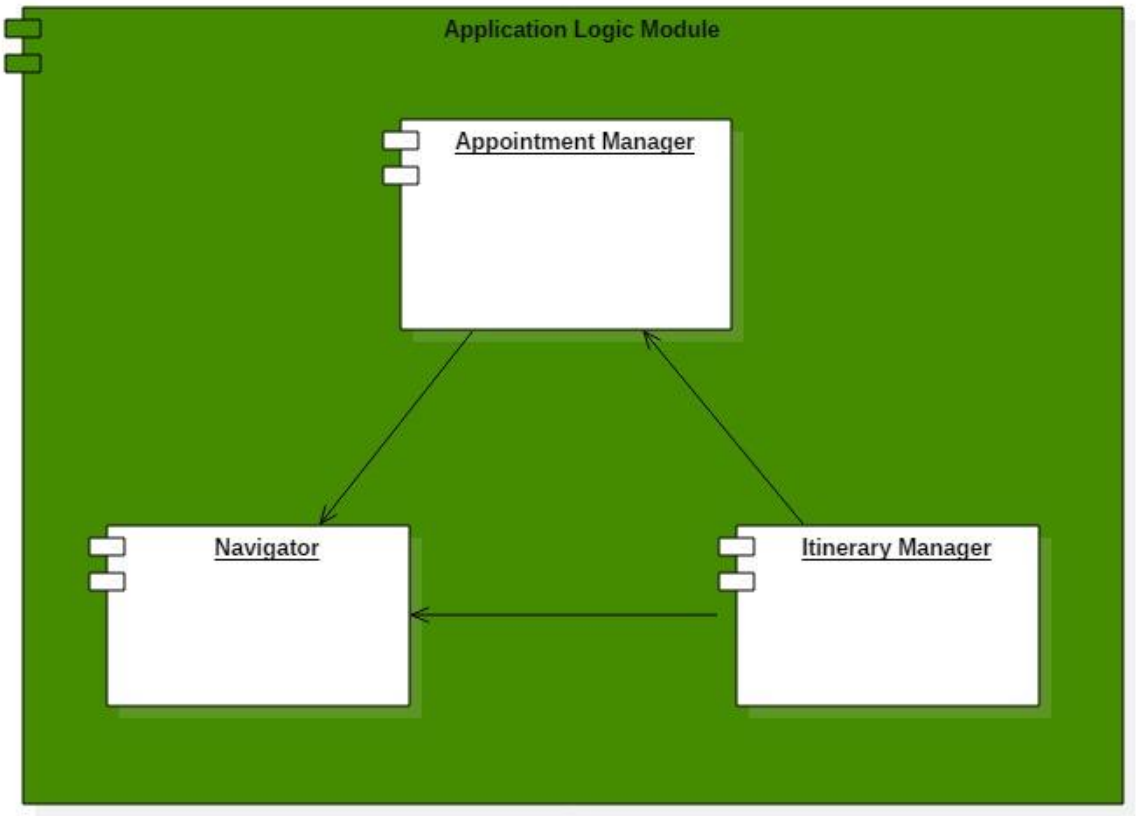

DBMS
Account Manager
Payment Manager
The operations of the Account Manager use some parts of the payment Manager, permitting to the user to pay the ticket if he wants, and the DBMS to have the data stored safely

6. Step

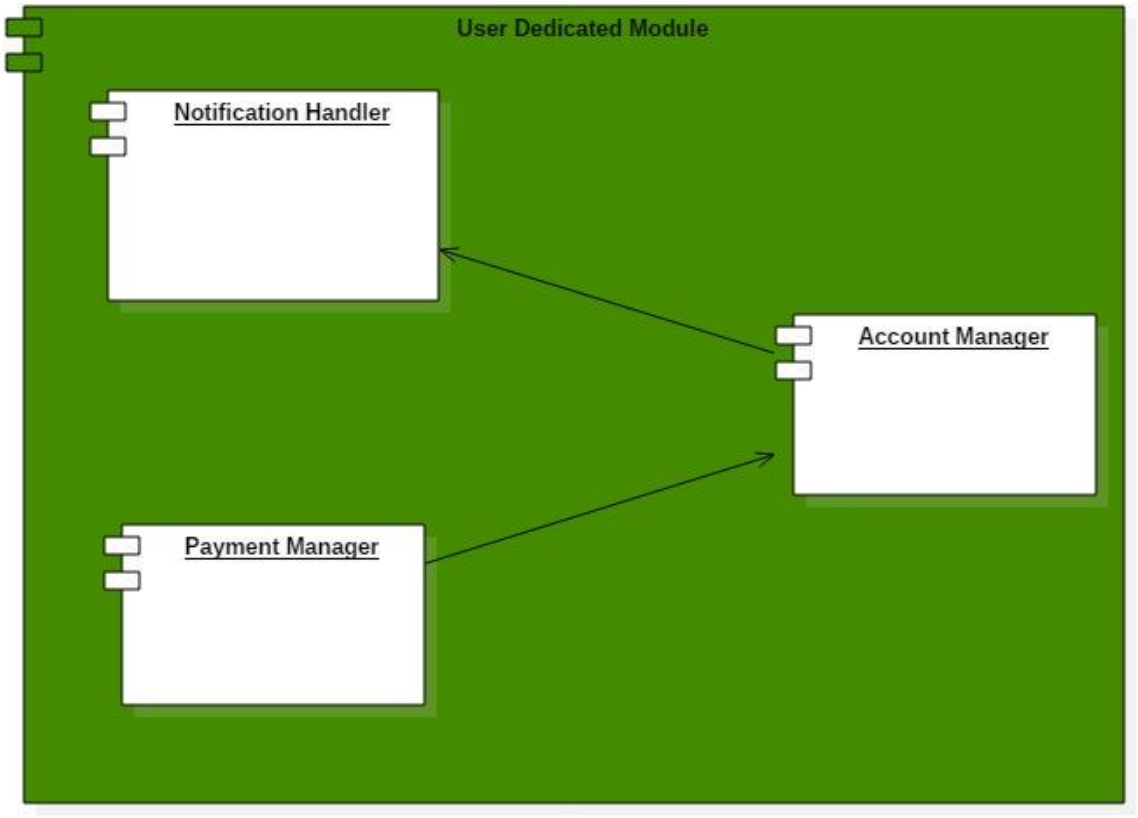

Itinerary Manager
Notification Handler
The Notification handler only needs Itinerary Manager to be operating in order to notify itinerary changes, starts and finish.

INTEGRATING MODULES

7. Step

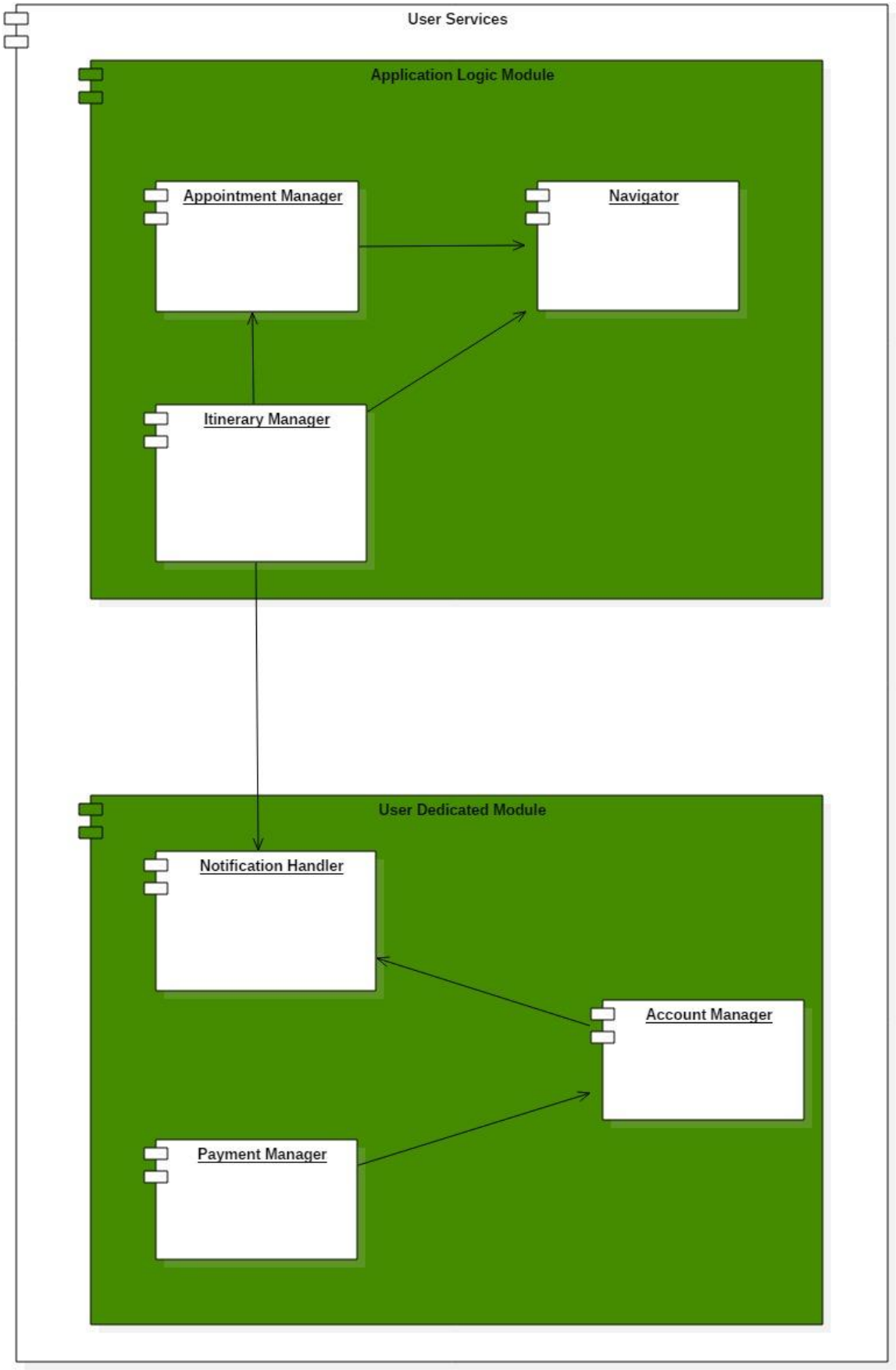


8. Step



INTEGRATING SYSTEM

9. Step



We also need the so called “drivers”, a software which manage the testing models creating the rights input for each one.

- NavigatorDriver: it guides tests invoking the methods of the Navigate Interface and input from Itinerary
- AccountDriver: it will invoke the profile Management Interface and input from Navigator
- AppointmentDriver: it will invoke the Calendar Management’s methods
- ItineraryDriver: it will invoke the Itinerary Management Interface’s methods
- PaymentDriver it will invoke the Buy Ticket Interface’s methods
- NotificationDriver: it will invoke the calls from Itinerary Manager

First phase: Account Manager

At the starting time, we first want to be sure about the user's details, testing if all the user's data (in terms of credentials and preferences) are correctly stored in the database.

TestID	t1
Name	Registration of a new account
Components to be tested	AccountManager, DBMS
Input	Username, Password, Email
Output	Check if the new account has been correctly stored in the database
Description	Account Manager contacts the DBMS in order to add the new account to the list of existing ones. It must happens only after clicking on the confirmation link sent by email.
Exception	UsernameAlreadyInUseException: the account must not be stored. InvalidEmailFormatException: the account must not be stored. PasswordConstraintsViolated: the account must not be stored.

TestID	t2
Name	Login of an existing account
Components to be tested	AccountManager, DBMS
Input	Username, Password
Output	Check if the login has been performed
Description	Account Manager contacts the DBMS in order to check the credentials.
Exception	NotExistingAccount: the client must remain in "not registered" status.

Second phase: Account Manager, AppointmentManager

From now on, we enter the most dense part of the project. Here we want to to test all the insertions dynamics.

For instance: insertion in lunch time should fail, it should be impossible to insert an appointment in the past, it should be impossible to insert overlapping appointments.

TestID	t3
Name	Insertion of a new appointment in an empty schedule
Components to be tested	AccountManager, AppointmentManager, DBMS
Input	Appointment details
Output	Check if the appointment has been inserted in the DBMS
Description	AppointmentManager stores the appointment details in the DBMS.
Exception	BreaksTimeException: if the appointment is held in the user's breaks time, it must not be stored.

TestID	t4
Name	Insertion of a new appointment in a non-empty schedule
Components to be tested	AccountManager, AppointmentManager, DBMS
Input	Appointment details
Output	Check if the appointment has been inserted in the DBMS
Description	Appointment Manager performs a consistency check and then contacts the DBMS in order to check whether the appointment has been stored.

Exception	OverlapsException: if there is an overlap, the appointment must not be stored. ImpossibleToReachException: if there is not a valid path to move between the previous appointment and the draft appointment (with respect to user's preferences, locations and travel time), the appointment must not be stored.

Third phase: AccountManager, AppointmentManager, ItineraryManager

In this phase, we plan to test how the computation of itineraries is performed, paying attention to the 5 categories highlighted in RASD (Shortest, Most Ecologic, Cheapest, MinimumChanges, MinimumWalkingDistance).

TestID	t4
Name	Computation of an itinerary between two appointments
Components to be tested	AccountManager, AppointmentManager, ItineraryManager DBMS
Input	2 Appointments' details
Output	Check if the itinerary has been inserted in the DBMS and it is suitable to get in time to the appointment
Description	AppointmentManager and the ItineraryManager perform a consistency check, then the ItineraryManager should compute the optimal path, according to user's preferences, retrieved by AccountManager
Exception	ImpossibleToReachException: if there is not a valid path to move between the previous appointment and the draft appointment (with respect to user's preferences, locations and travel time), the appointment must not be stored

Fourth phase: AppointmentManager, ItineraryManager, NotificationHandler

Here we test the notification feature of our system.

TestID	t5
Name	Notification of incoming appointment
Components to be tested	ItineraryManager, AppointmentManager, NotificationHandler
Input	Appointment details
Output	Check whether the application notifies the user with incoming appointment
Description	The system must notify the user of an incoming appointment 30 minutes earlier
Exception	-

Fifth phase: AppointmentManager, ItineraryManager, NotificationHandler, Navigator

Eventually, we test how all the components interact and cooperate, in order to let the user get in time to the appointment, checking functions like ticket purchase and reserving a vehicle.

TestID	t6
Name	Navigation toward an appointment location
Components to be tested	Navigator, ItineraryManager, AppointmentManager, Maps API, WeatherAPI, PublicTransportationServices, NotificationHandler
Input	A given appointment
Output	Check if the application guides you to the location
Description	The navigator must retrieve information about the appointment from the AppointmentManager, the path from the ItineraryManager and the maps must be provided by proper APIs to move into the map. By exploiting this information, it must provide the users with indications toward the appointment location
Exception	GenericDelayException/WeatherException/ TrafficException/StrikeException: the application must compute other itineraries and ask the user to select a new one (see RASD for further info).

TestID	t7
Name	Reserving a shared vehicle
Components to be tested	Navigator, ItineraryManager, AppointmentManager, Maps API, Vehicle Sharing Service
Input	Start a navigation near a vehicle sharing-service

Output	Check whether the application redirects to an external system.
Description	The navigator must retrieve all the information about vehicle sharing-services nearby and let the user click on the available vehicles. After the click, the application must redirect the user to the external system.
Exception	NotSuitableVehicleException: if the vehicle sharing service is not useful to get in time to the appointment, the navigator must not propose it to the user.

TestID	t8
Name	Buying a ticket
Components to be tested	Navigator, ItineraryManager, AppointmentManager, Maps API, PublicTransportationManager, PaymentManager
Input	Start a navigation with an itinerary that foresees public transportation.
Output	Check whether the application lets the user buy a transportation ticket.
Description	The navigator must retrieve the proper information from the scheduled public transportation service and ask the user if he wants to start a PayPal transaction to let the user buy a ticket, then the Payment Manager must handle the purchase.
Exception	StrikeException: the application must compute other itineraries and ask the user to select a new one (see RASD for further info). AlreadySubscribedException: if the user has got a subscription for the given public transportation service, the system must not ask the user to buy a ticket.

Tools

qTest

It is a scalable test management and automation tool. qTest has proven to make every step of the QA process faster, simpler and more efficient:
Manage Requirements, Test case repository, Test Execution, Defect Tracking, Reporting, and Integrations.

Furthermore qTEST is very common in Agile testing and development teams which could be a valid alternative to our proposed approach, not in terms of strategy but in terms of team-work organization.

Junit

As a Unit testing framework: Junit is the de facto standard library for unit testing in Java and it is supported out of the box by all major IDEs, following all our project choices.

In addition to these considerations, we plan to develop the application with the help of a static program analysis tool to continuous scan the code looking for bugs, vulnerabilities and bad-programming-practices. A good choice can be the use of SonarQube platform and Jenkins server.

7. Effort Spent

Alessandro Saverio Paticchio: ~28 hours
Andrea Tricarico: ~28 hours
Davide Santambrogio: ~28 hours

8. Document History

- Version 1.1:
  - Review of Component View Diagram;
  - Further explanation about test (we forgot to print the final version of the DD, where this content was already inserted);