POLITECNICO
MILANO 1863

Software Engineering II

Travlendar+

Design

Document

Authors:

Edoardo D'Amico
Gabbolini Giovanni
Parroni Federico

1st November 2017

Indice

Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to provide more technical details than the RASD about Travelendar+ system. This document is addressed to developers and aims to identify:

- The high level architecture
- The design patterns
- The main components and their interfaces provided one for another
- The Runtime behavior

1.2 Scope

Travlendar+ is an application that aims to ease the management of the daily appointments of a registered user by providing a desired schedule for those. The application runs within the context of a restricted geographic area and thus should consider only the travel means available in this scope. The application is able to interact with different data sources, in particular Travel Means APIs and User device GPS, and to arrange the appointments by exploiting these information. Once the schedule is computed, the application gives to the user the possibility to buy tickets for the involved travel means, then he/she can run it and follow directions provided by the application.

1.3 Definitions, Acronyms, Synonims

1.3.1 Definitions

Definition 1.3.1. The user data include:

- Schedules;
- Appointments;
- User parameters.

Definition 1.3.2. A device is a PC, a Tablet or a Smartphone in which run the last version of his O.S.;

Definition 1.3.3. A Schedule is a set of time-ordered and not overlapping appointments where their starting times are fixed and they're linked to each other by a path covered with a specific transportation mean;

1.3.2 Acronyms

- API: Application Programming Interface;

1.4 Revision history

1.5 Reference Documents

RASD document

1.6 Document structure

- Introduction: this section introduces the design document. It contains a justification of his utility
- Architecture Design: this section is divided into different parts:
    1. Overview: this section explains the division in tiers of our application and lists further details that were not specified in the RASD.

2. Components View: this sections gives a global view of the components of the application, other than a brief description, and describes how they communicate
3. Deployment view: this section shows the components that must be deployed to have the application running correctly
4. Runtime view: sequence diagrams are represented in this section to show how the components interact and collaborate during the fullfilling of a specific task. Each task is associated to each goal specified on the RASD document
5. Component interfaces: the interfaces between the components are presented in this section
6. Selected architectural styles and patterns : this section explains the architectural and design choices taken during the creation of the application

- Algorithms Design: this section describes the most important algorithmic parts. Pseudo code is used in order to hide unnecessary implementation details and abstract from a specific programming language
- Requirements Traceability: this section aims to explain how the decisions taken in the RASD are linked to design elements

Chapter 2

Architectural Design

2.1 Overview

Travlendar+ has a multi-tier architecture.

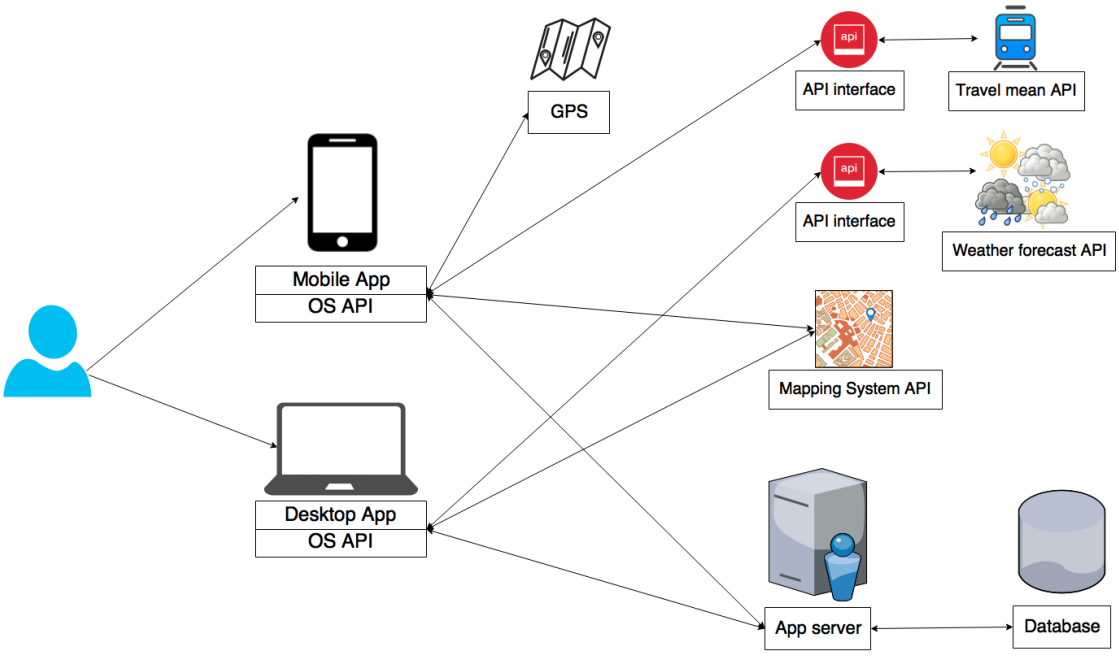

Figura 2.1: High level components

User can interact directly with both version of the application: desktop and mobile, both laying on their respective OS APIs. The schedule computation is performed locally

on the device in which the application runs by querying the GPS of the device and the external APIs:

- Mapping Service API
- Travel Means APIs
- Wheather Forecasts APIs

The communication between these components happens through different interfaces (wrappers), in order to improve the capability of the system to be expanded. In this way we provide a common pattern for the data acquired from the APIs, so that new ones can be easily included without the need of drastic changes on the central core. In the architecture there are two more components: the Application server, that offers authentication and sinchronization services and the Database used to grant the durability of the data. At the end of the registration phase a new record is allocated in the Database and it will be used for check the identity of a user during the login phase. The Database also stores the user data (1.3.1) so that they can be sinchronized in all the devices in which the user is logged in.

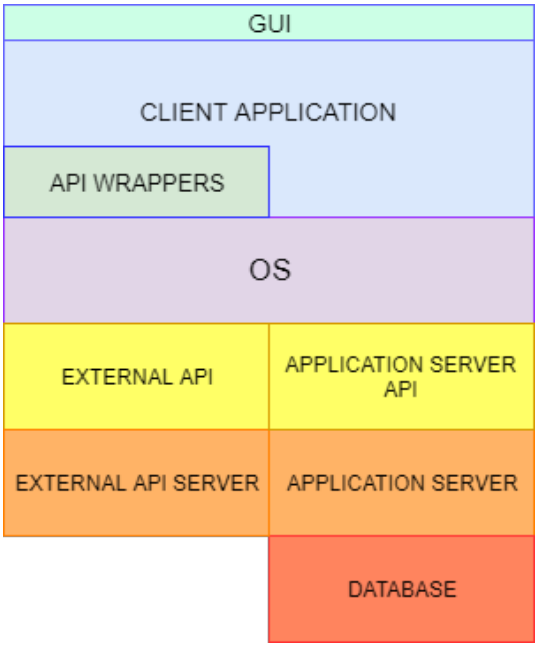

Figura 2.2: System stack

The figure above represents the stack architecture of our application and shows the hierarchy of the layers. Each level can communicate with the upper and lower layer.

2.2 Component View

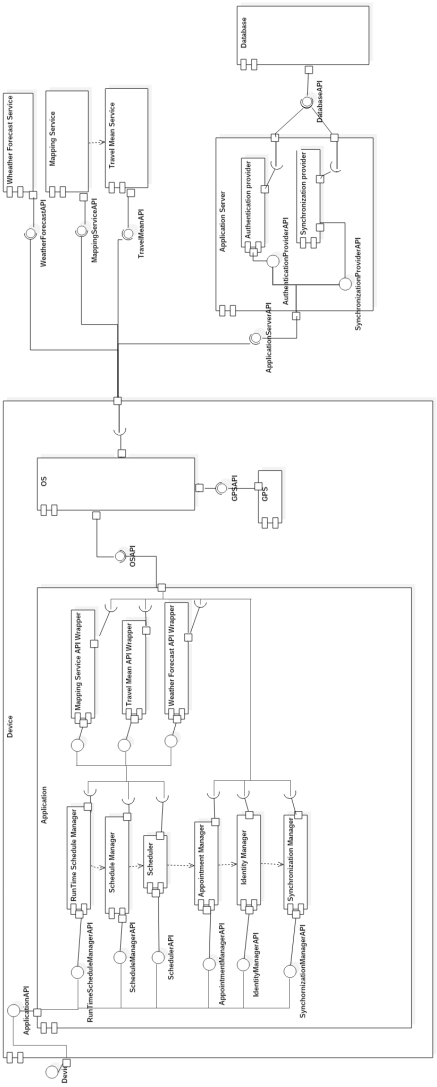

Figura 2.3: Component diagram

The application [1] includes several components:

[1] in the image is reported only the mobile version of the application since the desktop version is identical to it, with the only difference that it can't exploit GPS services, so for mantaining the graphic clear only one of the two is reported.

- Identity Manager: it requests a service to the authentication provider either for the login or the registration of a user.
- Appointment Manager: handles all the operations affecting appointment like modification, deletion and creation.
- Scheduler: it computes and saves the schedules of the user.
- Schedule Manager: handles all the operations affecting the schedule like showing and selecting the schedule to be run, notifying the user if a better schedule involving shared means has been found, buying the ticket for the means within the schedule.
- Runtime Schedule Manager: shows the directions and the progress of the running schedule.
- Synchronization Manager: it requests a synchronization to the Synchronization provider when needed.
- Mapping Service API Wrapper: manipulates data retrieved from a mapping service (e.g. Google Maps)in a format recognized by the application.
- Travel Mean API Wrapper: manipulates data retrieved from a travel mean API service (e.g. Mobike, ATM, etc) in a format recognized by the application.
- Weather Forecast API Wrapper: manipulates data retrieved from a weather forecast API service (e.g. Dark Sky) in a format recognized by the application.

The application Server is composed by the following two components:

- Authentication Provider: it handles the registration and the login phase of the user.
- Synchronization Provider: it has the purpose of synchronizing the user data (1.3.1) in the database when something is changed.

The three components representing services (Wheather Forecast API, Travel Mean API, Mapping Service API) aren't included in the system because they are external services but they have been represented since our system is using the interfaces exposed from them.

2.3 Deployment View

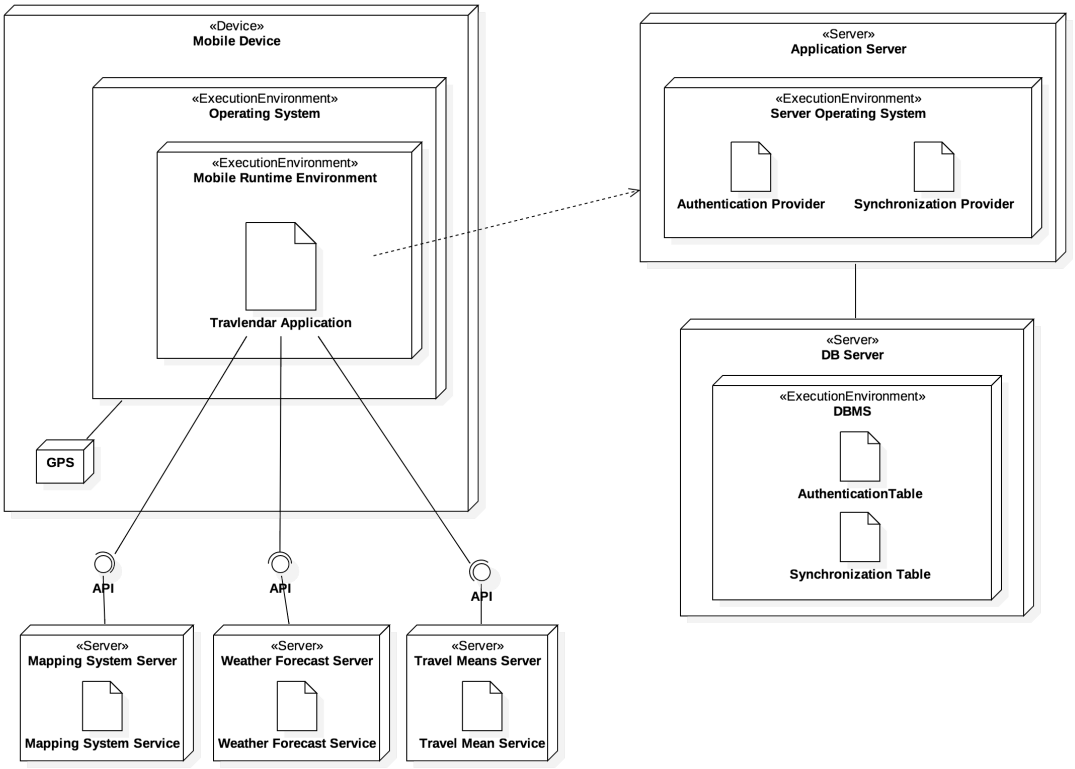

Figura 2.4: Deployment Diagram

2.4 Runtime View

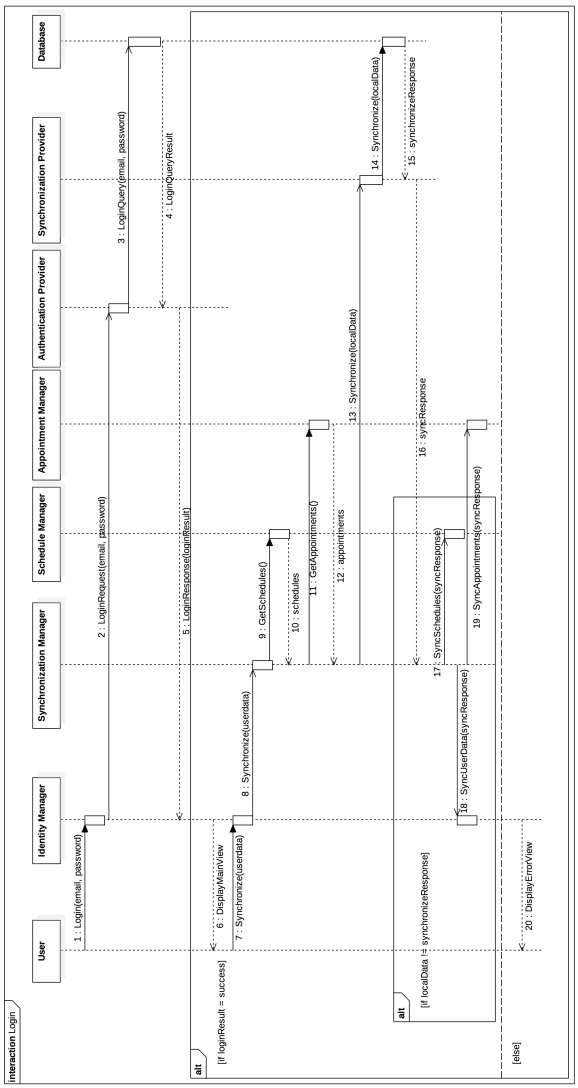

Figura 2.5: Login runtime view

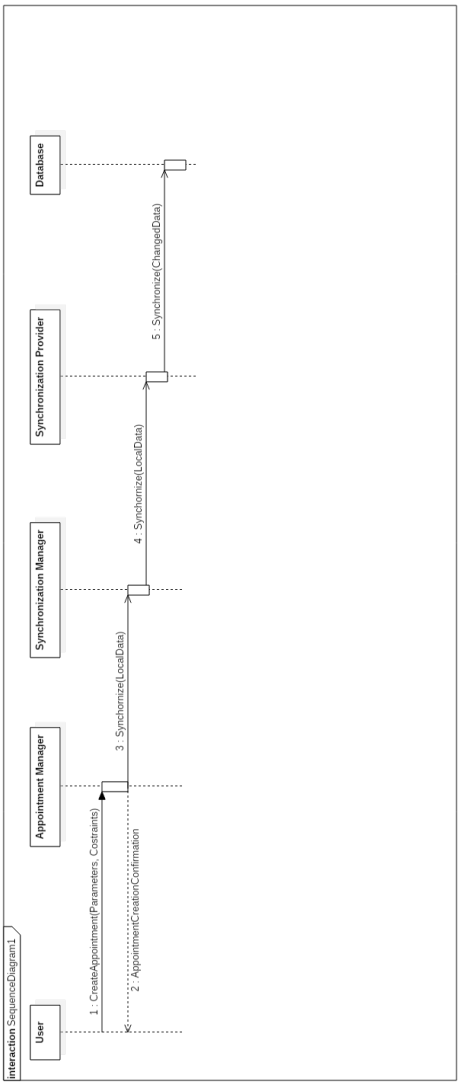

Figura 2.6: Appointment creation runtime view

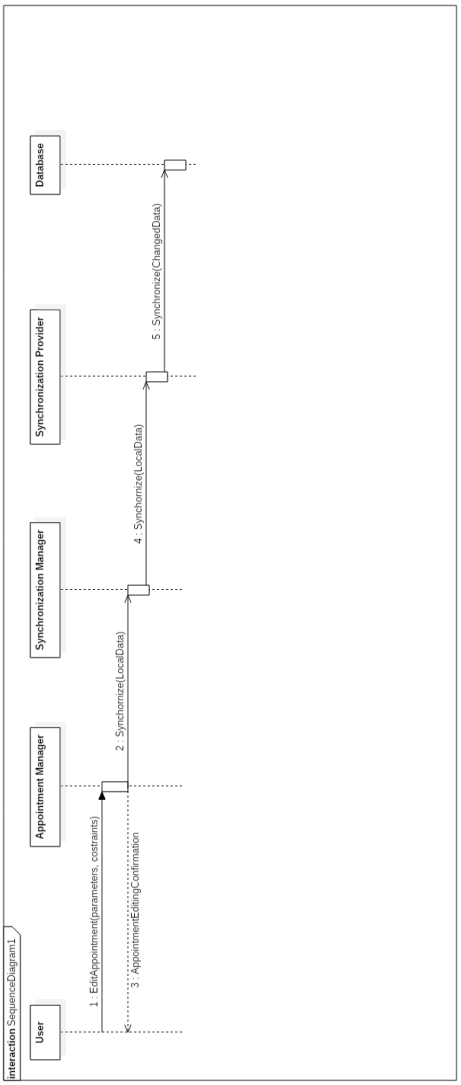

Figura 2.7: Appointment editing runtime view

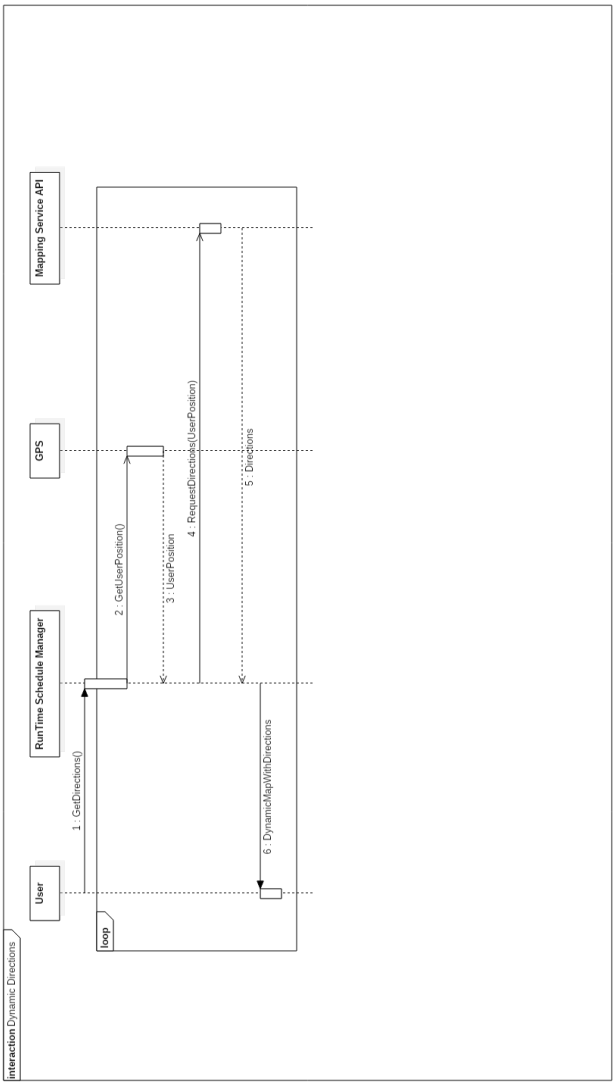

Figura 2.8: Dynamic directions runtime view

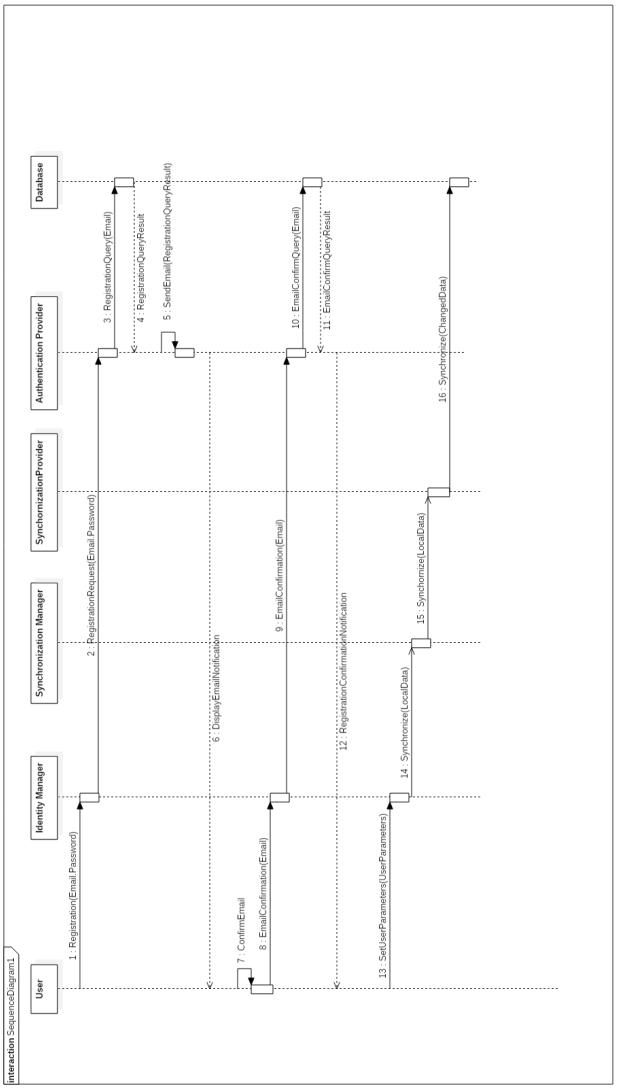

Figura 2.9: Registration runtime view

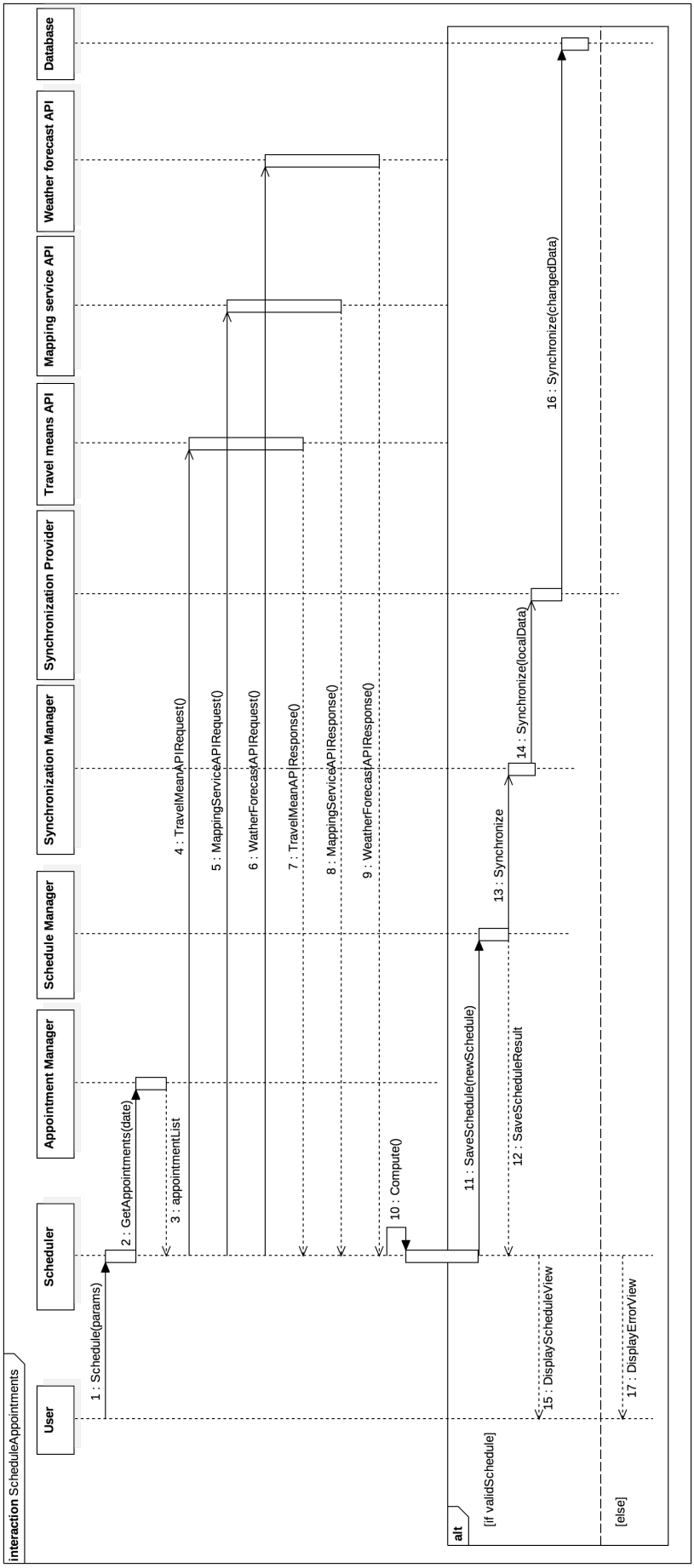

Figura 2.10: Schedule appointments runtime view

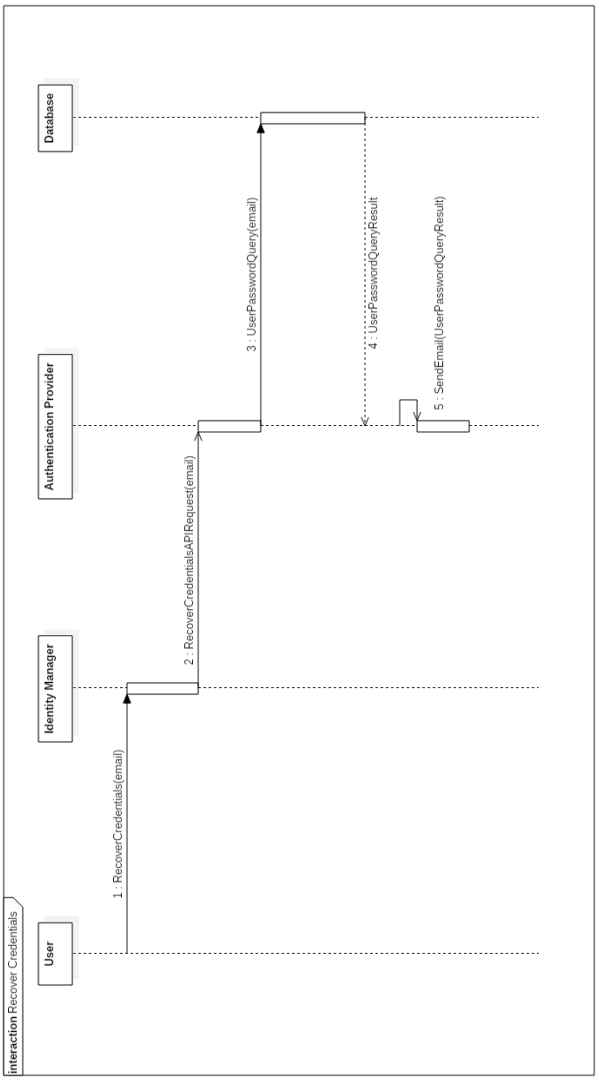

Figura 2.11: Recover Credentials Runtime View

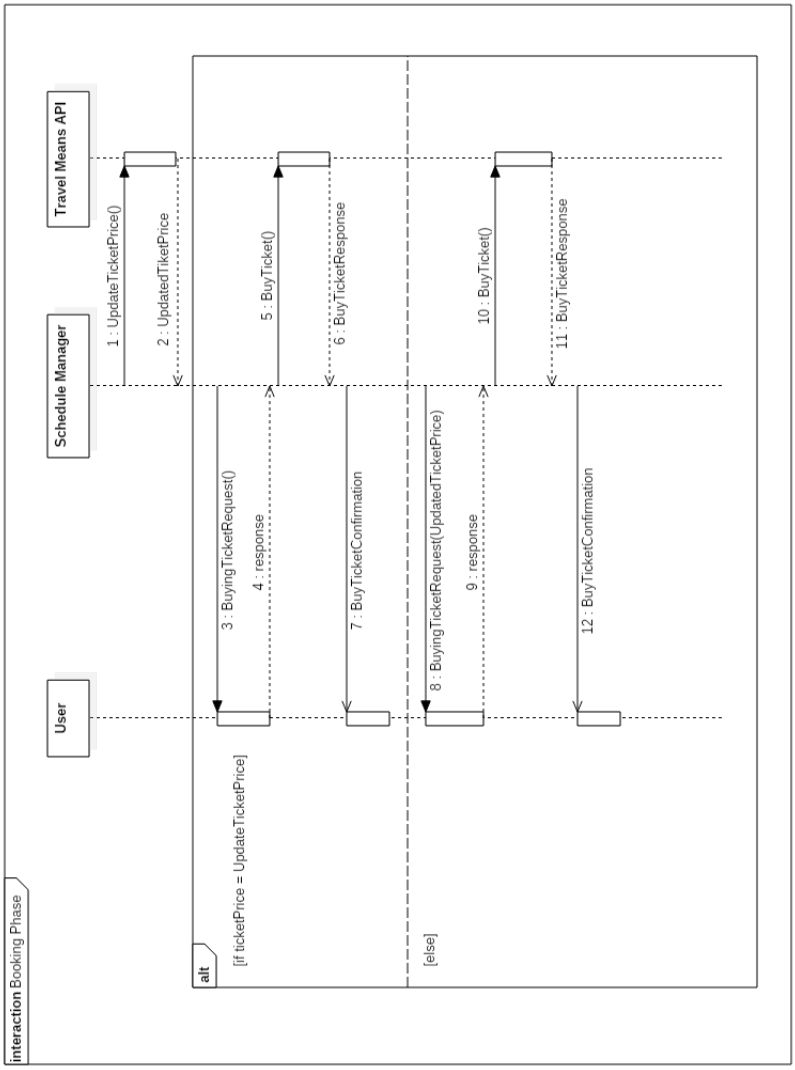

Figura 2.12: Booking Phase Runtime View

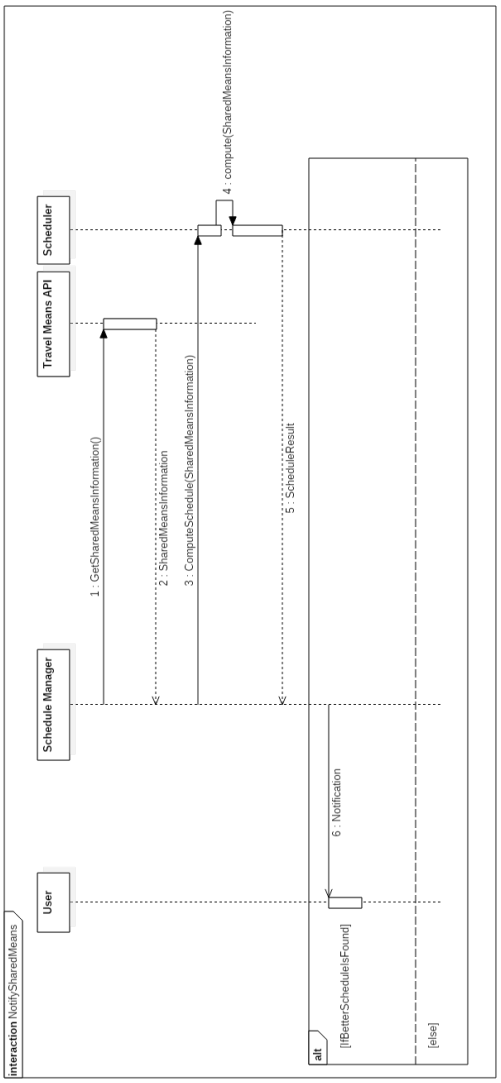

Figura 2.13: Notify Shared Means Runtime View

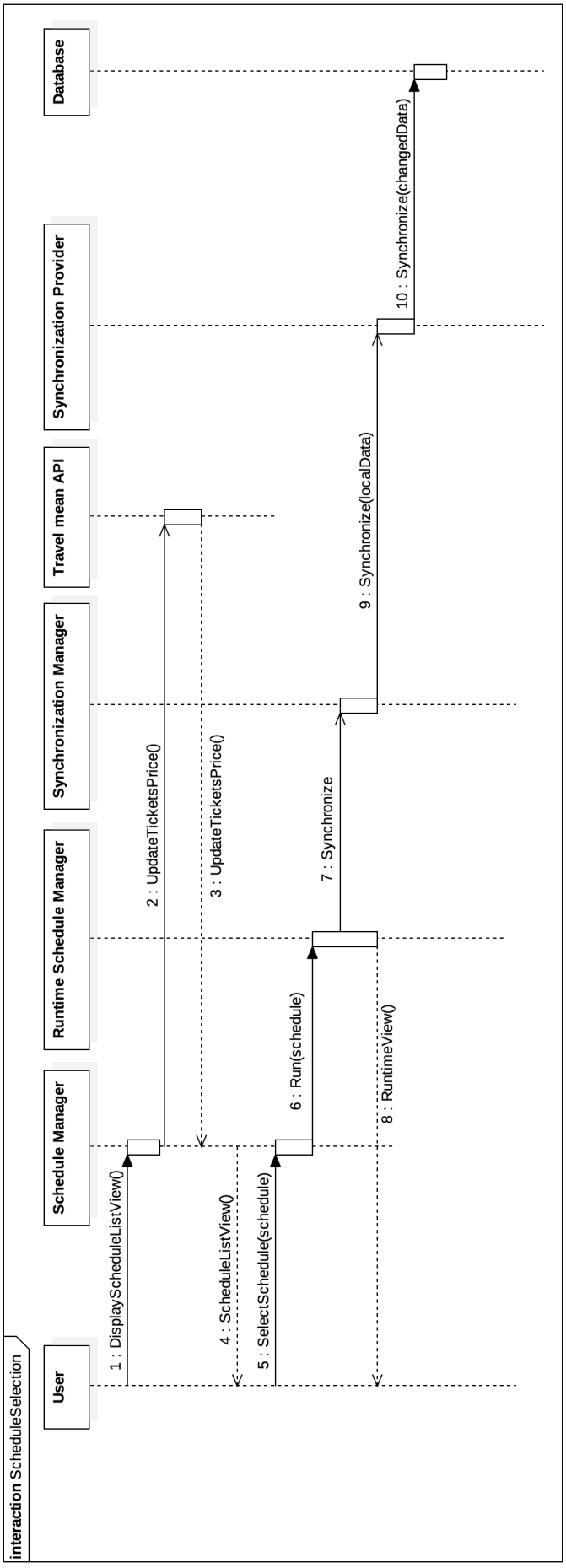

Figura 2.14: Schedule selection runtime view

2.4.1 Note on Runtime Views

In tbe diagrams the interaction involving Application Server, Application and API wrappers since they are not informative to show the behaviour of the application.

above only the interactions between internal components are shown.

2.5 Component Interfaces

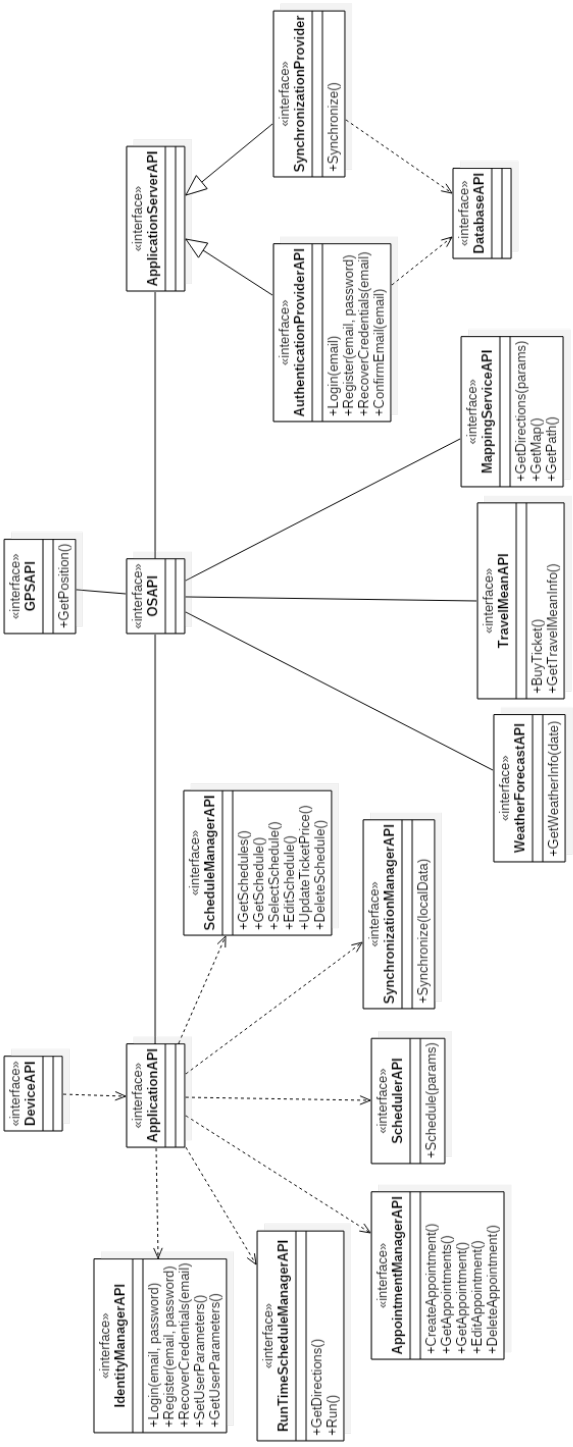

Figura 2.15: Schedule selection runtime view

2.6 Selected architectural styles and patterns

- MVC: all the classes of the application will adopt this pattern
- Adapter: API wrappers will adopt this pattern
- Client-Server: the application running on a device represents the client part, reqesting services to the Travlendar server
- OAuth: standard adopted by the exposed APIs of the server, in order to get authorization to make request from different clients
- Restful API: a particular kind of APIs that encapsulate the request/response body in a JSON string
- Singleton: restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. Some classes of our system can be built on this pattern, like the Travel Mean.

Chapter 3

Algorithmic Design

In this chapter are given the guidelines on how to implement the most important functionalities that the components of the system will offer. The pseudocode of the relevant method is shown.

3.1 Login

The Login involves the application the application server and the database. The last one is located outside the application server. To login in to the application the following steps must be done:

1. the user presses the Login button in the Login view;
2. the Identity manager (Login method) sends a request to the Application Server for authentication
3. A query on the database is performed to check the validity of the user credential and a response to the client is returned;

```
//client side
Login(username, password)
   response = SendRequest("api/login", username, password)
   if response.isValid
      token = response.GetToken
      Show(HomeView)
   else
      Show(ErrorMessage , "Invalid Credentials")
```

```
//server side
LoginRequest(username, password)
   result = database.query(username, password)
   if result == 1
      SendAuthenticationResponse (token)
   else
      sendAuthenticationError(error code)
```

3.2 CreateSchedule

correggere formula The process of computing a schedule is composed by the following steps:

1. Create the predecessor matrix (P) in which every cell (i,j) contains:

$$P_{ij} = \begin{cases} 1, & if Appointment_i must precede Appointment_j \\ 0, & if Appointment_i must follows Appointment_j \\ -1, & if Appointment_i can be scheduled both before or after appointment_j \end{cases} \tag{3.1}$$

2. Compute all the possible ordered arrangements of appointments with respect to the predecessor matrix

3. For each appointment in each arrangement, set the starting time according to travel time. This is estimated considering the euclidean distance and heuristics on the kind of travel mean between every pair of consecutive appointments in the arrangement, considering the bestTravelMean(**fare def**);

4. Check the feasibility of the arrangements and discard the ones which have overlapping appointments;

5. Choose the most convenient one, according to the optimization criteria.

6. Call mapping service Api to fix the effective routes between the appointments

```
ComputeSchedule(wakeUpTime, startingLocation, appts, constraints, optCriteria)
   p=CalculatePredecessorMatrix(appts)
   a=CalculateArrangements(appts, p, 0, 1)
   SetStartingTime(a, startingTime, startingLocation, constraint, optCriteria)
   s=ChooseBestSchedule(a)
   MappingServiceRequest(s)
   return s

CalculatePredecessorMatrix(appts)
   p=new Matrix[appt.size,appt.size]
   for(i=0 .. appts.size)
      for(j=i+1 .. appts.size)
         a1=appts[i]
         a2=appts[j]
         if a1.deterministic && a2.deterministic
            if a1.startingTime < a2.startingTime
               pred[i,j]=1
            else
               pred[i,j]=0
         elseif a1.deterministic && !a2.deterministic
            if a1.startingTime < a2.timeSlot.start
               pred[i,j]=1
            elseif a1.endingTime > a2.timeSlot.end
               pred[i,j]=0
            else
               pred[i,j]=-1
         elseif !a1.deterministic && a2.deterministic
               if a1.timeslot.end < a2.endingTime
                  pred[i,j]=1
               elseif a1.timeSlot.start > a2.startingTime
                  pred[i,j]=0
               else
                  pred[i,j]=-1
         elseif !a1.deterministic && !a2.deterministic
               if a1.timeSlot.end < a2.timeSlot.start
                  pred[i,j]=1
```

```
            elseif a1.timeSlot.start > a2.timeSlot.end
               pred[i,j]=0
            else
               pred[i,j]=-1
   return p


CalculateArrangements(appts, p, curri, currj)
   arrangement=new List
   for(i=curri .. appts.size-1)
      for(j=currj-1 .. appts.size)
         if p[i,j]==-1
            p0=p
            p0[i,j]=0
            CalculateArrangements(appts, p0, i, j)

            p1=p
            p1[i,j]=1
            CalculateArrangements(appts, p1, i, j)

            return

   a=ConvertPredMatrixToList(appts,p)
   arrangement.addLast(a)
   return arrangement


ConvertPredMatrixToList(appts, p)
//converts a "-1 free" predecessor matrix to an ordered list of appointments


SetStartingTime(a, startingTime, startingLocation, constraint, optCriteria)
   for(arr in a)
      dummyStartingAppt = new appointment(startingTime, startingLocation,
          duration=0)
      arr.addFirst(dummyStartingAppt)
      for(i=1 .. arr.size)
```

```
        appt1=arr[i-1]
        appt2=arr[i]
        travelMean=getBestTravelMean(appt1, appt2, constraint, optCriteria)
        travelTime=travelMean.estimateTime(appt1, appt2)
        if appt2.deterministic
           appt2.startingTravelTime = appt2.startingTime-travelTime
           appt2.travelMean=travelMean
           if appt1.endingTime > appt2.startingTravelTime
              error("schedule not feasible")
        else
           appt2.startingTravelTime =
               max(appt1.endingTime,appt2.timeSlot.start-travelTime)
           appt2.travelMean=travelMean
           if appt2.startingTravelTime > appt2.timeSlot.end
              error("schedule not feasible")


getBestTravelMean(appt1, appt2, constraint, optCriteria)
   l=getNotConstrainedTravelMeans(constraint)
   for(t in l)
      switch optCriteria
         case "MoneySpent"
            t.cost=estimateMoney(t, appt1, appt2)
         case "Time"
            t.cost=estimateTime(t, appt1, appt2)
         case "CarbonFootprint"
            t.cost=estimateCarbon(t, appt1, appt2)
   sortByCriteria(l, optCriteria)
   return l

ChooseBestSchedule(a)
   best=a[0]
   for(i = 0 .. a.size)
      sum=0
      for(appt in a[i])
         sum+=appt.travelMean.cost
      a[i].totalCost=sum
```

```
      if sum<a[0].totalCost
         best=a[i]
   return best

MappingServiceRequest(s)
   for (i=0 .. s.size-1)
      response=MappingServiceRequest(s[i], s[i+1])
      s[i].path=response.path
      s[i].startingTime=response.startingTime
```

3.3 Registration

The Registration involves the application the application server and the database. The last one is located outside the application server. To register in to the application the following steps must be done:

1. The user presses the Registration button in the Login view;
2. The Identity manager (Registration method) sends a request to the Application Server for authentication
3. A query on the database is performed to check the presence of the user credential and a confirmation email is sent
4. The user confirms the email by clicking on the designated link
5. The user's state on the database becomes confirmed

```
//client side
Register(username, password)
   response = SendRequest("api/registration", username, password)
   if !response.valid
      Show(ErrorMessage , "User Already Registered")



//server side
RegistrationRequest(username, password)
```

```
   result = database.query(username, password)
   if result == 0
      database.insertTuple(username, password)
      sendEmail(username)
   else
      sendAuthenticationError(error code)

EmailConfirmationRequest(username)
   result = database.modifyTuple(username, confirmed=true)
```

3.4 Synchronize

The synchronization in our system is the process that aims to keep the data consistent and updated between different devices. For example, when a user inserts an appointment on one of his devices, then the changes must be propagated to all his other devices, once the login is performed. The Synchronization involves the application, the application server and the database. The last one is located outside the application server. To sinchronize data across multiple devices two actions must be carried out:

- Upload of local data on the database when a single change on the client's local data occours(Synchronize Upwards);
- Download of data from the database, if an update is necessary, when login is performed(Synchronize Downwards).

```
//client background processes

SynchronizeUpwards(changedData)
   update = false
   while !update
      response = SendRequest("api/sync/up", token, changedData)
      if response
         update = true

SynchronizeDownwards()
   newData=SendRequest("api/sync/down", token)
```

```
   localData=newData


//server side

SynchUpwardsRequest(token, changedData)
   userID = database.getUser(token)
   if changedData.action == insert
      //changedData is a newly inserted element
      database.insert(changedData, userID)
   else if changedData.action == edit
      database.update(changedData, userID)
   else
      database.delete(changedData, userID)
   sendResponse("syncresult", true)

SynchUpwardsRequest(token, changedData)
   userID = database.getUser(token)
   data=database.getUserData(userID)
   sendResponse(data)
```

Chapter 4

User Interface Design

We refer to chapter 3.1.1 of the RASD, where we have given a complete overview of how the user interfaces will look like.

Chapter 5

Requirement Traceability

In each section section we will describe how the requirements of each goal are mapped in the design elements described above.

5.1 Goal 1

- *The system S.B.A to handle a registration phase in which the user will provide an e-mail and a password*
  - Login Interface
  - Identity Manager
  - Authentication Provider
  - Database
- *The system S.B.A. to verify the e-mail given by the user by sending a confirmation e-mail to his address*
  - Login Interface
  - Identity Manager
  - Authentication Provider
  - Database
- *The system should let the user to specify his parameters*
  - User Account Interface

5.2 Goal 2

- *The system S.B.A to recognize a registered user given an e-mail and a password*
  - Login Interface
  - Identity Manager
  - Authentication Provider
  - Database
- *The system S.B.A. to retrieve information from the user about his registration informations, i.e. his e-mail and password*
  - Login Interface
  - Identity Manager
  - Authentication Provider
  - Database
- The system must update the local date if some changes on them have been done from other devices.
  - Synchronization Manager
  - Synchronization Provider
  - Database

5.3 Goal 3

- *The system should be able to retrieve an e-mail address from the user*
  - Login Interface
- *The system should be able to send the password of a user to his e-mail given it*
  - Authentication Provider
  - Database

5.4 Goal 4

- *The system S.B.A. to retrieve information from the user about his appointments*
  - Appointment CRUD interface. **agg alle def**
- *The system S.B.A. to store an appointment in his memory*
  - Appointment Manager

5.5 Goal 5

- *The system should let the user change the parameters and the constraints of an inserted appointment*
  - Appointment CRUD interface. **agg alle def**
  - Appointment Manager
- *The system S.B.A to rewrite the appointment in his memory with his new parameters*
  - Appointment Manager

5.6 Goal 6

- *Allow the user to set the constraints of the schedule*
  - Schedule Interface
- *Allow the user to set the optimization criteria for the schedule*
  - Schedule Interface
- *Allow the user to set the variables for the schedule*
  - Schedule Interface
- *The system S.B.A. to gather information from external APIs about:*
  - *travel options with related travel option data;*

  - *weather forecast;*
  - *strike days;*
  - *delays.*

  - Mapping Service API Wrapper
  - Travel Mean API Wrapper
  - Weather Forecast API Wrapper

- *The system S.B.A. to select the best travel option according to the optimization criteria taking into account:*
  - *user constraint;*
  - *user parameters;*
  - *travel option data;*
  - *weather forecast;*
  - *information about strike day;*
  - *information about delays.*

  - Scheduler

- *The system S.B.A to store valid schedules requested by the user*
  - Schedule Manager

5.7 Goal 7

- *The system should let the user accept a schedule from the saved ones*
  - Schedule List Interface
  - Schedule Manager

5.8 Goal 8

- *The system S.B.A to book a travel mean through external API offered by third part application in which the user is signed*

  - Schedule Manager
  - Travel Mean API Wrapper

5.9 Goal 9

- *The system S.B.A to retrive the position of the user from his GPS*
  - GPSAPI
  - Runtime Schedule Manager
- *The system S.B.A. to retrive from an external API the directions to give to the user for reach the next appointment*
  - Mapping Service API Wrapper
  - Runtime Schedule Manager
- *The system S.B.A. to retrive from an external API the Graphical representation of the path that will be travelled by the user*
  - Mapping Service API Wrapper
  - Runtime Schedule Manager

5.10 Goal 10

- *The system should be able to send notifications to the user*
  - Schedule Manager
- *The system should be able to retrieve information about the availablity of shared travel means from external APIs without the user request*
  - Mapping Service API Wrapper
  - Schedule Manager

Chapter 6

Implementation Integration and TestPlan

In the next sections we will show the timeline of the implementation integration and testplan. This timeline is led by the functionalities, for each one we first implement the components needed, and then we proceed with the integration and with the test of the functionality.

6.1 User Authentication

Implemented and integrated Components:

- Authentication Provider
- Database
- Identity Manager

Functionality tested:

- User Registration
- User Login
- Recover credentials
- SetUserParameters
- GetUserParameters

6.2 Appointments management

Implemented and integrated components:

- Appointment manager

Functionality tested:

- Appointment creation
- Appointment editing
- Appointment deletion
- Appointments retrieving

6.3 External Api Data Retrieving

Implemented and integrated components:

- Mapping Service API Wrapper
- Travel Mean API Wrapper
- Weather Forecast API Wrapper

Functionality tested:

- Weather forecast data retrieving
- Strike day information retrieving
- Travel option data retrieving
- Get ticket price
- Purchase ticket

6.4 Static Schedule Management

Implemented and integrated components:

- Scheduler

- Schedule Manager

Functionality tested:

- Compute a schedule
- Schedules retrieving
- Schedule selection
- Schedule editing
- Schedule deletion
- Schedule price updating

6.5 Dynamic Schedule Management

Implemented and integrated components:

- Runtime Schedule Manager

Functionality tested:

- Directions retrieving
- Dynamic directions displaying

6.6 Synchronization

Implemented and integrated components:

- Synchronization provider
- Synchronization Manager

Functionality tested:

- Synchronize

Chapter 7

Effort Spent

- Federico Parroni: **44 hours**;
- Edoardo D'Amico: **44 hours**;
- Giovanni Gabbolini: **44 hours**.