Politecnico di Milano
AA 2017-2018

Computer Science and Engineering

Software Engineering 2 Project

Travlendar+

Design Document

Andrea Mafessoni - 899558
Andrea Mazzeo - 895579
Daniele Moltisanti - 898977

Deliverable:	DD
Title:	Design Document
Authors:	Andrea Mafessoni, Andrea Mazzeo, Daniele Moltisanti
Version:	1.0
Date:	26-November-2017
Download page:	https://github.com/AndreaMazzeo289/MafessoniMazzeoMoltisanti.git
Copyright:	Copyright © 2017, Andrea Mafessoni, Andrea Mazzeo, Daniele Moltisanti – All rights reserved

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of this document is to describe the architecture of the Travlendar+ system. It provide more technical details that the RASD.

1.2 Scope

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

1.3.2 Acronyms

1.3.3 Abbreviations

1.4 Document structure

2 Architectural design

2.1 Overview: High-level components and their interaction

2.2 Component view

2.3 Deployment view

2.4 Runtime view

2.5 Component interfaces

2.6 Selected architectural styles and patterns

3 Algorithm design

In the following paragraph are presented the most relevant and significant algorithm used in Travlendar+ application. More specifically are described the following algorithm:

- Compute travel;
- View daily schedule;
- Check appointments overlapping;
- Check appointments unreachability;
- Check travel alternatives;
- Check movement alternatives;

The application is written in Java, but the algorithms are shown in pseudocode.

3.1 Object class

Before to explain the algorithms details, it is needed to introduce the class Appointment, Travel and Movement that are the most important entities for the whole application.

3.1.1 Appointment class

The class Appointment has as attributes:

- The appointment name;
- The date;
- The start time;
- The place of arrival;
- The expected time of arrival;
- The appointment duration;
- The associated travel to reach the appointment;

Here the appointment class written in Java:

```
public class Appointment {

        String name;
        Date date;
        Time time;
        Place place;
        Time timeOfArrival;
        Time duration;
        Travel travel;
}
```

3.1.2 Travel class

The travel class has the following attributes:

- The related appointment;
- The departure point;
- The destination point;
- The desired time to leave;
- The movements list in which the travel is split;

Here the travel class written in Java:

```java
public class Travel {

        Appointment appointment;
        Place placeOfDeparture;
        Place placeOfArrival;
        Time timeOfDeparture;
        ArrayList<Movement> movements;
}
```

3.1.3 Movement class

The movement class has the following attributes:

- The departure point;
- The destination point;
- The estimated time;
- The time of departure;

Here the movement class written in Java:

```java
public class Movement {

        Place placeOfDeparture;
        Place placeOfArrival;
        Time estimatedTime;
        Time timeOfDeparture;
}
```

3.2 Algorithms

3.2.1 Create flexible appointment

When the user wants to schedule a flexible appointment, the system calls the follow function that controls if it is possible to create the appointment in the time range inserted If it is possible, means that no further appointments are between the inserted range and therfore the appointment is created, else the system shows to the user an error message.

```
function CREATEFLEXIBLEAPPOINTMENT(initRange, endRange, duration)
```

```
    if endRange-initRange < duration then
        NOTIFYUSER("Impossible to create flexible appointment")
        return impossible
    else if exists appointment between initRange and initRange+duration then
        CREATEFLEXIBLEAPPOINTMENT(appointment.time+appointment.duration, endRange, duration)
    else
        return CREATEAPPOINTMENT(...)
    end if
end function
```

3.2.2 Compute travel

The function computeTravel is used to create a new travel for a specific appointment. The followed procedure is composed by four steps:

1. Read the different information from the appointment passed as parameters.
2. Call the function queryMaps. this uses Google Maps API to compute the travel, the response is JSON object and the function parse this and extracts all information related to the computed travel.
3. For each steps, which put together with the other steps produces the travel, is created a new Movement and it is added to MovementList presents in Travel class.
4. Call the function computeWeatherCondition, that with specific API can compute the weather condition, and if the travel includes some walking or bicycling movement and it is expected 'rain', the system shows to the user a message.

```
function COMPUTETRAVEL(appoinment, placeOfDeparture, timeOfDeparture, preferences)
    READ(appointment.place)
    READ(preferences.travelPreferences)
    travel = QUERYMAPS(placeOfDeparture,place,travelPreferences)
    for all movement detected in travel do
        createdMovement = CREATEMOVEMENT(movement)
        ADDTOMOVEMENTLIST(createdMovement)
    end for
    READ(appointment.date)
    READ(appointment.time)
    weather = COMPUTEWEATHERCONDITION(placeOfDeparture, place, date, time)
    if wheater is 'rain' and ( (travelPreferences is 'green') or (exists one movement : movementType is 'walk' or 'bike') ) then
        NOTIFYUSER("Rain expected, not reccomended use of bike or walks")
    end if
end function
```

3.2.3 View daily schedule

When the user clicks on view daily schedule button, the system computes the daily schedule through this function and show to the user all appointments expected for the selected day and all travel to reach them. This method has only one parameter, the day desired to compute the schedule.
The function is composed by four steps:

1. Extract first appointment expected in the selected day (appointment-i) and check its unreachability.

2. Extract the second appointment (appointment-i+1) and check its unreachability.
3. If both appointments are reachable then it is necessary to check the overlapping between them.
4. If all controls are successfully passed then it is possible to compute the travel for the appointment-i.
5. Go on with the next appointment and repeats the loop until all appointments are examinated.

```
function VIEWDAILYSCHEDULE(day)
    for all appointment in day do
        if CHECKUNREACHABILITY(appointment-i) is unreachable then
            MANAGEUNREACHABILITY(appointment-i)
        end if
    end for
    if CHECKUNREACHABILITY(appointment-i+1) is unreachable then
        MANAGEUNREACHABILITY(appointment-i+1)
    else if CHECKOVERLAP(appointment-i, appointment-i+1) is overlap then
        MANAGEOVERLAP(appointment-i,appointment-i+1)
    end ifCOMPUTETRAVEL(appointment-i,appointment-i.travel.placeOfDeparture, appoinment-i.travel.departureTim
INCREMENT(i)
end function
```

3.2.4 Check overlap

When the system has to check the overlapping between two appointments invokes this function that has as parameter the two appointments. The function controls which appointment starts before and then controls if the begin time of the second appointment overlaps with end time of the first appointment.

```
function CHECKOVERLAP(app1, app2)
    if app1.time is before of app2.time and (app2.time is before (app1.time + app1.duration)) then
        return overlap
    else if app1.time is before (app2.time + app2.duration) then
        return overlap
    end if;
    return no overlap
end function
```

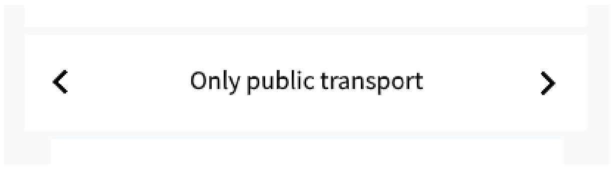
3.2.5 Check unreachability

This function checks the unreachability of one appointment passed as parameter. To check the unreachability it is necessary to control if the arrival time of one appointment is compatible with departure time added to estimated travel time.

```
function CHECKUNREACHABILITY(appointment)
    if appointment.timeOfArrival is before (appointment.travel.timeOfDeparture + travel duration)
then
        return unreachable;
    else
        return reachable;
    end if
end function
```

3.2.6 Check travel alternative

The user can check all available travel alternatives and modify the actual one with another. This is possible switching the different alternatives.



The function checkTravelAlternative reads the selected alternative and computes the travel with the inserted TravelType.
View more detailed these steps:

1. There is a control on a travel type selected from the user and there are two possible cases:
   (a) 'only-own-car' or 'only-public-transport' or 'green: when one these travel type is selected, it is created a new travel alternative based on related travel mode (driving for only-own-car, transit for only-public-transport and bicycling for green).
   (b) 'faster' or 'cheaper': in this case, it is created a set that contains all possible travel computable.
2. Then it is selected the travel alternative:
   (a) In case of cheaper or faster travel, it is necessary to find among all computed travels the cheaper or the faster.
   (b) In other cases, return the only one alternative computed.

```
function CHECKTRAVELALTERNATIVE(travel, travelType)
    if travelType is 'only-own-car' then
        compute driving travel alternative
    else if travelType is 'only-public-transport' then
        compute transit travel alternative
    else if travelType is 'green' then
        compute bicycling travel alternative
    else if travelType is 'faster' or travelType is 'cheaper' then
        compute a set of differents travel alternatives
    end if
    if travel alternative exists then
        if travelType is 'faster' then
            find faster travel in the computed set and return it
        end if
        if t thenravelType is 'cheaper'
            find cheaper travel in the computed set and return it
        end if
        return the travel alternative found
    end if
end function
```

3.2.7 Check movement alternative

When the user views a movement details and clicks on a "means icon" from the bar, the system calls this function that read the movement type selected and computed the relative movement.
If the user selects 'car-sharing' or 'bike-sharing' it is added a new movement to reach the car or bike.

```
function CHECKMOVEMENTALTERNATIVE(movement, movementType)
    if movementType is ’car’ then
        compute driving movement
    else if movementType is ’walk’ then
        compute walking movement
    else if movementType is ’public-transport’ then
        compute transit movement
    else if movementType is ’bike’ then
        compute bicycling movement
    else if movementType is ’car-sharing’ then
        compute walking movement to reach the car
        add a further driving movement
    else if movementType is ’bike-sharing’ then
        compute walking movement to reach the bike
        add a further bicycling movement
    end if
    if computed movement exists then
        return movement alternative
    end if
end function
```

4 User interface design

4.1 Registration/Login

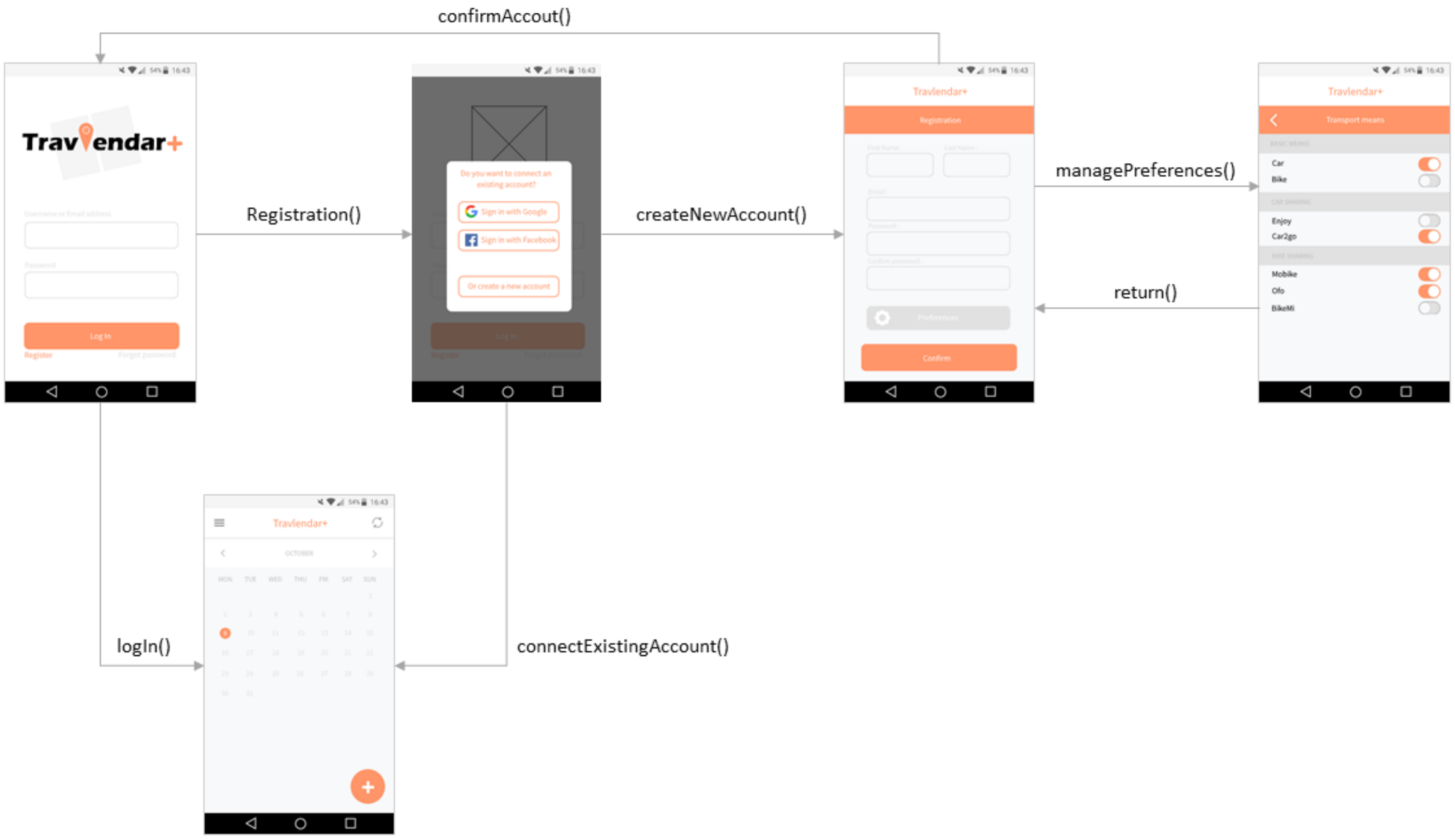


Figure 1: User Interface: registration and login

4.2 Calendar views

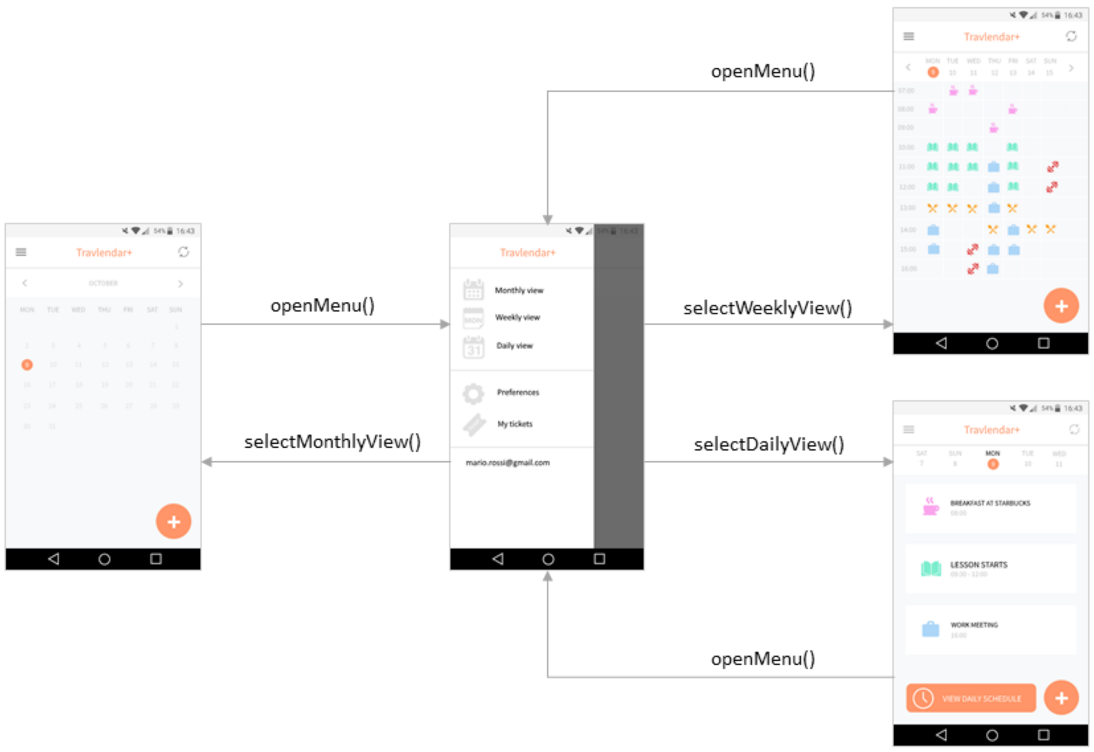


Figure 2: User Interface: calendar views and menu

4.3 Create new appointment

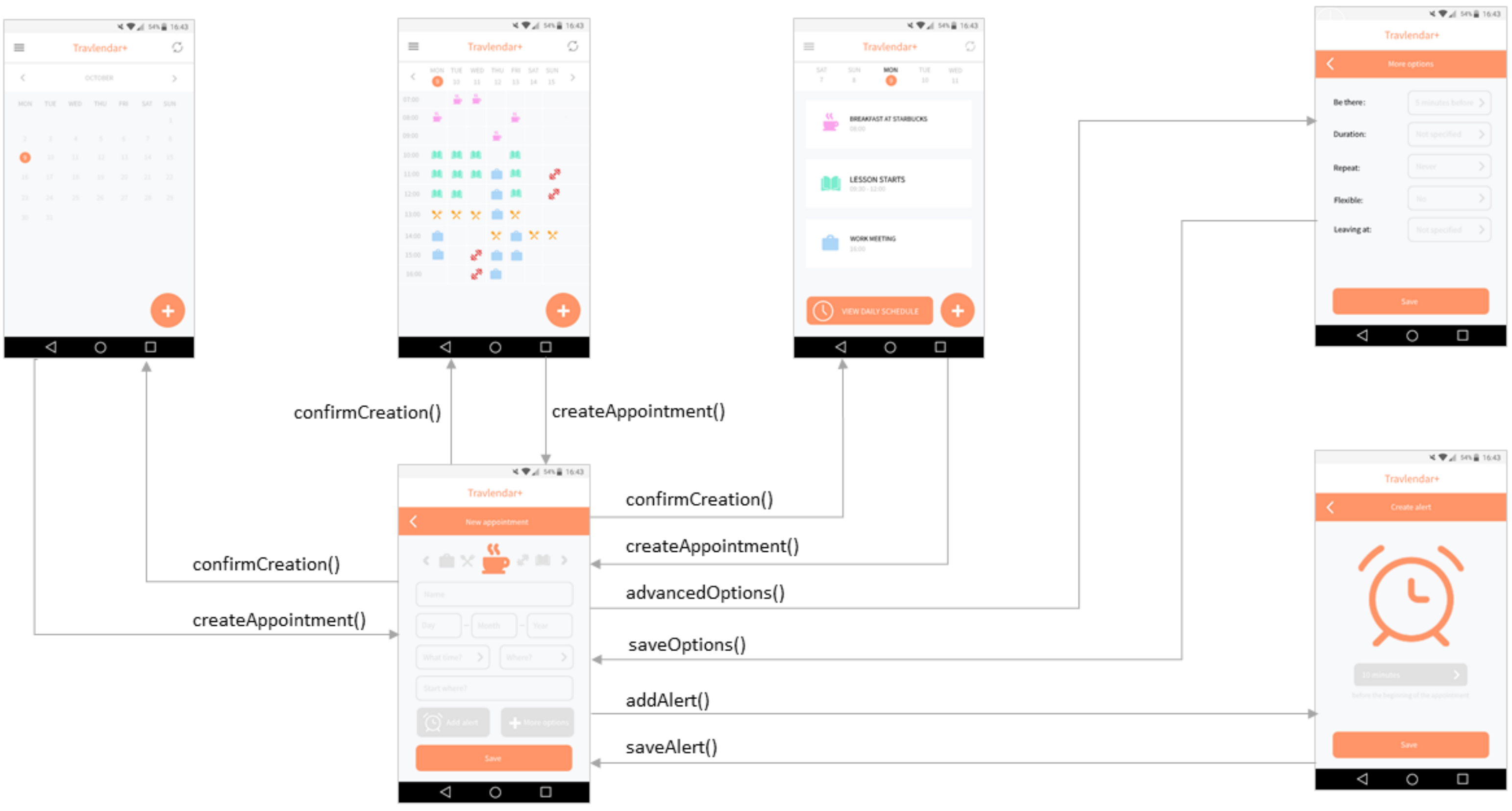


Figure 3: User Interface: creation of new appointment

4.4 Edit existing appointment

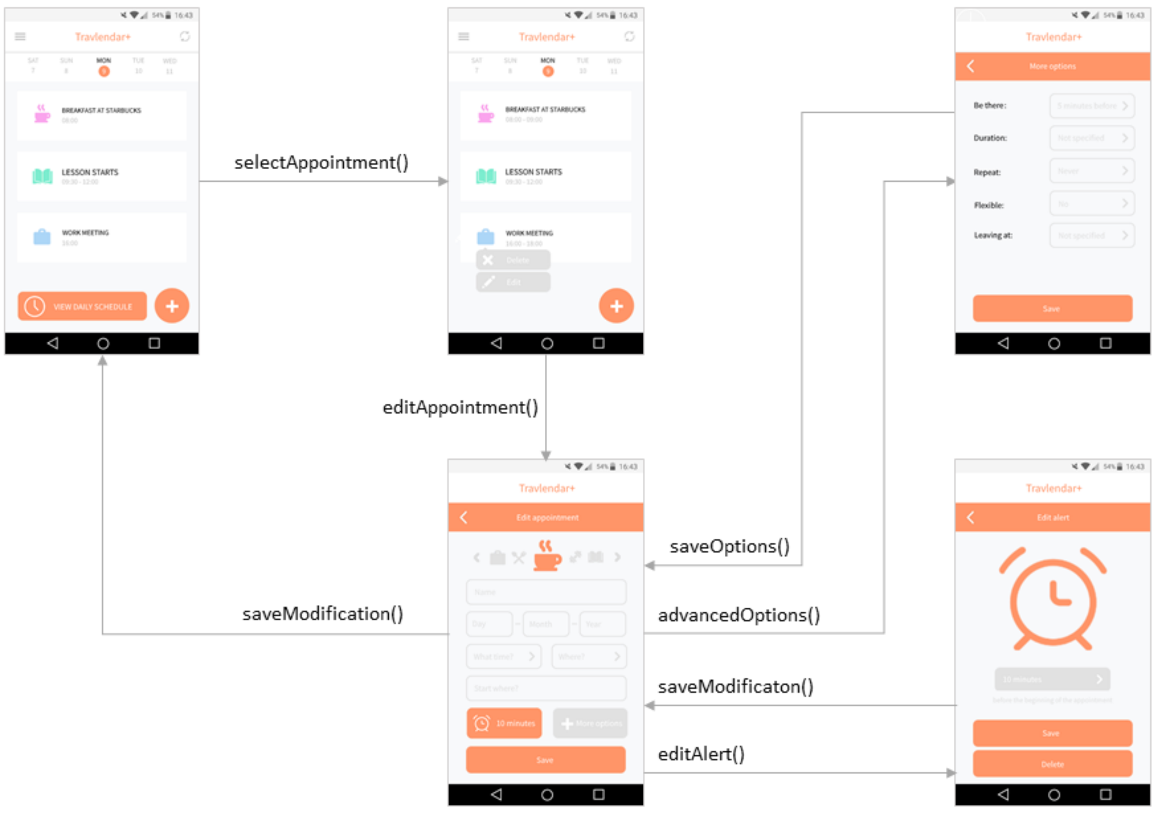


Figure 4: User Interface: edit of an existing appointment

4.5 manage preferences

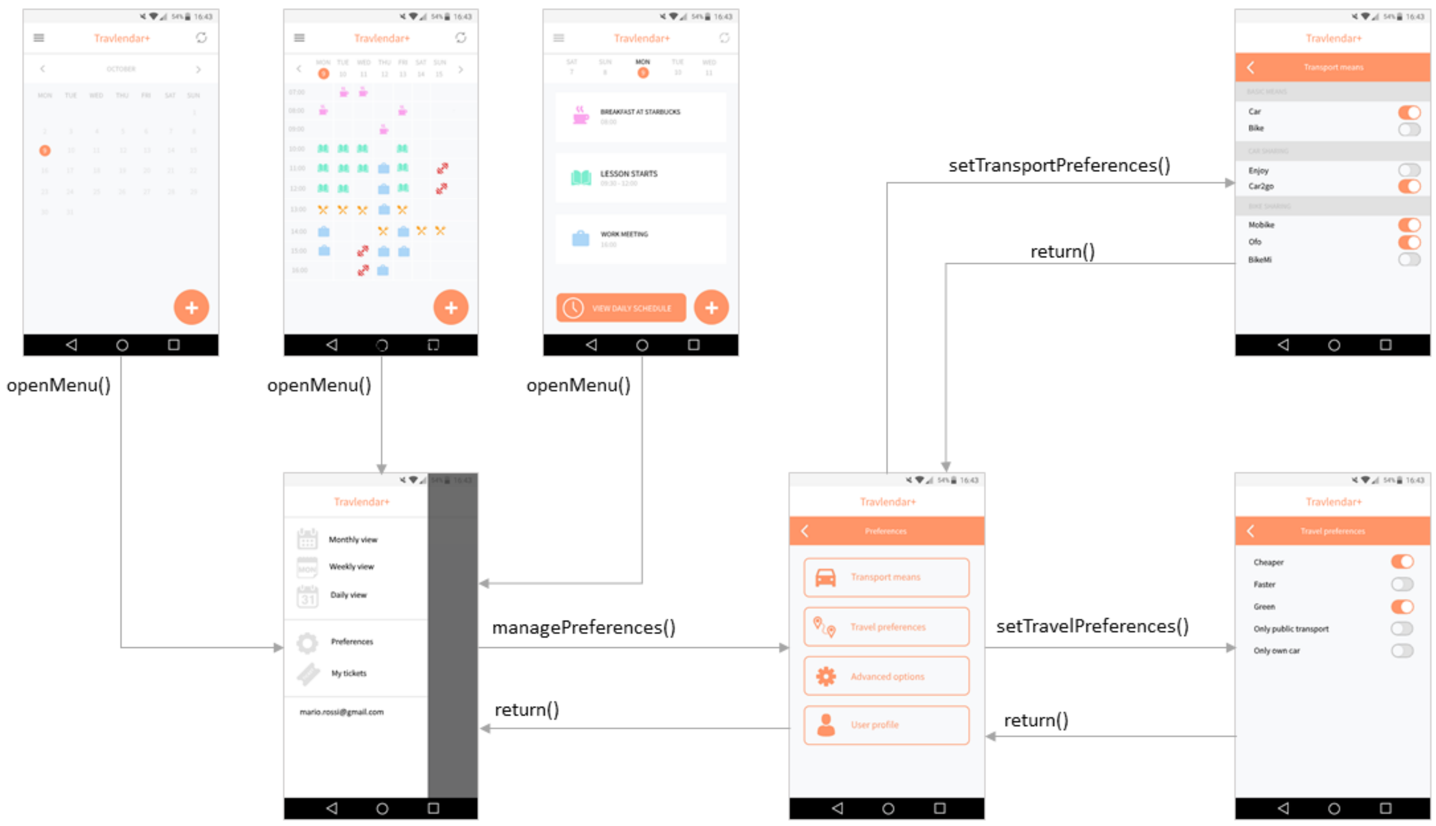


Figure 5: User Interface: manage preferences

4.6 View daily schedule and travel/movement details

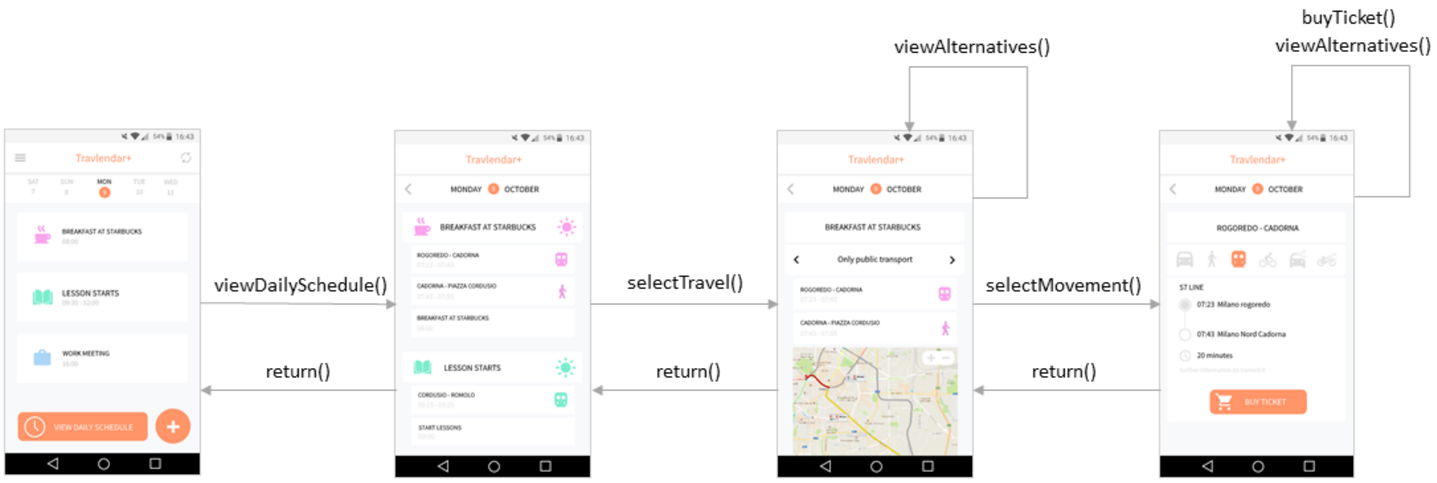


Figure 6: User Interface: daily schedule and travel/movement details

4.7 Buy and view tickets

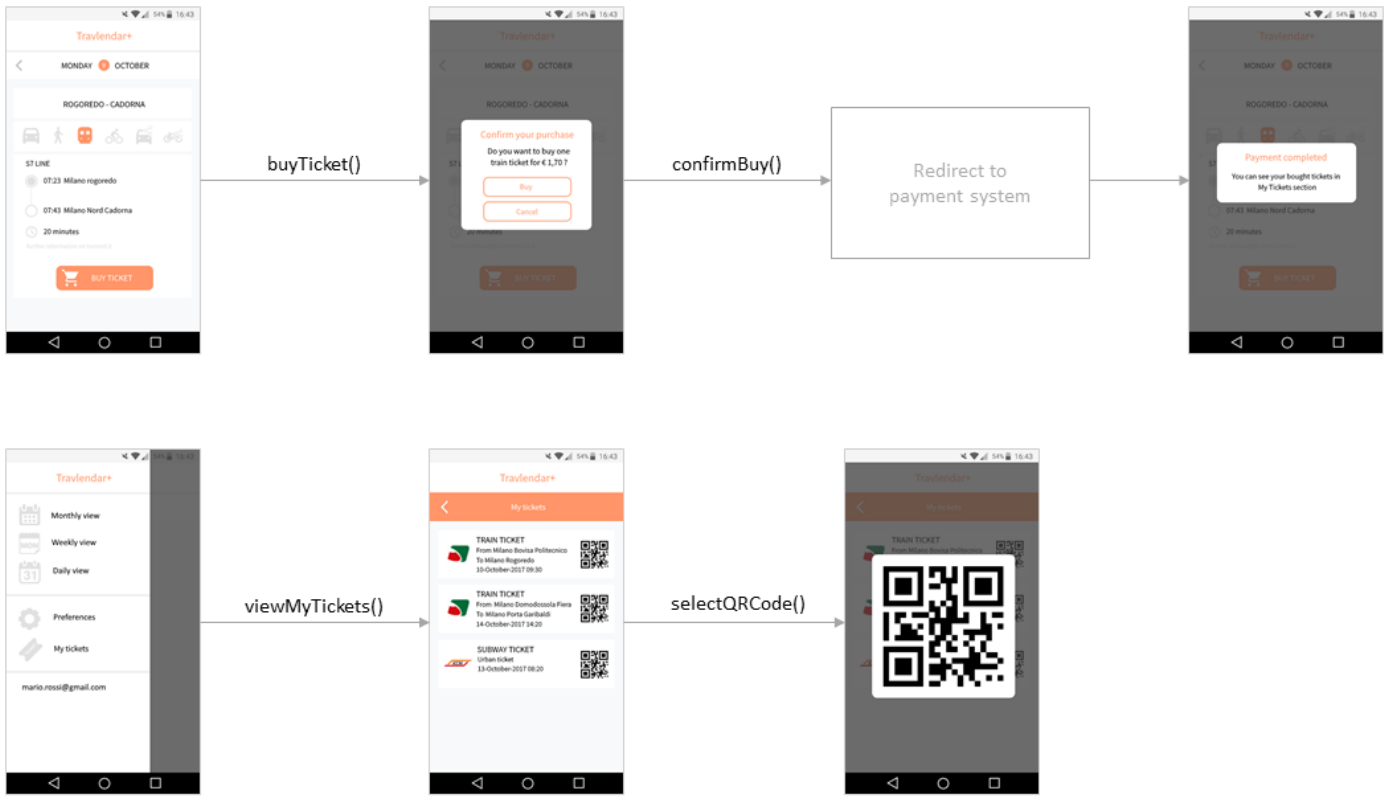


Figure 7: User Interface: buy ticket and ’My tickets’ section

5 Requirements traceability

- **[R1] The user must be logged into the system to access application features.**
  As explained in the IITP part, there is dependency between Travel and Appointment Manager and Account Manager: the account managing infrastructure is created first and travel/appointment data are acquired from the database.

- **[R2] The user must be able to choose the option of creating a new appointment.**
  Appointment Manager implements the functionality createAppointment(). See section 4.3: Create new appointment.

- **[R3] The user must be able to choose the option of editing a selected appointment.**
  Appointment Manager implements the functionality editAppointment(). See section 4.4: Edit existing appointment.

- **[R4] The user must be able to choose the option of deleting a selected appointment.**
  Appointment Manager implements the functionality deleteAppointment().

- **[R5] The system must be able to provide the user with an overview of his calendar and the user must be able to view all appointments fixed in a certain period.**
  The system gets information about all the created appointment from the database and display them to with the interfaces described in the User Interfaces section of this document.

- **[R6] The user must be able to select a chosen day from the overview of his calendar.**
  The system keeps track of the different dates (see Class Diagram: Date class) and offers the daily calendar option of display. See section 4.2: Calendar views.

- **[R7] The user must be able to select a specific appointment in his calendar.**
  Account Manager loads the information about the different appointments (see Class Diagram: Appointment class). See section 4.4: Editing existing appointment to see the correspondent user interfaces.

- **[R8] The system must ask the user to provide all information needed for the creation of a new appointment, such as place and time of start and overall duration.**
  This requirement is covered by the createAppointment() functionality of Appointment Manager. See section 4.3: Create new appointment.

- **[R9] The system must check if the information provided by the user are correct.**
  Functionalities covered during phases of creation/editing. See Sequence diagrams on section 5.2 of RASD document.

- **[R10] The system must check if an appointment overlaps with other events and must eventually notify it to the user.**
  See section 3.2.4: Check overlap algorithm.

- **[R11] The system must give the user access to all details of a selected appointment and the user must be allowed to edit the information needed.**
  Account Manager loads the information about the different appointments (see Class Diagram: Appointment class) and display them to the user. It also implements the functionality editAppointment(). See section 4.4: Editing existing appointment to see user interfaces.

- **[R12] The user must be able to set advanced information for a created appointment.**
  See advancedOptions() in section 4.3: Create new appointment.

- **[R13] The user must be able to set an appointment as flexible, specifying the interval of time.** Functionality of Appointment Manager. See advancedOptions() in section 4.3: Create new appointment and the Flexible Appointment class in the Class Diagram.
- **[R14] The user must be able to set an appointment as repeatable, specifying the desired days.** Functionality of Appointment Manager. See advancedOptions() in section 4.3: Create new appointment.
- **[R15] The system must schedule any flexible or repeatable appointment in the correct way, avoiding overlapping with other appointments.** The system is provided with algorithm for overlapping checking and flexible appointments scheduling. See section 3.2 for further info.
- **[R16] The appointment intended to be modified must have been previously successfully created and not already deleted.** The process of editing appointments is made by Appointment Manager and works only with existing appointments in the database. A process is stored in the database only at the creation and removed during the deletion. See section 2.4: Runtime view.
- **[R17] The user must confirm the creation of the new appointment.** createAppointment() functionality of Appointment Manager. See confirmCreation() in section 4.3: Create new appointment.
- **[R18] The user must confirm any appointment modification.** editAppointment() functionality of Appointment Manager. See saveModification() in section 4.3: Create new appointment.
- **[R19] The system must save the user modifications in memory and the calendar must be updated.** Appointment Manager interacts with the DBMS: after any modification, the database is updated. See section 2.4: Runtime view.
- **[R20] The system must remove a deleted appointment from the memory and delete every alert related to it.** Appointment Manager interacts with the DBMS: after any modification, the database is updated. This functionality is implemented by deleteAppointment().
- **[R21] The user must be able to switch between different possible calendar, such as daily calendar, weekly calendar and monthly calendar.** See section 4.2: Calendar view.
- **[R22] The system must be able to provide information about the scheduled travels for a chosen day, showing the transport means and the estimated time required from each travel.** User can recall the function viewDailySchedule() from his application, and Travel Manager takes care of provide all the information in detail. See section 2.4: Runtime view.
- **[R23] The system must choose the best option between the possible travel alternatives according to the preferences expressed in the user profile settings and the information about external weather.** This requirement is covered by the computeTravel() and loadPreferences() functionality of Travel Manager. See section 3.2.2: Compute travel for further info on the algorithm.
- **[R24] The user must be able to select a specific travel in his daily schedule.** See selectTravel() in section 4.6: View daily schedule and travel/movement details.

- **[R25] The system must provide detailed information about the travels selected by the user, such as the trace route on the map and the weather conditions.**
  This requirement is covered by the computeTravel() functionality of Travel Manager. More specifically, see section 2.4 for runtime view and section 3.2.2: Compute travel for further info on the algorithm.

- **[R26] The system must provide the user with an overview of the possible travel alternatives for the chosen travel, specifying all details for each one.**
  Travel Manager functionalities. see section 3.2.6: Check travel alternative for further info on the algorithm and viewAlternatives() in section 4.6: View daily schedule and travel/movement details for user interfaces.

- **[R27] The user must be able to filter the travel alternatives furnished.**
  Travel Manager functionalities. see section 3.2.6: Check travel alternative for further info on the algorithm.

- **[R28] The user must be able to choose a favorite travel option different from the displayed default one.**
  Travel Manager functionalities. See section 3.2.6: Check travel alternatives for further info on the algorithm.

- **[R29] The user must be able to select a specific movement in a travel.**
  See selectMovement() in section 4.6: View daily schedule and travel/movement details.

- **[R30] The system must provide detailed information about the movements selected by the user, such as the specific trace route on the map and the price of the ticket.**
  Travel Manager functionalities. All information are provided by the external APIs and collected by the system as explained in section 2.4. See section 4.6: View daily schedule and travel/movement details for user interfaces.

- **[R31] The user must be able to choose an alternative transport mean for a selected movement, if there are any.**
  Travel Manager functionalities. See section 3.2.7: Check movement alternative for further info on the algorithm.

- **[R32] The system must update the daily schedule according to the travel option chosen by the user and the user must be able to see the new updated schedule.**
  Travel Manager functionality. The user can choose a travel alternative (see section 3.2.6: Check travel alternative for info on the algorithm) and Travel Manager updates info.

- **[R33] The system must give to the user the possibility of buying the ticket for the selected travel.**
  This requirement is covered by Ticket Manager functionalities. See section 4.6: Component interfaces for full functionalities overview.

- **[R34] The system must save a copy of the bought tickets.**
  Tickets manager interacts with DMBS to store tickets data after the purchase. See section 2.4 for runtime views.

- **[R35] The user must be able to access to a ticket page from the home page.**
  Bought tickets are saved in the database during the purchase process (see section 2.4 for runtime view) and the user can access them by clicking on "My tickets) on the side panel menu. See section 4.7: Buy and view tickets for user interfaces.

- **[R36] The system must provide a list of all the bought tickets and the user must be able to select and view a specific one.**
  This requirement is covered by the viewTickets() functionality of Travel Manager. See section 4.7: Buy and view tickets for user interfaces.

- **[R37] The user must be able to access the preferences panel of his account.**
  Account Manager functionality. See section 4.5: Manage preferences for user interfaces.

- **[R38] The system must give the user the possibility of setting various preferences, such as owned and preferred travel means, address of Home and other general travel preferences.**
  Account Manager functionality. See section 4.5: Manage preferences for user interfaces.

- **[R39] The user must be able to edit the provided preferences when needed.**
  The preferences panel is always accessible from the home page (See section 4.5: Manage preferences for user interfaces). Account Manager interacts with DBMS, so any modification will be update in the database.

- **[R40] The system must give the user the possibility of adding an alert to an appointment while it is being created or modified.**
  Functionality of Appointment Manager. See section 5.2.3 and 5.2.4 or RASD document: Appointment creation and Appointment editing for info on Sequence diagrams and sections 4.3/ 4.4 of this document for user interfaces

- **[R41] The user must be able to choose a desired interval of time for the warning alert.**
  Functionality of Appointment Manager. See section 5.2.6 of RASD document: Alert editing for info on Sequence diagrams and sections 4.3/ 4.4 of this document for user interfaces.

- **[R42] The user must confirm the alert creation and the system must save the insertion in the memory.**
  Appointment manager interacts with DBMS to store information in the system memory. See section 5.2.6 of RASD document: Alert editing for info on Sequence diagrams and sections 4.3/ 4.4 of this document for user interfaces.

- **[R43] The user must be able to modify or remove the inserted alert when needed.**
  Appointment manager functionalities are always accessible by the user through the correspondent interfaces.

- **[R44] In case of any alert modification made by the user, the user must confirm the modification and the system must save all changes.**
  Appointment manager interacts with DBMS and updates data after any modification. See section 5.2.6 of RASD document: Alert editing for info on Sequence diagrams and sections 4.3/ 4.4 of this document for user interfaces.

6 Implementation, integration and test plan

6.1 Overview

As explained in the RASD and DD documents, the system requires the following components to be fully developed and working:

- **Travlendar+ mobile application, working on Android v 4.0.3 and later;**
- **JBoss 7.0.1 Java Server, running the Travlendar+ application services;**
- **MySQL 5.7.19 DBMS for users data managing;**

Furthermore, the Travlendar+ application services component is composed by the following subsystems, wich must be developed and correctly integrated:

- **Appointment manager**, which makes use of Calendar APIs and requires DBMS connection;
- **Travel manager**, which implements Weather and Maps APIs;
- **Account manager**,
- **Tickets manager**,

Finally, the following connection must be implemented and full working:

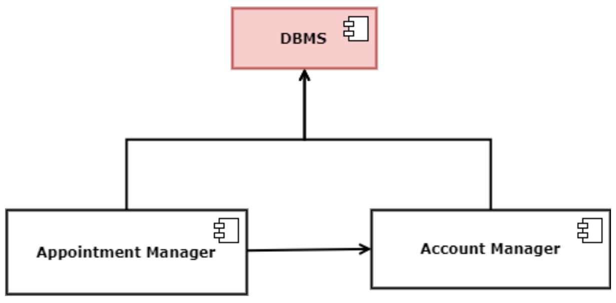
- **Remote Method Invocation** connection between client and application server, using JRMP protocol;
- **JDBC** connection between application server and DBMS.

6.2 Strategy

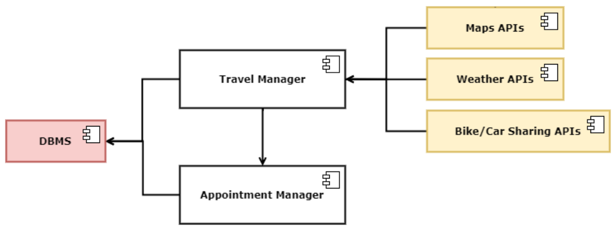
In order to guarantee observability and efficacy, the testing strategy follows an incremental integration logic. The chosen approach is bottom-up: this choice allows us to gradually build the system and continuously provide feedbacks as soon as the single parts are finished. Our goal is to first develop sub-component that are able to work independently and do not require connections with other parts of the systems. Then, once that all sub-components are fully developed and tested, we plan to proceed in connecting them and fulfill the dependencies to implement gradually the main super-component functions. Final part of the integration will require the main component to be full working and tested, and consists in testing the correct working of connections between main-components.

6.3 Dependecies

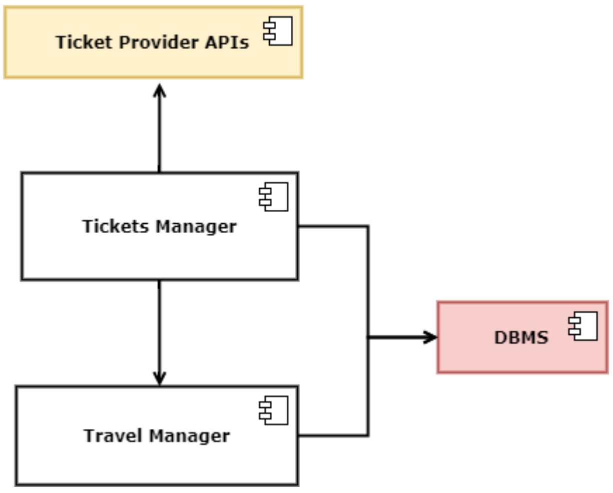
The interaction with DBMS is essential for the correct working of all the four main components of the Application services. Plus, Appointment Manager requires Account Manager to be implemented to ensure the correctness of the links between different accounts and relative calendars:



In order to work properly, Travel Manager makes use of different external APIs such as Maps APIs, Weather APIs and different APIs from car sharing and bike sharing services. The correct functionality of these interfaces must be checked before proceeding in developing the component. In addition, Travel Manager requires an integration with Appointment Manager to acquire data relative to the different locations of the appointments.



Finally, Tickets Manager strictly depends on the correct working of the APIs of the chosen ticket provider to pay and acquire travel tickets. It also requires Travel Manager to work properly, because the user can accesstickets functionalities only after selecting a movement from the daily schedule.



Given the previous information, an appropriate plan of integration and testing could be the following one:

Week 27/11/17 – 4/12/17:

- DBMS: build a simple Java application that interacts with MySQL 5.7.19 Database Manage System by JDBC and correctly store and manage data from a small database.
- ACCOUNT MANAGER: expand the application, allowing different users to register and insert personal info.
- MAPS APIs: testing APIs with small example applications to ensure correct working.
- WEATHER APIs: testing APIs with small example applications to ensure correct working.

Week 4/12/17 – 11/12/17:

- CAR/BIKE SHARING APIs: testing APIs with small example applications to ensure correct working.
- ACCOUNT MANAGER: fully development and testing of all component functionalities: allow multiple users to register and login, manage personal data and connect existing Facebook/Google account.

- APPOINTMENT MANAGER: expand the application by giving the user the possibility of creating, editing and delete simple appointments (no alert/advanced option). Test the correct storage in the database and the correct integration with Account Manager.
- start working on network platform: develop a small application that makes use of RMI technology to invoke methods on an example application running on JBoss 7.0.1 Java Server.

Week 11/12/17 – 18/12/17:

- APPOINTMENT MANAGER: fully development and testing of all component functionalities: allow the user to add advanced options to the different appointments. Test full integration with Account Manager.
- TRAVEL MANAGER: integrate use of Maps/Weather/Car&Bike Sharing APIs with data provided by Appointment Manager to compute travels for each appointment. Testing of the functionalities.
- TICKET PROVIDER APIs: testing APIs with small example applications to ensure correct working.
- Export the application on JBoss 7.0.1 Java Server, testing of RMI methods to access the application features from remote.

Week 18/12/17 – 8/1/18:

- TRAVEL MANAGER: fully development and testing of all component functionalities: the application accesses correctly appointment and user data to compute travels. Test complete integration with Appointment Manager.
- TICKET MANAGER: expand the application by adding functionalities for tickets purchase using the tested APIs. Test integration with Travel Manager.

Week 8/1/18– 18/1/18:

- General testing of full application functionalities.

7 Effort spent