

**Travlendar+ project Daverio Pietro,  
Fiorillo Alessandro**



**POLITECNICO**  
MILANO 1863

# **Design Document**

---

<b>Deliverable:</b>	DD
<b>Title:</b>	Design Document
<b>Authors:</b>	Daverio Pietro, Fiorillo Alessandro
<b>Version:</b>	1.0
<b>Date:</b>	26-November-2017
<b>Download page:</b>	<a href="https://github.com/FiorixF1/DaverioFiorillo">https://github.com/FiorixF1/DaverioFiorillo</a>
<b>Copyright:</b>	Copyright © 2018, Daverio, Fiorillo @ Politecnico di Milano

---

## Contents

<b>Table of Contents</b>	<b>3</b>
<b>List of Figures</b>	<b>4</b>
<b>List of Tables</b>	<b>4</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Purpose	5
1.2 Scope	5
1.3 Definitions, Acronyms, Abbreviations	5
1.4 Revision History	5
1.5 Reference Document	5
1.6 Document Structure	6
<b>2 Architectural Design</b>	<b>7</b>
2.1 Overview	7
2.2 High level components and their interactions	7
2.3 Component view	8
2.4 Deployment view	9
2.5 Runtime view	10
2.6 Component interfaces	13
2.7 Selected architectural styles and patterns	13
2.8 Other design decisions	13
<b>3 Algorithm Design</b>	<b>14</b>
3.1 Check overlapping	14
3.2 Add appointments and routes	15
3.3 Remove routes of an appointment	17
3.4 Remove an appointment	18
3.5 Updating an appointment	19
3.6 Calculating a route	20
<b>4 User Interface Design</b>	<b>22</b>
4.1 Mock ups	22
4.2 UX diagram	26
<b>5 Requirements Traceability</b>	<b>28</b>
<b>6 Implementation, Integration and Test Plan</b>	<b>33</b>
6.1 Integration Strategy	33
6.1.1 Entry Criteria	33
6.1.2 Elements to be integrated	33
6.1.3 Integration Strategy	34
6.1.4 Integration sequence	34
6.2 Test Plan	36
6.2.1 Testing Strategy	36
6.2.2 Testing Sequence	36
<b>7 Effort Spent</b>	<b>37</b>

## List of Figures

1	Component diagram of the system . . . . .	8
2	Deployment diagram of the proposed implementation . . . . .	10
3	User login . . . . .	11
4	Add an appointment . . . . .	11
5	Add a route . . . . .	12
6	Send a notification (push or SMS) . . . . .	12
7	Check overlapping between two events . . . . .	15
8	Add an appointment . . . . .	16
9	Add a route . . . . .	17
10	Remove all routes of an appointment . . . . .	18
11	Remove an appointment . . . . .	19
12	Update an appointment . . . . .	20
13	Calculate routes . . . . .	21
14	Login screen . . . . .	22
15	Sigup screen . . . . .	23
16	Home screen with calendar . . . . .	23
17	Add an appointment . . . . .	24
18	Add a route . . . . .	24
19	Set preferences . . . . .	25
20	Select a route . . . . .	25
21	UX Diagram . . . . .	27
22	Bottom-Up integration and testing components diagram. . . . .	35

## List of Tables

# 1 Introduction

## 1.1 Purpose

The purpose of this derivable is to provide an architectural design solution for Travlendar+ that fulfils all functional and non-functional requirements expressed in the linked Requirement Analysis and Specification Document.

The following document will also provide an overview of an implementation, integration and test plan solution, bounded with this derivable.

The target audiences of this paper are developers, testers teams and analysts involved in the project.

## 1.2 Scope

Travlendar+ is a calendar-based application thought to support user thorough his appointment scheduling and transport management.

The system will provide to user:

- the possibility to insert scheduled appointments
- the insertions, modification and organization of different types of preferences
- registration and login in order to access backup/sync options, beside a more advance preferences management
- different route choices to reach the location of appointments

(for further details look at RASD document).

The system will offer his services to user through a Web GUI and a mobile application supporting different operating systems and device format.

## 1.3 Definitions, Acronyms, Abbreviations

- **RASD** Requirement Analysis and Software Specification
- **DD** Design Document
- *Appointment*
- *Route*
- *Path*
- *MA*
- **GUI** Graphic User Interface
- **WebUI** Wew User Interface

## 1.4 Revision History

## 1.5 Reference Document

- *Travlendar+ RASD document*
- *"Mandatory\_Project\_Assignment.pdf"* : assignment given of the project

## 1.6 Document Structure

**1 - Introduction** The first section offers an overview on the content of the following document, highlighting the purpose of this derivable and recalling a brief description of the problem itself. It also contains references to other documentation linked to this project.

**2 - Architectural Design** Firstly, the architectural design section presents an overview of a proposed system architecture to accomplish RASD specification.

Secondly, it steps into the system identifying behaviour of components, their interfaces and interoperability with other components, inside or outside the system.

Finally, it explains the thought behind design choices and it gives a list of patterns employed.

**3 - Algorithm Design** This section provides a list of the most significant algorithms, used by system, expressed in object oriented pseudo-code. They are given in order to specify some critical operation steps.

**4 - User Interface Design** This section offers a look upon the user interfaces in order to give a good representation of how the UI will look like to users by mean of interfaces mock-up. Furthermore, it' offered a deeper inquiry on the user interaction with the system through UX and BCE diagrams.

**5 - Requirement Traceability** Section implied in the traceability purpose of requirements, defined in RASD document, with components identified by current derivable in order to increase the observability of requirements fulfilment in following parts of system development and testing.

**6 - Implementation, Integration and Test Plan** It defines the strategy and provides a sequential plan for the implementation, integration and test processes, describing the sequence in which components are developed, integrated and tested together.

**7 - Effort Spent** Appendix showing the commitment required by the project to the team.

## 2 Architectural Design

### 2.1 Overview

This chapter will illustrate the architecture of the system with its logical and physical components and how they interact to reach the goals. The sections of this chapter are:

- *High level components and their interactions*: it provides a general idea about the physical components of the system
- *Component view*: it shows the structures of the logical components and how they communicate with each other
- *Deployment view*: it describes the physical structure of the system and shows where each component will be deployed. It will also present the possible technologies that will be used for the implementation.
- *Runtime view*: it presents how the components communicate with each other inside the most important scenarios that will be handled by the application
- *Component interfaces*: it describes the interfaces between the components of the system
- *Selected architectural styles and patterns*: it explains the patterns and architectural paradigms that have been used in some aspects of the architectural design
- *Other design decisions*: it presents further design choices that are worth mentioning

### 2.2 High level components and their interactions

The high level structure of the system follows a three-tier architecture. The client side is developed as a website and a mobile application, which are the interface of the user to the services of the system. The server contains all the business logic and is divided into several components. Each of them has a specific role and acts as an interface for other components and uses the interface provided by other components. Some other services participate in the functioning of the system but are outside the server: these are the external services, used for calculating routes, solve addresses into GPS coordinates and checking the weather conditions. Other external components let the application send push notification and SMS to the phone of a user. Finally, the last tier is realized by the database server, which is external to the main server hosting the business logic.

## 2.3 Component view

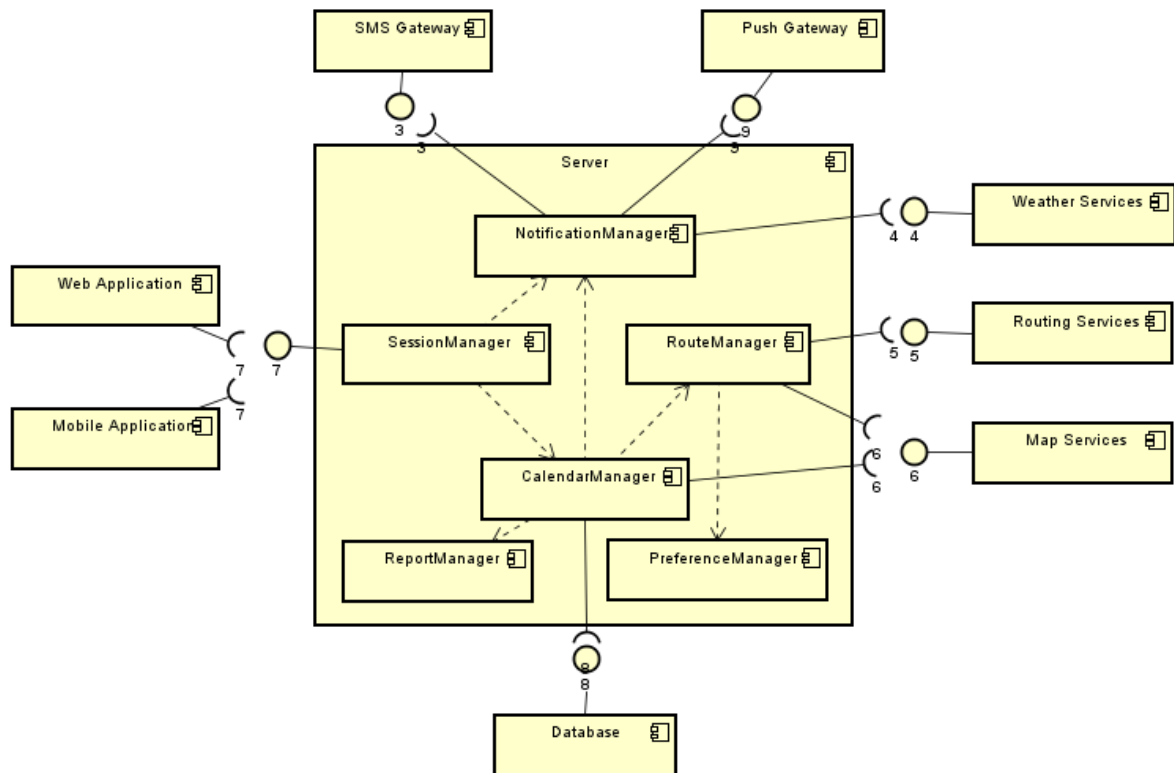


Figure 1: Component diagram of the system

- *Web Application*: client side component for the application via web browser
- *Mobile Application*: client side component for the application via mobile device
- *SessionManager*: handles login, sign up and general account functionalities
- *NotificationManager*: handles notifications for weather forecast, incoming routes and account recovering by phone number
- *CalendarManager*: handles the calendar and its appointments
- *RouteManager*: handles routes and their generation
- *ReportManager*: handles functionalities for reporting issues and suggestions
- *PreferenceManager*: handles the settings of the user, in particular for route computation
- *SMS Gateway*: service for sending SMS
- *Push Gateway*: service for sending push notifications
- *Weather Services*: external API for weather forecast
- *Map Services*: external API for maps
- *Route Services*: external API for routing



- *Database*: the database of the system

The SessionManager exposes an interface for client users and it is a sort of dispatcher of the requests, since the available functionalities are different for registered users and normal users.

All the functionalities of the system are divided among the corresponding manager. They need some external services for fulfilling their task, which are represented as external components. Other external components are the Push Gateway and the SMS Gateway, which are not mandatory but provide a better user experience.

## 2.4 Deployment view

The architecture is divided in the following tiers:

- **Client tier**: it is the interface of the application to the user, executable as a website on a browser or a graphical interface on a mobile device. The website is realized with the classical tools HTML5 and CSS, while the mobile application is available on Android and iOS operating systems.
- **Web tier**: it contains the web server which receives the HTTP requests from the web client and answers with HTML pages generated by a script engine. It can be implemented by the Tomcat web server and the JSP containers of Java EE.
- **Application tier**: it contains the core of the application with all the running components. It can be implemented using the Glassfish application server and the logic is contained in stateless Java beans. The communication with database happens with the JPA interface.
- **Database tier**: it contains the DBMS with the persistent data. It is accessible only by the application tier. A possible implementation will use the MySQL database.

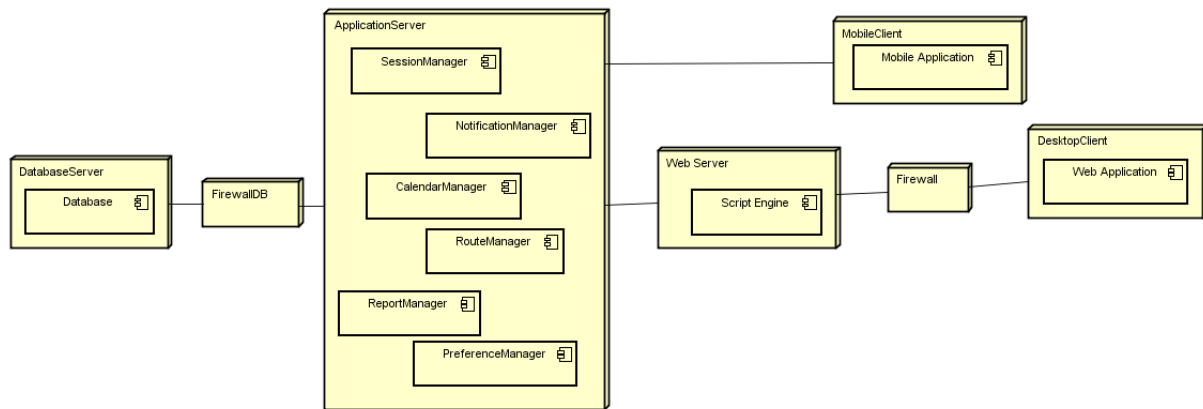


Figure 2: Deployment diagram of the proposed implementation

## 2.5 Runtime view

The following sequence diagrams show the interaction between the components of the system while fulfilling the main features of the application. The features have been chosen in such a way that every component appears at least once in any sequence diagram, so that every component can be observed while operating in a scenario.

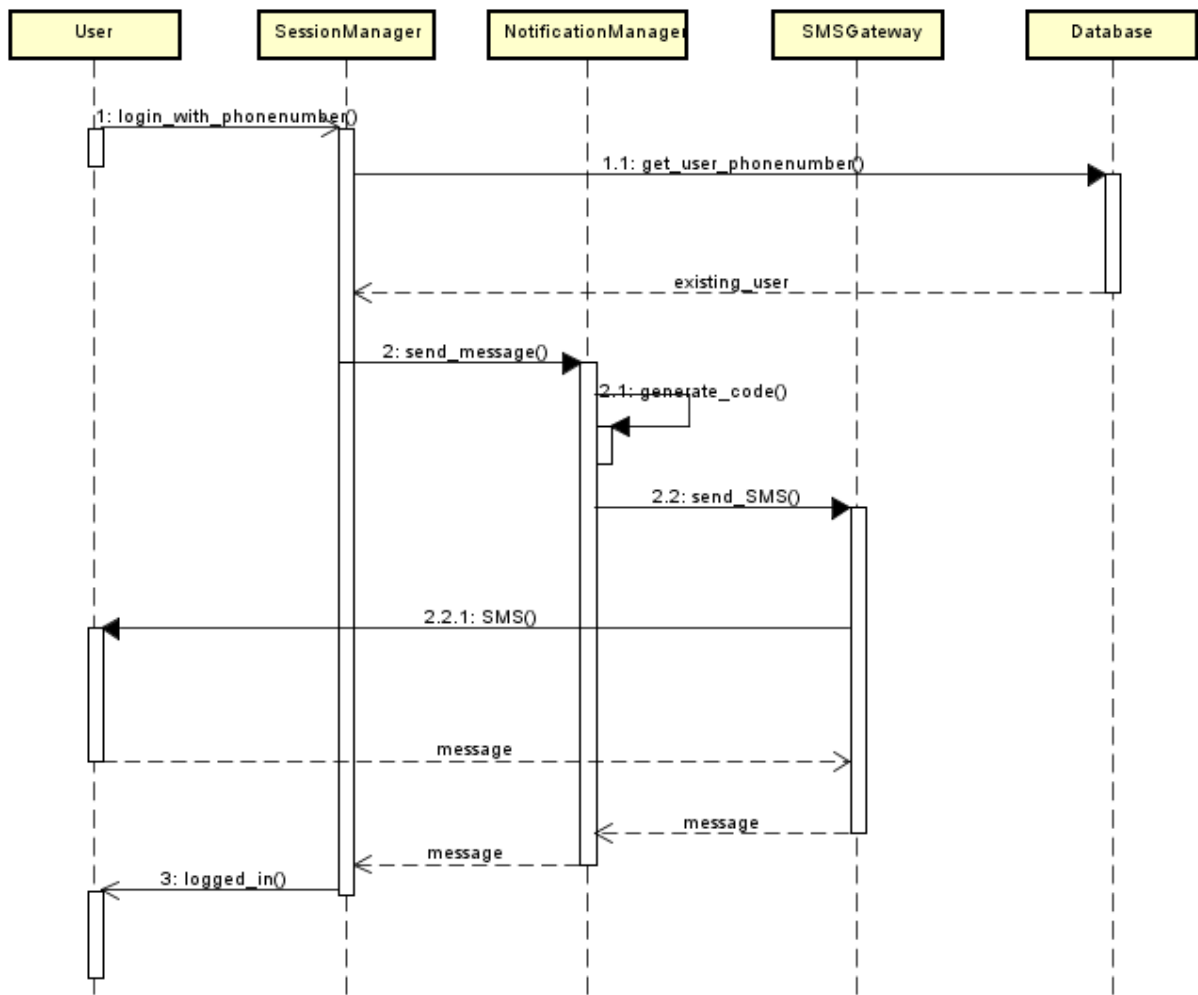


Figure 3: User login

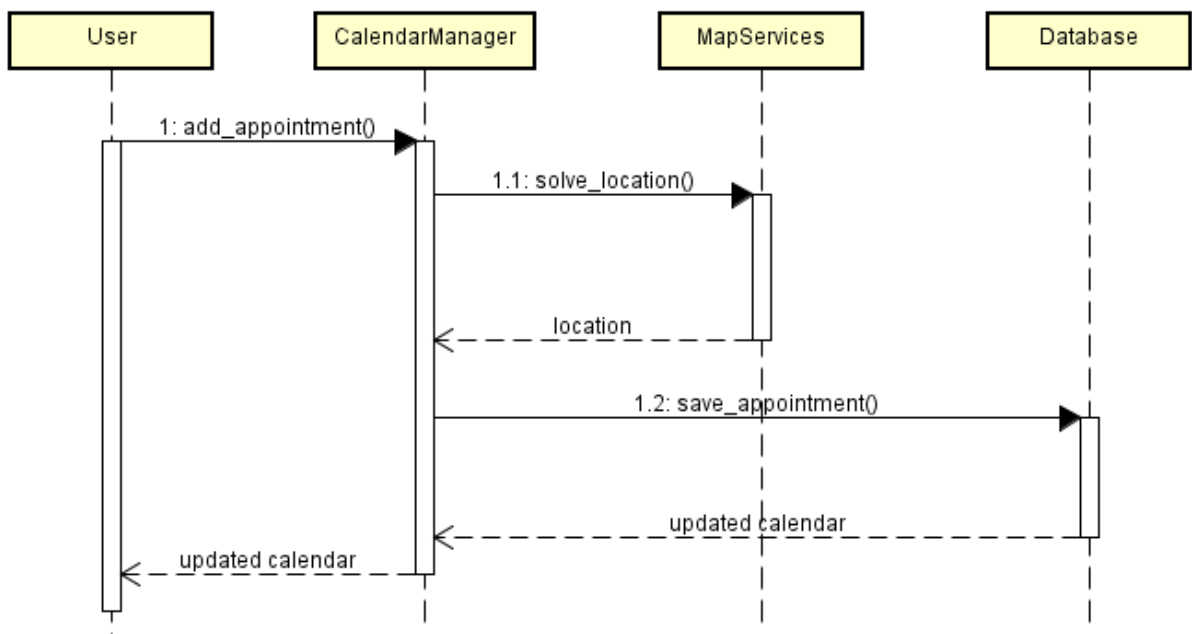


Figure 4: Add an appointment

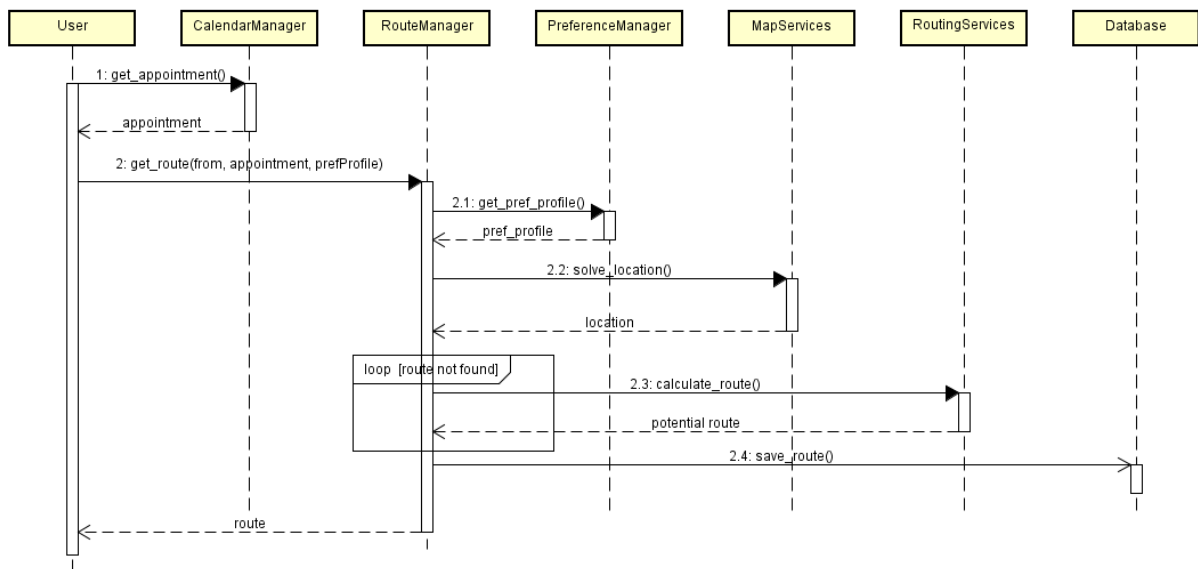


Figure 5: Add a route

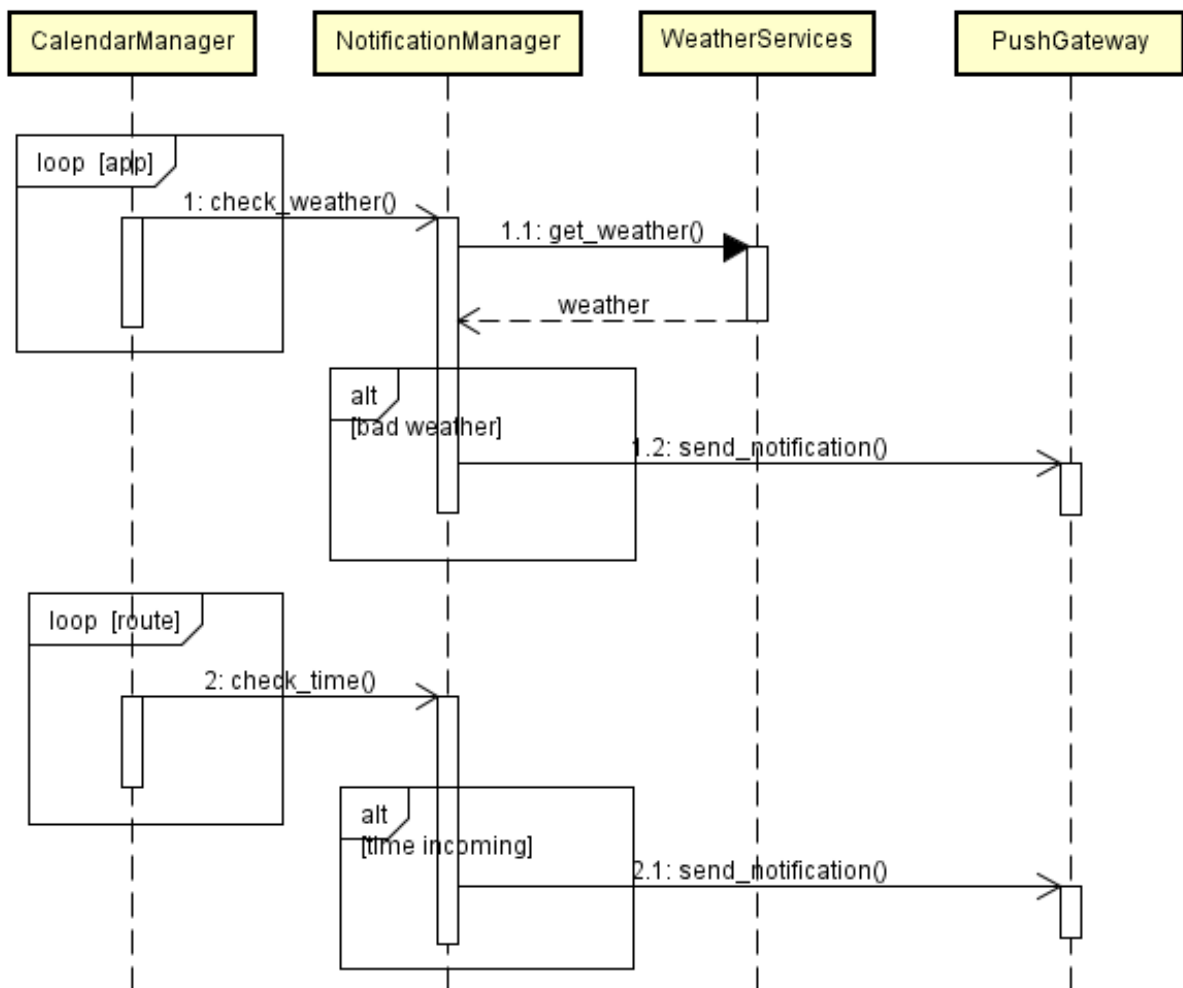


Figure 6: Send a notification (push or SMS)

## 2.6 Component interfaces

As shown in the component diagram, all components are interconnected. Here follows an explanation of the services offered and used by each component.

- The *SessionManager* offers an interface to the client tier. It gives the possibility for a user to login or sign up, using also the phone number. All the requests made by a user pass through this interface, checking if a user has the right to make that request and then forwarding it to the correct component.
- The *NotificationManager* offers methods to send push notifications and SMS messages. These methods are used by the *SessionManager* for allowing a user to login with a phone number, and by the *CalendarManager* to warn a user that a registered appointment or route is going to begin. This component relies on external services for delivering the messages.
- The *CalendarManager* is the container of the methods to handle the calendar and the appointments. It uses the interfaces of *RouteManager* and *ReportManager* for managing routes and reporting issues. Also map services are required when solving the addresses of the appointments.
- The *RouteManager* has the task to create, calculate and manipulate routes according to the needs of a user. It needs the external map and routing services and uses the *PreferenceManager* to get the necessary constraints while building a route.
- The *PreferenceManager* allows to create and manage the preference profiles of a user.
- The *ReportManager* allows to register issues, reports and suggestions written by users.

## 2.7 Selected architectural styles and patterns

As shown previously, the architecture is basically divided into three tiers where:

- The **client tier** is a graphical interface for the user with little data manipulation
- The **server tier** contains most of the business logic
- The **database tier** stores all the required data

With this perspective, the most suitable design patterns are the following:

- **Facade pattern:** a single interface is shown outside the server to the clients. The requests will be then forwarded to the necessary manager.
- **Proxy pattern:** the *SessionManager* acts not only as a request dispatcher, but also as a request filter, to distinguish the rights of a normal user from those of a registered user.
- **Adapter pattern:** the application makes high use of external services, which are unlikely to not change, be expanded or be replaced by other services. This is why the internal components of the server won't communicate directly with the services, but will use the methods offered by some adapter classes.
- **Observer pattern:** for allowing push notifications, it is reasonable to use this simple and robust pattern.

## 2.8 Other design decisions

In Section 2.1.3 of RASD a list of software interfaces are provided. Those will be the external services that will be used inside the application for routing and weather forecast. For map services, an open API will be used.

## 3 Algorithm Design

In this section we will describe the most important algorithms for the application Travlendar+ with a description and a snippet of Java code.

### 3.1 Check overlapping

In this application it is often needed to check if an appointment or route is not overlapping with other events in the calendar of the user. For this purpose, a specific method is required.

Input:

- *cal*: the calendar of the user who is adding the event
- *app*: an object representing the appointment to add or the appointment to which the route to add is linked
- *begin*: the starting time of the event as instance of `DateTime`
- *end*: the ending time of the event as instance of `DateTime`

Output:

- It returns true if the event does not overlap with other events, false otherwise.

The method treats as a special case the overlapping of a route with its own appointment: since it is allowed for a route to end after the corresponding appointment starts, the method won't return false if this case happens.

```

1 public boolean checkOverlapping(Calendar cal, Appointment app, DateTime begin, DateTime end) {
2     // check if the user is free during the timeslot begin-end
3     Set<Appointment> appsBefore = cal.getAppointments(begin.getDay()); // get appointments in the same day of the starting time
4     Set<Route> routesBefore = cal.getRoutes(begin.getDay()); // get routes in the same day of the starting time
5     Set<Appointment> appsAfter = cal.getAppointments(end.getDay()); // get appointments in the same day of the ending time
6     Set<Route> routesAfter = cal.getRoutes(end.getDay()); // get routes in the same day of the ending time
7
8     // special case for routes: app-route CAN overlap with its own appointment because it just raises a warning
9     appsBefore.remove(app);
10    appsAfter.remove(app);
11
12    // always check events starting and ending during the timeslot
13
14    // appointments overlapping with the starting time
15    for (Appointment app : appsBefore) {
16        if (app.beginTime < begin && app.endTime > begin || app.beginTime >= begin && app.endTime <= end) {
17            return false;
18        }
19    }
20
21    // routes overlapping with the starting time
22    for (Route route : routesBefore) {
23        if (route.beginTime < begin && route.endTime > begin || route.beginTime >= begin && route.endTime <= end) {
24            return false;
25        }
26    }
27
28    // appointments overlapping with the ending time
29    for (Appointment app : appsAfter) {
30        if (app.beginTime < end && app.endTime > end || app.beginTime >= begin && app.endTime <= end) {
31            return false;
32        }
33    }
34
35    // routes overlapping with the ending time
36    for (Route route : routesAfter) {
37        if (route.beginTime < end && route.endTime > end || route.beginTime >= begin && route.endTime <= end) {
38            return false;
39        }
40    }
41
42    return true;
43 }

```

Figure 7: Check overlapping between two events

### 3.2 Add appointments and routes

With the previously described method, adding an appointment or route becomes easy. Just check if there are no overlapping and then add the event to the calendar. There are two versions of the method based on the type of the event to add (appointment or route).

Input:

- *user*: an object representing the user who is doing the request. It is needed to retrieve its calendar.
- *app*: an instance of the class Appointment with the needed attributes
- *route*: an instance of the class Route with the needed attributes

Output:

- It modifies the calendar if the event is acceptable, otherwise it raises an exception.

The method which adds an appointment can be inserted inside the class that manages the calendar, while the method which adds a route should be added inside the class Appointment, since there is a one-to-one relationship between appointments and routes.

```
1 // inside CalendarManager
2 public void addAppointment(User user, Appointment app) {
3     Calendar cal = user.getCalendar();
4     DateTime begin = app.getBeginTime();
5     DateTime end = app.getEndTime();
6
7     if (checkOverlapping(cal, app, begin, end)) {
8         cal.insertAppointment(app);
9     } else {
10         throw new OverlappingException();
11     }
12 }
```

Figure 8: Add an appointment



```
1  // inside class Appointment
2  public void addRoute(User user, Route route) {
3      Calendar cal = user.getCalendar();
4      DateTime begin = route.getStart().getStartTime();
5      DateTime end = route.getEnd().getEndTime();
6
7      if (checkOverlapping(cal, this, begin, end)) {
8          this.route = route;
9          cal.insertRoute(route);
10     } else {
11         throw new OverlappingException();
12     }
13 }
```

Figure 9: Add a route

### 3.3 Remove routes of an appointment

In some cases, given an appointment, it will be needed to remove both the route going to the appointment and the route starting from it.

Input:

- *user*: the user to which the appointment is linked

Output:

- It removes both the incoming and the outgoing route from the appointment

The method will be implemented inside the class Appointment.

```

1  // inside class Appointment
2  public void removeAllRoutes(User user) {
3      Calendar cal = user.getCalendar();
4
5      // remove the route to the appointment
6      this.removeRoute(user);
7
8      // remove the route starting from the appointment, if one exists
9      Set<Route> routesOfTheDay = cal.getRoutes(this.getEndTime().getDay());
10     Route toBeRemoved = null;
11     for (Route r : routesOfTheDay) {
12         if (r.getStart().getPosition().equals(this.getPosition())) {
13             toBeRemoved = r;
14             break;
15         }
16     }
17     if (toBeRemoved != null) {
18         cal.removeRoute(toBeRemoved);
19     }
20 }

```

Figure 10: Remove all routes of an appointment

### 3.4 Remove an appointment

As stated in the RASD, when deleting an appointment, also its incoming and outgoing routes must be automatically removed. Thanks to the previous method, this becomes an easy task.

Input:

- *user*: the user to which the appointment to delete is linked.
- *app*: the appointment to delete

Output:

- The appointment is removed with its own routes.

The method is inside the class manager along with other methods that deal with appointments.

```
1 // inside CalendarManager
2 public void removeAppointment(User user, Appointment app) {
3     // when removing an appointment, also the routes starting from or ending to it must be deleted
4     app.removeAllRoutes();
5
6     // then remove the appointment from the calendar
7     cal.removeAppointment(app);
8 }
```

Figure 11: Remove an appointment

### 3.5 Updating an appointment

When modifying the attributes of an appointment, it is possible to change the time or the location of it. If this happens, the previously calculated routes become meaningless and so they shall be removed as well.

Input:

- *user*: the user to which the appointment is linked
- *app*: the appointment to modify
- *newBegin*: the new starting time
- *newEnd*: the new ending time
- *newPosition*: the new location

Output:

- The appointment is updated with the new attributes. If necessary, its routes are deleted.

```

1 // inside CalendarManager
2 public void updateAppointment(User user, Appointment app, DateTime newBegin, DateTime newEnd, Position newPosition) {
3     Calendar cal = user.getCalendar();
4     DateTime oldBegin = app.getBeginTime();
5     DateTime oldEnd = app.getEndTime();
6     Position oldPosition = app.getPosition();
7
8     // if starting or ending time or position changes, routes going to or coming from the appointment must be deleted
9     if (!oldBegin.equals(newBegin) || !oldEnd.equals(newEnd) || !oldPosition.equals(newPosition)) {
10         app.removeAllRoutes();
11     }
12
13     // then update with new attributes
14     app.setBeginTime(newBegin);
15     app.setEndTime(newEnd);
16     app.setPosition(newPosition);
17 }

```

Figure 12: Update an appointment

### 3.6 Calculating a route

The most important algorithm in the application is the calculation of the route to an appointment given a set of preferences and constraints.

Input:

- *user*: the user asking for the route
- *from*: the starting position
- *to*: the position to reach
- *start*: the starting hour as instance of `DateTime`
- *end*: the arriving hour as instance of `DateTime`

Output:

- A list of routes, each one minimizing a specific parameter

The algorithm calculates a set of possible routes between two points with respect to a list of constraints: the available vehicles, the priority of transport means, the timeslots of the vehicles, the maximum allowed length on foot or bicycle, a set of pauses (including Flexible Lunch) and the need to minimize four parameters (length, duration, number of changes, CO2 consumption).

```

1 // given the points and starting/ending time get a list of possible routes
2 public List<Route> calculateRoutes(User user, Position from, Position to, DateTime start, DateTime end) {
3     // load the preference profile, the set of vehicles in order of priority and the list of pauses
4     PreferenceProfile profile = user.getPreferenceProfile(); // TODO: select a specific profile preference
5     Set<TransportMean> availableVehicles = profile.getOrderedTransportSet();
6     List<TravelPause> pauses = profile.getTravelPauses();
7
8     List<Route> possibleRoutes = new ArrayList<Route>();
9     for (TransportMean tm : availableVehicles) {
10         // don't calculate routes on foot or bicycle if too long
11         if ((tm instanceof OnFoot || tm instanceof Bicycle) && distance(from, to) > profile.getMaxLengthOnFoot()) {
12             continue;
13         }
14         // don't calculate routes for vehicles which are out of their timeslot
15         DisableMean forbiddenTimeslot = tm.getForbiddenTimeslot();
16         if (forbiddenTimeslot.getStart() <= start && forbiddenTimeslot.getEnd() >= start
17             || forbiddenTimeslot.getStart() <= end && forbiddenTimeslot.getEnd() >= end
18             || forbiddenTimeslot.getStart() >= start && forbiddenTimeslot.getEnd() <= end) {
19             continue;
20         }
21         // calculate the route with that vehicle
22         // queryRoute returns a route with all necessary data (length, duration, changes, CO2)
23         Route possibleRoute = queryRoute(from, to, start, end, tm);
24
25         // modify the route according to the given pauses
26         for (TravelPause tp : pauses) {
27             // if a route part interleaves with a pause, delete it
28             for (RoutePart rp : possibleRoute.getRouteParts()) {
29                 if (rp.getStartTime() <= tp.getStart() && rp.getEndTime() >= tp.getStart()
30                     || rp.getStartTime() <= tp.getEnd() && rp.getEndTime() >= tp.getEnd()
31                     || rp.getStartTime() >= tp.getStart() && rp.getEndTime() <= tp.getEnd()) {
32                     possibleRoute.removePart(rp);
33                 }
34             }
35
36             // if some route parts have been deleted, calculate a new route for that part out of the pause
37             for (RoutePart rp : possibleRoute.getRouteParts()) {
38                 RoutePart next = rp.getNext();
39                 // detect two disconnected route parts
40                 if (!rp.getEnd().equals(next.getStart())) {
41                     // check if you have more time before or after the pause
42                     int timeBeforePause = tp.getStart() - rp.getEndTime();
43                     int timeAfterPause = next.getStartTime() - tp.getEnd();
44                     // add new route with addRoutePart(newRoute, rp1, rp2)
45                     if (timeBeforePause > timeAfterPause) {
46                         // check if the pause is flexible: in that case there is less constraint on the duration of the route
47                         if (tp.isFlexible()) {
48                             possibleRoute.addRoutePart(queryRoute(rp.getEnd(), next.getStart(), rp.getEndTime(), tp.getEnd() - tp.getMinimumDuration(), tm), rp, next);
49                         } else {
50                             possibleRoute.addRoutePart(queryRoute(rp.getEnd(), next.getStart(), rp.getEndTime(), tp.getStart(), tm), rp, next);
51                         }
52                     } else {
53                         if (tp.isFlexible()) {
54                             possibleRoute.addRoutePart(queryRoute(rp.getEnd(), next.getStart(), tp.getStart() + tp.getMinimumDuration(), next.getStartTime(), tm), rp, next);
55                         } else {
56                             possibleRoute.addRoutePart(queryRoute(rp.getEnd(), next.getStart(), tp.getEnd(), next.getStartTime(), tm), rp, next);
57                         }
58                     }
59                 }
60             }
61         }
62
63         // add it to a list
64         possibleRoutes.add(possibleRoute);
65     }
66
67     // find routes that minimize each parameter
68     List<Route> finalRoutes = new ArrayList<Route>();
69     if (possibleRoutes.size() > 0) {
70         Route leastLength = leastDuration = leastChanges = leastCO2 = possibleRoutes.get(0);
71         for (Route r : possibleRoutes) {
72             if (r.getLength() < leastLength.getLength()) {
73                 leastLength = r;
74             }
75             if (r.getDuration() < leastDuration.getDuration()) {
76                 leastDuration = r;
77             }
78             if (r.getChanges() < leastChanges.getChanges()) {
79                 leastChanges = r;
80             }
81             if (r.getCO2() < leastCO2.getCO2()) {
82                 leastCO2 = r;
83             }
84         }
85         finalRoutes.add(leastLength);
86         finalRoutes.add(leastDuration);
87         finalRoutes.add(leastChanges);
88         finalRoutes.add(leastCO2);
89     }
90
91     return finalRoutes;
92 }
93

```

Figure 13: Calculate routes

## 4 User Interface Design

### 4.1 Mock ups

The following mock-ups give an idea of how the mobile user interface will be structured. The software used for realizing these mock-ups is Pencil, which doesn't produce very nice looking interfaces, but it is helpful for creating a structure that will be followed when realizing the real user interface.

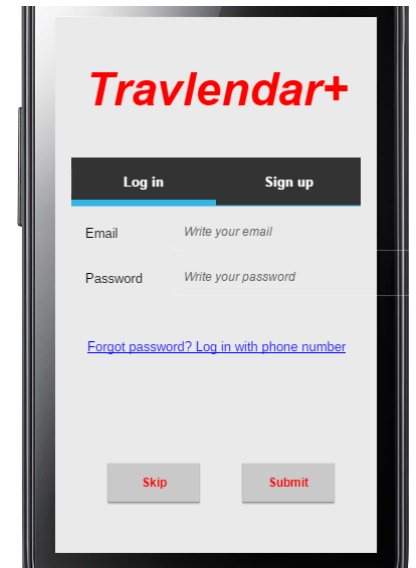
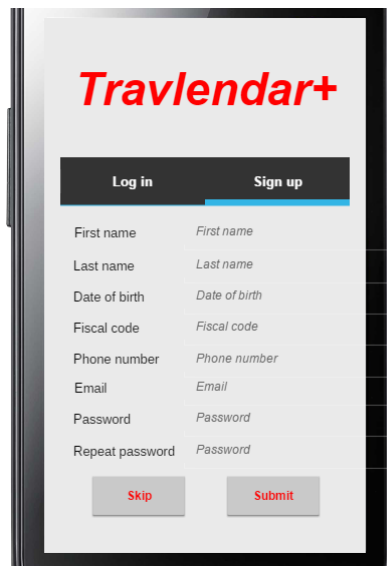


Figure 14: Login screen



The image shows a mobile application screen for signing up. At the top, the app name "Travlendar+" is displayed in red. Below it, there are two tabs: "Log in" and "Sign up", with "Sign up" being the active tab. The form contains several input fields for user registration: First name, Last name, Date of birth, Fiscal code, Phone number, Email, Password, and Repeat password. Each field has a placeholder text. At the bottom, there are two buttons: "Skip" and "Submit".

**Travlendar+**

Log in Sign up

First name First name

Last name Last name

Date of birth Date of birth

Fiscal code Fiscal code

Phone number Phone number

Email Email

Password Password

Repeat password Password

Skip Submit

Figure 15: Sigup screen



The image shows a mobile application screen displaying a calendar for November 2017. The calendar is a grid with days of the week (M, T, W, T, F, S, S) and dates. The 24th of November is highlighted with a blue background. Below the calendar, there is a section titled "24th November" showing a list of appointments with their times and locations. Each appointment has "Change" and "Delete" buttons. At the bottom, there is an "Add appointment" button.

**Calendar**

**November 2017**

M	T	W	T	F	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

**24th November**

8:30 - 10:00	Lecture in L26	Change	Delete
10:00 - 10:15	Route to N	Change	Delete
10:30 - 12:00	Lecture in N	Change	Delete

Add appointment

Figure 16: Home screen with calendar

**Appointment**

**New appointment**

Date: 24th November

Starting time: 00:00

Ending time: 01:00

Location:

map and geolocation from map services

Description:

**Add route** **Submit**

Figure 17: Add an appointment

**Route**

Route for appointment  
on 24th November at 11:00

Starting location and time

☒ From previous appointment

☐ From here and now

☐ From custom location and time

Time: 00:00

Location:

map and geolocation from map services

**Preferences** **Submit**

Figure 18: Add a route



**Preferences**

Preferences for appointment  
on 24th November at 11:00

Vehicles	Priority	Validity
<input checked="" type="checkbox"/> Car	High	00:00 01:00
<input checked="" type="checkbox"/> Bicycle	Low	00:00 01:00
<input checked="" type="checkbox"/> On foot	Low	00:00 01:00
<input checked="" type="checkbox"/> Bus	Normal	00:00 01:00
<input checked="" type="checkbox"/> Tram	Normal	00:00 01:00
<input checked="" type="checkbox"/> Underground	High	00:00 01:00
<input checked="" type="checkbox"/> Train	Low	00:00 01:00
<input checked="" type="checkbox"/> Taxi	Low	00:00 01:00
<input checked="" type="checkbox"/> Car sharing	Low	00:00 01:00
<input checked="" type="checkbox"/> Bike sharing	Low	00:00 01:00

Maximum length  
on foot or bicycle (km)

☒ Break 00:00 01:00

☒ Flexible

[Add break](#)

Figure 19: Set preferences

**Route**

☒ Minimum length

10:30 --- Walk to ...  
--- Take bus number ...  
--- Proceed until stop ...  
--- Walk to ...  
11:00 You have arrived

Length: 2 km Changes: 0  
Duration: 30 m Carbon: ...

☐ Minimum duration

☐ Minimum number of changes

☐ Minimum carbon footprint

Figure 20: Select a route

## 4.2 UX diagram

The following UX diagram shows how a user of the application would navigate through the different screens while using the available features. Each screen is represented as a class with stereotype «screen» and contains the possible functionalities, while the classes with stereotype «input form» contains attributes that the user must provide inside a specific screen. The diagram has been created following the structure of the mockups in the previous paragraph.

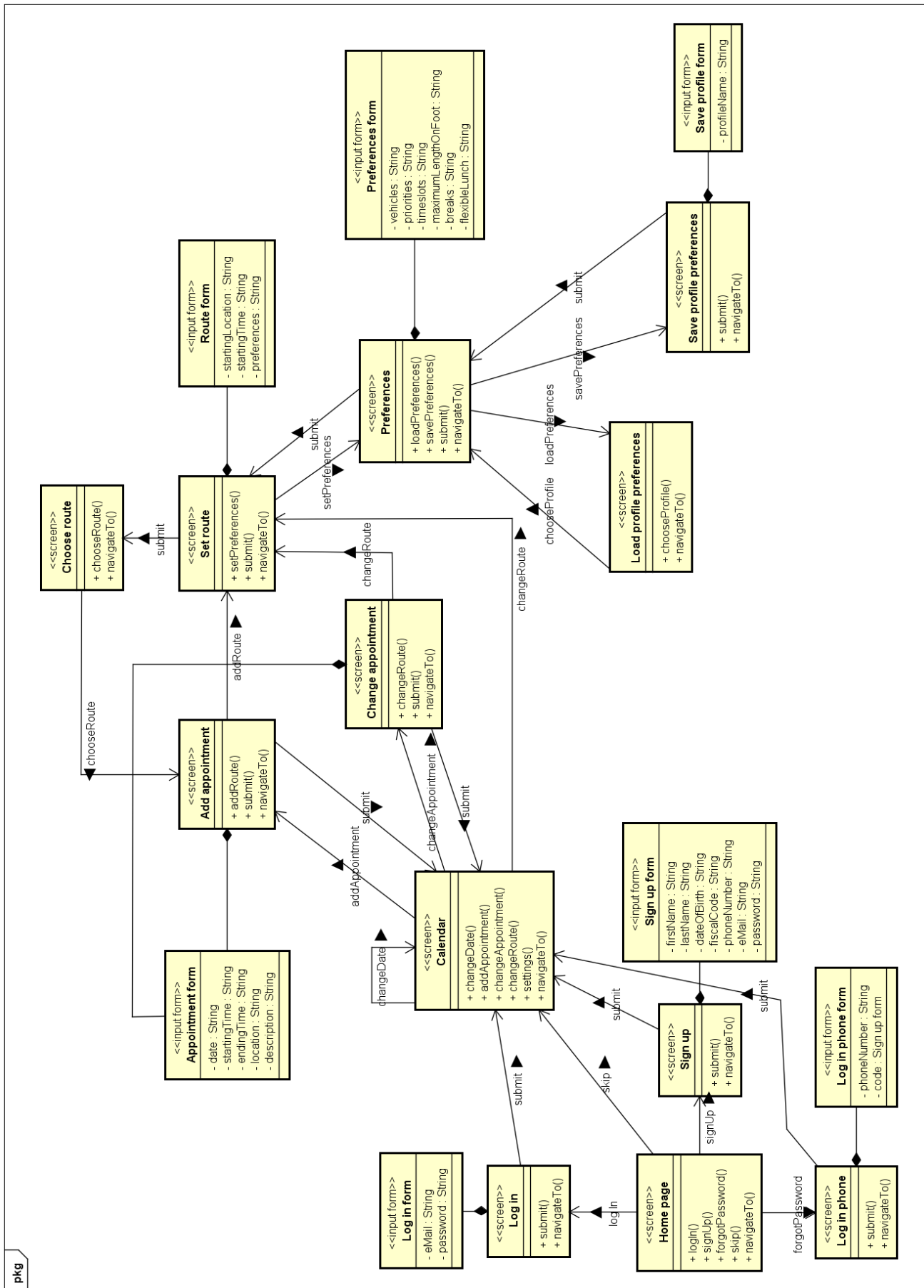


Figure 21: UX Diagram

## 5 Requirements Traceability

The purpose of this design project is to build up a system able to fulfil all requirements proposed in the bounded RASD document.

In the following section, it is reported the list of project's goals, and for each of them is given the set of components involved in related operations.

G1 Allow the user to add an appointment

[G1.1] The user can add the date of the appointment through a calendar

- Web Application
- Mobile Application

[G1.2] The user can select the location through a map

- Web Application
- Mobile Application
- SessionManager
- CalendarManager
- Map Services

[G1.3] The appointment must be processed by the system

- SessionManager
- CalendarManager
- Web Application
- Mobile Application

G2 - Provide a route to the user for reaching the appointments

[G2.1] The user must reach on time his/her appointments

- CalendarManager
- RouteManager
- Routing Services

[G2.2] The user can choose the starting location and time of the route

- Web Application
- Mobile Application
- SessionManager
- CalendarManager
- RouteManager

[G2.3] Generate routes according to the preferences of the user

- RouteManager
- PreferenceManager

- Database

[G2.4] The application provides a route for each objective, minimizing each of these attributes: length, duration, number of changes, carbon footprint

- RouteManager
- PreferenceManager

[G2.5] Always provide a route

- RouteManager
- CalendarManager
- NotificationManager
- Push Gateway

[G2.6] During strike days, public transportation must not be available

- CalendarManager
- ReportManager

[G2.7] Report unfavourable weather conditions

- NotificationManager
- Push Gateway

[G2.8] Update unfavourable weather conditions

- NotificationManager
- Weather Services

[G2.9] The user can save one route among the shown ones

- Web Application
- Mobile Application
- SessionManager
- CalendarManager
- Database

G3 Allow the user to sign up into the application

[G3.1] The registration must allow the univocal recognition of the user

- Web Application
- Mobile Application
- SessionManager
- Database

G4 Allow the user to log in

[G4.1] The system allow the login through e-mail and password

- Web Application
- Mobile Application
- SessionManager
- Database

[G4.2] The application allow the login through telephone number

- Web Application
- Mobile Application
- SessionManager
- Database

G5 Allow the user to add his own preferences

[G5.1] The user must be logged

- Web Application
- Mobile Application
- SessionManager
- Database

[G5.2] The preferences are synchronized between the application and the database

- SessionManager
- PreferenceManager
- Database

[G5.3] The user can choose the available transport means

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

[G5.4] The user can add a priority to the means of transport

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

[G5.5] The user can add the maximum length of routes on foot or by bicycle

- Web Application
- Mobile Application
- SessionManager

- PreferenceManager
- Database

[G5.6] For each vehicle the user can choose a time slot of validity

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

[G5.7] The user can set Flexible Lunch

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

[G5.8] The user can add breaks for fixed moments of the day

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

[G5.9] The user can add a private car or bicycle

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

[G5.10] The user can organize his preferences in "Preferences Profiles"

- Web Application
- Mobile Application
- SessionManager
- PreferenceManager
- Database

G6 Allow the user to manage his account

[G6.1] The user must be able to remove appointments and routes

- Web Application

- Mobile Application
- SessionManager
- CalendarManager
- Database

[G6.2] The user must be able to modify appointments and routes

- Web Application
- Mobile Application
- SessionManager
- CalendarManager
- RouteManager
- Database

[G6.3] The user must be able to delete his account

- Web Application
- Mobile Application
- SessionManager
- Database

G7 Allow the user to report events and disservices

[G7.1] The user can report strikes using the application

- Web Application
- Mobile Application
- SessionManager
- ReportManager
- Database

[G7.2] The user can report faults, malfunctions and suggestions

- Web Application
- Mobile Application
- SessionManager
- ReportManager
- Database



## 6 Implementation, Integration and Test Plan

### 6.1 Integration Strategy

#### 6.1.1 Entry Criteria

Integration and testing processes should be performed on each single unit as soon as the component has been fully developed. In addition, integration of different components should be performed only if all these criteria are satisfied:

- RASD and DD documents can be considered "stable", therefore it isn't expected any other modification of requirements or structure of the system, furthermore both deliverables ought to be distributed to all developer team involved in the project.
- There are some dependencies in integration and testing plan that have to be satisfied in order to guarantee the possibility to perform useful cross-components tests on functionalities, because of intrinsic bounds of the architecture.
  1. *Map Services* and *Route Services* for *RouteManager*
  2. *SMS Gateway* and *Push Gateway* for *NotificationManager*
  3. *Map Services* for *MA* and *WebUI*
  4. *DBMS* for *CalendarManager*, *ReportManager* and *PreferenceManager*

#### 6.1.2 Elements to be integrated

It is possible to distinguish three different categories of components, basing the grouping on set of functionalities covered, their interaction and integration dependencies.

- **FrontEnd components:** set of units involved in the management of interactions between the system and user. Its modules are distributed among web and client tiers.
  1. *Mobile Application*
  2. *Web Application*
  3. *SessionManager*
- **BackEnd components:** units that provides the business logic for all system's features. Entirely located on application tier.
  1. *SessionManager*
  2. *CalendarManager*
  3. *RouteManager*
  4. *PreferenceManager*
  5. *NotificationManager*
- **Services components:** atomic components, without any dependency, each one provides a single basic functionality to the system. They are mainly located on the application tier.
  1. *DBMS*
  2. *Route Services*
  3. *Map Services*
  4. *WeatherServices*
  5. *Push Gateway*
  6. *SMS Gateway*

### 6.1.3 Integration Strategy

Given the foretold component categorization, it is possible to easily set up a bottom up strategy for the integration of implemented components. This plan aims to reduce the whole testing effort needed to deploy complex drivers and stubs employed in the integration process.

The following integration plan will give higher priority to implementation, integration and unit test to components that doesn't rely on undeveloped ones. According to that scheme, the first developing iteration will focus on *services components*, followed by units with less unsatisfied dependencies.

It is also possible to parallelize large part of the development process working on parallel branches of the integration sequence diagram, smoothing the entire development process, without modifications to the integration and testing strategy.

Actually, the development of some component parts can be developed separately and asynchronously respect the unit itself. This practise is encouraged in order to offer the possibility to testing vertically some core functionalities of the system before the end of development. Further details on thread testing strategy in [6.2](#).

It is provided a list of functionalities and unit parts that can be implemented and integrated interdependently for testing purpose:

- **Registration and Login**

*Web Application*: UI with login/registration functionalities

*Mobile Application*: UI with login/registration functionalities

*SessionManager*: login/registration and client communication implemented

*DBMS*: implemented

- **Appointment management** (add/remove/modify an appointment) without routes definition

*Web Application*: UI with appointment management functionalities

*Mobile Application*: UI with appointment management functionalities

*SessionManager*: appointment management and client communication implemented

*CalendarManager*: appointment management implemented

*DBMS*: implemented

- **Preference management** (add/remove/modify preferences), requires *Registration and Login*

*Web Application*: UI with preference management functionalities

*Mobile Application*: UI with preference management functionalities

*SessionManager*: preference management and implemented

*PreferenceManager*: preference management implemented

*DBMS*: implemented

### 6.1.4 Integration sequence

Directed arc from component-A to component-B is intended as a dependency of component-B on functionalities offered by component-A. The implementation, integration and testing scheme starts from the bottom with first to be developed units, and propagates to the top, stepping forward only if all dependencies are satisfied.

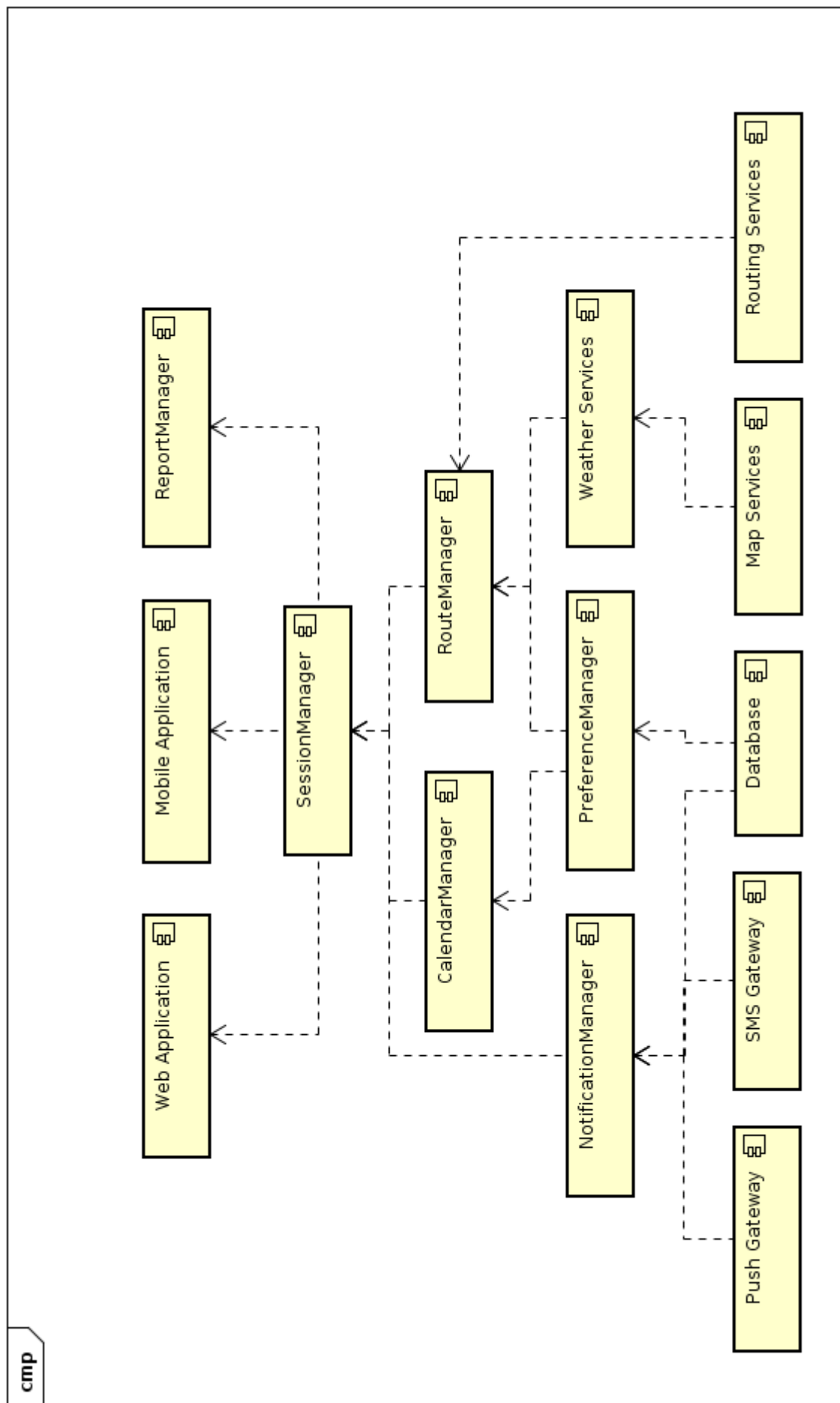


Figure 22: Bottom-Up integration and testing components diagram.

## **6.2 Test Plan**

### **6.2.1 Testing Strategy**

### **6.2.2 Testing Sequence**

## 7 Effort Spent

<b>Daverio</b>	20/11/2017	1h15m
	21/11/2017	1h30m
	22/11/2017	2h45m
	24/11/2017	1h
	25/11/2017	3h
	26/11/2017	7h
	<b>Total</b>	16h30m

<b>Fiorillo</b>	15/11/2017	1h30m
	16/11/2017	1h30m
	17/11/2017	1h30m
	18/11/2017	3h
	22/11/2017	1h
	23/11/2017	1h30m
	24/11/2017	2h50m
	25/11/2017	6h
	26/11/2017	4h
	<b>Total</b>	23h50m