Politecnico di Milano

A.Y. 2017-2018

TRAVLENDAR+

DD

Design Document

Version 1.0

Greco Sonia *matricola 893706*

Guzzo Francesco *matricola 898035*

Release Date: 26.11.2017

TABLE OF CONTENTS

1. INTRODUCTION

1.1 PURPOSE

The purpose of this document is to increase the level of details, already presented in the RASD, about Travlendar+ project. As a matter of fact, this document contains information about the high-level architecture, with the description of the styles, patterns and all the design decisions that have been taken and chosen for the realization of this project. Furthermore, the aim of this document is also to give an idea of the final view of the application, as far as the software components and the interfaces are concerned. More importance is given to the algorithms and implementation & testing phase, as well.

1.2 SCOPE

Travlendar+ is a useful mean for the everyday life of all kinds of people, as already shown in the RASD. The application aims to help the users in the daily issue of keeping track of all the events scheduled in their agenda and finding the optimal mobility option to reach them, among all the possible ones. The user can personalize his/her calendar, adding some information about him/herself. For example, it is possible to add personal mobility means, personal disabilities (that could exclude, for example, driving or walking in the choice of the transport mean), a personal flexible slot of time for lunch break and other personal preferences. The scope of the system is, also, to help users understand if a meeting is reachable in the due time, keeping him/her sure that they will never be late.

1.3 DEFINITIONS, ACRONYMS, ABBREVIATIONS

- RASD: Requirement Analysis and Specification Document
- DD: Design Document
- UML: Unified Modeling Language
- RMI: Remote Method Invocation
- DMZ: Demilitarized Zone
- EJB: Enterprise Java Beans
- JNDI: Java Naming and Directory Interface
- JPA: Java Persistence API

1.4 REVISION HISTORY

First release of the document: 26th November 2017.

1.5 REFERENCE DOCUMENT

The following documents have been used as references:

- Travlendar+ RASD version 1.0
- Mandatory Project Assignment.pdf

1.6 DOCUMENT STRUCTURE

Section 1 of this document presents an overall description of the purpose and the scope of the document itself.

In section 2, the reader will find a sequence of diagrams which explain the software architecture of the Travlendar+ application. Component view diagrams, deployment view diagrams, runtime view (sequence diagrams) and component interface diagrams will be provided and described.

The third section presents the main algorithms of the application logic.

In Section 4 we show some examples of the user's interface Mobile App and in Section 5, a description of how we plan to implement and respect the requirements expressed previously in the RASD is provided.

Lastly, in Section 6, we discuss about implementation and testing plan, based on the architecture's structure and components' integration.

2. ARCHITECTURAL DESIGN

2.1 OVERVIEW

This section of the document contains the architectural design of the project.

The reader will find:

• *Component View Diagrams* – These diagrams explain how the components and subcomponents of the project are wired and connected to each other. They are also resumed in the section 6.3 of the present document, to easily represent the integration of the software's components.

• *Deployment View Diagram* – This diagram represents the tiers that compose the system. In this case we use a 3 tiers-Architecture, consisting in a mobile application, an application server and a database server.

• *Runtime View Diagrams* – Here we include the sequence diagrams already provided in the *RASD* but in an updated and more detailed way.

• *Component Interfaces* –This section includes UML component diagrams of the Client Mobile Services and the Server Mobile Services.

• *Architectural Styles and Patterns* – Here we include the main styles and software design patterns we plan to use in the development of the project.

2.2 COMPONENT VIEW

The following diagrams show how components and interfaces interacts between each other.

In Figure 2.2.1, the Client-Server relation is described. The following components represent *client mobile services* and the related interface of *server mobile services* - both client and server have specific components that are described and grouped into subsystems in the following.

- *Client mobile services* specification consists in a component diagram related to personal data management, such as user preferences, daily schedule, creation/deletion of events, profile configuration and strikes notification.
- *Server mobile services* specification consist into two components capable to gather information from different APIs they are related with Google's and weather API.

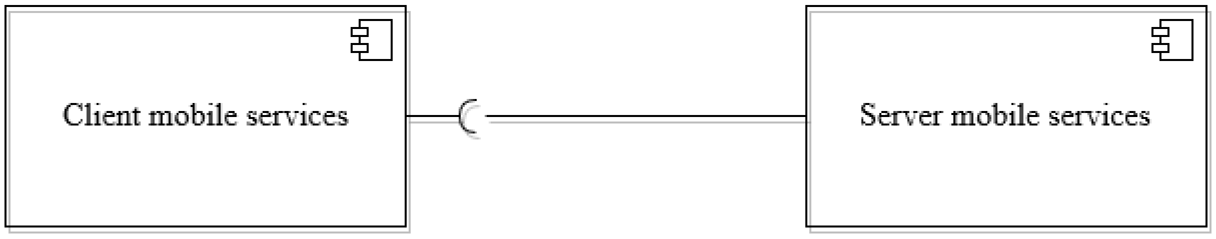


Figure 2.2.1 Client-Server relation

As shown in Figure 2.2.2, the components into *Client mobile services* subsystem are the following:

- *Event module management* provides to the client system two different interfaces, the *create event* and the *delete event* interface. The component is directly linked with the local DBMS of the application where all the personal data of the user are stored.
- *Preferences module* provides to the user all the different interfaces he needs to modify the settings that will influence the computation of the schedule; exploiting the related interface, the user will be able to modify walking and biking time, eco choices and set lunch time and lunch time ranges. *Modify preferences* is the related interface. The component is directly linked with the local DBMS of the application where all the personal datas of the user are stored.
- *Strike notification module* provides the interface *addStrike* that allows the user to set a strike for a specific mobile mean in a specic day. The component is directly linked with the local DBMS of the application where all the personal datas of the user are stored.
- *Account manager* provides the *login* and *register* interfaces that allow the user to set up his account for the application. The component is both linked to the local DBMS and the remote components on the server side.
- *Schedule* provides the user the possibility to refresh the current journey, search for alternative or cheaper paths or just acknowledge the daily schedule. The related interfaces are : *show alternative, cheapest path, show schedule*. The component is linked to both the local DBMS and the remote components on the server side.

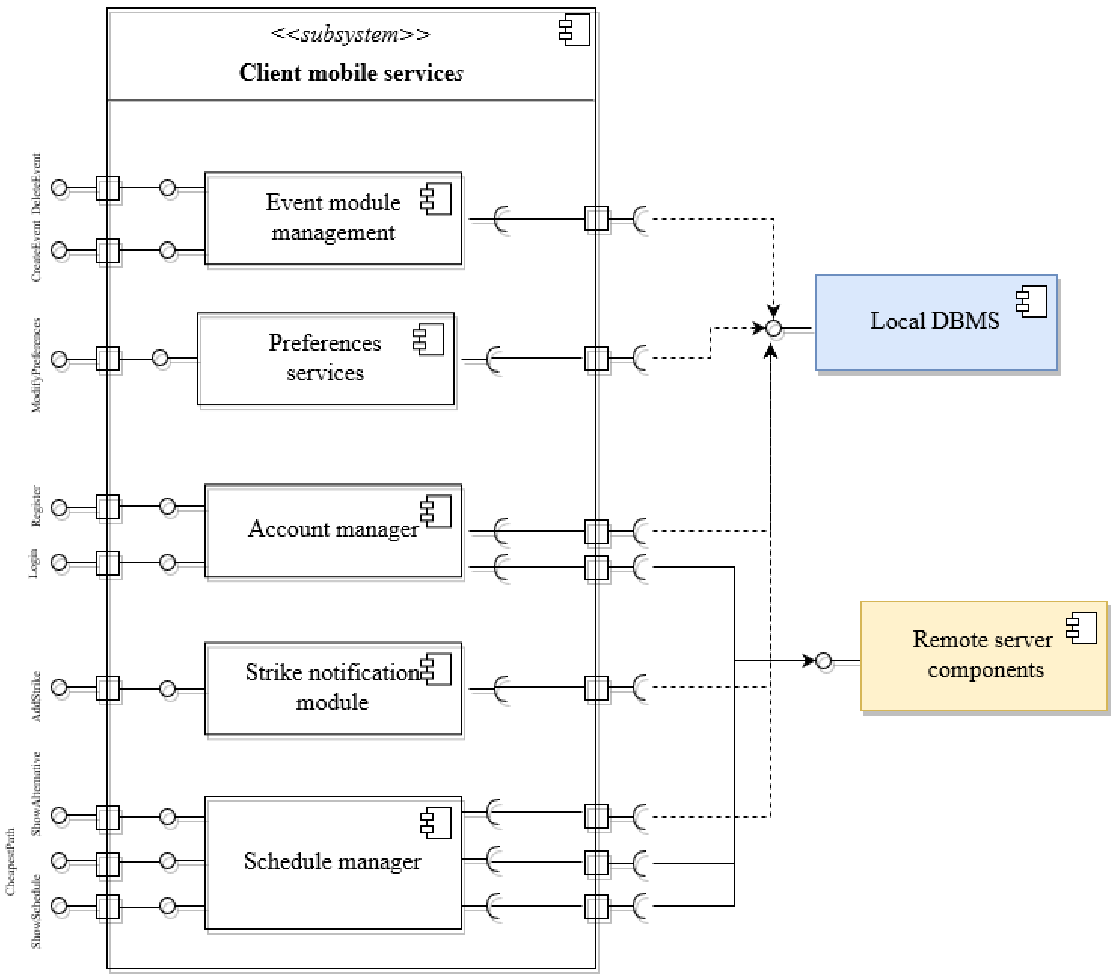


Figure 2.2.2 Client Mobile Services – component view

As shown in Figure 2.2.3, the components into *Server mobile services* subsystem are the following:

- *Preferences remote management* provides the interface *influence route computation* which receives the user's preferences from the client side and guides the route computation. The component is linked with the server side DMBS.
- *Account manager* impements the interface *account validation* whose aim is to register in the server DBMS user's credentials and validate them, allowing him/her to use the Travlendar+ application. The component is linked with the server side DMBS.
- *Route management* with the interface *route computation* is the main component of the subsystem and calculates the routes between the events of the user schedule gathering infos from Google's API and wheather API. The component is both linked to the remote DBMS and the APIs.

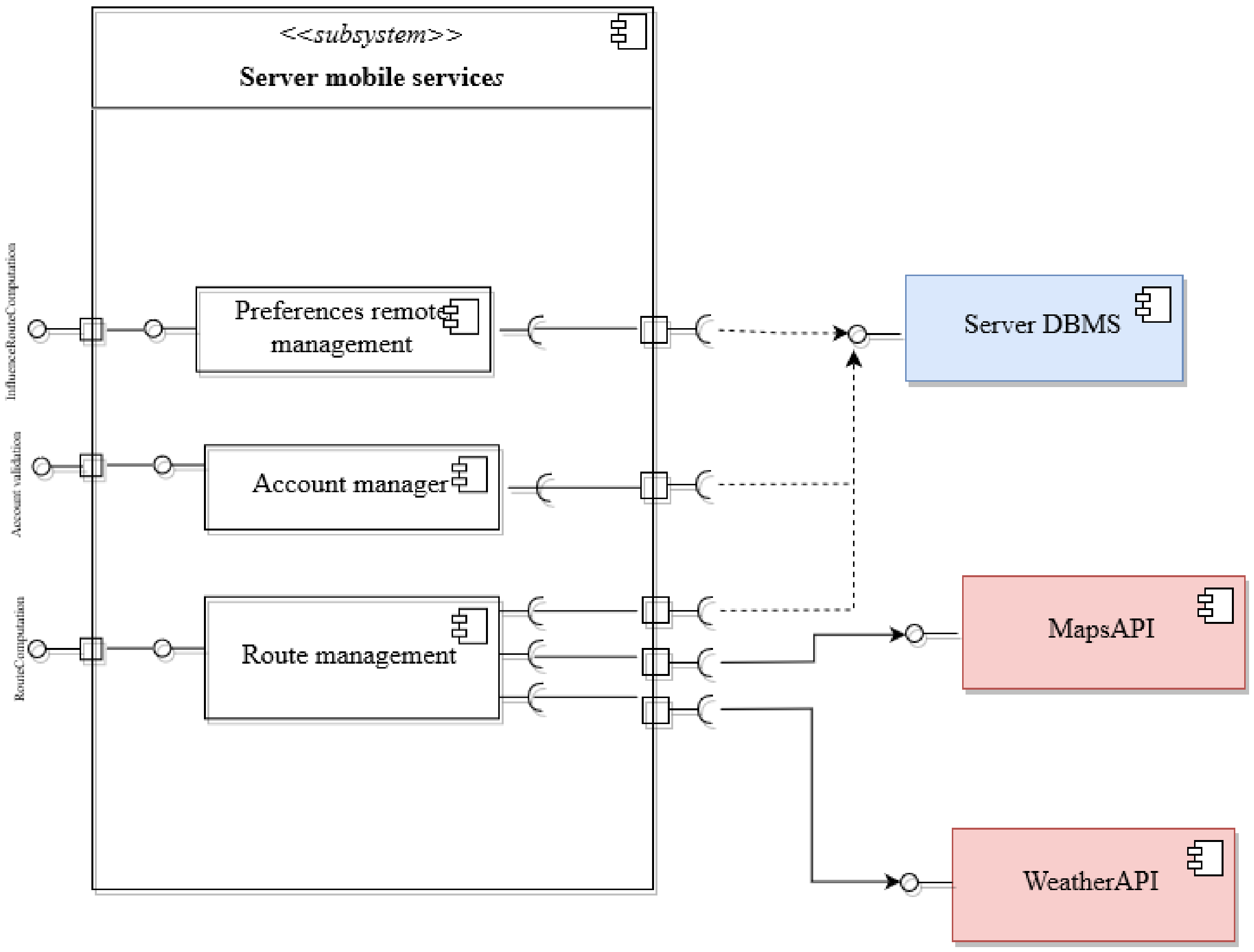


Figure 2.2.3 Server Mobile Services – component view

2.3 DEPLOYMENT VIEW

The system architecture is divided into 3 tiers and based on the JEE framework:

- *Tier 1*: This tier represents the client side that consists in the mobile application. The container of the mobile application is a structure composed by the interface, the local logical layer and the local DBMS. The container is linked through *RMI* to a DMZ zone.
- *Tier 2*: This tier represents the application server. It is placed into two firewalls to create a DMZ zone. The related container is based on the Enterprise Java Beans (EJB) software components. In order to access the beans in the application server, the RMI system has to communicate with the EJB container. Inside the container, there are *stateful* beans; each bean has a state defined by its instances and remains passive until a request from a client is received. The application server disposes of JNDI service which, through JNDI APIs, can look up by name to client side data source and make a *resource injection* to wake up a bean in the EJB container. The application server uses the java persistent API(JPA) to manage stored data and map entity to relations into the database server.
- *Tier 3*: This tier is the database tier that consists only of the server side entity-relation DBMS implemented through *oracle database.*

Figure 2.3.1 shows a graphical representation of what said so far.

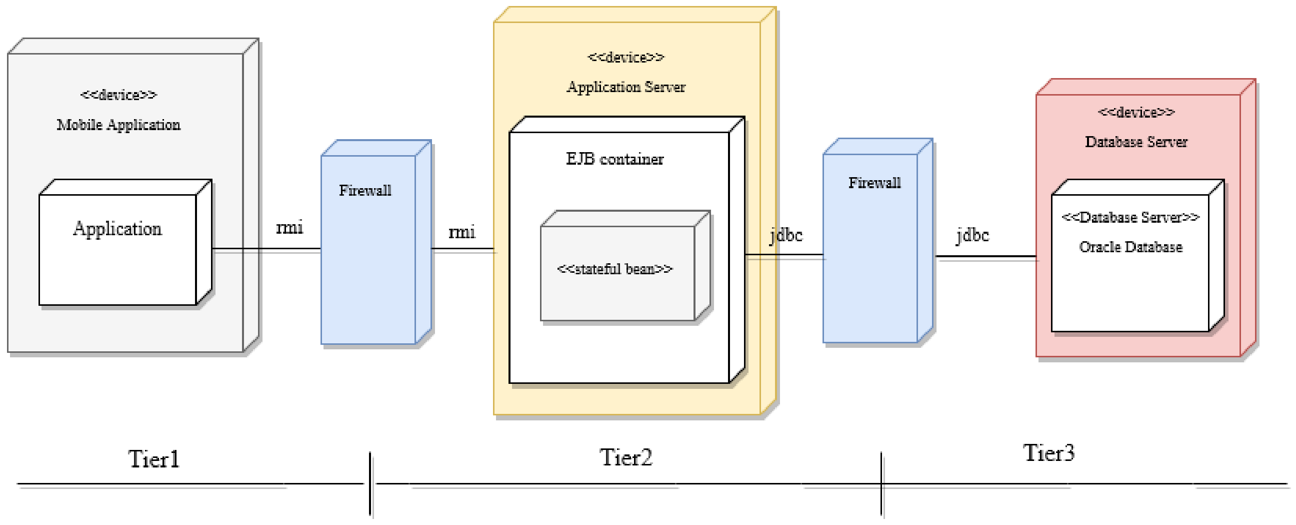


Figure 2.3.1 Deployment view

Therefore, the following are recommended implementation:

- *Client side*: android development tool;
- *Application server*: EJB software components and JPA.
- *Database server*: the database may be implemented through oracle database.

2.4 RUNTIME VIEW

In the present section, we provide the same sequence diagrams already provided in the *RASD*, but with more details with respect to the architecture components described in this document.

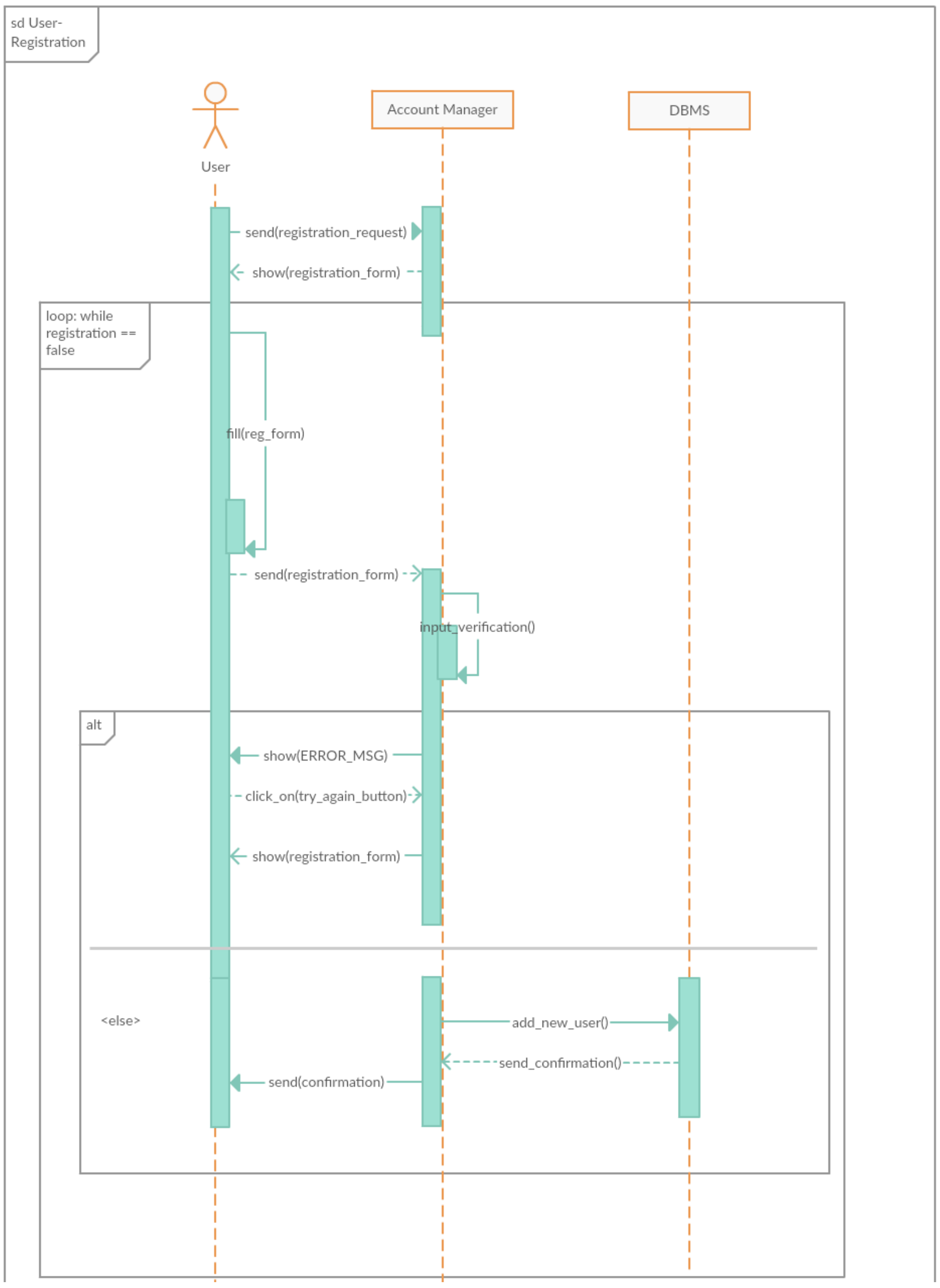


Figure 2.4.1 Registration – sequence diagram

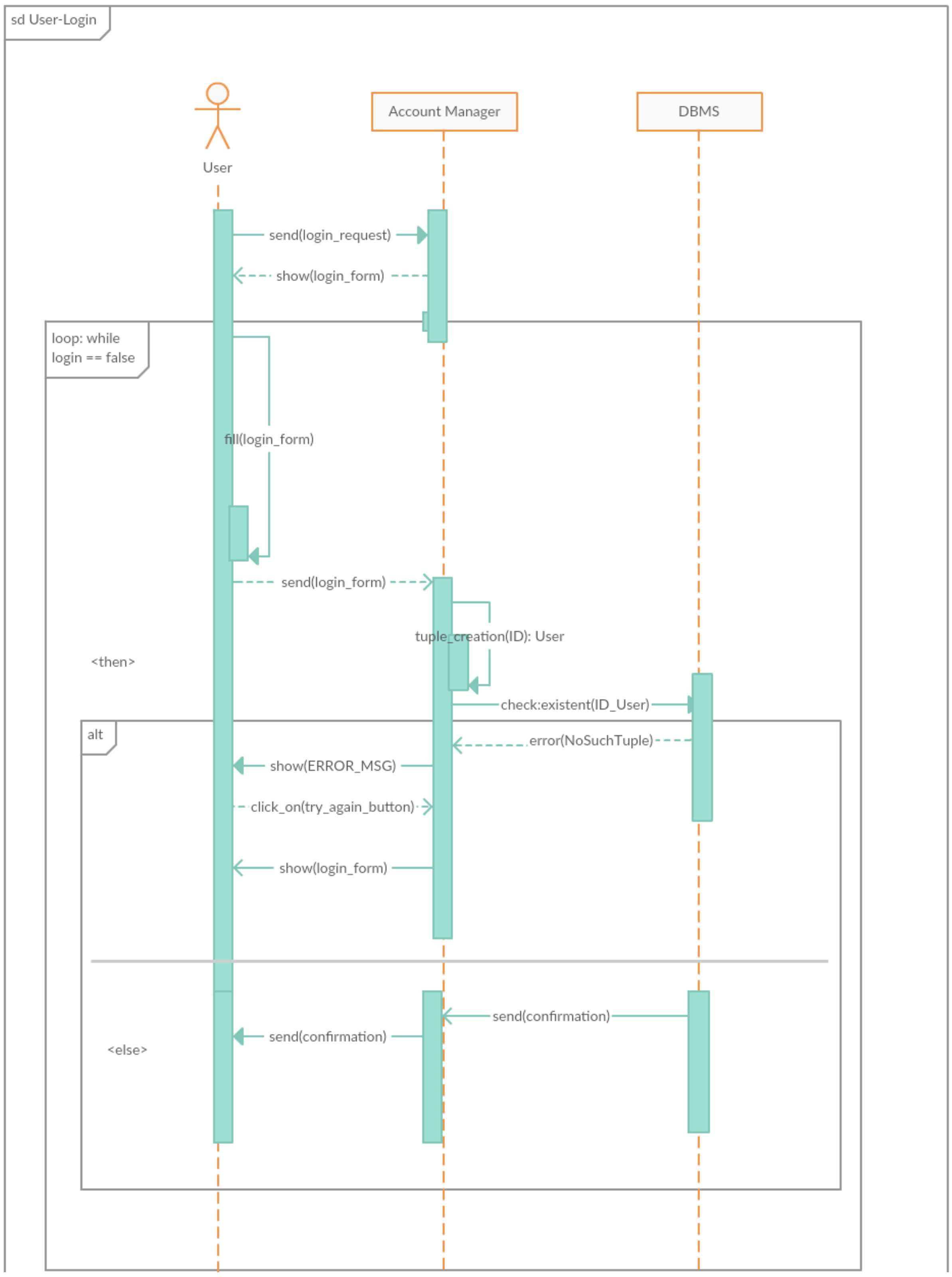


Figure 2.4.2 Login – sequence diagram

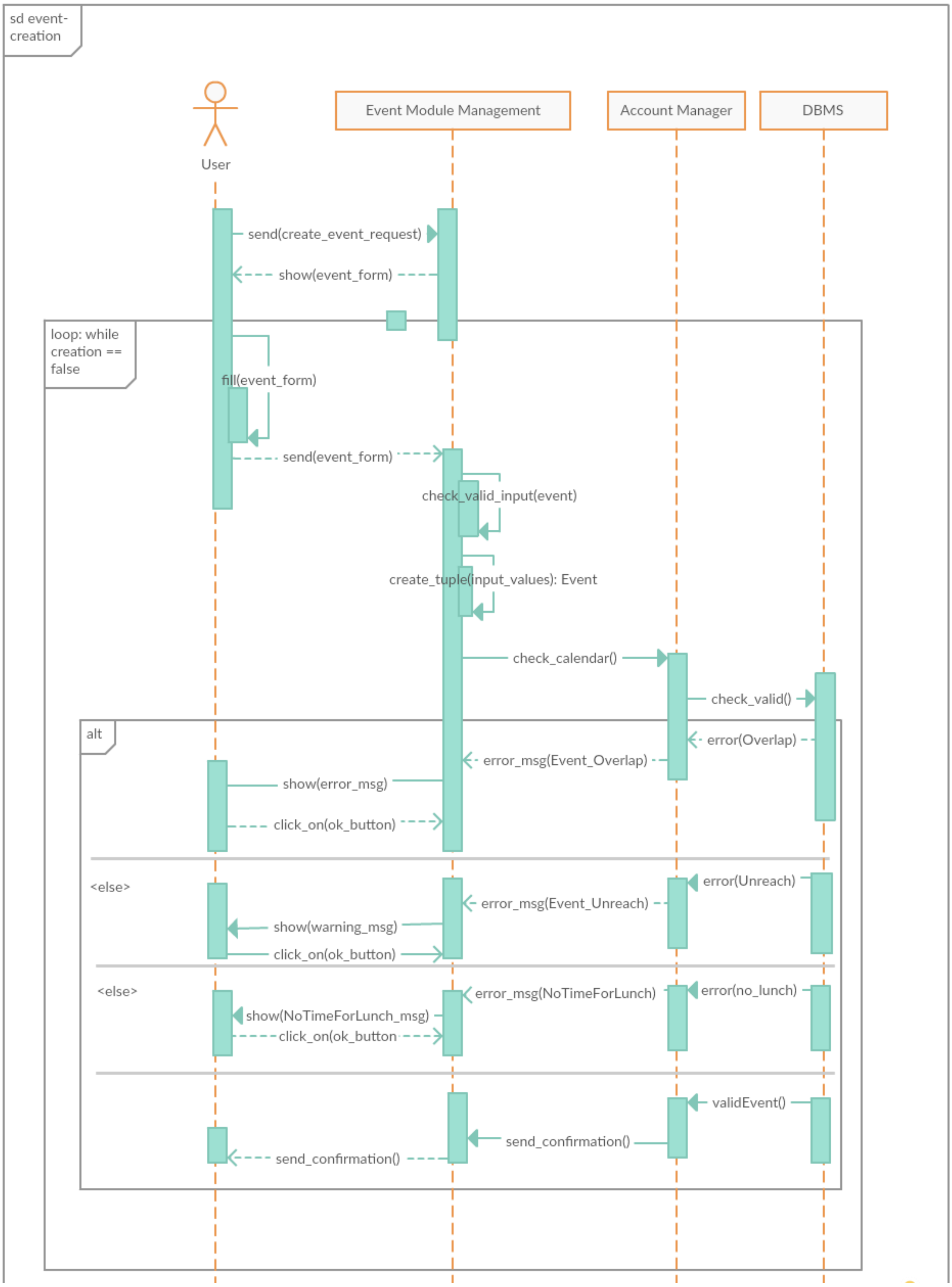


Figure 2.4.3 Event creation – sequence diagram

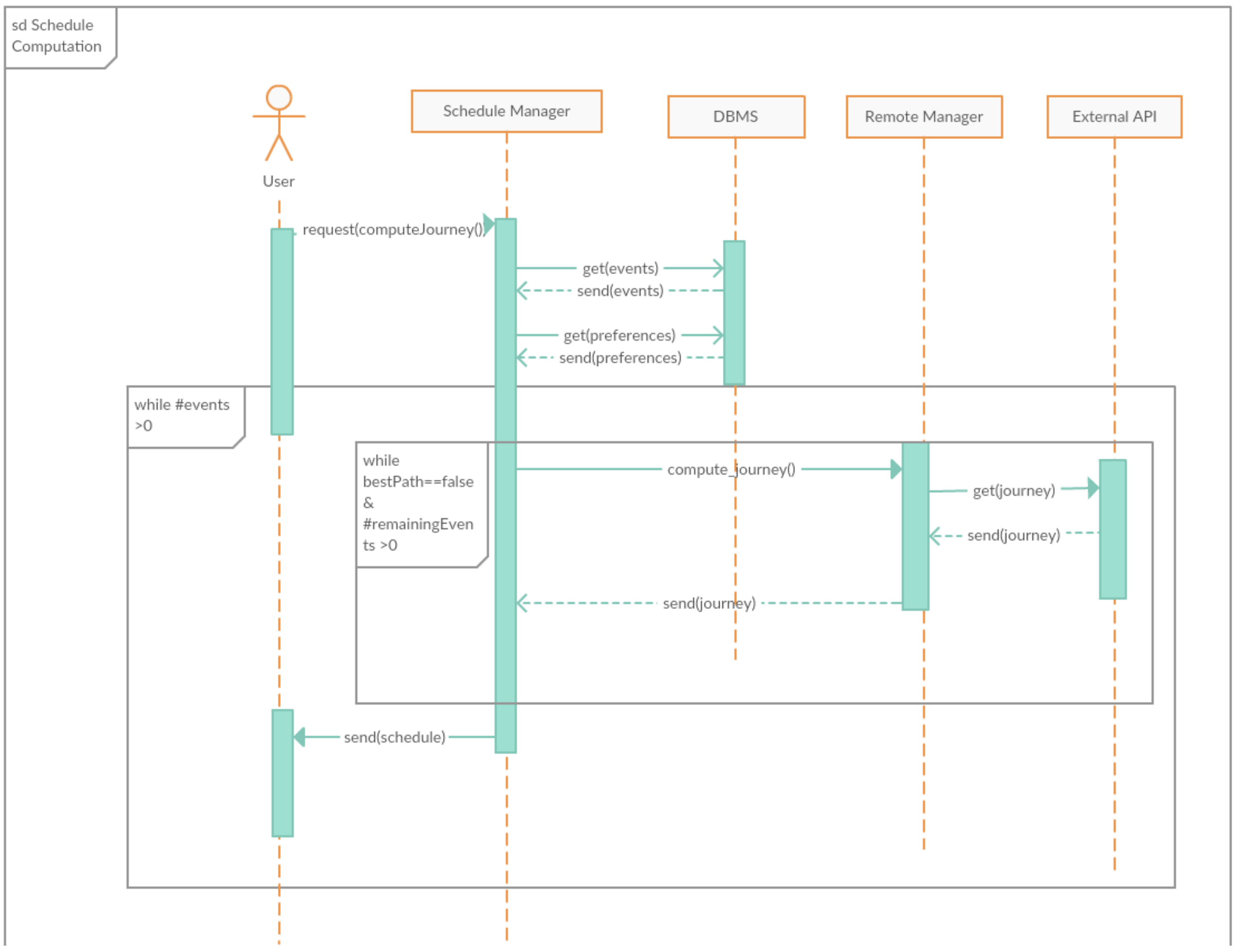


Figure 2.4.4 Schedule creation – sequence diagram

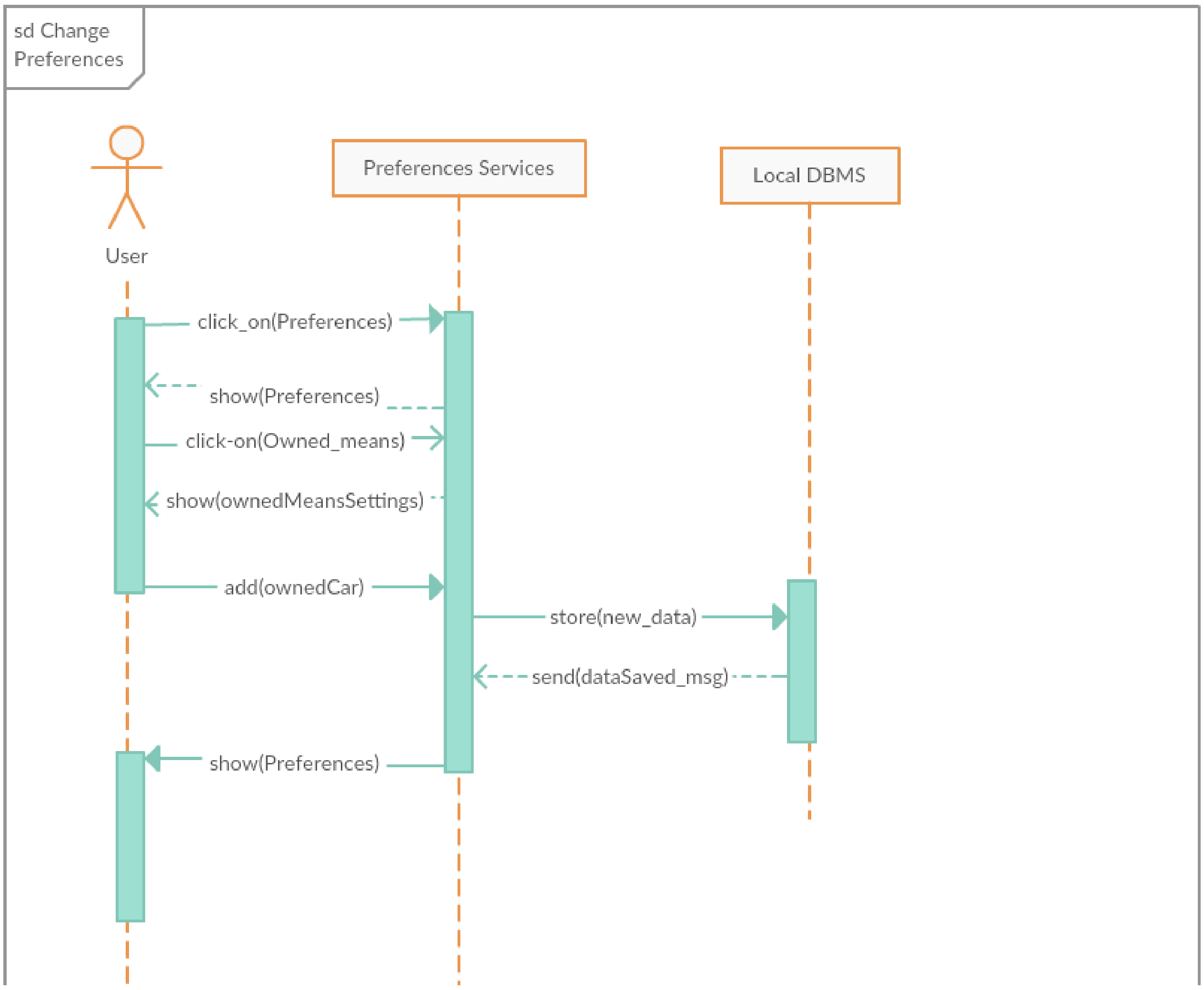


Figure 2.4.5 Edit Preferences – sequence diagram – an example

2.5 COMPONENT INTERFACES

This section includes UML component diagrams of the Client Mobile Services and the Server Mobile Services, shown below, respectively, in Figure 2.5.1 and Figure 2.5.2.

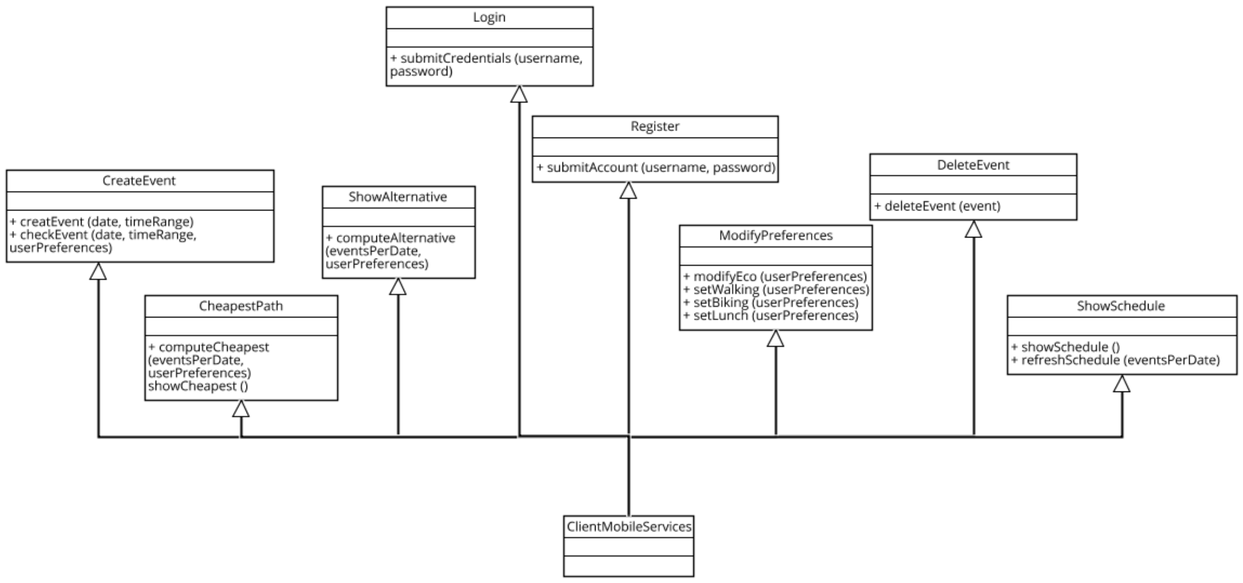

Login
+ submitCredentials (username, password)
Register
+ submitAccount (username, password)
CreateEvent
+ creatEvent (date, timeRange)
+ checkEvent (date, timeRange, userPreferences)
ShowAlternative
+ computeAlternative (eventsPerDate, userPreferences)
DeleteEvent
+ deleteEvent (event)
ModifyPreferences
+ modifyEco (userPreferences)
+ setWalking (userPreferences)
+ setBiking (userPreferences)
+ setLunch (userPreferences)
CheapestPath
+ computeCheapest (eventsPerDate, userPreferences)
showCheapest ()
ShowSchedule
+ showSchedule ()
+ refreshSchedule (eventsPerDate)
ClientMobileServices

Figure 2.5.1. Client Mobile Services – UML

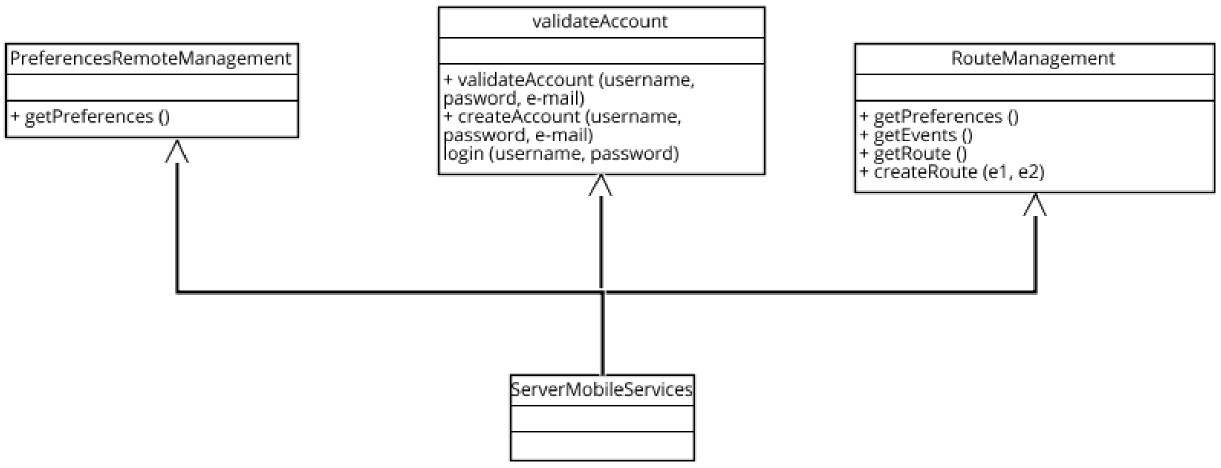

validateAccount
+ validateAccount (username, pasword, e-mail)
+ createAccount (username, password, e-mail)
login (username, password)
PreferencesRemoteManagement
+ getPreferences ()
RouteManagement
+ getPreferences ()
+ getEvents ()
+ getRoute ()
+ createRoute (e1, e2)
ServerMobileServices

Figure 2.5.2. Server Mobile Services - UML

2.6 SELECTED ARCHITECTURAL STYLES AND PATTERNS

2.6.1. Overall architecture

As shown in Figure 2.6.1 and already explained in previous sections, we plan to divide our project's architecture into 3 tiers:

• Thick Client
• Application Logic (server and local)
• Database (external and local).

We decided to use a *thick client* architecture to distribute the application logic between the server and the client.

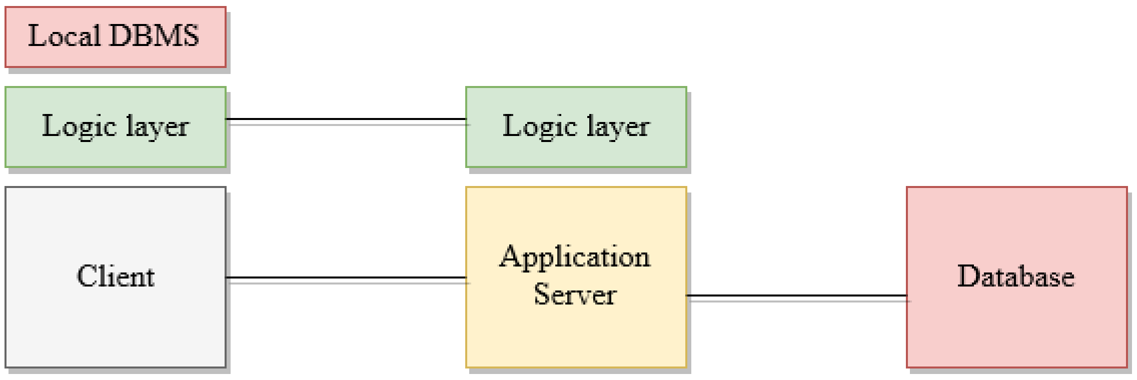


Figure 2.6.1. 3 tiers-architecture.

2.6.2. Design pattern and styles

The design pattern used in the project are the following:

• **Observer** pattern: an object has a list of observers that are notified whenever something changes in the state of the object. It is helpful in the implementation of event handling systems like our project.

• **Proxy** pattern**:** it acts like a placeholder object between different entities. It usually consists in some pointers pointing to complex objects: whenever an action is made on the proxy objects, it is sent forward to the complex object.

• **Visitor** pattern: this pattern is very useful because it provides a way to add new functions to existent objects without modifying their structures. It is mainly used where a service of messages between objects and operations is implemented.

• **MVC**: this is one of the most common software design pattern that divide the project into three connected parts: a Model, a View and a Controller. The model contains the main data of the system, the view represents the interface with the user and the Controller manages connections between them and reporting updates and events.

• **Decorator** pattern: this typical object-oriented design pattern is mostly used when there is the need to add different behavior or characteristics to an object. In our case, we use it to define meetings scheduled in the calendar. For example, we create the object event and then we add different qualities, like time, location, etc.

• **State** pattern: the system is represented as a state machine, where every state is a derived class from the state interface.

3. ALGORITHM DESIGN

This section includes the main algorithms used to implement the logic of the application. They are written in pseudo-code, for the easiness of the reader's understanding.
The first explained algorithm is the so called *3 Checks-Algorithm*. We use this to verify if a new created event can be inserted in the user's calendar.

These 3 checks consist in:
- the event cannot overlap with previously inserted events.
- the event must be reachable, otherwise a warning message will be sent.
- the event must respect the time slot kept for lunch break. If inserting the event in th e calendar, the slot for the lunch break won't be sufficient, according to user's preferences, then the event won't be saved.

```
3 Checks algo :

//create event
Create new EventBuilder eventBuilder(getDateFromUser(),getTimeRangeFromUser(), userPreferences);

//into event builder class constructor
public class EventBuilder{
        public constructor Event(date, timeRange){
                Create new EventChecker eventChecker(date,timeRange, userPreferences);
                checkResult == eventChecker.CHECKEVENT();

                if checkResult == "FirstCheckFailure" then
                        Create Warning firstTypeWarning("OverlappingProblem");
                elseIf chekResult == "SecondCheckFailure" then
                        Create Warning secondTypeWarning("ReachabilityProblem");
                elseIf check Result = "ThirdCheckFailure" then
                        Create Warning secondTypeWarning("LunchTimeProblem");
                elseIf checkResult == "CheckSuccessful" then
                        Create Event event(date, timeRange);
        }
//...
}

//into event checker class
public class EventChecker{
        public constructor eventChecker(date, timeRange, userPreferences){
                this.date = date;
                this.timeRange = timeRange;
                this.userPrefs = userPreferences
        }

        //metodo check event
        string CHECKEVENT(){
        boolean = FIRSTEVENTCHECK(getEventPerDate(date),timeRange);
        if boolean == false then return "FirstCheckFailure";
        else boolean = secondEventCheck(getEventPerDate(date),timeRange);
                if boolean == false then return "SecondCheckFailure";
                else boolean = thirdEventCheck(getEventPerDate(date),timeRange);
                        if boolean == false then return "ThirdCheckFailure";
                        else return "CheckSuccessful";
        }
        //into event checker functions
        bool firstEventChecker(eventPerDate, timeRange){
        forEach e(int) in  eventPerDate :
                if eventPerDate[e].getTimeRange().intersects(this.timeRange) == true then return false;
                else return true;
        }

        bool secondEventChecker(eventPerDate, timeRange){
                if #(eventPerDate) == 0 than return true;
                else boolean == CHECKREACHABILITY(eventPerDate, timeRange);
                        if boolean = true then return true;
                        else return false;
        }

        bool thirdEventChecker(eventPerDate, timeRange){
                lunchTimeRange = userPrefs.getPreferences().getPreferredLunchTimeRange();
                if LunchTimeRange.intersects(timeRange) == false then return true;
                else
                        lunchTimeWasted = lunchTimeRange.getIntesectionTimeRange(timeRange);
                        remainingLunchTime = date.getRemainingLunchTime();
                        if lunchTimeWasted > remainingLunchTime then return false;
                        if remainingLunchTime.existsUnique()<userPrefs.getPreferences().getLunchTimeRange() than return false;
                        if REACHABILITYTIMETEST(eventPerDate,lunchTimeWasted,remainingLunchTime,timeRange) than return true;
                                else return false;
        }
//...
}
```

The second algorithm explains how the journey computation is implemented. Journey computation is the main feature of the application: it calculates the best available path between the events in the user's calendar, always respecting his/her preferences.

```
Journey computation algo:

Create new Schedule schedule(getCurrentDate(), getUser());

//into schedule class
public class Schedule{
        public constructor Schedule(currentDate, user){

                Create new GoogleAPI googleApi;
                Create new WeatherAPI weatherApi;

                this.user = user;
                this.currentDate = currentDate;
                this.eventsPerDate = currentDate.getEvents();
                Create new Event startingPoint(currentDate, 0);
                this.currentEvent = startingPoint;
        }

        //...
        Bool createSchedule(){
                Create new HashMap schedule(Event, Route);
                forEach e(int) in eventPerDate:
                        schedule.put(eventPerDate[e], getRoute(eventPerDate[e]);
                        currentEvent = eventPerDate[e];
        }

        //...
        Route getRoute(chosenEvent){
                return createItinerary(currentEvent,chosenEvent,user.getPreferences());
        }

        //...
        Route createItinerary(event1, event2, preferences){
                if preferences.getEco() and weather.getWeather(currentDate) == "good"
                        and googleApi.computeRoute(event1,event2,"bike", user.getTravelMeans())<=user.getMaximumBikingTime
                        than return googleApi.computeRoute("ECO", user.getTravelMeans());
                elseIf currentDate.getStrike() != "NOSTRIKE" than
                        return googleApi.computeRoute(currentDate.getStrike(), user.getTravelMeans());
                else return googleApi.computeRoute("FASTEST", user.getTravelMeans());

        }
}
```

4. USER INTERFACE DESIGN

In the RASD, we have already shown the main characteristics of the interfaces of the project. In the figures below, we provide an updated and improved version of them.

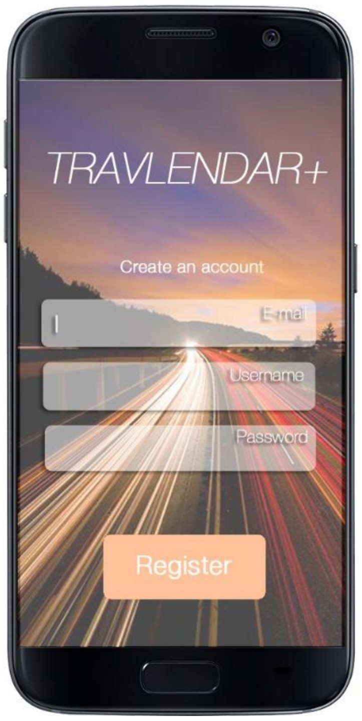


Figure 4.1 Registration

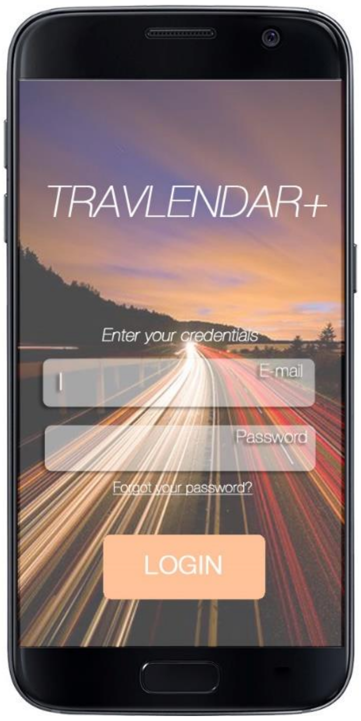


Figure 4.2 Login

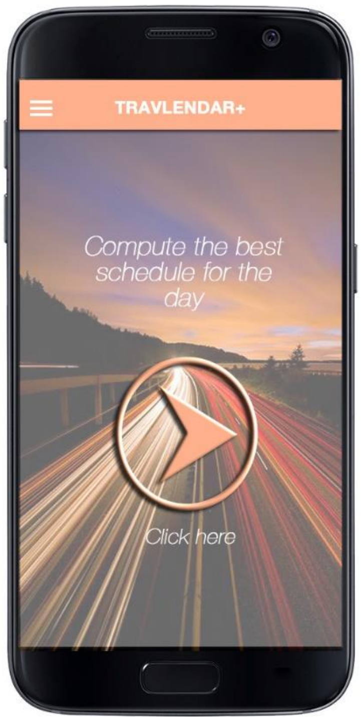


Figure 4.3 Home screen

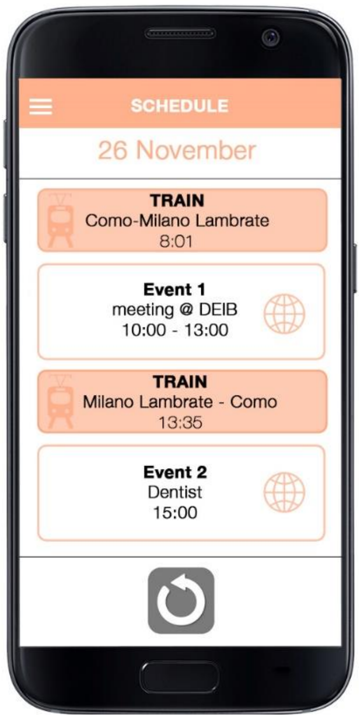


Figure 4.4 Schedule

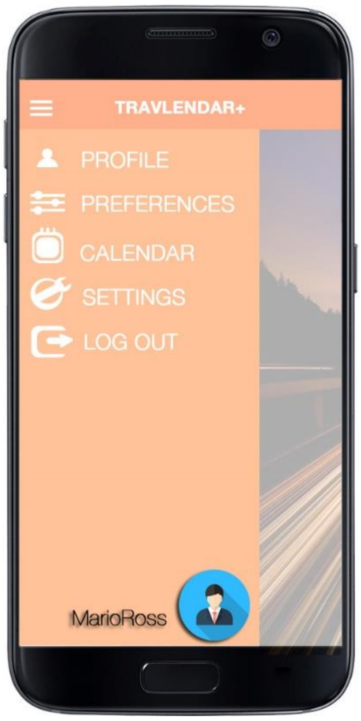
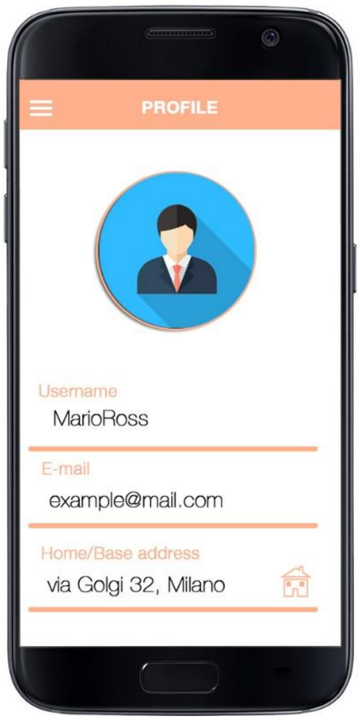


Figure 4.5 Options

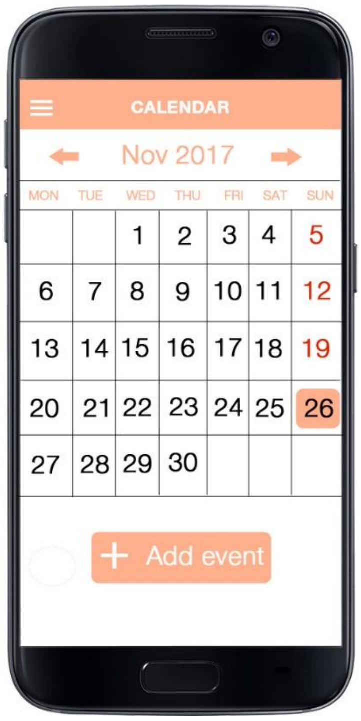


Username
MarioRoss

E-mail
example@mail.com

Home/Base address
via Golgi 32, Milano

Figure 4.6 Profile



MON	TUE	WED	THU	FRI	SAT	SUN
		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			

Figure 4.7 Calendar

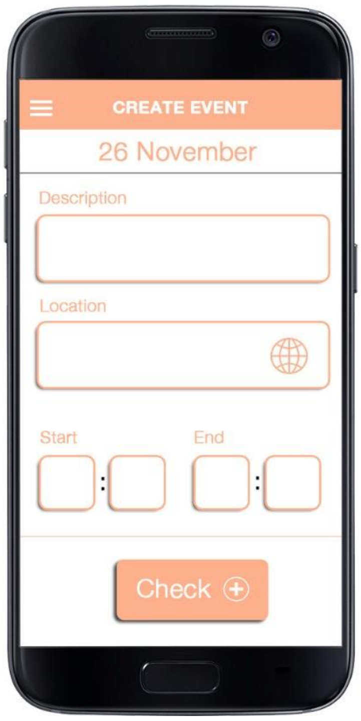
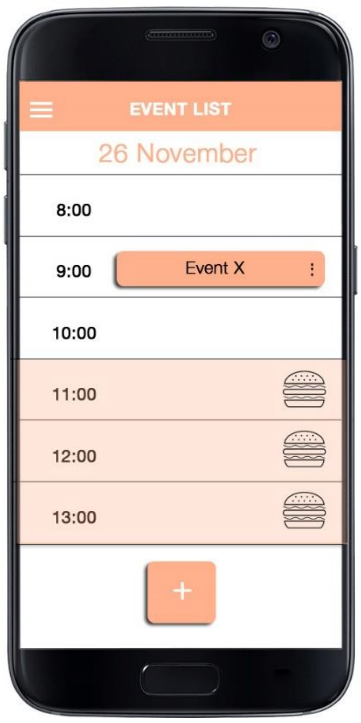


Figure 4.8 Event creation


EVENT LIST

26 November

8:00

9:00 Event X

10:00

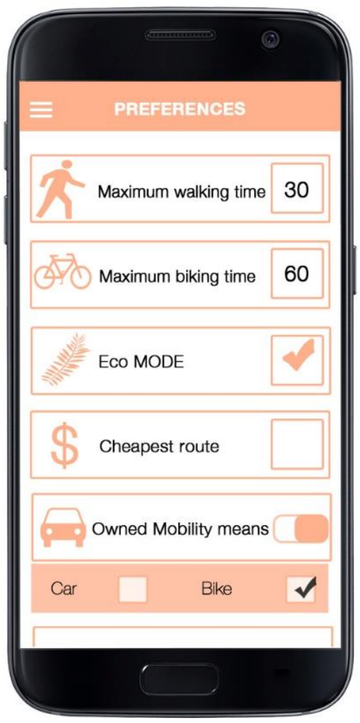
11:00

12:00

13:00

+

Figure 4.9 Event list


PREFERENCES

Maximum walking time 30

Maximum biking time 60

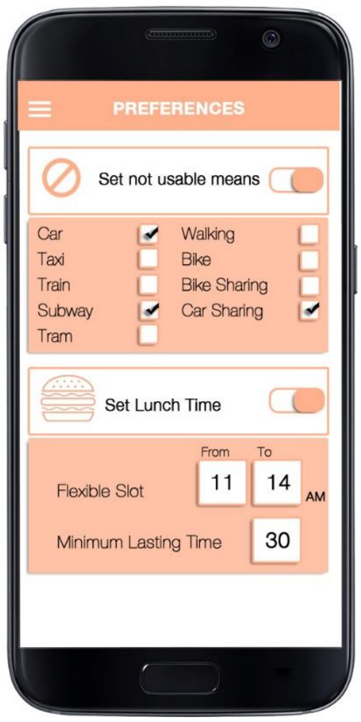
Eco MODE ✓

Cheapest route ☐

Owned Mobility means

Car ☐ Bike ✓

Figure 4.10 Preferences - A


PREFERENCES

Set not usable means

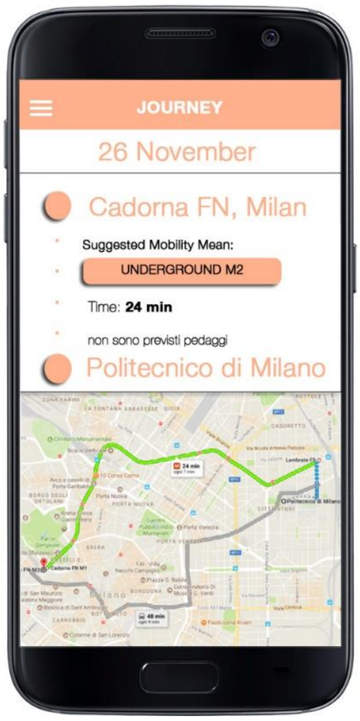
Car	✓	Walking	☐
Taxi	☐	Bike	☐
Train	☐	Bike Sharing	☐
Subway	✓	Car Sharing	✓
Tram	☐		

Set Lunch Time

Flexible Slot From 11 To 14 AM

Minimum Lasting Time 30

Figure 4.11 Preferences - B


JOURNEY

26 November

Cadorna FN, Milan

Suggested Mobility Mean:

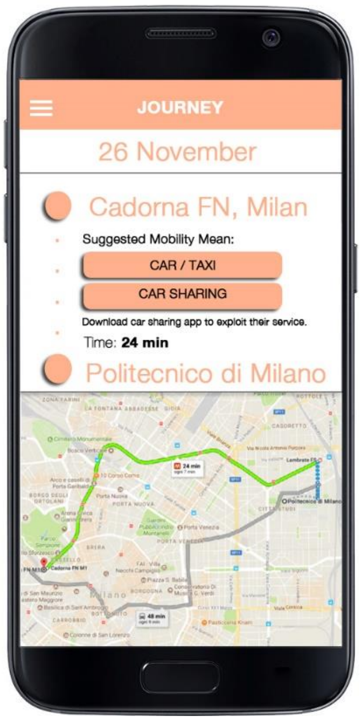
UNDERGROUND M2

Time: **24 min**

non sono previsti pedaggi

Politecnico di Milano

Figure 4.12 Journey - A


JOURNEY

26 November

Cadorna FN, Milan

Suggested Mobility Mean:

Download car sharing app to exploit their service.

Time: **24 min**

Politecnico di Milano

Figure 4.13 Journey - B

5. REQUIREMENTS TRACEABILITY

We designed the software architecture in order to respect the requirements and the goals previously expressed in the RASD. For every goal, we provide all the components of the application that are used to reach them and the respective requirements.

• *[G1] Registration.*

Requirements: [R1] [R2]

- Account Manager
- Server DBMS
- Remote Server Components

• *[G2] Login.*

Requirements: [R3] [R4]

- Account Manager
- Server DBMS
- Remote Server Components

• *[G3] Allow an User to create/delete meetings and send warning messages if needed.*

Requirements: [R4] [R5] [R6] [R7]

- Account Manager
- Local DBMS
- Remote Server Components
- Event Module Management

• *[G4] Allow an User to notify a public transport strike.*

Requirements: [R4] [R8]

- Account Manager
- Local DBMS
- Remote Server Component
- Strike Notification Module

• *[G5] Allow an User to insert personal preferences to modify the calculation of the best path.*

Requirements: [R4] [R9]

- Account Manager
- Local DBMS
- Remote Server Components
- Preferences service

- Walking Time Setter

- Biking Time Setter

- Lunch Setter

• *[G6] Allow an User to choose to minimize carbon footprint of the journey.*

Requirements: [R4] [R10]

- Account Manager

- Local DBMS

- Remote Server Components

- Preferences Service

- Eco Setter

• *[G7] Allow an User to insert in the Preferences his/her owned mobility means.*

Requirements: [R4]

- Account Manager

- Local DBMS

- Remote Server Components

- Preferences Service

• *[G8] Allow an User to choose the least expensive path in the computation of the best path.*

Requirements: [R4] [R11]

- Account Manager

- Local DBMS

- Remote Server Components

- Preferences Service

• *[G9] Allow an User to acknowledge the best path to follow to reach the daily meetings.*

Requirements: [R4] [R12]

- Account Manager

- Local DBMS

- Remote Server Components

- Schedule Manager

- Route Management

- Server DBMS

- Maps API

- Weather API

- *[G10] Allow the application to notify a User if there are changes on the pre-defined route.*

  Requirements: [R4] [R12] [R13]

  - Server DBMS

  - Remote Server Components

  - Schedule Manager

  - Route Management

  - Maps API

  - Weather API

- *[G11] Allow the application to keep a slot of time for lunch break, if requested.*

  Requirements: [R4] [R14]

  - Account Manager

  - Preferences Remote Management

  - Server DBMS

  - Schedule Manager

  - Lunch Setter

6. IMPLEMENTATION, INTEGRATION AND TEST PLAN

6.1 Overall description

The purpose of this last section is to explain how we plan to implement the project, in which order the components of the system will be realized, how we plan to integrate all the subcomponents and how we plan to test them.

The reader will find a sequence of diagrams representing the relations and integration between the components and subcomponents belonging to the system and finally, the description of the planned testing activities for each integration subsystem.

As a main assumption, we must guarantee that low level modules, like external APIs - Maps and Weather Forecast, are available and used properly according to the legal conditions of the providers and only after having subscribed proper usage plans with them.

6.2 Order of implementation

The overall idea is to implement the components that not depend on other components, at the beginning, in order to make the process more linear and with less probability of making mistakes.

A description of the system's components and subcomponents order of implementation will follow – an arrow going from a component A to a component B means that the component A must be implemented before B.

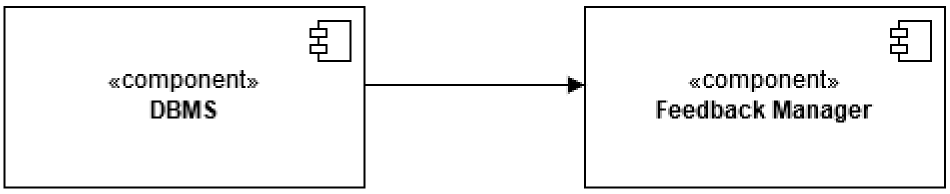


Figure 6.2.1 Feedback manager

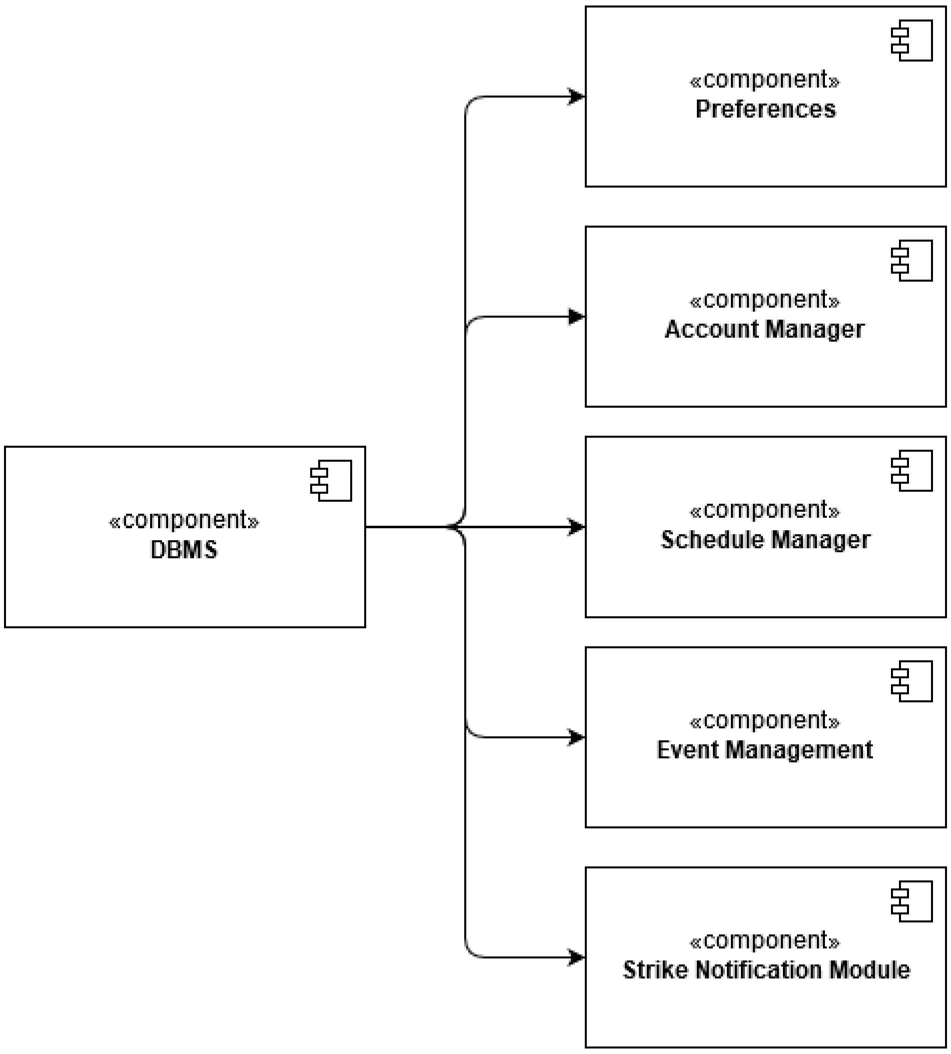


Figure 6.2.2 User's options

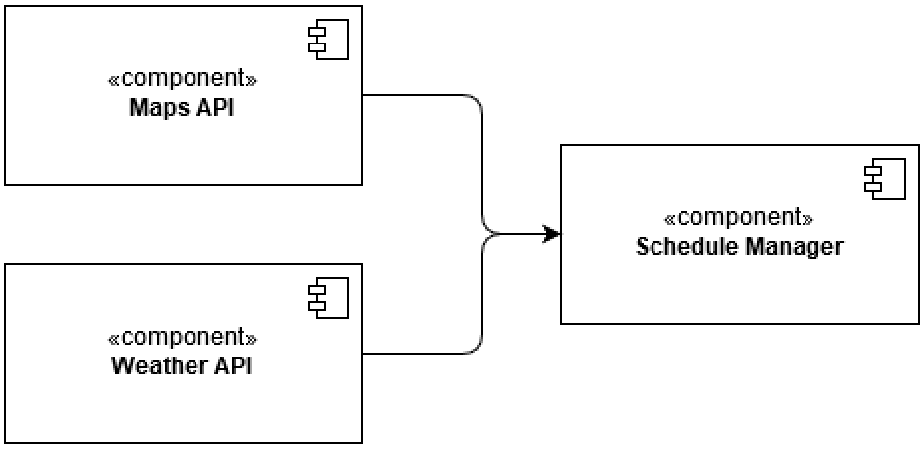


Figure 6.2.3 Schedule manager

6.3 Components Integration and Testing

As far as the integration of components is concerned, we can mainly refer to Figure 2.2.2 and Figure 2.2.3 of the present document, provided in the *Component View* section.

Respectively, we will refer to the integration represented in Figure 2.2.2 as **I1 – Customer Service** and to Figure 2.2.3 as **I2 – Schedule manager**.

The following test cases refer to Customer Service.

TEST ID	I1 T1
Components	Mobile Application, User Mobile Application, Account Manager, Server DBMS
Input specification	Account registration and login
Output specification	User registered or logged in
Description	This test aims to control if the registration and login processes are correctly performed. This means, it checks if all the fields of the registration and login forms respect the application security rules and that, as far as the login process is concerned, once filled the form, the user actually has an already registered account.
Environmental needs	-

TEST ID	I1 T2
Components	User Mobile Application, Event Module Management, Local DBMS, Remote Server Components
Input specification	Insertion/deletion of a new event
Output specification	Event correctly inserted in the user's calendar or rejected by the system.
Description	This test aims to control the insertion of a new event by the user. It checks that all the fields of the event form are correctly filled, that the new event doesn't overlap with previously inserted events or that it is not reachable. In this latter case, the test has to check that a warning message is sent.
Environmental needs	-

TEST ID	I1 T3
Components	User Mobile Application, Preferences Services, Local DBMS, Walking Time Modifier, Biking Time Modifier, Lunch Setter, Eco Setter, etc.
Input specification	User changes his/her personal preferences
Output specification	New preferences are set and saved in the local DBMS
Description	This test aims to check if the preferences are correctly set and changed and that they are then saved in the local DBMS
Environmental needs	-

The following test refers to the *Schedule Manager*.

TEST ID	I2 T1
Components	Mobile Application, Schedule Management, Server DBMS, Maps API, Weather API, Route Management
Input specification	Compute and display schedule for the day
Output specification	Show schedule for the current day
Description	This test aims to check if the computed schedule for the day respects all the requirements. The schedule must respect the personal preferences of the user and by the best available option.
Environmental needs	I1 T2

7. EFFORT SPENT

The total amount of hours spent working on the project:

Sonia Greco	Date	Time
	7.11.2017	5 h
	11.11.2017	2 h
	14.11.2017	3 h
	19.11.2017	4 h
	20.11.2017	3 h
	21.11.2017	4 h
	22.11.2017	3 h
	23.11.2017	4 h
	24.11.2017	4 h
Francesco Guzzo	Date	Time
	7.11.2017	5 h
	11.11.2017	2 h
	14.11.2017	3 h
	19.11.2017	4 h
	20.11.2017	3 h
	21.11.2017	4 h
	22.11.2017	3 h
	23.11.2017	4 h

8. REFERENCES

Documents:

• Travlendar+ RASD

Software and tools used in the development of the DD and the RASD:

• Creately – *online diagram editor* [https://creately.com/]

• Adobe Photoshop [http://www.adobe.com/it/products/photoshop/]

• Draw.io [https://www.draw.io]