
Politecnico di Milano

A.Y. 2017/2018

Software Engineering 2: ***Travlendar+***

# Design Document

Matteo Biasielli - Emilio Capo - Mattia Di Fatta

v. 1.0

# Table of Contents

# 1. Introduction

## 1.1 Document purpose

This document has to be intended as a general guide for the correct development of the Travlendar+ application. The content of this document follows and is based on the content of the RASD document. This document is meant to be a reference for any person who has an interest in the project. This includes, but is not limited to, development team members, stakeholders and end users.

## 1.2 Definitions, Acronyms, Abbreviations

### 1.2.1 Definitions

- **User**: actor that is using the application and may want to access all functionalities.
- **Application**: with the term application we are talking about the desktop version, the website and mobile version of the Travlendar+ system.
- **Scheduling**: action performed by a user that is adding a new activity to his personal calendar.
- **Flexible Activity**: An activity with starting and ending time larger than the duration.
- **Fixed Activity**: An activity with fixed starting and ending time.

### 1.2.2 Acronyms

- **RASD:** Requirements Analysis and Specification Document
- **DD:** Design Document
- **UI:** User Interface
- **API:** Application programming interface
- **UXD**: User Experience Diagram
- **UML**: Unified Modeling Language
- **GPS**: Global Positioning System
- **DECS**: Dynamic Event Check System

### 1.2.3 Abbreviations

- **[Gn]:** the n-th goal
- **[Rn]:** the n-th requirement
- **[NFRn]:** the n-th non-functional requirement
- **[An]:** the n-th assumption
- **[Cn]:** the n-th constraint
- **[UIn**]: the n-th user interface example

## 1.3 Reference Documents

- Mandatory project assignments for the A.Y. 2017/2018 available on the beep's page of the Software Engineering 2 course.
- Projects examples and other documents available on the beep's page of the Software Engineering 2 course.
- RASD Document available on the Delivery Folder on the repository https://github.com/MatteoBiasielli/BiasielliCapoDifatta .

## 1.4 Document Structure

- **Introduction:** This is the very first part of the document.
  In this section it's possible to retrieve general information about the Design Document. The purpose and intended audience of the document are specified here.
  In addition, Acronyms, Definitions and Abbreviations are defined in this section in order to make it easier, more concise and clearer to read the rest of the Design Document.
- **Architectural Design:** This part represents the second chapter of the document. Here the reader can find the architecture of the system components at various levels and contexts.
  First of all, a high-level overview of the components and the way they're connected is provided in this section. Following this, some components will be analysed in detail and their internal architectures will be showed for a matter of clarity.
- **Algorithm Design:** The most important algorithms that will be implemented in our application are described here, both with natural language and with java code/pseudocode.
- **User Interface Design:** Some User Interface samples have already been provided in the RASD document but they'll be extended and some will be added in this section of the Design Document.
  In addition, further explanation about the already existing UI will be added here, together with a detailed mapping of the User Interfaces into functional requirements and non-functional requirements.
- **Requirements Traceability:** Design choices are mapped into functional and non-functional requirements here.
- **Implementation, integration and test plan:** In this section you can find a brief description of the order in which we will implement and test components of our system.

## 1.5 Revision History

- **v. 0.1 [28 Oct 2017]**: added the whole “Introduction” section.
- **v. 0.2 [06 Nov 2017]:** Added “Algorithm Design” and “User Interface Design” sections.
- **v. 0.3 [07 Nov 2017]:** Added part of Architectural design
- **v. 0.4 [17 Nov 2017]:** Completed all parts
- **v. 1.0 [22 Nov 2017]:** Finalized DD

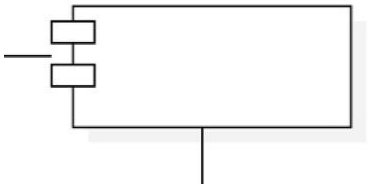
# 2. Architectural Design

In this section, we first provide a general overview on Travlendar+ system by means of a general Component Diagram. Then it is given a Component View on some important components again by means of Component Diagrams.
Moreover, a Deployment View and a Runtime View of the system is provided in order to describe nodes with their components, protocol for their interaction and how they are expected to work (by means of Sequence Diagrams).
Eventually, we describe in detail all components' interfaces that can be found in the general Component Diagram.

## 2.1 Convention

In the following section, conventions to design the component diagrams are enlisted.

### *Component*



Each component in the following diagrams are represented by means of the shape above. Notice that the two arcs in the sample represent relations with other components. Inside the shape the name of the component is specified and nothing else.

### *Interface and relations*

With the circle shape above we intend to represent an interface between two or more components (see next convention). Notice that the interface is provided by the component(s) on the right and exploited by the component(s)

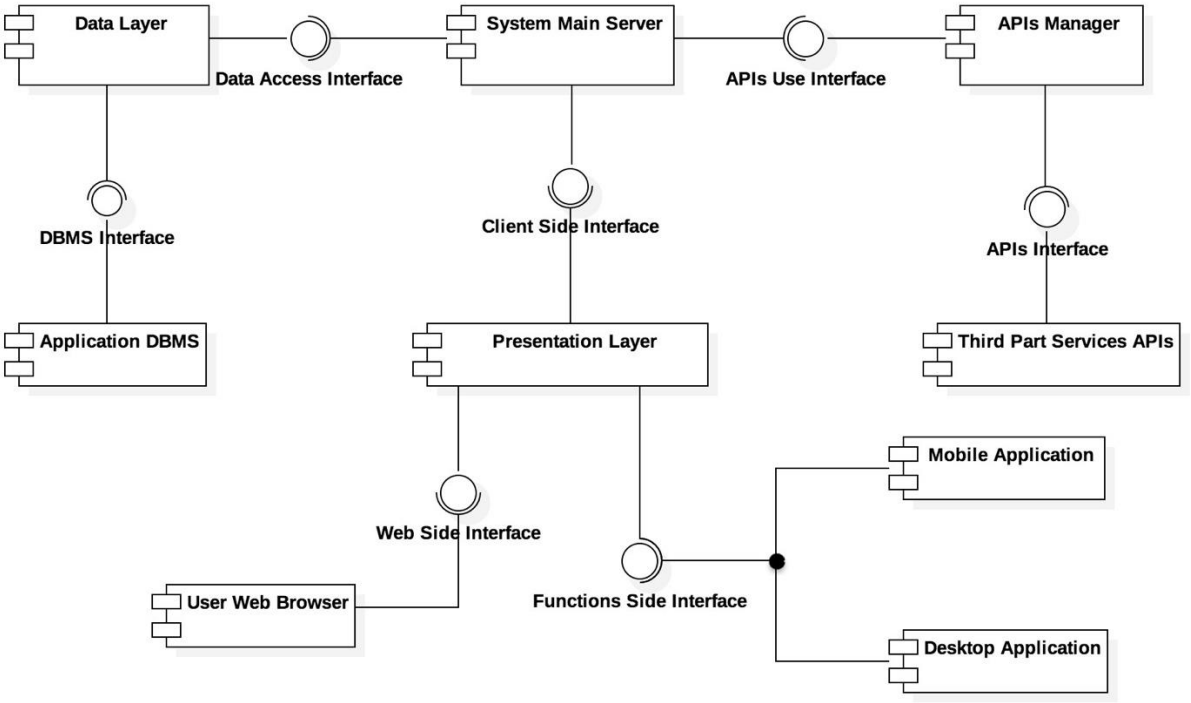
on the left by means of a relation arc. Below the shape a name is specified for the interface.

### *Join relations*

With the conventions above we intend to represent two (or more) components that exploit, on the left, (or provide, on the right) the same interface. For sake of clarity, in the diagrams join arcs are marked with a dot.

## 2.2 System Overview

The following diagram describes the whole Travlendar+ system with all its components (i.e. software modules).



### *Data Layer*

This component deals directly with system's DBMSs allowing other components to write, read and update data in the DBs properly (i.e. in a secure, consistent way) using the proper DBMS. It provides a single interface outward in order to systematize access to data stored in DBs and hide the internal complexity and implementation of DBMSs and DBs.

### *Application DBMSs*

This particular component is used as generalization of the two DBMSs used in Travlendar+: one to manage users' personal data and calendar, one to manage all relevant internal information used by Travlendar's server to works properly (for further information see Component View section).
It provides an appropriate interface to allow the Data Layer component, and only it, to query DBMSs.

### *System Main Server*

This is the core component of the server side of the system and of the application in general. It's composed by three essential components: the Computation Unit, the User Side Unit and the Dynamic Event Check System. It manages, with the aid of the APIs Manager and the Data Layer, all the Travlendar's functionalities described in the RASD document, except for the presentation side, managed entirely by the Presentation Layer (see below for further information).

### *APIs Manager*

This component is used to homogenize the different kinds of APIs provided by third part services, in other words it adapts the external APIs to the system in order to make them easily usable. By doing this, the APIs Manager tries also to optimize the access and the usage to the APIs by means of an internal optimizer component. It provided the System Main Server with the APIs Use Interface, used to exploit third part services while masking their implementation.

## *Third Part Services APIs*

This component is just a collection of all needed APIs with their internal representation, tools and external references. It represents external entities w.r.t. our system, useful in this kind of diagrams just to model their interaction with the system.

## *Presentation Layer*

The Presentation Layer component is the one appointed to show the front-end of the Travlendar+ system. Making use of the Client Side Interface provided by the System Main Server, it grants access to system's data to users w.r.t. confidentiality (i.e. only authorized users/clients will have access to data they're allowed to access and that are made accessible) by means of a web browser and the mobile and desktop application.
It provides two different interfaces, the Web Side Interface and the Functions Side Interface, as the user can interact in two ways with the system: through the website where he can download the application, find useful information (e.g. FAQs) and additional documentation, or through the desktop/mobile application by means of which he can exploit Travlendar+ functionalities.

## *Mobile Application*

The Mobile Application component represents an abstraction of the mobile version of Travlendar+, used in this diagram to show its interaction with the whole system. It makes use of the Functions Side Interface to connect to the system.
The core function of this component is to keep track of the itineraries contained in the travel options provided by the Computation Unit in the System Main Server in order to send notification to the user and notify him of possible/critical changes in their travel options.

### *Desktop Application*

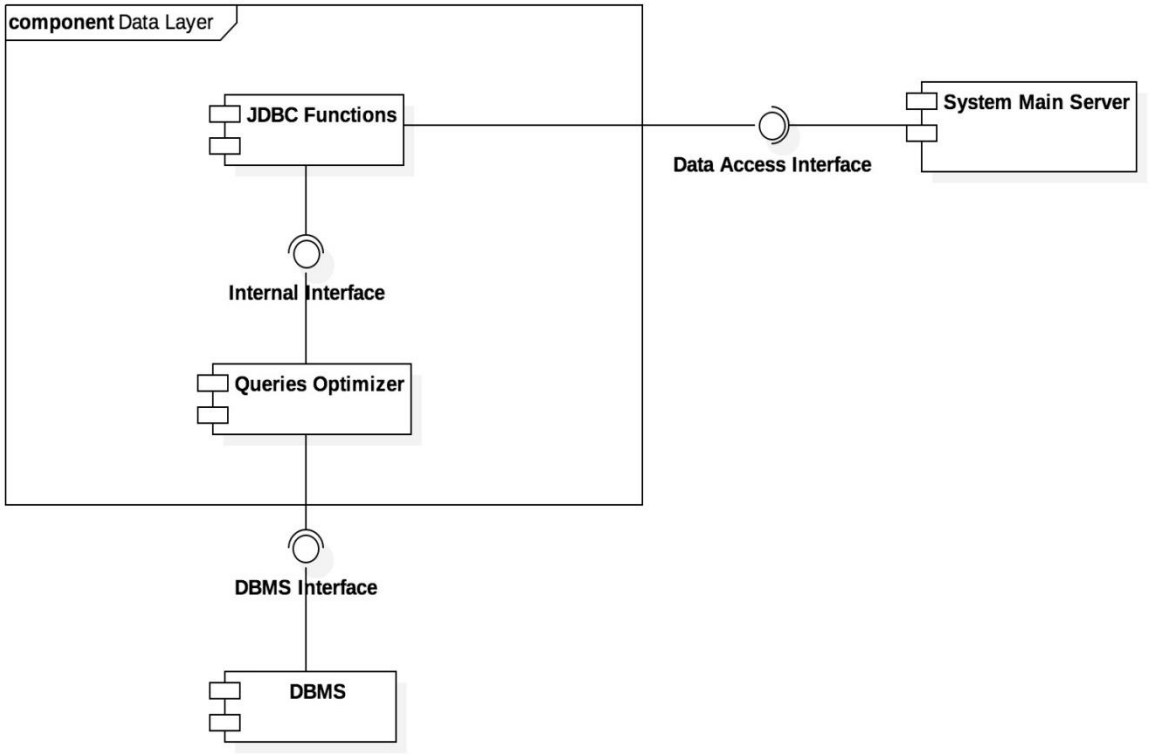
The Desktop Application component represents an abstraction of the desktop version of Travlendar+, used in this diagram to show its interaction with the whole system. It makes use of the Functions Side Interface to connect to the system.
The core function of this component is to keep track of the itineraries contained in the travel options provided by the Computation Unit in the System Main Server in order to send notification to the user and notify him of possible/critical changes in their travel options.

### *User Web Browser*

This component models the web browser used by the user to access the website. It's an external object with respect to our system, but its corresponding component is necessary in order to model its interactions with the website.

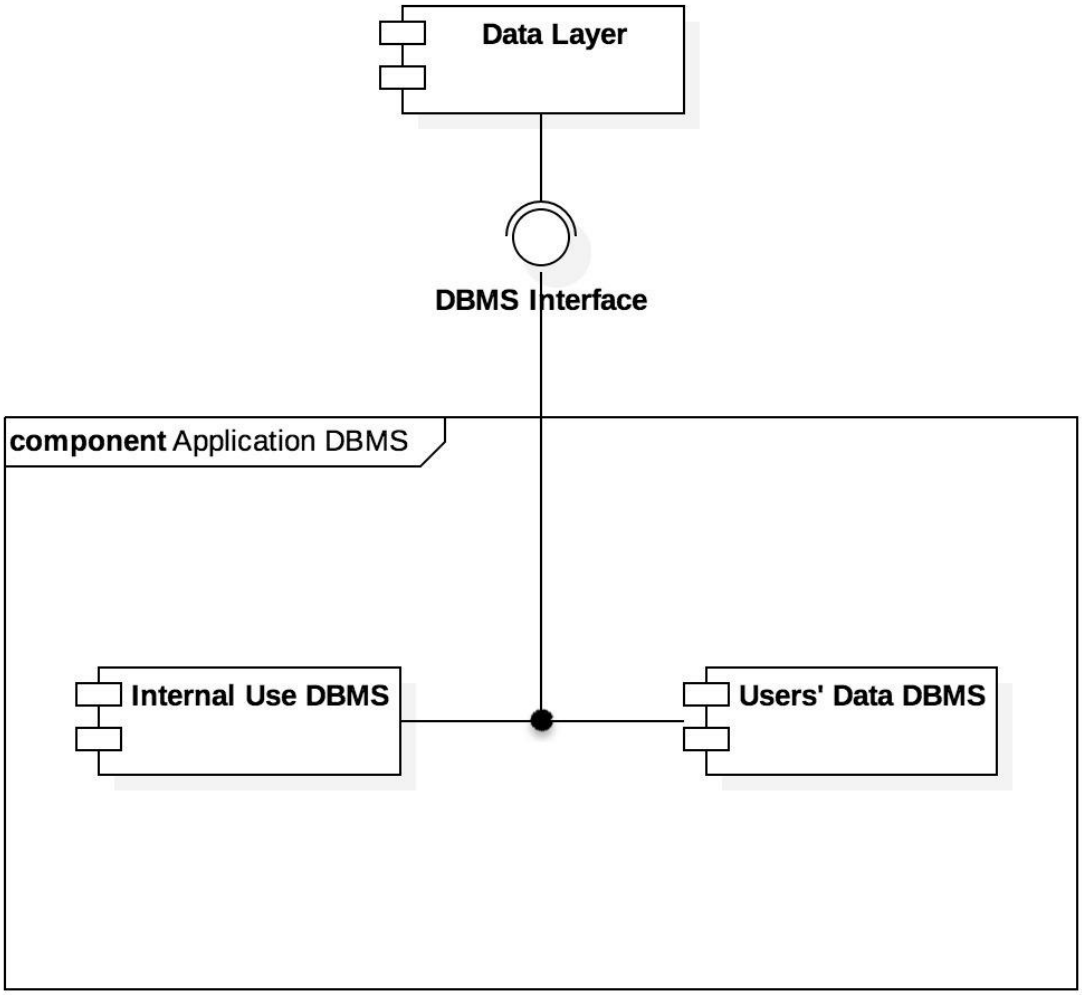
## *2.3* Component View

### *Data Layer*



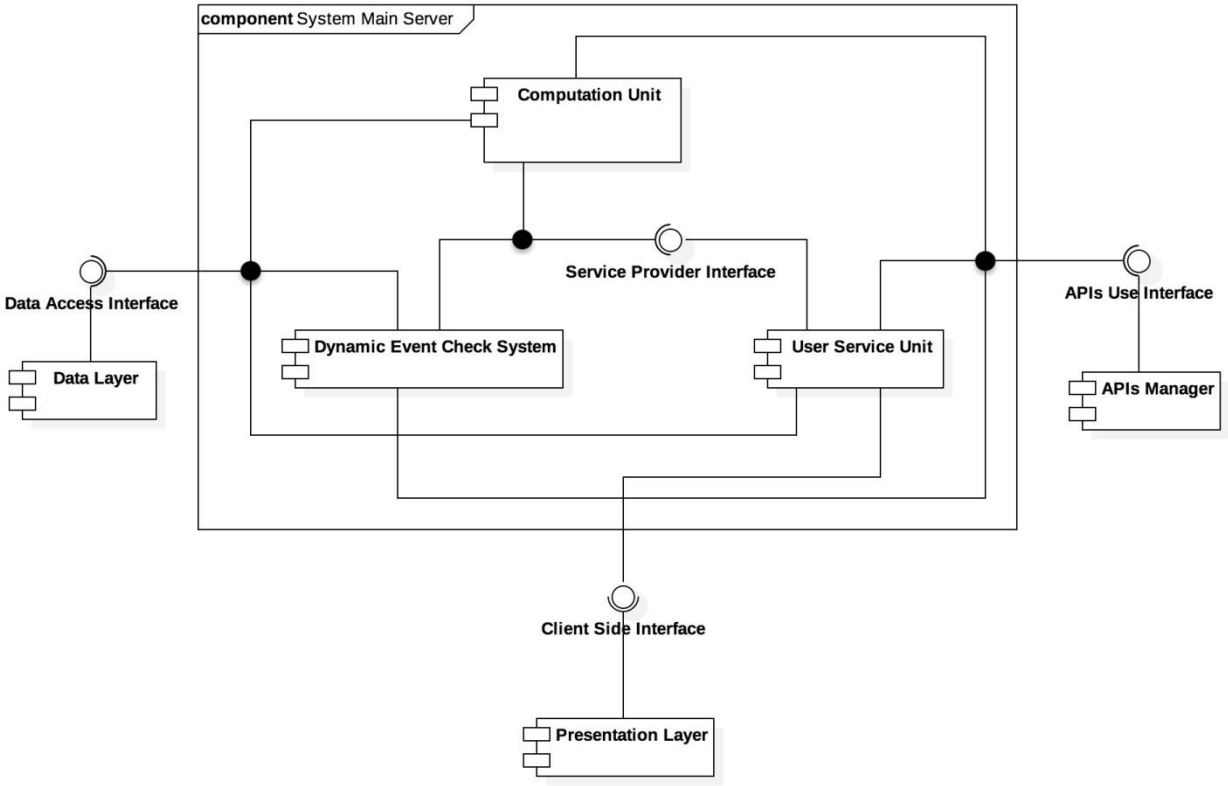

The Data Layer module is composed by two sub-components: the JDBC Functions Component and the Queries Optimizer Component. The former is a collection of tools and references used to integrate and use the JDBC library (Java library for DBs handling). The latter is used to optimize the formulation of queries by means of JDBC tools.

## *Application Database Management Systems*



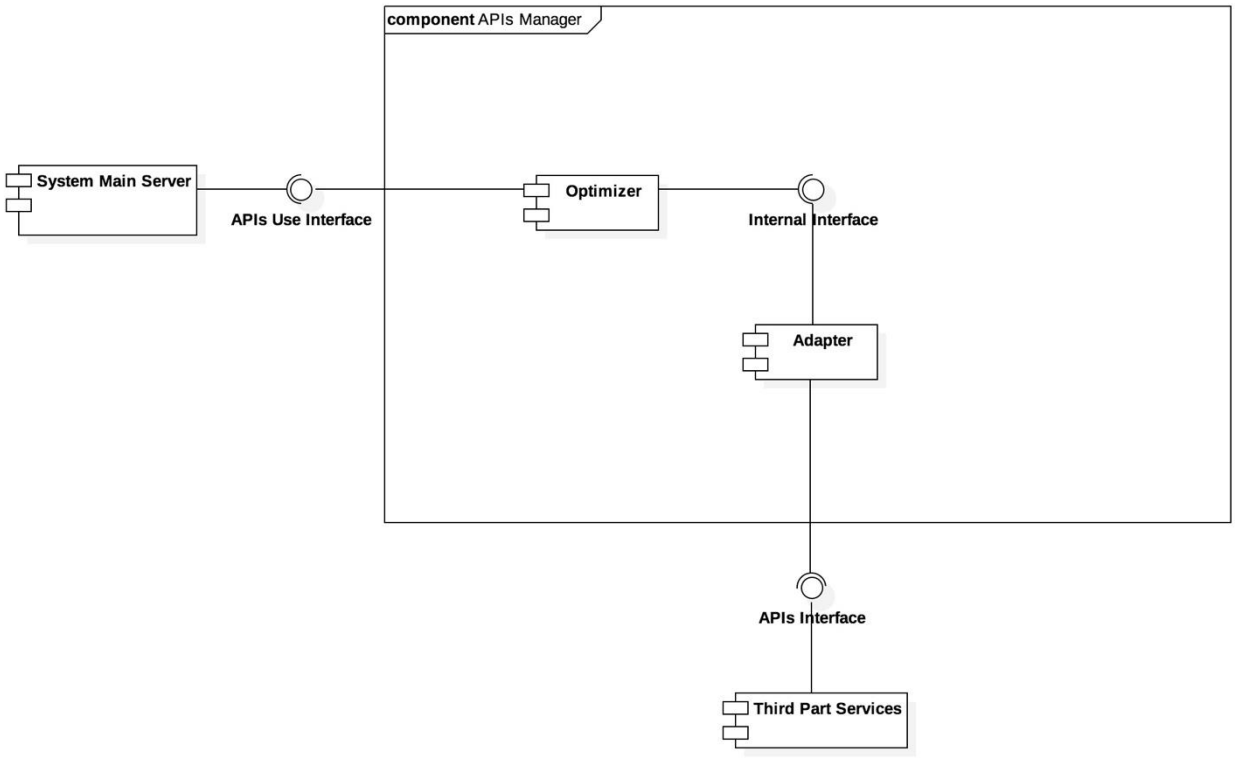

The Application DBMSs, as mentioned in the previous section, is a generalization of the two DBMSs used in the Travlendar+ system.
The Internal Use DBMS component models the DBMS that deals with the DB in which the system stores all the data needed to make the Travlendar+ application and server works properly.
The Users' Data DBMS component models the DBMS used query the DB in which the system stores users' credentials, calendars and sensitive data. All these data are encrypted.

## *System Main Server*



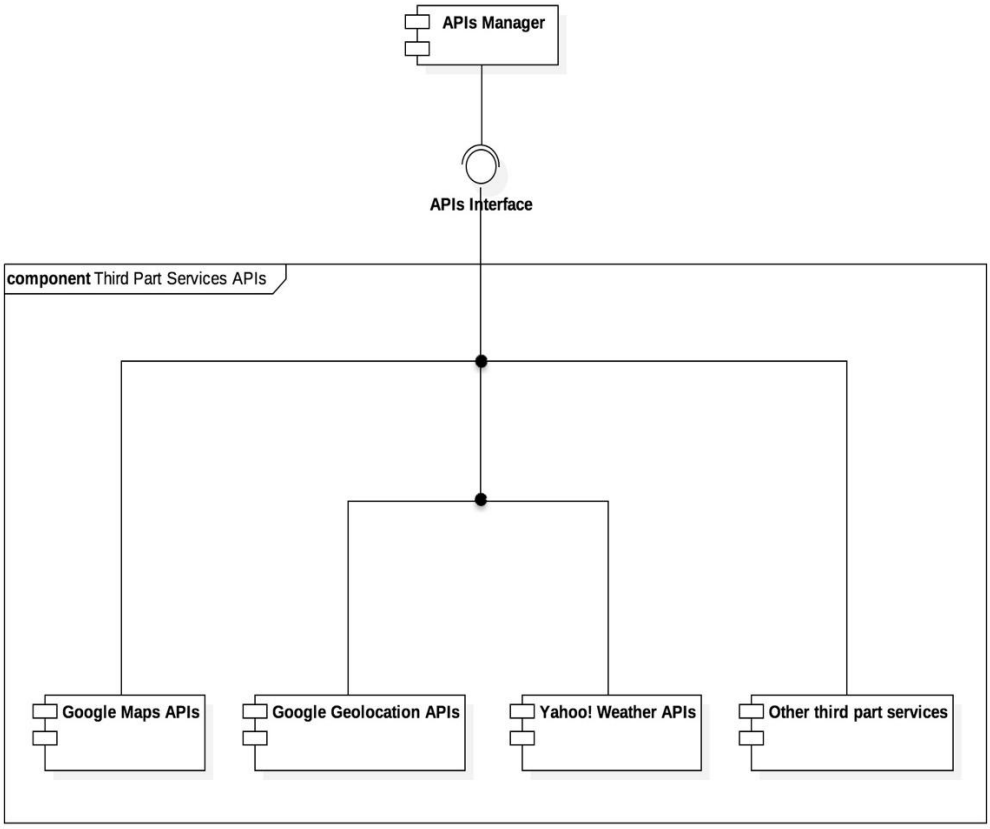

The System Main Server is the core component of our server. It acts as "brain" for Travlendar+ since it must compute the best travel option when needed for each user (by means of the its Computation Unit), check dynamically (i.e. when events happen it's listening) for events which can change travel options for some users and reports these to the local application(by means of the Dynamic Event Check System) and provide all these data (and some more) to the user in a proper way (by means of the User Service Unit).
Since this component has to be scalable and high-performing we could decide to distribute this component over several physical machines in case of necessity.
The communication between its component is allowed by the Service Provider Interface, an internal interface provided by the User Service Unit to transmit data outwards.
All the three internal components makes use of the Data Access Interface through which they can query DBs properly and of the APIs Use Interface through which they can exploits third part services.

## *APIs Manager*



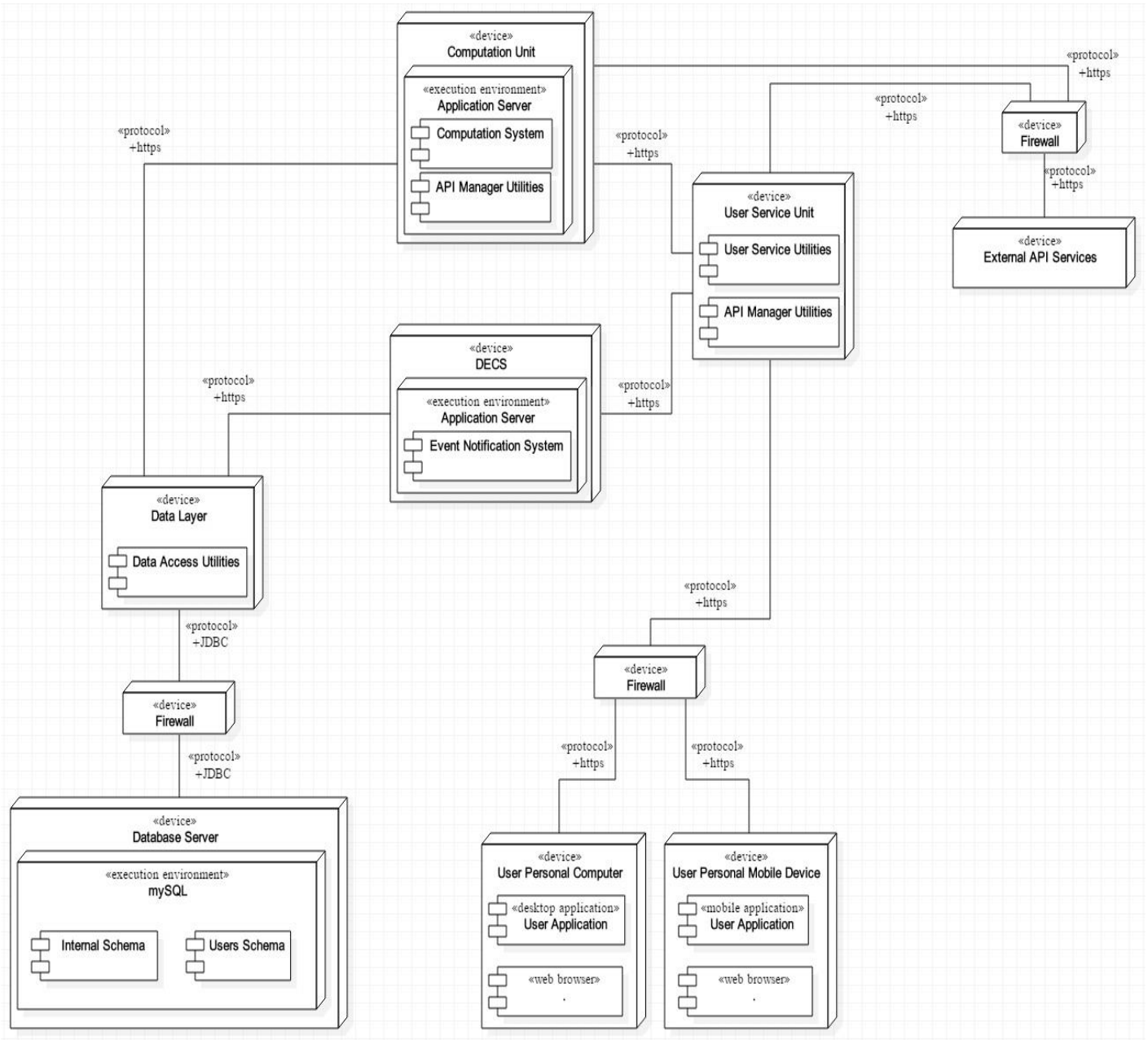

The APIs Manager is internally composed by two components and an interface to let them communicate.
The Adapter component is used to make homogenous all different kinds of APIs exploited by the system by encapsulating them according to OO principles for sake of clarity and order. It also provides the Internal Interface to let other components make use of these APIs.
Moreover, the Optimizer tries to optimize the access to APIs and the computation in using them.

## *Third Part Services APIs*




This component is a container of all commercial partners' APIs. The system makes use of the OO principle of 'encapsulation' to efficiently and methodically have a reference to all these external tools. This way, this component of our system is easily extensible.

## 2.4 Deployment View



The purpose of this section is to show how the hardware deployment for our system has been designed.

Each node represented is physical and it possibly has a given execution environment and different components related to the various functionalities it offers or has access to.

The reasons for our decisions are briefly listed here:

1) The three-tiers architecture is a good compromise between security and complexity, keeping the presentation, application and data layers separated. This also permits a more coherent implementation of the MVC design pattern.

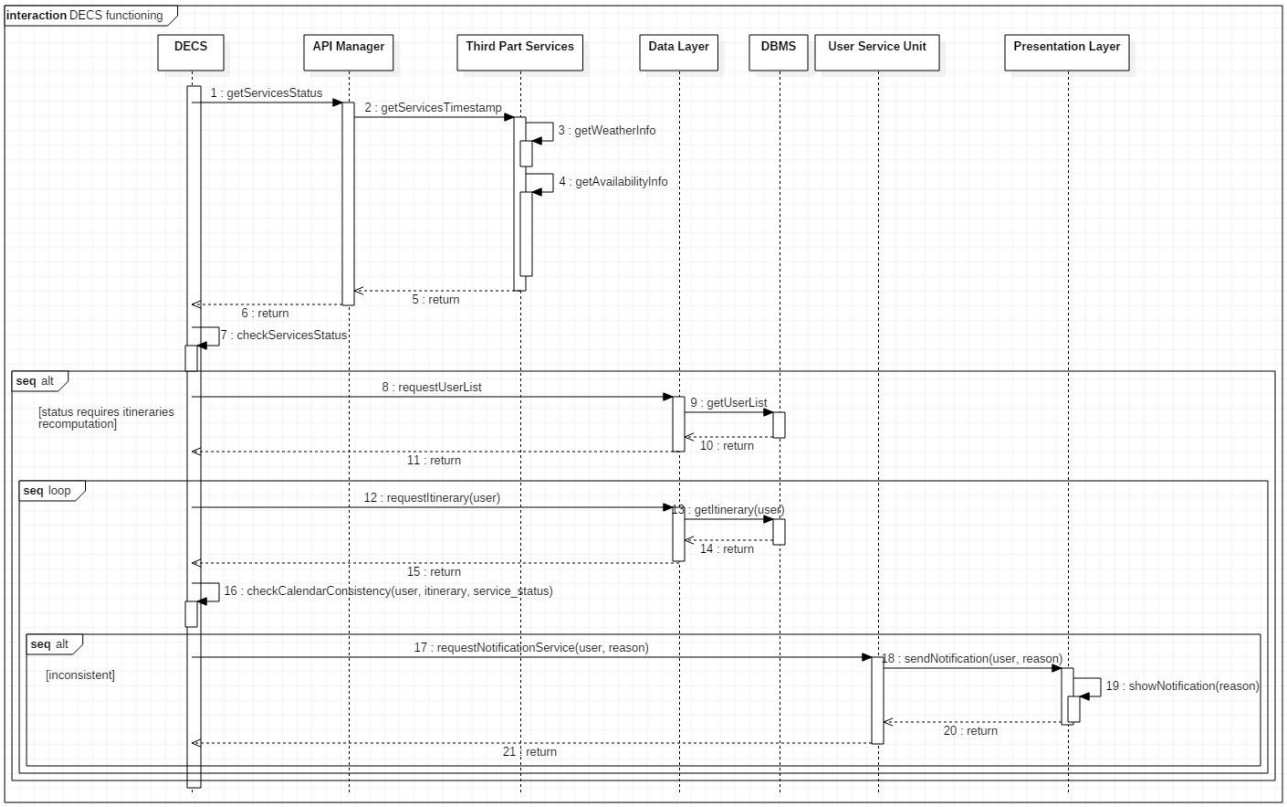
2) The system has been deployed on different physical nodes in order to avoid overloads.

3) The external API services have been modeled as a single external node for simplicity.

## 2.5 Runtime View

This chapter of the Architectural Design section is dedicated to showing how the previously identified components interact together to guarantee the correct functioning of the different features offered by our system.
Since most interactions are quite redundant, mainly requiring the cooperation of the Computation Unit and the Data Layer services, this chapter only shows the most complex interactions, where the correct functioning of almost every component is required to provide a given feature, while for the simplest just some cases are shown in detail, as the others can be easily derived from these ones.

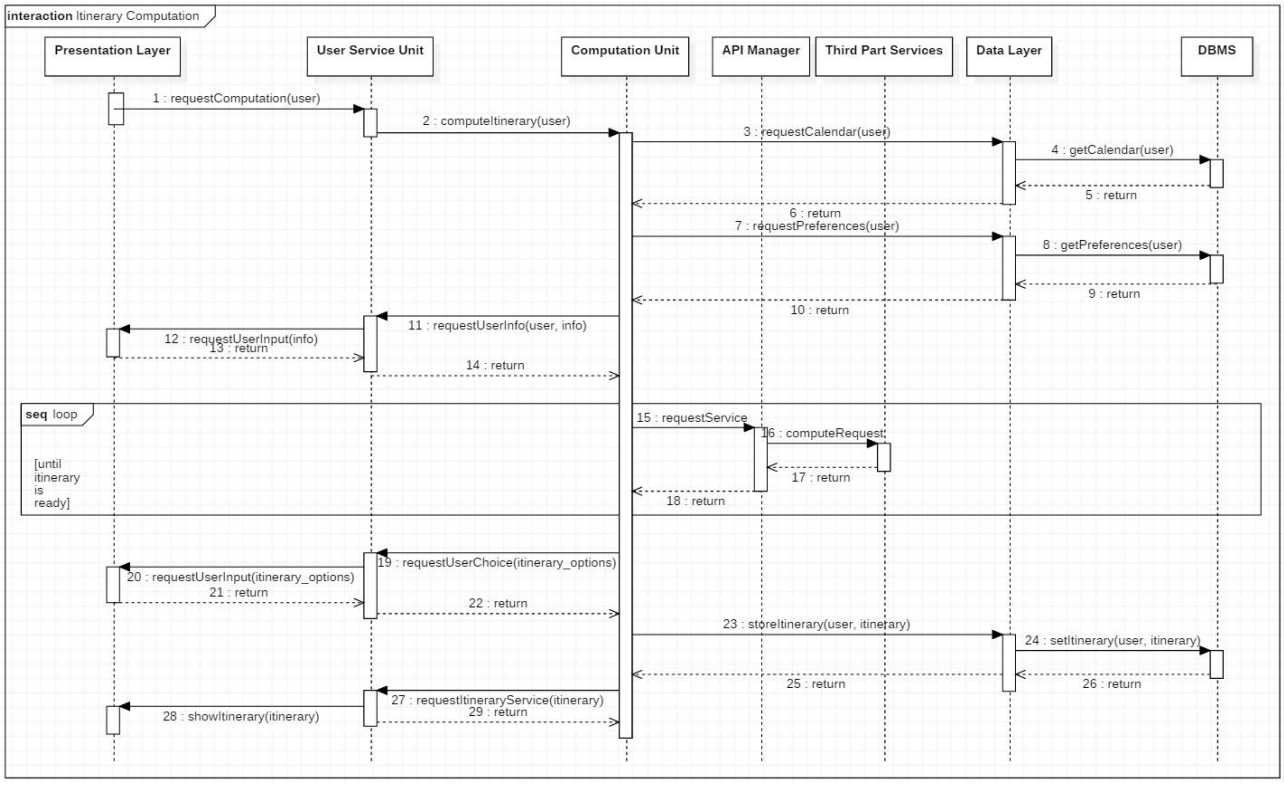
### *DECS functioning*




The DECS asks to the API Manager to retrieve the information regarding the services status. The API Manager gets a timestamp of the services, that is, the status of all services at a given point in time. In the case of weather forecast, it means getting the current weather condition, while in case of other services, it simply gets the information on the availability on those services (e.g. if the

underground metro service is available, if there is a strike, etc.). This information is returned to the DECS, which analyses it to see if it reveals possible threats for the user itineraries (e.g. if it's raining, users who were supposed to use the bicycle must be warned). If that's the case, the DECS requests the Data Layer to hand a copy of the users list, which is fetched from the DBMS. At this point, for each user, the consistency of their itinerary is verified. If their itineraries require modifications due to a change in the weather condition or availability of a given service, then the DECS prepares a notification to the user and sends it through the Notification Service provided by the User Service Unit. At this point it's up to the user to ask the system to compute a new itinerary.

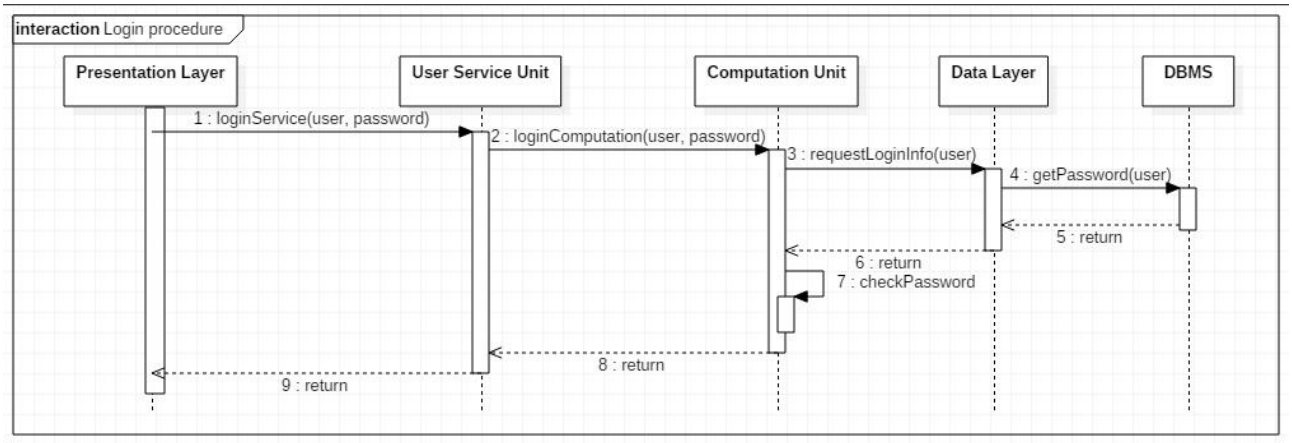
## *Itinerary computation*




Starting from the Presentation Layer (either from the desktop application or the mobile application, as it's irrelevant), a request of computation is forwarded to the User Service Unit, which notifies the Computation Unit of the request to compute a new itinerary coming from a certain user. At this point, the Computation Unit begins an information gathering phase: first, through the Data Layer, the Computation Unit asks to retrieve the calendar and preferences related to that user. Then, if necessary, the Computation Unit might ask the

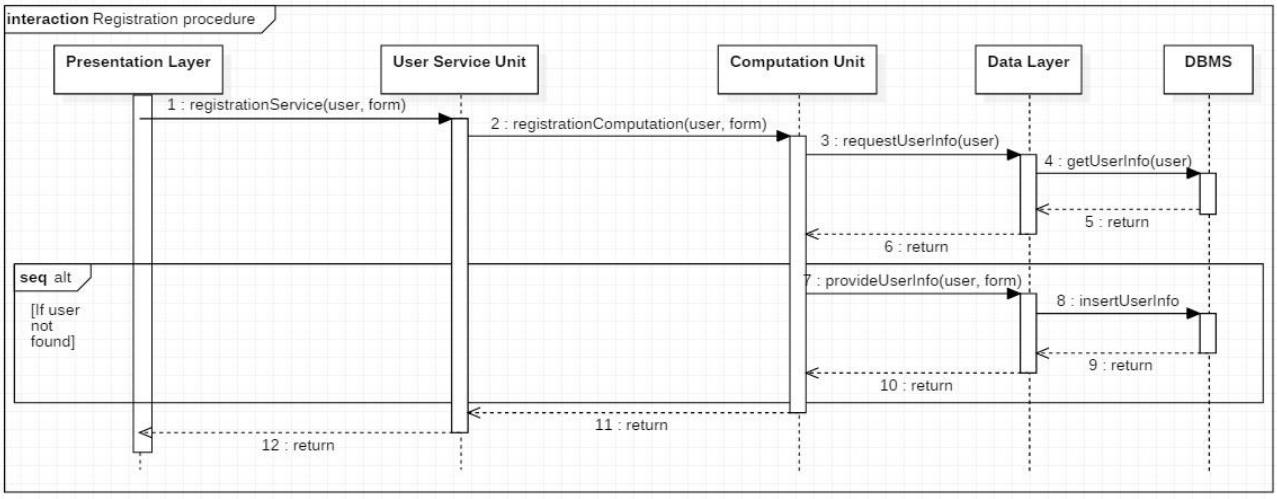
user for some details, through the User Service Unit. like specified in the algorithm section. At this point, the Computation Unit has all it needs, so a computation phase starts where, in loop, many different services are used to compute the different possible paths. When a list of potential itineraries is ready, it is immediately forwarded to the user through the User Service Unit. After making his choice, the itinerary is saved into the DBMS through the Data Layer and the selected itinerary is showed to the user.

## *Login Procedure*




This procedure is one of the simplest in our system. Through the login service provided by the User Service Unit, a login request is sent to the Computation Unit. Upon receiving the password retrieved from the DBMS through the Data Layer, the Computation Unit checks the validity of the login and the result is sent to the Presentation Layer through the User Service Unit.

### *Registration Procedure*




This procedure is quite similar to the login procedure, except it also requires storing data onto the database. In fact, after receiving the registration computation request forwarded by the User Service Unit, the Computation Unit looks for the username of the user who's trying to register onto the database. If the username is not found, the user can be registered, otherwise the user is notified of the issue.

## 2.6 Component Interfaces

In this sub-section, we'll focus on all interfaces in the System Component Diagram providing a detailed description.
Notice that our description will focus only on interfaces in the high-level Component Diagram, since inner interfaces' function and usage is almost trivial.

### DBMS Interface

This software interface is provided by the DBMSs Component (see System Overview Diagram for further information) and consists in all JDBC tools (i.e. classes, methods etc.) to query the DB using Java. Check out https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-usagenotes-statements.html for further information about integration between Java (the chosen programming language) and MySQL (the chosen DBMS for the server) and http://www.oracle.com/technetwork/java/javase/jdbc/index.html for further information about JDBC API.

## Data Access Interface

This interface is provided by the Data Layer Component who, in the intended implementation, will encapsulate the JDBC tools exposed by the DBMS Interface (see above) in order to hide the internal implementation and to make the access to the DBs (i.e. queries) easier. For a future implementation, all possible queries useful for the Travlendar+ system will be encapsulated in order to provide access control to data (static approach).

## APIs Use Interface

This Interface is provided by the APIs Manager. Its scope is to encapsulate APIs tools/libraries in Java classes and parse Third Part Service responses in order to integrate them in our Java system and make them available to the System Main Server and to the user.

## APIs Interface

This APIs Interface deals directly with external tools and services (i.e. APIs) used by Travlendar+. Since the component Third Part Services represents external entities (included in the Component Diagram to model their interactions with the system), this interface provided our system with access to external tools (i.e. APIs).

## Client Side Interface

This interface is provided by the System Main Server in order to allow the Presentation Layer to retrieve data from the DBs, from the APIs or from the System Main Server itself. The Presentation Layer exploits it to gain access to data and then prepare these data to be sent to users.

## Web Side Interface

This interface is provided the Presentation Layer and is the one users' web browsers exploit in order to gain access to Travlendar+ web-site. It just makes available "raw" data retrieved by the Presentation Layer and to be displayed on the web-site in a usable format.

### Functions Side Interface

This interface is provided by the Presentation Layer and is the one mobile/desktop application uses to gain access to the server (i.e. retrieve data from the DB, ask for computation and receive its result, exploit Third Part Services functionalities and so on).

## 2.7 Selected architectural styles and patterns

In this section we enlist all design patterns and architectures used to model our system. All of them can be found in the Component Diagrams of section 2.2 and 2.3.

### 3-tiered Architecture

We decided to model our system on three tiers: the Client Tier, the Logic and Presentation Tier and the Data Tier.
The Client Tier includes the mobile and desktop version of Travlendar+ application and the users' web browser. The Logic and Presentation Tier includes the Presentation Layer component, the Main System Server component and the APIs Manager component. Lastly, the Data Tier includes the Data Layer component and the DBMSs.
The Data Tier have been thought to be allocated on a dedicated physical machine, separated by the Logic and Presentation Tier for security and scalability (reliability) purposes. The Logic and Presentation Tier could be distributed on different physical layers in order to improve its scalability, reliability and performances.

### Client-server architecture

Due to the functionalities to be provided and the structure described in the Component Diagram, a client-server architecture will be used. The client side of the system is clearly represented by the application and by the user's web browser, while the server side is represented by the remote components that provide the service (i.e. the Presentation Layer, the System Main Server, the APIs Manager, the Data Layer and the DBMSs).

## Encapsulation

The core OO principle of encapsulation is widely used in our system in order to separate components. This allows us to perform a better access control to resources and modules and to hide the implementation and the complexity of each component.

## MVC (Model-View-Controller) design pattern

The MVC is one of the most used design pattern since it allows developers to make a clear distinction between the logic (controller), the presentation/UI (view) and data (model) of the system. For this reason our system will be developed following this design pattern.

## Adapter and facade design pattern

Designing the Travlendar+ system we often need to make heterogeneous modules interact each other. For this reason, we often use the Adapter design pattern, for instance in order to adapt JDBC functions or APIs tools to our system (see Component Diagrams). Moreover, with the Facade design pattern we are able to decide which methods (i.e. which actions) are permitted on the resources, for example which queries the System Main Server can perform or which APIs tool it can exploit.

# 3. Algorithm Design

In this section, the most important algorithms, that will need to be implemented in order to make our application work properly and satisfy the goals, will be described with details.

## 3.1 Best Travel Option

### 3.1.1 Natural Language Description

The problem is: given a user, a starting point and an arrival point, the algorithm has to compute the best travel option that leads from the starting point to the ending point and that satisfies the preferences expressed by the user.

The system running the algorithm has, thanks to the assumptions made in the RASD document, access to all the APIs of the third part systems involved:

- Google Maps;
- Car Sharing Systems;
- Bike Sharing Systems;
- Yahoo! Weather.

The access to APIs is granted through the API manager (see class diagram), so it's fair to group them by category, as listed above.

In order to avoid useless calculations and delays, the Algorithm first checks the user's preferences to check which vehicles the user has declared as available and which Vehicle Sharing services he agreed to take into consideration.

The list of available vehicles is modified as follows:

- If the starting point corresponds to the user's home, then the list is not modified.
- Otherwise, the user is asked which of the vehicles that he owns are available (that should be the one he used to go out before or none if he went out using another transport) at the moment and the list is restricted. If the user does not reply within 30 seconds, the list is considered empty.

Then, the weather is checked through the API manager and the lists are then restricted again as follows:

- If the user is home and forecasts say it's going to rain in a moment the user should be travelling, then bike is removed from both lists;

- If the user is not home and forecasts say it's going to rain then bike sharing is removed from the list of Vehicle Sharing systems available(if present);
- Otherwise, the lists are left untouched.

Now, the following calculations are performed:

1. For each vehicle in the available vehicle's list, through the API manager, the travel option from the starting position to the ending position using the vehicle is computed.
2. For each category of Vehicle Sharing system available, the nearest one is located. Then, for each of them, the best travel option corresponds to the best travel option to reach the vehicle plus the best travel option from the vehicle's position to the destination (using the vehicle). In order to have a consistent final travel option, both sub-travel options must satisfy preferences.
3. The best travel option using public transport (and on foot) are computed.

Then:

- For each travel option in the list 1, if the user is home, considering that if he goes out with a certain vehicle he may want to use it until he goes home again, it is checked is a good travel option using that vehicle is available among all other events of the day. If not, the travel option is removed from list 1.
- If the user is not home and list 1 contains a travel option, all other lists are emptied because he will need to use his vehicle.

Following, the travel options that don't satisfy the user's preferences are removed from the lists. At this very moment, if the user is using any special modality (e.g. minimize cost modality), it is verified correctly, leaving in the lists only the travel means that satisfy them. Furthermore, travel options are deleted also if they take more than the available time. The last step is:

- If at least one list is not empty, the travel option that takes less time is taken from each list and presented to the user.
- If all the lists are empty, it means that the user can't arrive to the place on time and have his preferences satisfied at the same time. In this case, the user is presented the travel options as they were before the application of the preferences and the user is warned of the situation.

## 3.2 Overlapping New Activity Check

### 3.2.1 Natural Language Description

The problem is: given a user and an activity, the algorithm has to check if the activity can be added to the calendar.
A first check has to be done on the activity:

- If the activity to be added is a fixed activity, then:
  - For each **fixed** activity Act already present in the user's calendar, if Act has either one or both starting and ending moment strictly included in the time span defined by the starting and ending moment of activity to be added, return **false**.
    This guarantees that there's no overlapping with fixed activities;
  - If the algorithm got to this point, it means that the activity to be added does not overlap with other fixed activities.
    Considering now a calendar that contains also the activity to be added, if at least one of the flexible activities doesn't have anymore a possible "placement" for the effective activity duration in the range defined by the starting and ending moment, return **false**;
  - Return **true** if false has not been returned in the previous checks;
- Otherwise, it means that the activity to be added is a flexible activity. In this case, if there is a possible "placement" for the effective activity duration in the range defined by the starting and ending moment and all other flexible activities can still be placed, return **true**. Otherwise, return **false**.

### 3.2.2 Java Code

In this section, only strictly necessary methods are represented. Further information and the complete code can be found in the attachments and/or in the delivery folder.

- **Break class**

```
1  package Travlendar;
2
3  import java.util.ArrayList;
5
6  public class Break implements Activity{
7      private Date startDate;
8      private Date endDate;
9
10     /**minutes**/
11     private long duration;
12
13     /***************CONSTRUCTORS****************/
14     /**
15      * @param s is the start date
16      * @param e is the ending date
17      * @param d is the duration
18      */
19     public Break(Date s, Date e, long d){
24     /**
25      * @param act is the break to copy
26      */
27     public Break(Break act) {
32     /***********************************************/
33
34     /**
35      * @param c is the calendar for which to verify whether the break can
36      * be added into it or not.
37      * @return true if the calendar containing the break to be added is
38      * still coherent, false otherwise.
39      */
40     @Override
41     public boolean canBeAddedTo(Calendar c) {
42         ArrayList<Break> b=Break.copyList(c.getBreaks());
43         b.add(new Break(this));
44         ArrayList<FixedActivity> fa=FixedActivity.copyList(c.getFixedActivities());
45         return c.canBeACalendar(fa, b);
46     }
47
48     /**
49      * @param b is the array list of breaks to copy
50      * @return a new ArrayList object containing a copy of all
51      * the breaks in the ArrayList given as parameter
52      */
53     public static ArrayList<Break> copyList(ArrayList<Break> b){
59
60
61     /************GETTERS*****************/
62     public Date getStart(){
65     public Date getEnd(){
68     public long getDuration(){
71 }
72
```

- **FixedActivity class**

```java
package Travlendar;

import java.util.*;

public class FixedActivity implements Activity{
    private Date startDate;
    private Date endDate;

    /*****************CONSTRUCTORS**********************/
    /**
     * @param s is the start date
     * @param e is the ending date
     */
    public FixedActivity(Date s, Date e){
        startDate=s;
        endDate=e;
    }
    /**
     * @param act is the FixedActivity to copy
     */
    public FixedActivity(FixedActivity fa){
        this.startDate=new Date(fa.startDate.getTime());
        this.endDate=new Date(fa.endDate.getTime());
    }

    /***********************************************/

    /**
     * @param a is a fixed activity to compare with the caller object
     * @return true if the callers's ending time comes before or is equal to
     * the parameters's starting time. false otherwise
     */
    public boolean isBefore(FixedActivity a){
        return endDate.before(a.startDate) || endDate.equals(a.startDate);
    }

    /**
     * @param a is a fixed activity to compare with the caller object
     * @return true if the callers's starting time comes after or is equal to
     * the parameters's ending time. false otherwise
     */
    public boolean isAfter(FixedActivity a){
        return this.startDate.after(a.endDate) || startDate.equals(a.endDate);
    }

    /**
     * This method is used to verify that breaks can have a placement in the calendar.
     * @param b is the break to parse to FixedActivity.
     * @param s is the starting time of the new FixedActivity.
     * @param e is the ending time of the new FixedActivity.
     * @return the new FixedActivity produced from break b and the given times.
     * @throws InvalidInputException when s comes before b's starting time or
     *              e comes after b's ending time. Basically
     *              the exception is thrown when s and e are not coherent with the break
     *              that is going to be parsed.
     */
    static FixedActivity parseFixedActivity(Break b, Date s, Date e) throws InvalidInputException{
        if(s.before(b.getStart()) || e.after(b.getEnd()))
            throw new InvalidInputException();
        return new FixedActivity(s,e);
    }
```

```
62      /**
63       * @param c is the calendar for which to verify whether the FixedActivity can
64       * be added into it or not.
65       * @return true if the calendar containing the FixedActivity to be added is
66       * still coherent, false otherwise.
67       */
68      @Override
69      public boolean canBeAddedTo(Calendar c) {
70          ArrayList<FixedActivity> calendarActivities= c.getFixedActivities();
71          for(FixedActivity fa:calendarActivities)
72              if(!this.isBefore(fa) && !this.isAfter(fa))
73                  return false;
74          ArrayList<FixedActivity> fixApp=new ArrayList<>();
75          ArrayList<Break> breaks=new ArrayList<>();
76          boundSubCalendar(c,fixApp, breaks);
77          fixApp.add(this);
78          return c.canBeACalendar(fixApp,breaks);
79      }
80
81
82      /**
83       * Supports canBeAddedTo method. This method reduces the complexity of the process
84       * followed to verify whether a FixedActivity can be added to a calendar or not
85       * by finding a sub-calendar of the original calendar on which to verify the conditions.
86       * Since the sub-calendar will probably contain less activities than the original calendar,
87       * operations that will be executed are significantly less.
88       * @param c is the original calendar
89       * @param fixApp is the ArrayList that, at the end of the process, will contain
90       *          all the FixedActvities of the sub-calendar.
91       * @param breaksis the ArrayList that, at the end of the process, will contain
92       *          all the Breaks of the sub-calendar.
93       */
94      private void boundSubCalendar(Calendar c,ArrayList<FixedActivity> fixApp, ArrayList<Break> breaks) {
95          ArrayList<FixedActivity> fa=c.getFixedActivities();
96          ArrayList<Break> b=c.getBreaks();
97          for(Break br:b){
98              if((br.getStart().after(this.startDate) || br.getStart().equals(this.startDate))
99                      && br.getStart().before(this.endDate) ||
100                         br.getEnd().after(this.startDate) &&
101                     (br.getEnd().before(this.endDate) || br.getEnd().equals(this.endDate))){
102                     breaks.add(br);
103             }
104         }
105         for(FixedActivity fAct: fa){
106             for(Break br:breaks){
107                 if((br.getStart().after(fAct.startDate) || br.getStart().equals(fAct.startDate))
108                         && br.getStart().before(fAct.endDate) ||
109                         br.getEnd().after(fAct.startDate) &&
110                     (br.getEnd().before(fAct.endDate) || br.getEnd().equals(fAct.endDate))){
111                     fixApp.add(fAct);
112                     break;
113                 }
114             }
115         }
116         fixApp=copyList(fixApp);
117     }
118
119
120     /**
121      * @param b is the array list of FixedActivity to copy
122      * @return a new ArrayList object containing a copy of all
123      * the FixedActivities in the ArrayList given as parameter
124      */
125     public static ArrayList<FixedActivity> copyList(ArrayList<FixedActivity> fa){
131 }
132
```

- **Calendar class**

```
1  package Travlendar;
2
3  import java.util.ArrayList;
5
6  public class Calendar {
7      private ArrayList<FixedActivity> fixedActivities;
8      private ArrayList<Break> breaks;
9
10     /**Written in minutes, SCATTO represents and must equal to the minimum time
11      * granularity of the activities*/
12     private static int SCATTO=15;
13
14     /***********CONSTRUCTORS****************/
15     Calendar(){
19     Calendar(ArrayList<FixedActivity> fa, ArrayList<Break> b){
23
24     /*****************************************/
25     /**
26      * Adds a FixedActivity to the calendar
27      * @param a is the activity to be added
28      * @throws CannotBeAddedException is the activity can't be added
29      */
30     public void addActivity(FixedActivity a) throws CannotBeAddedException{
37     /**
38      * Adds a Break to the calendar
39      * @param a is the break to be added
40      * @throws CannotBeAddedException is the activity can't be added
41      */
42     public void addActivity(Break a) throws CannotBeAddedException{
49
50     /**
51      * Verifies if a list of FixedActivities is consinstent
52      * @param fa is the list of FixedActivities
53      * @return true if the list's elements don't overlap, false otherwise
54      */
55     private boolean isConsistent(ArrayList<FixedActivity> fa){
56         int size=fa.size();
57         if(size<=1)
58             return true;
59         for(int i=0; i<size-1;i++)
60             for(int j=i+1;j<size;j++)
61                 if(!fa.get(i).isBefore(fa.get(j)) && !fa.get(i).isAfter(fa.get(j)))
62                     return false;
63         return true;
64     }
65
66     /**
67      * Verifies if a calendar containing the given activities can exist
68      * @param fa is the list of FixedActivities
69      * @param b is the list of Breaks
70      * @return true if the lists together can represent a calendar. false otherwise
71      */
72     public boolean canBeACalendar(ArrayList<FixedActivity> fa, ArrayList<Break> b){
73         return recursiveCanBeACalendar(fa,b,b.size());
74     }
75
76     /**
77      * Supports canBeACalendar
78      * @param fa is the list of FixedActivities
79      * @param b is the list of Breaks
80      * @param s is the size of b
81      * @return true if the lists together can represent a calendar. false otherwise
82      */
```

```
83      private boolean recursiveCanBeACalendar(ArrayList<FixedActivity> fa, ArrayList<Break> b, int s){
84          if(s==0)
85              return isConsistent(fa);
86          Break bApp=b.get(s-1);
87          Date start=bApp.getStart();
88          Date end= new Date(start.getTime()+bApp.getDuration()*60*1000);
89          while(true){
90              try {
91                  FixedActivity faToAdd=FixedActivity.parseFixedActivity(bApp, start,end);
92                  fa.add(faToAdd);
93                  if(recursiveCanBeACalendar(fa, b, s-1))
94                      return true;
95                  start=new Date(start.getTime()+SCATTO*60*1000);
96                  end=new Date(end.getTime()+SCATTO*60*1000);
97                  fa.remove(faToAdd);
98              } catch (InvalidInputException e) {
99                  break;
100             }
101
102         }
103         return false;
104     }
105
106
107     /********************GETTERS*********************************/
108     public ArrayList<FixedActivity> getFixedActivities(){
109         return fixedActivities;
110     }
111     public ArrayList<Break> getBreaks(){
112         return breaks;
113     }
114 }
```

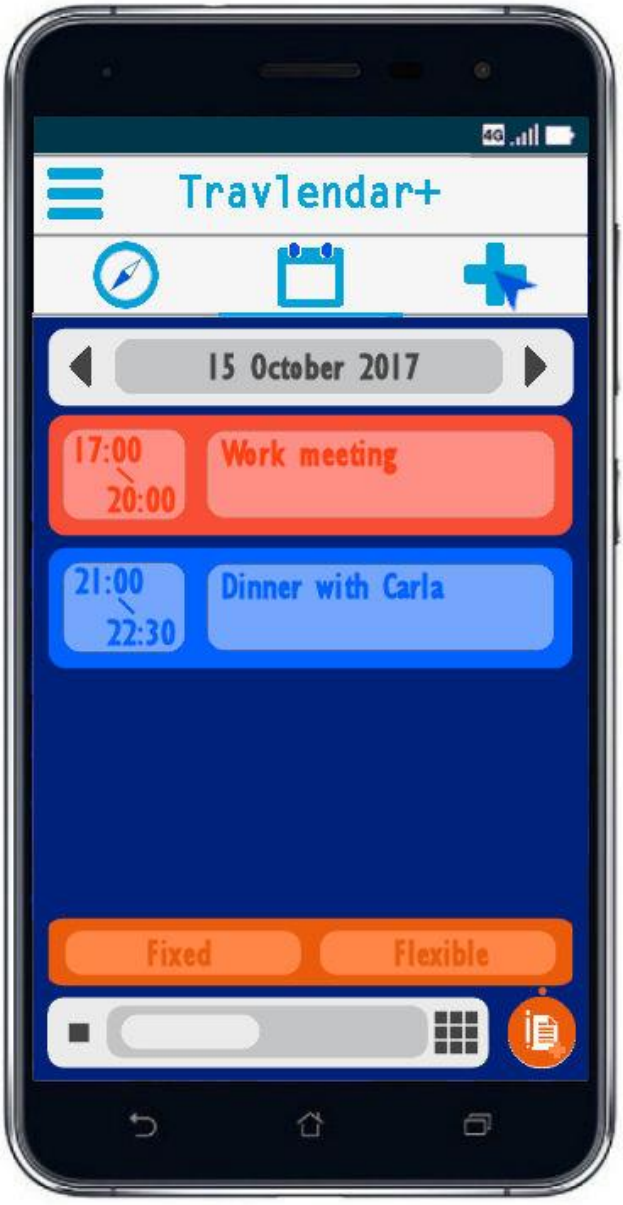
Note that this small example has been provided with the only purpose to show the structure of the algorithm (that actually works because this is a complete code). Some previous or further operations that are not strictly related to the execution of the algorithm (e.g. load activities data from a DataBase or storing the new Activity in the DB after adding it to a calendar) have not been represented .

# 4. User Interface Design

## 4.1 Additional UIs

Though some UI samples have already been provided in the RASD document ( see from [UI1] to [UI8] in RASD), for a better mapping of our design choices into the requirements another one is provided in the DD document.

**[UI9]**



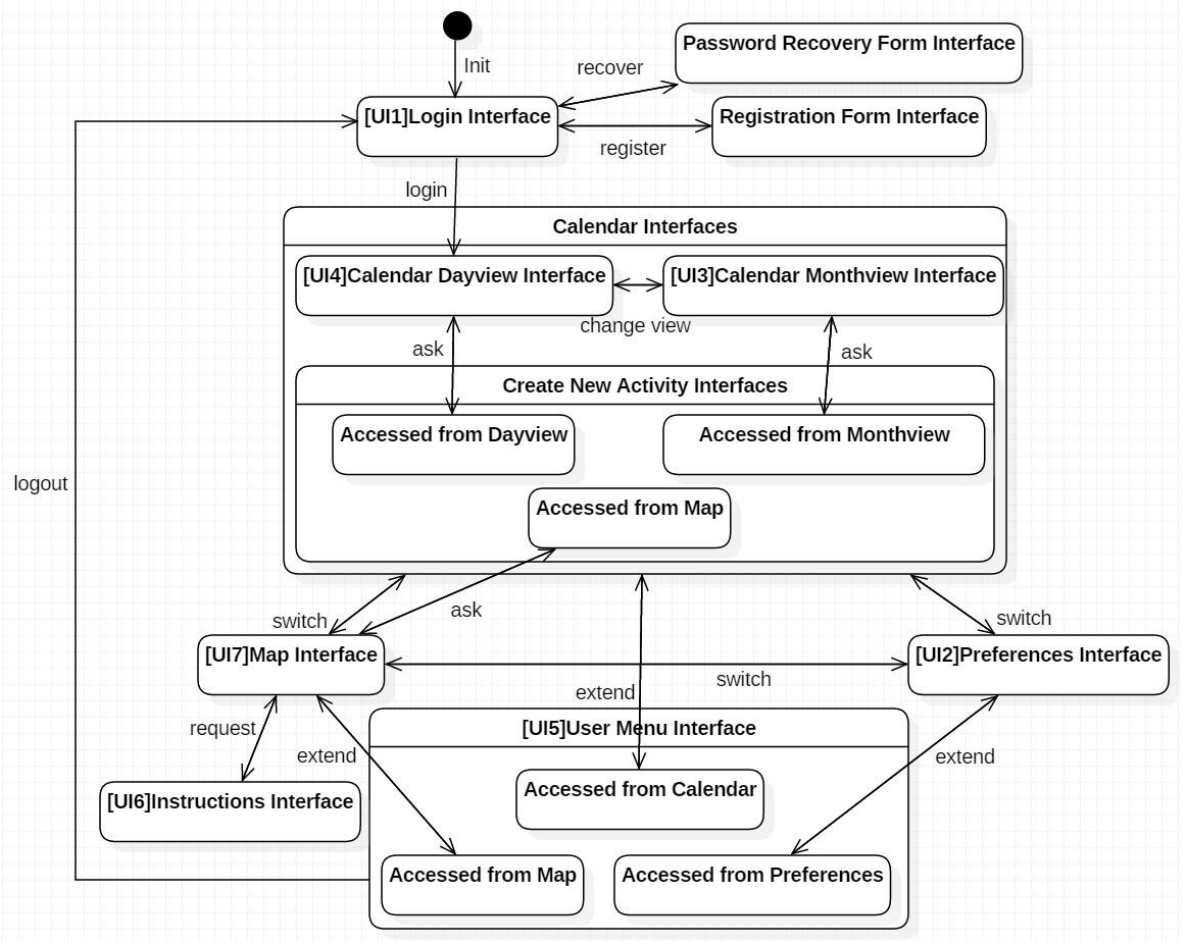

This UI shows how users can differentiate between fixed and flexible activities. This is obtained through a small menu that pops up after tapping on the Create New Activity button.

## 4.2 Mapping UIs into Requirements

Considering the functional requirements, that have already been described in detail in the RASD document, here's a brief explanation of how our User Interface samples are coherent with the requirements and help us to satisfy them in order to reach the defined goals ( -> RASD ):

- [UI4] (and [UI9]) show the day's activities. By tapping on them it's possible to reach the screen from which it's possible to modify an activity. They match with [R1];
- [UI1], the login screen, matches with [R2]. Furthermore, it also matches with [R3] and [R4] because it shows it's possible to change password and to register in the system as a new user;
- [UI8] and [UI9] match with [R5] because they show it's possible to create fixed and flexible activities;
- [UI2] satisfies [R6], [R6.1] and [R9] because it shows some of the possible preferences, including the possibility to take into consideration bike and car sharing;
- [UI6] and [UI7] obviously match with [R11] because they show mobility options and travelling phase.

## 4.3 UI Navigation State Chart



The above statechart highlights the most relevant connections between the different UIs presented into the RASD, with the correspondent notation. Specifically, the major points to notice are:

1) From the Calendar Dayview Interface, which is the first interface the users are presented after the login, they can reach all the main functionalities of Travlendar+.

2) The condition of the Calendar Interfaces macrostate is always preserved, that is, if for instance one user decides to create an activity from one of the Calendar Interfaces and then to switch to the Preferences Interface to set up some constraints, when switching back to the Calendar Interfaces he or she will still be presented the Activity Creation Interface, thus preserving any data that was inserted. This explains the meaning of the arrows connected to the whole macrostate instead of the single microstates.

3) The "accessed from..." states serve to keep track of the interface from which a given functionality was accessed, so that the users will then be brought back to the correct interface. The core functionality accessed from those states, though, is still the same.

4) Some interfaces have been omitted for simplicity, such as the distincted ones for fixed and flexible activities creation. This is because there are very few differences between them, so they can be treated as the same from the point of view of this statechart.

# 5. Requirements Traceability

In this section, we provide a mapping between functional and non-functional requirements specified in details in the RASD and design components described mainly in the Architectures Design section of this document. We explain how each requirement is fulfilled by means of one or more components.

## 5.1 Functional Requirements mapping

**[R1] Allow the users to manage already existing activities -** This requirement is fulfilled by, and so mapped to, the System Main Server (and partially by the Data Layer) and more in detail by the User Service Unit and the Computation Unit. The former deals with the users and listens for their requests, the latter is able to perform computation. Both two components are able to query and write the DBs.

**[R2] Users should be able to log in to Travlendar+ -** Functionalities described by this requirement are provided by the User Service Unit which is able to receive requests and forward them to the Computation Unit ,to let the users log in the system (by means of the Data Layer).

**[R3] Users should be able to register to Travlendar+ -** Functionalities described by this requirement are provided by the User Service Unit which is able to receive requests and forward them to the Computation Unit ,to let the users register into the system (by means of the Data Layer).

**[R4] Users should be able to change their password whether they forget it -** This requirement is fulfilled by means of the Computation Unit which is able to query the DBs (through the Data Access Interface provide by the Data Layer) to retrieve data of the users and manage the password recovery procedure.

**[R5] Users should be able to schedule new activities –** This requirement is fulfilled in our system by the System Main Server (with the interaction with the APIs Manager and the Data Layer). In fact, the Computation Unit inside the System Main Server is the component appointed to compute the best travel options exploiting APIs and to deal with DBs.

**[R6] Users should be able to set their own preferences that will be taken into account and will be applied to schedules every this is possible and reasonable** and **[R6.1]** ***Specification:*** **The user can also set flexible activities (e.g. flexible lunch), and, in particular, the modality "Minimize Carbon Footprint" will be present –** Again, we map these requirements to the Computation Unit and to the User Service Unit since the former is the component that will take into account and apply users' preferences and flexible activities when computing the best travel options, and the latter is able to deals with users and their choices.

**[R7] When necessary, users should be supported in buying public transports tickets directly on Travlendar+ and/or redirected on the correct external page –** This requirement is mapped into the Computation Unit since this is the component that exploit APIs when interacting with the users.

**[R8] Users should be warned when they're scheduling an activity that is not physically possible due to a lack of time or that overlaps with other activities –** This requirements is clearly mapped into the Computation Unit inside the System Main Server, and also by the local application (both mobile and desktop one) since these are the components that search for consistency and overlapping issues.

**[R9] Mobility solutions involving car and bike sharing systems must be taken into account, when possible, and proposed to the user when they represent the optimal solution –** This requirement is fulfilled by means of the Computation Unit and the APIs Manager. The former computes travel options and the latter provides access to third part service.

**[R10] Users should receive a notification (e.g. email, push notification) a little before the time they have to leave to go to the next appointment –** This requirement is trivially fulfilled by the Dynamic Event Check System whose scope is to check users' calendars and generate notification.

**[R11] The application should identify the best mobility option. Moreover, this should be done by appointment and by day (e.g., the app should suggest that you leave your home via car in the morning because meetings during the day will not be doable via public transportation) –** Since this requirement deals with computation, it is mapped into the Computation Unit inside the System Main Server.

## 5.2 Non-Functional Requirements mapping

**[NFR1] After a user is logged in, he should be able to reach every functionality in less than 3 taps/clicks –** This requirement is fulfilled by the designed structure of the whole system and by the GUI in the mobile and desktop application.

**[NFR2] The mobile application, when it will be developed, should work properly on Android, iOS (the most widely used) and, optionally, on Windows Phone –** This requirement is mapped to the Presentation Layer since this is the component appointed to make data compatible with users' devices.

**[NFR3] The desktop application should work at least on Windows 7 or higher and on MacOS X or higher -** This requirement is mapped to the Presentation Layer since this is the component appointed to make data compatible with users' OS.

**[NFR4] The system should be available at least 99,9% of the time over a year –** This requirement is fulfilled by the System Main Server and by the distribution we are going to perform (as specified in the component diagram descriptions).

**[NFR5] The website should work fine at least on the best-known browsers (Safari, Google Chrome, Mozilla FireFox, Internet Edge) -** This requirement is mapped to the Presentation Layer since this is the component appointed to make data compatible with users' web browser.

# 6. Implementation, integration and test plan

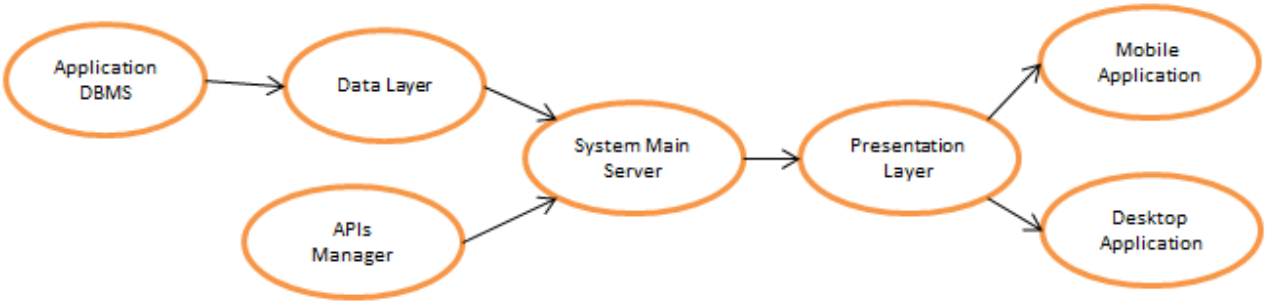
Considering the diagram showed in the **2.2 System Overview** section as reference, it's easy to notice which components (depend) on other components. There are components, anyway, that are independent. Considering also that there are no circular dependencies, we can conclude that we are able to implement components according to dependencies, that means that a component is implementable if and only if it does not depend on other components or its dependencies have already been implemented. Let's analyse dependencies:

- **Application DBMS** is independent;
- **Data Layer** depends on **Application DBMS**, since it has to build queries and send them to the **Application DBMS** component;
- **System Main Server** depends on the **Data Layer** and **APIs Manager**, since it needs both of them to perform its operations (controller);
- **APIs Manager** relies on the **Third Part Services APIs** in the sense that it can accomplish its role if and only if the **Third Part Services APIs** work fine. However, we don't need to implement the **Third Part Service APIs** because they have obviously been implemented already by the respective owners, so we can consider the **APIs Manager** independent;
- **Presentation Layer** relies on the **System Main Server**;
- **Mobile Application** relies on the **Presentation Layer**;
- **Desktop Application** relies on the **Presentation Layer**.

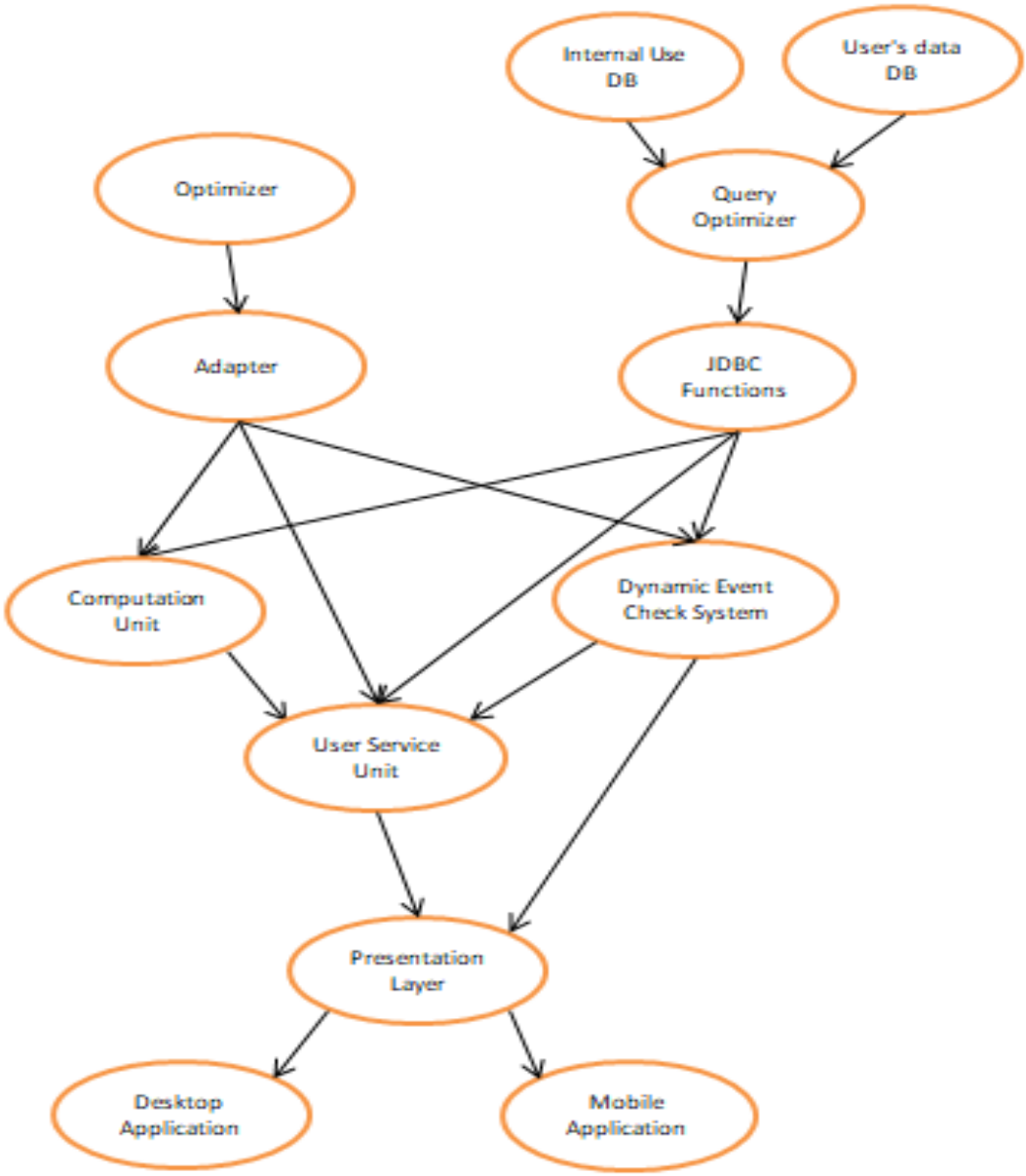
For example, a feasible developing order becomes: Application DBMS -> Data Layer -> APIs Manager -> System Main Server -> Presentation Layer -> Mobile Application -> Desktop Application.

Note that the browser is not represented here because it doesn't need to be developed.

As a matter of clarity, and to highlight other possible orders, we can represent dependencies as a graph, where A -> B if and only if B depends on A.




The same reasonment can be applied considering the Component view for each node of the graph. Expanding all nodes of the previous graph, we obtain a new and more detailed graph that expresses priorities, showed below.



Considering the way we designed the implementation process, it appears natural to proceed to the integration and test phases with this strategy:

- When the **Application Database** and the **Data Layer** have been completely projected, the **Data Layer** must be completely tested, together with all the queries it exposes to query the Database. Since the **Data Layer** implements all and only the queries that are useful to the system itself, this will also help to understand if the DB has been projected and implemented correctly;
- Similarly to the Data Layer, the **APIs Manager** exposes all the calls that the system needs to do to external systems. Provided that the Third Part Systems are reachable ,it needs to be tested completely, in order to be sure that the calls are formatted and implemented correctly and that the system receives the results it needs;
- Since the **System Main Server** is implemented after the **Data Layer** and the **APIs Manager** are fully implemented and tested (so we can say they are perfectly functional), it can be integrated with them from the beginning. Moreover, the **System Main Server** should be tested in all its functionalities.
- For the same reasons, the **Presentation Layer** can be integrated with the **System Main Server** immediately.
- When the clients (**Mobile Application** and **Desktop Application**) are developed, they just need to be integrated with the whole system, that is already implemented, integrated and tested and thus can be considered fully and perfectly functional.

# 7. Effort Spent

This section will provide detailed information about the number of hours spent on this document.

**Matteo Biasielli**, matr. 893590

| **Section(s)** | **Number of hours** |
|---|---|
| 28-oct-17 Introduction | 1 |
| 29-oct-17 Algorithm Design | 3 |
| 31-oct-17 Algorithm 2 Pseudocode | 1 |
| 01-nov-17 Algorithm 2 Pseudocode | 2 |
| 02-nov-17 Algorithm 2 Pseudocode | 2 |
| 02-nov-17 UI design | 1 |
| 05-nov-17 Google Directions Api | 1.5 |
| 06-nov-17 Review | 1 |
| 05-nov-17 Google Directions Api | 1 |
| 07-nov-17 Group Review | 1 |
| 09-nov-17 Data Layer Implementation | 5 |
| 11-nov-17 Data Layer Implementation | 5 |
| 16-nov-17 Preparing JEE | 5 |
| 17-nov-17 Group Review | 4 |
| 22-nov-17 Group Review | 2 |
| | |
| | |
| | |
| | |
| **TOTAL:** | 35,5 |

**Mattia Di Fatta,** matr. 893608

| **Section(s)** | **Number of hours** |
|---|---|
| 28-oct-17 System Component Diagram | 1 |
| 29-oct-17 System Component Diagram + single Component Diagram | 2.5 |
| 30-oct-17 single Component Diagram | 2 |
| 31-oct-17 Started Descriptions of Component Diagrams | 2 |
| 1-nov-17 Descriptions of Component Diagrams | 2 |
| 4-nov-17 Descriptions of Component Diagrams | 2 |
| 6-nov-17 started component interfaces | 1.5 |
| 07-nov-17 Group Review | 1 |
| 10-nov-17 Ended component interfaces + arch styles and patterns | 3 |
| 11-nov-17 Requirements Traceability | 2 |
| 17-nov-17 Group Review | 4 |
| 22-nov-17 Group Review | 2 |
| | |
| **TOTAL:** | 25 |

**Emilio Capo,** matr. 899842

| **Section(s)** | **Number of hours** |
|---|---|
| 4-nov-17 Runtime View | 1.5 |
| 5-nov-17 Runtime View | 1.5 |
| 07-nov-17 Group Review | 1 |
| 8-nov-17 Runtime View | 2 |
| 9-nov-17 Deployment diagram | 1 |
| 15-nov-17 UI state chart | 1.5 |
| 16-nov-17 General Review | 3 |
| 17-nov-17 Group Review | 4 |
| 22-nov-17 Group Review | 2 |
| | |
| | |
| | |
| | |
| | |
| **TOTAL:** | 17,5 |