

Politecnico di Milano

A.Y. 2017/2018

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

**Implementation & Testing**

**Document**

Matteo Biasielli - Emilio Capo - Mattia Di Fatta

v. 0.1

**Table of Contents**

**1. Introduction**

1.1. Document purpose..………………………………………………...………………….3

1.2. Description of the problem………………………………………………………….3

1.3. Definitions, Acronyms, Abbreviations………………………………………….3

1.3.1 Definitions……………………………………………………………………….3

1.3.2 Acronyms………………………………………………………………………...4

1.3.3 Abbreviations…………………………………………………………………..4

**2. Requirements mapping and Functionalities**

2.1. Functionalities

2.2. Requirements mapping

**3. Adopted development frameworks**

**4. Structure of the source code**

**5. Test Plan**

**6. Performed tests and analysis**

6.1. Unit Test..………………………………………………...………………….

6.2. Integration testing…………………………………………………….

6.3. Performances evaluation……………………………………………….

**7. External References**

**8. Tools Used**

**9. Effort Spent**

**1. Introduction**

* 1. **Document purpose**

This document focuses on the implementation of details the project Travlendar+. The application’s purpose is to support users in handling out one of the most difficult nowadays’ challenges: organization. No previous versions of this application were developed.

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. **Description of the problem** *[reported from RASD document]*

The aim of the project is to create an all-in-one system that unites services that are nowadays offered by various different applications (e.g. Calendar, Travel Scheduler). In order to use Travlendar+, final users must be registered and logged in.

Users should be able to schedule their activities directly through the application and, by taking into account travelling times, constraints and preferences expressed by the user, Travlendar+ should:

* Identify the best mobility option;
* Support the user in buying public transport tickets, if necessary;
* Locate the nearest car or bike sharing, if they represent the best solution;
* Warn the user when a place can’t be reached in the available time.

In general, Travlendar+ should make it easier to organize complex schedules, by finding the best compromises between time optimization and the users’ needs and preferences.

* 1. **Definitions, Acronyms, Abbreviations**
     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.
* **Break**: Flexible Actvity.
  + 1. **Acronyms**
* **RASD:** Requirements Analysis and Specification Document
* **UI:** User Interface
* **API:** [Application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface)
* **UML**: Unified Modeling Language
* **GPS**: Global Positioning System
  + 1. **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
* **[Fn]:** the n-th functionality

**2. Requirements mapping and Functionalities**

* 1. **Functionalities**

In the allotted time for the implementation, we could develop the Server (which structure is discussed later in this document), the Windows Client and Android Client. Considering that **this is a prototype**, we focused on the basic features and what we achieved is that the functionalities we could provide are the following:

* **[F1] Login:** A Login system is available and it’s mandatory to login into the system to use the application;
* **[F2] Registration:** Users that are not registered yet are able to register into the system. Note that for the momentthe registration procedure asks nothing but username and password because those were the only vital data required to provide basic login functionalities. It can be easily extended;
* **[F3] View Calendar:** After the Login, on both clients it’s possible for users to view their calendar with activities that were previously added;
* **[F4] Add a Fixed Activity:** After the Login , users can schedule fixed activities (see definition). After scheduling such an activity, they receive a response from the server containing the result status of the request (OK or an error code) and eventually a notification. There are many different notifications but it’s worth it to focus the attention on a few of them:
  + A notification is received if the added activity does not make the calendar be inconsistent (no overlapping) but the user will not be allowed to be on time to the just added activity;
  + A notification is received if the added activity does not make the calendar be inconsistent (no overlapping) but the user will not be allowed to be on time to the activity that comes immediately after the just added one;
  + A notification is received if the added activity does not make the calendar be inconsistent (no overlapping) but the user will not be allowed to be on time to one activity and the above two cases are not verified.
* **[F5] Add a Flexible Activity (Break)**: After the Login , users can schedule flexible activities (see definition). After scheduling such an activity, they receive a response from the server containing the result status of the request (OK or an error code) and eventually a notification. There are many different notifications but it’s worth it to focus the attention on one of them:
  + A notification is received if the added activity does not make the calendar be inconsistent (no overlapping) but the user will not be allowed to be on time to one activity and the above two cases are not verified.
* **[F6] Update an existing activity:** After the Login, users can update their previously added activities. They can modify any field of activities and they can also change it from fixed to flexible or vice versa. The answer/notification they receive is the same as for [F6] and [F5];
* **[F7] Delete an existing activity:** After the Login, users can delete their previously added activities.
* **[F8] Add a tag:** a tag is a tuple <Position, Address, text>. It represents an address that the user will refer to with a keyword text that he sets as well. Most likely there are some places that users will have to write down really often. The point of the tag system is to allow users to indicate those address by just selecting the tag from the tag list instead of writing down the whole address every time. After the Login, users can add new tags. We completely rely on Google Geocoding and Google Reverse Geocoding APIs for this service, to check that inserted addresses are valid.
* **[F8] Delete a tag:** a tag is a tuple <Position, Address, text>. It represents an address that the user will refer to with a keyword text that he sets as well. Most likely there are some places that users will have to write down really often. The point of the tag system is to allow users to indicate those address by just selecting the tag from the tag list instead of writing down the whole address every time. After the Login, users can delete already existing tags.
* **[F9] Preferences:** After the Login, users can indicate theur travel preferences. There are plenty of possibilities and each of them is always accepted. Though, is a user sets too strict preferences it won’t be possible to find possible routes or to estimate the travel time for an activity in order to be allowed to send notifications to advice the user to get ready to go out. Should this happen, users will be informed.
* **[F10] Notifications:** users are notified every day about the weather. Moreover, users are notified when they should get ready to leave for the next activity. Clients execute the notifications synchronization process every minute;
* **[F11] Travel:** Users can request to be shown a possible travel mean (or more than one) for the next activity. If this is not possible due to their too strict preferences, users are advised to change preferences and retry.

**2.2 Requirements mapping** *[Requirements reported from RASD]*

* **[R1]** Allow the users to manage already existing activities.

This is achieved by [F3], [F6], [F7].

* **[R2]** Users should be able to log in to Travlendar+.

This is achieved by [F1].

* **[R3]** Users should be able to register to Travlendar+.

This is achieved by [F2].

* **[R5]**  Users should be able to schedule new activities.

This is achieved by [F4], [F5].

* **[R6]** Users should be able to set their own preferences that will be taken into account and will be applied to schedules every time this is possible and reasonable.

This is achieved by [F9].

* **[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.

This is achieved by [F9], [F5].

* **[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 is achieved by [F4], [F5].

* **[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.

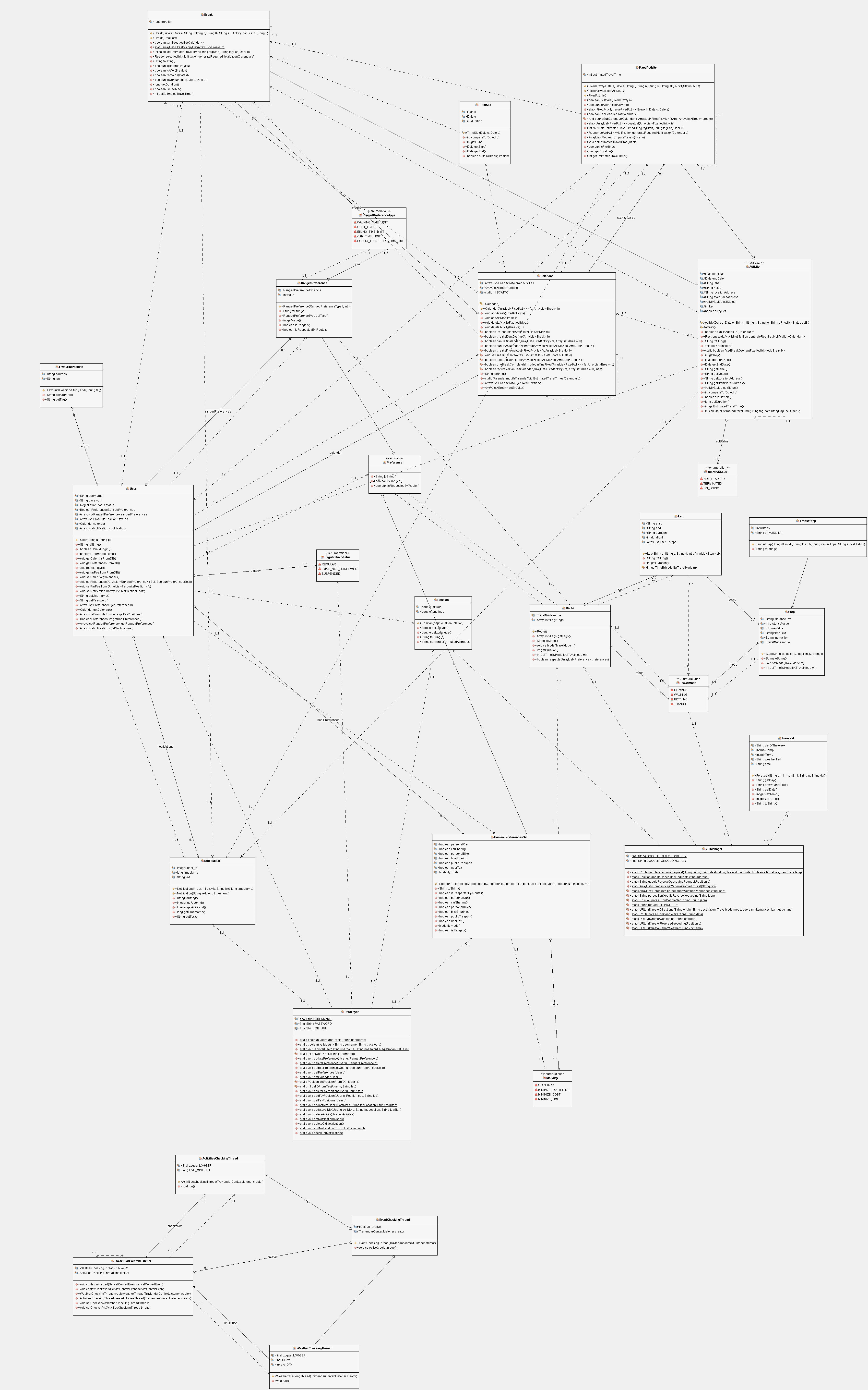
This is achieved by [F11].

* **[R10]** Users should receive a notification (e.g. email, push notifications) a little before the time they have to leave to go to the next appointment.

This is achieved by [F10].

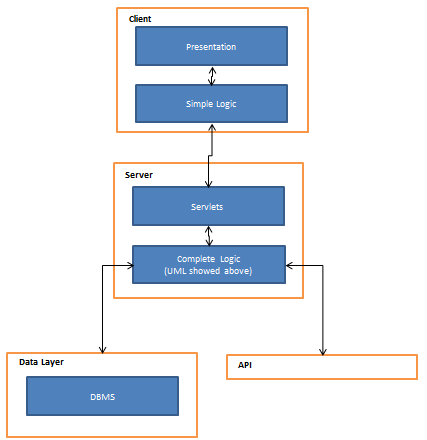
**4. Structure of the source code**

We developed three independent projects: “Travlendar Server”, “Travlendar Desktop Client” and “Travlendar Android Client”. Here’s the UML diagram of the **server’s logic**, which is one of the most important parts:

****

Note that in the diagram we represented only classes and packages that are relevant to the real comprehension of the structure. This means that we removed Exceptions, Servlets and Response objects to keep only the logic in the diagram. The original diagram is way larger and a bit uncomprehensible, but for the sake of completeness it’s available in the delivery folder.

The following simple schema, however, helps to understand the structure of the whole project and communications:



As said in the schema, the UML diagram showed above represents the most important part of the “Complete Logic” component. When the server receives a request, the receiving servlet just “delivers” to the request to the logic and then takes the answer and sends it back to the client.

The “Complete Logic” component is able to query the Data Layer and the external system’s APIs through the classes “Data Layer” and “API Manager”, that offer the necessary query methods and that are both showed in the UML diagram. Those two classes completely encapsulate the requests to the related external systems so that the logic doesn’t have to care about the connection details.

Due to the simplicity of the clients, their structure is not represented here.

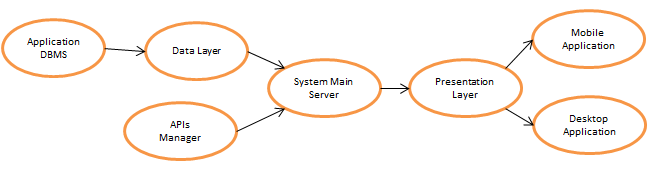
**5. Test Plan** *[Made-up prior to the starting of the Implementation - Reported from DD]*

Considering the diagram showed in the **2.2 System Overview** section of the DD 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:

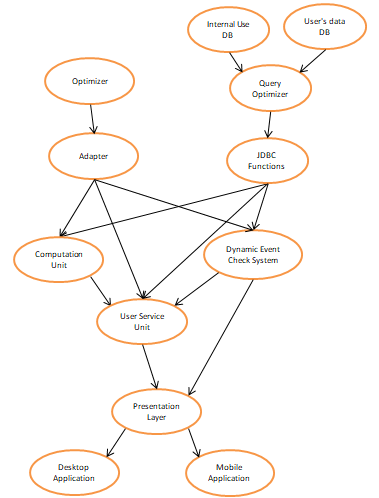
* **Application DBMS** is independent;
* **Data Layer** depends on **Application DBMS**, since it has to build queries and send them to the **Application DBMS** component;
* **System Main Server** depends on the **Data Layer** and **APIs Manager**, since it needs both of them to perform its operations (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** implementsall 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.

**6. Performed Tests and Analysis**

**6.1 Unit Tests**

Among the classes of the Complete Logic showed in section 4, we performed unit tests where relevant. To be more precise, we tested the following classes:

* Calendar;
* FavouritePosition;
* RegistrationStatus;
* User;
* BooleanPreferencesSet;
* Modality;
* RangedPreference;
* RangedPreferenceType.

In particular, along with Calendar, we also tested Activity, FixedActivity and Break. Those classes include the most relevant and complicated part of the complete logic, because they contain methods necessary to exploit and verify all the conditions necessary to schedule, modify and remove activities. Furthermore, they’re also responsible of generating the notifications of all the various cases that may happen when adding/updating an activity.

During the test process we were then able to locate bugs and, by observing execution time of tests, to find the most complex procedures and optimize them so that we can exploit all the computations in a fair amount of time, in order to avoid users having an endless wait just to schedule an activity.

Considering the percentage of tested code, the nature and the importance of the unit-tested modules , we’re quite confident in saying that those stand-alone modules are bug-free.

**6.2 Integration Testing**

**6.3 Performances Evaluation**

Considering that our server should be ready and able to satisfy requests from **several** users, it’s understandable that **performances**, and in particular **response time**, represent a significant quality indicator for our application.

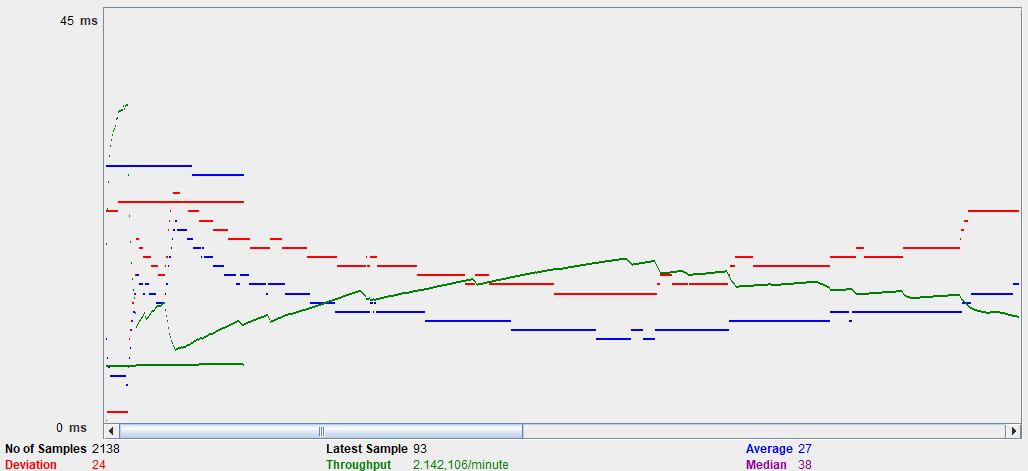
We performed the following tests:

* **Single Login Response Time**

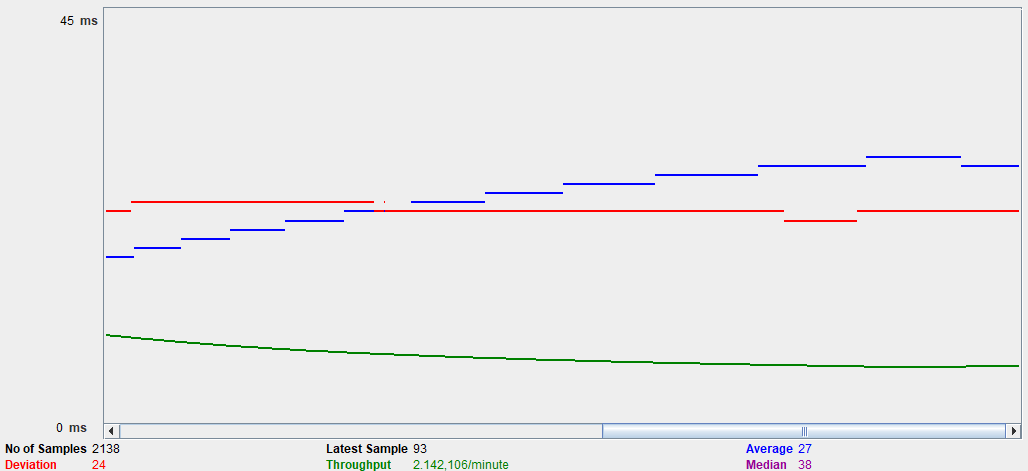
This is not extremely significant when evaluating performances of an application that should serve thousands of users at the same time, but it is a nice starting point and a good basis for the other tests because we can have a “measurement basis” thanks to this. A single login just requires a few simple queries to the database so thanks to this we know how long it takes to satisfy an “easy” request and we can compare this with the other results.

In the graph showed below it’s possible to observe the evolution of the average response time (blue) and the Throughput (green)

Beginning of the test:



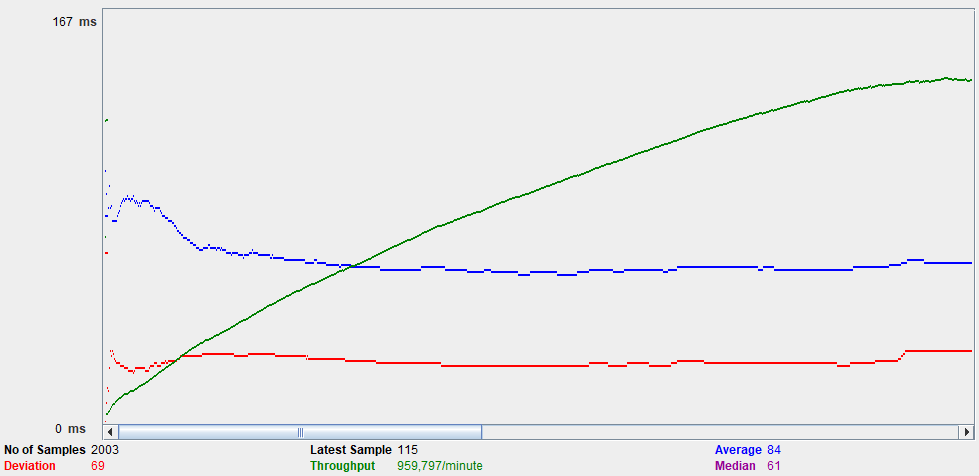
Asymptotically:



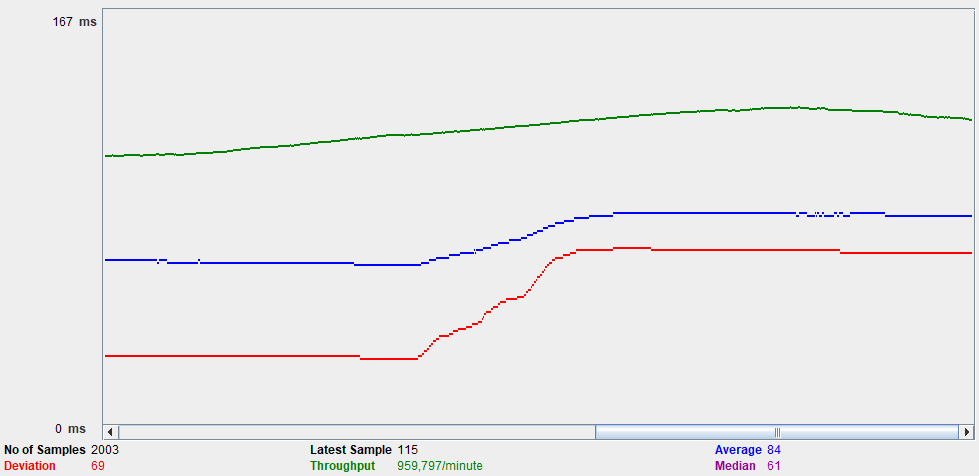
After stabilizing, the Throughput is such that a single login takes about 4.7 milliseconds.

* **Massive Login Response Time**

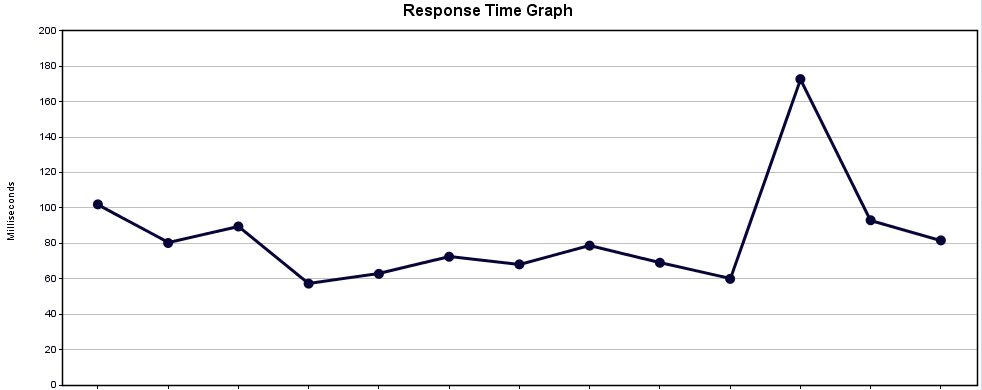
To have a comparison with the previous test, we tried then to send 1 thousand of login requests per minute (with peaks at the beginning and at the end of the minute) and we had the results are shown below.



Asymptotically:



Response Time Graph:



In the Graph, each step is 10 seconds. The peak is one of the moments almost all the requests have been sent together and it’s observable that requests are still satisfied in an acceptable time.

Some more general statistics:

