Software Development Plan for Git Dashboard

Version 1.1

Document Created By:
Mexican Royal Rumble

Team Members and contacts:
Ana Matilde Branco da Conceição
Contact: matilde.branco96@gmail.com

Christopher Jin Liu
Contact : liuchristopher1993@gmail.com

Emanuel dos Santos Matos
Contact: emanuel.24@hotmail.com

Guilherme Carlos Mateus Peixoto
Contact: gpeixoto@student.uc.pt

Marco André de Matos Pereira Gomes
Contact: mamgomes@student.dei.uc.pt

Maria Beatriz Monteiro Araujo
Contact: beatriz.araujo96@hotmail.com

Nuno Miguel Mota Gonçalves
Contact: nmmg@student.dei.uc.pt

30/12/2016

This page intentionally left blank.

Table of Contents

Section 1 - Scope **5**

1. Introduction 5

1.1 Purpose 5

1.2 Document Conventions 5

1.3 Intended Audience and Reading Suggestions 5

1.4 Product Scope 5

1.5 References 5

2. Overall Description 6

2.1 Product Perspective 6

2.2 Product Functions 6

2.3 User Classes and Characteristics 6

2.4 Operating Environment 6

2.5 Design and Implementation Constraints 6

2.6 User Documentation 7

2.7 Assumptions and Dependencies 7

3. External Interface Requirements 7

3.1 User Interfaces 7

3.2 Hardware Interfaces 7

3.3 Software Interfaces 7

3.4 Communications Interfaces 7

4. System Features 7

4.1 File Hierarchy Navigator 7

4.2 Artifact’s History Visualization 8

4.3 User login 8

4.4 View Master Branch Commits 8

4.5 Commits per user 9

5. Other Nonfunctional Requirements 9

5.1 Performance Requirements 9

5.2 Safety Requirements 9

5.3 Security Requirements 9

5.4 Software Quality Attributes 10

5.5 Business Rules 10

Section 2 - Architectural & Design Specifications **11**

1.Overall description of software environment 11

2.Detail of the system structure in layers Image 12

4.What is the structure of the code? What are the major components? How are they interconnected? 14

Section 3 - Implementation **15**
1. Project Overview 15
2. Software Process Model 15
2.1 Scrum 15
2.2 Extreme Programming (XP) 15
2.3 Test-Driven-Development (TDD) 15
3. Technologies and tools 16
3.1 IDEs 16
3.2 Code Collaboration 16
3.3 Documentation 17
3.4 Testing 17
3.5 Backend 18
3.6 Frontend 18
3.7 Group Communication 18
3.8 File Sharing 18
4. Conventions 19
4.1 Code 19
4.2 Branching 19

Section 4 - Quality Assurance Specifications **21**
1. Software Quality Standards 21
1.1 Metrics 21
2. Development Cycle 21
2.1 Agile Methodologies 21
2.2 Code Conventions 21
2.3 Design Patterns 22
2.4 Documentation 22
3. Testing 22
3.1 Types 22
3.2 Avoiding Biases 22
4. Continuous Integration 23
4.1 Automatic Procedures 23
5. Code Reviews 23
5.1 Before Merging 23
5.2 After Merging 23
6. Code Inspection 24
7. Static Analysis 24
8. Logging 24
8.1 Our API instance 24
8.2 Runnable API instance 24
9. Acceptance Tests 25
9.1 File Hierarchy Navigation 25

9.2 Artifact's History Visualization 25

Section 5 - Considerations **26**

5.1 Architectural & Design Specifications 26

5.2 Software Requirements Specification 26

5.3 Implementation 26

5.3.1 Problems 26

5.4 Quality Assurance Specifications 27

5.5 Team behaviour 27

5.5.1 Task management 27

5.5.2 Performance 27

5.5.3 Cohesion 29

5.5.4 Final thoughts 30

Section 1 - Scope

1. Introduction

1.1 Purpose

The purpose of this document is to present a detailed description of the Git Dashboard software engineering project. It will explain its purpose, functionality and overall architecture.

1.2 Document Conventions

Throughout the whole document we may:

1. Refer to a “version control system” simply as a “VCS”.
2. Refer to our team’s implemented user stories as “IDUS” (internally developed user stories).
3. Refer to other team’s implemented user stories as “EDUS” (externally developed user stories).
4. Refer to a “virtual private network” as a “VPN”.
5. Refer to “departamento de engenharia informática” as “DEI”.

1.3 Intended Audience and Reading Suggestions

This document is intended for both developer teams and project managers. This version of the document contains a basic overview of the project.

1.4 Product Scope

Our application will be a Dashboard for the Git VCS. It will be designed to maximize productivity of a software development team by providing tools to assist on project management. More specifically, it’s designed to help a team visualize and evaluate their project as it progresses, to enable early identification of possible improvements.

1.5 References

None.

2. Overall Description

2.1 Product Perspective

This dashboard is a new way to work with Git, as it adds features that don't natively exist on such VCSs. The whole project can be separated into 2 main components: the front-end, as a webpage that will display meaningful information to the user; the back-end, which will perform all necessary calculations from the information stored in a Git repository and provide the front-end with data.

2.2 Product Functions

- IDUS:
    1. File hierarchy navigation - displaying all the files and directories in a Git repository's path.
    2. Artifact's history visualization - displaying the commit history for any given file.
- EDUS:
    1. User login - allowing a user's login.
    2. View master branch commits - see all master branch commits
    3. Commits per user - see all commits per user

2.3 User Classes and Characteristics

TBD

2.4 Operating Environment

The IDUS were developed to work as a service. Having that said, the components that provides all the necessary information are running on an external machine, which can be accessed from any given environment. The fronted we provide, on the other hand, is a webpage that can run on any given modern web browser (such as: Safari, Chrome, Firefox, Microsoft Edge, etc...).

2.5 Design and Implementation Constraints

As a service, our IDUS run on a Java backend. This is then accessed from an HTML / JavaScript fronted, which poses no apparent constraints to the implementation. Also, this frontend can be easily replaced with any other kind of interface (developed in any programming language) without the need to fiddle with the backend.

2.6 User Documentation

All our documentation can be found at es2016.dei.uc.pt. Keep in mind that this website can only be accessed from within DEI's network (or through a VPN). More specific documentation can also be found in our repository as JavaDoc.

2.7 Assumptions and Dependencies

Our implementation assumes that the user will have an active internet connection and that there is a proper backend running (either locally or externally). All other dependencies are handled internally in the backend, on a remote machine.

3. External Interface Requirements

3.1 User Interfaces

Simple webpage developed with HTML / JavaScript / CSS.

3.2 Hardware Interfaces

None.

3.3 Software Interfaces

Rest API running on an external server.

3.4 Communications Interfaces

GitLab's Rest API.

4. System Features

4.1 File Hierarchy Navigator

4.1.1 Description and Priority

Container used to display all the folders and files located on a given Git repository.
High priority

4.1.2 Stimulus/Response
Sequences by clicking on a folder on the left-hand side panel, all its subfolders and files will be displayed on the right-hand side panel.

4.1.3 Functional Requirements
This feature needs a connection with git so you can navigate and see the files in a repository.

4.2 Artifact's History Visualization

4.2.1 Description and Priority
Container with information regarding commits' history of a file.
High priority

4.2.2 Stimulus/Response
Sequences Non-interactive component. Gets updated with changes pushed to the repository.

4.2.3 Functional Requirements
This feature needs a connection with git so it can access the files' commit history.

4.3 User login

4.3.1 Description and Priority
Container where the user can input it's login data to enter the project's main page.
High priority

4.3.2 Stimulus/Response Sequences
The user inputs it's username and password so he can click on the 'login' button to enter the project's main page. If the asked data is wrong there will appear a error message.

4.3.3 Functional Requirements
This feature needs a connection to the internet so it can check if the asked data is valid or not.

4.4 View Master Branch Commits

4.4.1 Description and Priority
Container with information regarding each commit made in the master branch of the project.
Medium priority

4.4.2 Stimulus/Response Sequences
Non-interactive component. Gets updated with changes pushed to the project's master branch.

4.4.3 Functional Requirements
This feature only needs a connection with git so it can access the project's master branch commit list.

4.5 Commits per user

4.5.1 Description and Priority
Container with information regarding each commit made by the selected user.
low priority

4.5.2 Stimulus/Response
Sequences Non-interactive component. Gets updated with changes pushed to the repository.

4.5.3 Functional Requirements
This feature needs a connection with git so it can access the user commit list regarding the project.

5. Other Nonfunctional Requirements

5.1 Performance Requirements

None.

5.2 Safety Requirements

The frontend should be able to handle backend failures. The backend should be able to handle GitLab's API failures.

5.3 Security Requirements

None.

5.4 Software Quality Attributes

Software quality shall be discussed in a later section.

5.5 Business Rules

None.

Section 2 - Architectural & Design Specifications

1.Overall description of software environment

The project will be divided into two major components:

- Front-end: web page created using Bootstrap, along with custom HTML/CSS/JavaScript.
- Back-end server created using Java. It will make use of two main different technologies:
    1. Jersey - used to implement a RESTful API.
    2. GSON - used to serialize/deserialize JSON objects.

Our **front-end**, to control the graphical user interface (GUI), will:
1. Send HTTP requests to the back-end.
2. Wait for a JSON reply.
3. Deserialize it.
4. Populate the HTML.

On the other hand, our **back-end** will, simultaneously:
1. Listen to HTTP requests from the front-end.
2. Interpret the request.
3. Make all necessary calls to the GitLab API.
4. Bundle the relevant information into Java objects.
5. Serialize them to JSON.
6. Send them to the front-end.

Moreover:

1. JUnit will be used server-side to test all the implemented functionality, along with continuous integration using GitLab CI (running on GitLab.com's shared runners).
2. Log4j2, also server-side, will be used for system logging.
3. Maven will be used to build, trigger unit test runs, execute and manage backend's dependencies.

2.Detail of the system structure in layers Image

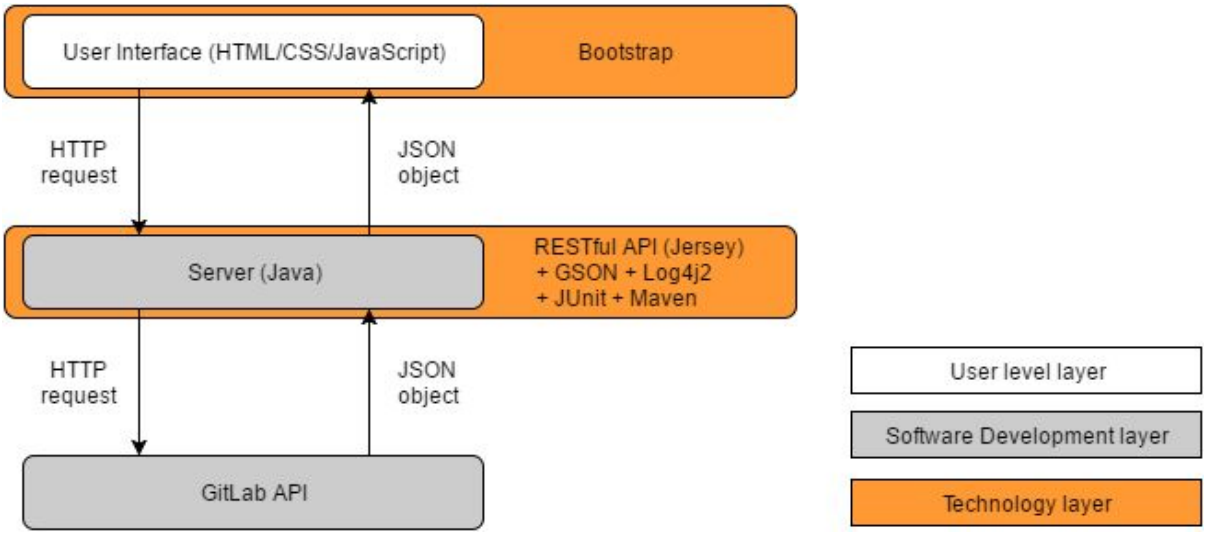


Figure 1. High Level Diagram Image

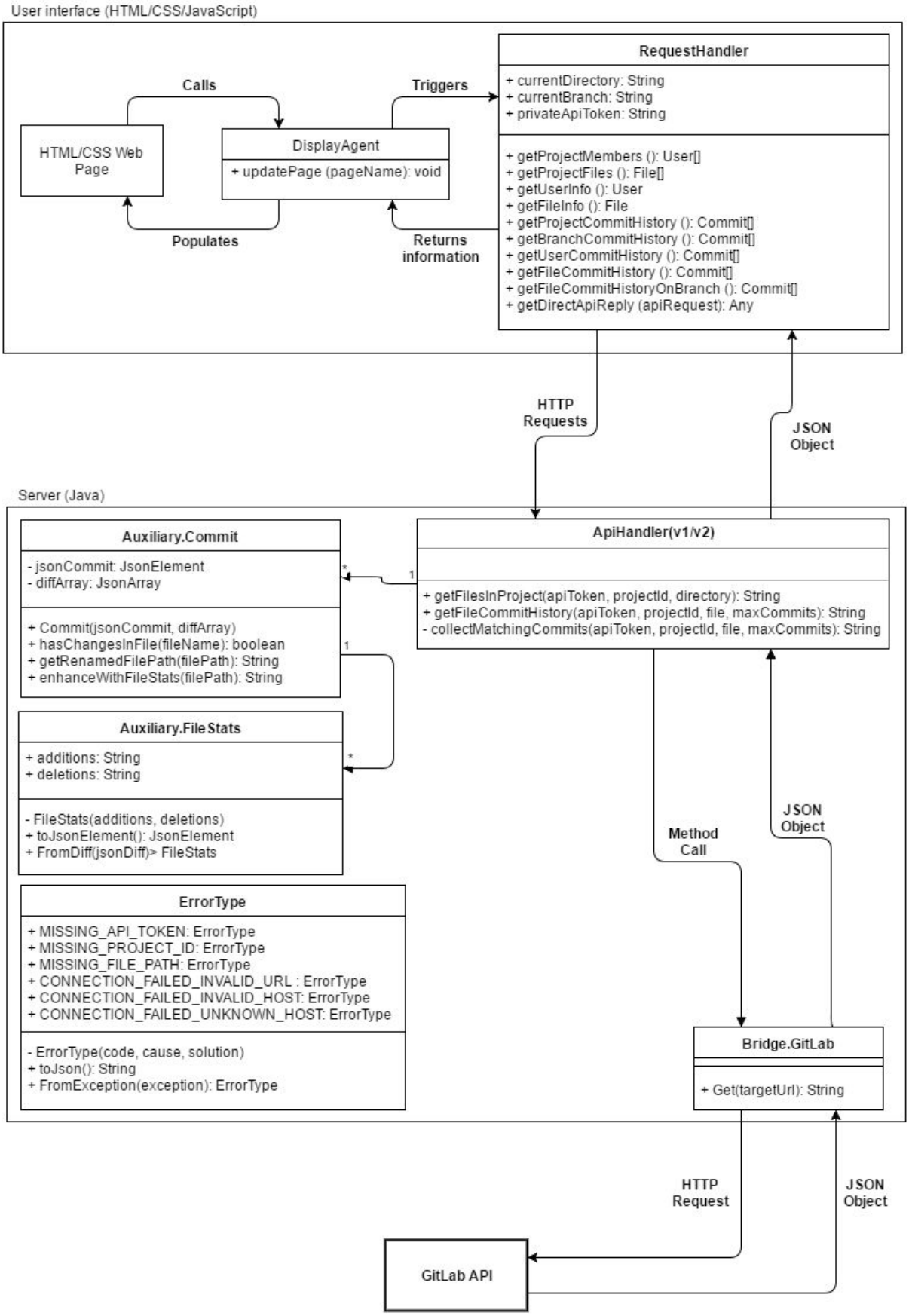


Figure 2. Low level diagram

4.What is the structure of the code? What are the major components? How are they interconnected?

The overall idea is already present in *Figure 2. Low Level Diagram* but there are some aspects worth clarifying:

1. There are two versions of the API (one supporting git.dei.uc.pt and the other supporting gitlab.com). Each of these versions is controlled by its own handler class (ApiHandlerV1 and ApiHandlerV2, specifically). These two have very similar structure and identical functionality and their differences are transparent to the user.
2. There is a class, Bridge.GitLab, that acts as a middleman between our API and GitLab's. It also takes care of performing all the necessary checks and error handling caused by this external component.
3. ErrorType is used by Bridge.GitLab and/or ApiHandler to inform the user of any error that occurs.
4. The path of any request until completion is as follows:
    - *RequestHandler -> ApiHandler(v1 or v2) -> Bridge.GitLab -> GitLab API -> Bridge.GitLab -> ApiHandler(v1 or v2) -> RequestHandler*

Section 3 - Implementation

1. Project Overview

This project can be separated into two main components: (i) a Git dashboard using web technologies and (ii) a RESTful API implemented in Java. Basic functionality is based on a request sent from the dashboard to our API which, in turn, returns useful information from a given Git repository. Although the number of components could be minimized (by implementing every feature directly with web technologies), running our API on a remote server will allow other groups to use it as a service - without the need to copy & paste code. Taking the above into consideration, our group's members are mainly working in one of the described components. Due to language requirements, some conventions and/or tools may vary but these will be discussed in the next few sections.

2. Software Process Model

In this project we decided to use a mix of Agile methodologies:

2.1 Scrum

● Used throughout the whole project.
● Standard Scrum roles: (i) product owner, (ii) scrum master and (iii) scrum team member.
● Weekly Scrum stand-up meeting (~5 min per person).
● Mainly one week sprints (with exceptional two week sprints).
● Small iterations of: (i) code development, (ii) testing and (iii) code refactoring.

2.2 Extreme Programming (XP)

● Pair programming used in web development.
● Fifteen to twenty minute shifts.

2.3 Test-Driven-Development (TDD)

● Although it's not currently used by any of the group's members, we intend to integrate it with some of the development sections of the project.
● Small iterations of: (i) test development, (ii) code development, (iii) code refactoring. Some things to keep in mind regarding the methodologies described above:
● During weekly scrum meetings, each person is given 5 minutes of uninterrupted speech. Each member explains what he/she did during the previous week, the problems he/she had

and possible improvements to his/her work. No other member is allowed to intervene or make any comments (it's supposed to be a monologue, not an open discussion).
● During Scrum iterations: (i) code development should be a small increment in functionality; (ii) code testing should cover the as much of the implemented functionalities as possible; (iii) code refactoring has to be done to both test and source code accordingly.
● In pair programming, the developer that is reviewing the code is not allowed to physically interfere with the other's work - he/she cannot type any lines of code, only suggest different approaches/solutions.
● During TDD iterations: (i) all written tests should be failing test and simply target a small increment in functionality; (ii) developed code simply aims to make all tests pass, nothing more; (iii) refactoring similar to the one described in the Scrum methodology.

3. Technologies and tools

3.1 IDEs

As it is, the project setup allows for painless collaboration with several IDEs

3.1.1 Intelllj

The IDE of choice in our team. No plugin installation is necessary, since it already comes bundled with Maven.

3.1.2 Eclipse

Using this IDE you will need to setup Maven manually. As an alternative, you can always install M2Eclipse.

3.1.3 Other

As it is, you can use any other Jetbrains' IDE to contribute to this project. Disclaimer: We currently do not use or encourage the use of NetBeans.

3.2 Code Collaboration

For this specific topic we use a series of tools to ease integration between team members' work and maximize the reliability of our production code.

3.2.1 Git

We are currently using Git, a third generation Version Control System, to handle contributions to our project. Also, our group opted for a feature branching approach. It basically dictates that all the changes related to a certain functionality have to be contained in a single branch and, only after it's fully tested and validated, merged with the 'master'

branch. Also, the ‘master’ branch is protected and doesn’t allow any direct contributions or changes to it.

3.2.2 Continuous Integration (GitLab CI)

We set up a continuous integration GitLab CI instance to continually build and test our pushed code. The status of a specific run can be found as an icon next to a corresponding commit.

3.2.3 GitLab

We are using GitLab.com’s platform to handle merge requests, from implemented features. After a merge request is open, it can only be merged once a couple of conditions are met: (i) the project is building correctly; (ii) all the tests are passing; (iii) at least three members reviewed the code and accepted the request. If any of these are not met, GitLab.com will prevent the branch from being merged.

3.2.4 Trello

We are using Trello, instead of GitLab.com’s issue-tracking and milestone systems to assign/manage tasks for any given sprint and monitor development progress, because we found the later to be lacking in functionality/customizability.

3.3 Documentation

3.3.1 JavaDoc

Current documentation on the source code exists in this format - and will continue to exist for future API versions.

3.3.2 Web Page

All the documentation for every component of the project, plus examples, will reside in our website and written manually with Markdown syntax.

3.4 Testing

As we are using different languages in our project we needed to put in place two different tools. There is no hard limit for test code coverage, since this will greatly vary depending on the situation and will have to be considered by the developer. Despite everything, the test requirements for this project on either framework are the same: both unit tests and regression tests are mandatory.

3.4.1 JUnit

JUnit will be used to test all our backend Java code.

3.4.2 QUnit

QUnit seems to be a good choice to test our Javascript code, although we may consider other alternatives in the near future.

3.5 Backend

This component of the project requires its own set of technologies to implement the desired functionality.

3.5.1 Jersey
Will be used to implement a RESTful API, similar to GitLab's.

3.5.2 GSON
Will be used to serialize/deserialize JSON objects server-side.

3.5.3 Log4j2
Will be used for logging.

3.5.4 Maven
Will be used to trigger unit tests, compile, build and run the API.

3.6 Frontend

None of the technologies used in the frontend are functionally necessary but we thought could be visually interesting.

3.6.1 Bootstrap
Used to create the group's webpage and Git dashboard (final project).

3.7 Group Communication

3.7.1 Slack
It is used both as a communication means and team-event scheduler.

3.8 File Sharing

Apart from code collaboration, some assets and reports also need to be easily accessible to every team member. To avoid merge conflicts problems through Git (caused by binary files' metadata) we decided to use other technologies.

3.8.1 Google Drive

It is used as a means to share temporary assets (like images) as well every team-related document (created using any of the Google Docs' technologies).

3.8.2 Google Docs

Using Google's sharing platform:

- Sheets is used for statistics calculations and graph creation;
- Docs is used for weekly report and milestones creation;

4. Conventions

4.1 Code

Below are presented all the code conventions used for all the different programming languages used in our implementation (Java, Javascript and HTML). To make it easier, instead of defining our own conventions, we took advantage of existing public resources to guide our development.

4.1.1 Java

We are currently following Oracle's Java code conventions.

4.1.2 Javascript

We are currently following w3schools' Javascript code conventions.

4.1.3 HTML

We are currently following w3schools' HTML code conventions.

4.2 Branching

In our feature-branching approach on Git, described previously, we decided to define conventions for both a branch's lifecycle and naming.

4.2.1 Life Cycle

● Whenever you are working on a different feature, or simply extending an existing one, create a new branch.
● Don't code different features on the same branch, keep every branch as modular and simple as possible.
● As soon as a feature is fully implemented, open a merge request and wait for other team members to comment.
● After getting all the necessary fixes in place, merge your branch and fix any conflicts.
● Do it all over again.

4.2.2 Naming

● Always use one of three keywords as the prefix for any branch name: core, extra or experimental.
● Every word contained in a branch's name should be separated with an underscore, like “core_branch”.
● All the letters in a branch's name should be lowercase.
● The name of a branch should let everyone know what is being done in there.

Section 4 - Quality Assurance Specifications

1. Software Quality Standards

1.1 Metrics

We do not use developer-defined quality metrics, such as a test suite's code coverage, as these may lead to unrealistics *perfect* expectations of the final product. Instead, we base the product's metrics on its requested final state. In other words, if it satisfies the initially established requirements and core functionality, it's a *perfect* product.

2. Development Cycle

To improve the quality of our product, we carefully defined our processes according to the project's needs, right from the beginning of the development cycle:

2.1 Agile Methodologies

- Both Scrum's and Test-Driven Development's (TDD) small iterations keep every new functionality tested.
- Both Scrum's and Test-Driven Development's (TDD) small functionality increments allow for easier backtracking and debugging.
- Extreme Programming's (XP) pair programming allows for better code quality and structure from an earlier stage.

2.2 Code Conventions

- Well defined code conventions allow for better code consistency, readability and maintainability.
- Using widely accepted code conventions, such as Oracle's Java conventions, takes advantage of their reliability - refined by many, over years of experience.

2.3 Design Patterns

- Along with code conventions, design patterns improve code consistency, readability and maintainability, along with a more structured and simpler design.
- Using widely accepted design patterns, such as Gang of Fours' design patterns, takes advantage of their cross-team structural understanding.

2.4 Documentation

- All the code created and integrated with our project should be well documented, as no code can be considered *good* without it.
- Our group presents people - completely unrelated to the development process - to our documentation looking for feedback, to understand how transparent our project's structure is.

3. Testing

We develop tests for every component of our product, with different purposes and coverages, to find and fix as many problems as possible at an early stage.

3.1 Types

- Unit tests
- Regression tests
- Integration tests

3.2 Avoiding Biases

- Our tests check the expected boundaries' values - **exact boundary** values, **outside boundary** values and **inside boundary** values - against its expected behaviour.
- Some tests use **arbitrary values** as input for our modules - when there are so many possible scenarios it isn't a viable option to test all of them - which prevents the programmer from specifying subsets of pre-defined tests.
- Tests may be written by a person unrelated to a module's development - to avoid *"my baby" syndrome*.

Also, our test suite reflects our product's requirements through tested functionality, which allows us to keep track of the overall product's quality during development.

4. Continuous Integration

4.1 Automatic Procedures

- At every push to our repository, a *verification* and a *validation* steps are ran, detecting any and every failing builds/tests as soon as possible.
- If the above step detects a problem in our code, the branch where it exists cannot be merged until fixed - and the developer gets notified immediately.
- As more and more tests are introduced in our test suite, running the *validation* step over the entire set will become impossible. To maintain product quality, while not clogging up development, we will then select a subset of core test to run as *smoke tests* - still ran at every push to our repository - only running the remaining when a *merge request* is opened - keeping possible mistakes from getting into production.

5. Code Reviews

The reviewing stage takes place once a team member opens a *merge request* and can be subdivided into two steps:

5.1 Before Merging

- Reviewing can only start once both *verification* and *validation* steps pass, for the target branch.
- Every software developer in the team should review and comment on the open *merge request*.
- No *merge request* can be accepted while there are still unresolved comments.
- At least three team members have to accept the changes.

5.2 After Merging

- Every software developer should review and comment on the merge commit - or subsequent conflict fixes.
- The developer responsible for the *merge request* can only start a new iteration once there are no more unresolved comments.

6. Code Inspection

We use something similar to Fagan's inspection methodology, according to certain criteria:

- We only do it when a broken functionality is detected, by a failing test.
- We apply it to the smallest subset of code possible - the code causing the failing test and its immediate dependencies.

Although this approach takes away the inspection's advantage of finding errors earlier in the development process, it is more cost-efficient. Nevertheless, any improvement or restructuring made to the targeted subset of code will then still be propagated as needed - by each developer independently.

7. Static Analysis

As many of these tools would require a developer to get familiar with formal verification beforehand - to be able to verify more complex theorems in our code - we thought this approach would not be feasible/advantageous in the available time (as a good formal verification of a smaller subset would be time-equivalent to better testing and inspection of wider portions of code, the compromise *reliability vs. coverage* had to be made).

8. Logging

Even after all of the above steps, after the code gets into production and eventually distributed, we still monitor its behaviour through logging. This is so we can more easily maintain a reliable product and constantly adapt it to a user's needs.

8.1 Our API instance

On our running instance of the API, we immediately use our logging system as a means to detect runtime problems, undesired program states or issues with any given module or package.

8.2 Runnable API instance

As we also provide a runnable version of our API (if any given client wants to run his own local instance), anyone can submit the log files created automatically on his machine, for our future inspection.

9. Acceptance Tests

9.1 File Hierarchy Navigation

1. Open dashboard page.
2. The page should have a navigation menu on the left with three elements:
   - Dashboard;
   - Files;
   - Graphs;

3. The page should also display every team member inside a box, on the right.
4. You can scroll on ‘Team Members’ display box.
5. Click the ‘Files’ button on the navigation menu.
6. In the middle of the page, you will see a container with every file/folder in the current directory, with an icon on the left and their respective names on the right.
7. Click on any folder to see its contents.
8. Verify change in location (files and folders displayed should be different).
9. Click ‘...’ folder to go back.
10. Verify change in location (files and folders displayed should be different).
11. Hover any file with mouse.
12. The text ‘Ver Commits’ should appear below each file on hover.

9.2 Artifact’s History Visualization

1. Follow every step in ***8.1*** above, until step ***6***.
2. Click ‘Ver Commits’ text below a file.
3. All of the files previously displayed should disappear and a timeline should be displayed in the same navigation division.
4. The timeline should be a centered vertical white line with red boxes on its left and right side and a red ‘Now’ circle centered on top.
5. Each red box should have information pertaining to the clicked artifact’s commits (code, date and name).
6. Scrolling should be possible for further reading, if there are enough commits beings visualized.
7. Hover ‘go back’ button with mouse.
8. A color inversion should happen on the same button.
9. Click the ‘go back’ button.
10. The timeline should disappear and the centered display division should change to the previous folder and file display (file hierarchy navigation).

Section 5 - Considerations

// TODO due to lack of time, sometimes team kept building features on the same branch to avoid opening a merge request and waiting for reviews
// TODO due to lack of time, some features went by poorly tested and got into production
// TODO although some of these setbacks, our team tried to follow the defined processes as well as it could
// TODO the granularity of the tasks sometimes made it tougher to make them completely independent

5.1 Architectural & Design Specifications

Despite the fact that our group was basically one of the few to opt for such an architecture, as initially described in *M1*, we managed to follow what was initially planned with two minor differences:

1. **[Improvement]** We eventually replaced every dependency with a single one, *Maven*. This technology allowed us to create a single bundle of dependencies, managed from within the project itself, hugely simplifying the project's setup process.
2. **[Improvement]** We simplified the backend packaging and kept only strictly necessary structures and classes.

All of these changes reflected on the updated version, in *Section 2*.

5.2 Software Requirements Specification

This component of the project had to be revised, since new course objectives were defined. All of the changes can be checked on *Section 1*.

5.3 Implementation

Our group basically used *Section 3* as a guidebook throughout the whole development process. This also had to be updated (to include group's file sharing technologies).

5.3.1 Problems

Although we all did our best to follow these rules, some aspects still deviated a bit from their intended course of action:

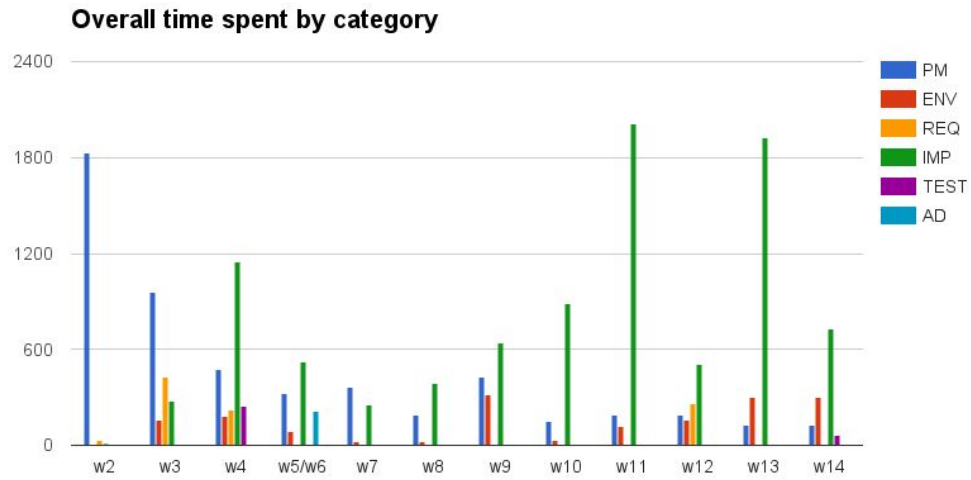
1. Due to time constraints, sometimes team members kept building features on the same branch to avoid opening a merge request and waiting for code review.
2. Due to time constraints, some features went by poorly tested and got into production.

5.4 Quality Assurance Specifications

With a concern for post-distribution product quality, we updated *Section 4* with logging technologies, used to keep everything under control. This was done to better fix and adapt the product to the user's needs.

5.5 Team behaviour

5.5.1 Task management



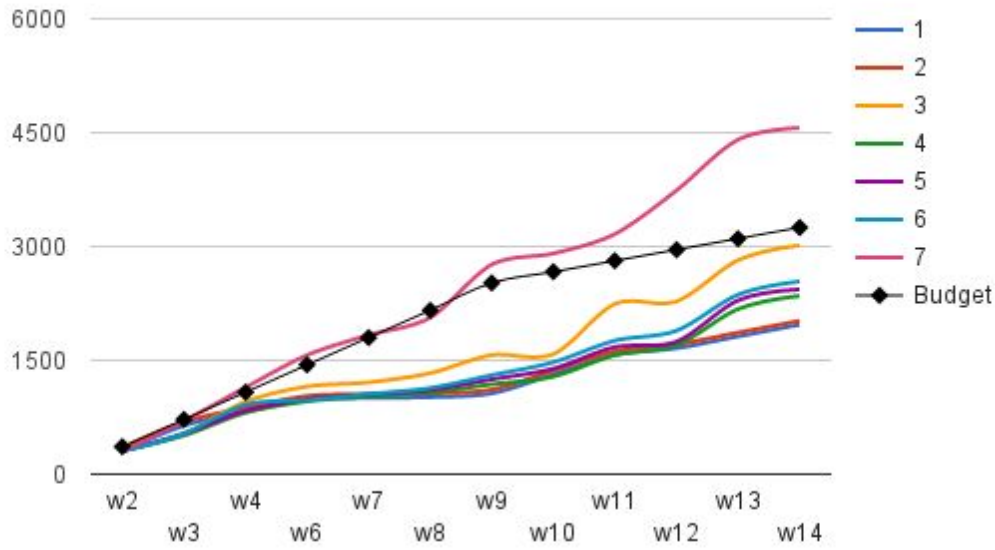
Analysing the above chart, we can see a couple of things:

[Good] The team shifted their efforts from *PM* and *REQ* to *IMP* and *ENV* early on, which would mean their goals were well set and processes defined.
[Bad] Using the methodologies defined in *Section 3*, every iteration should bring along both *IMP* and *TEST*, which isn't the case (the group clearly neglected testing).
[Neutral] As we kept a constant focus on improving and developing the backend for our IDUS, we also kept bringing up *ENV* throughout the development process. We don't consider this to be either a good or a bad sign, simply shows some changes needed to be made in order to better allocate for the constant updates.
[Sporadic] A focus on *REQ* surfaces again on week 12, caused by the revised project's goals and functionality.

5.5.2 Performance

Considering we just barely met the requirements, any time spent on other activities instead of implementation (even if testing) would mean the goals would not be met. Having said that,

this does not necessarily mean the workload was too heavy, it may mean that the group simply didn't work enough - which can be seen in the below graph.

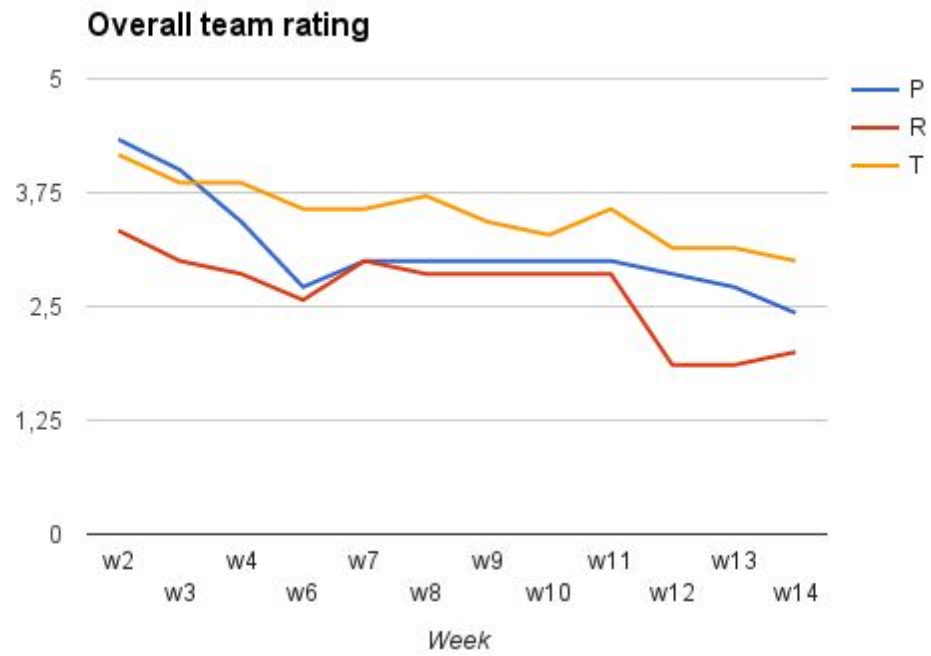


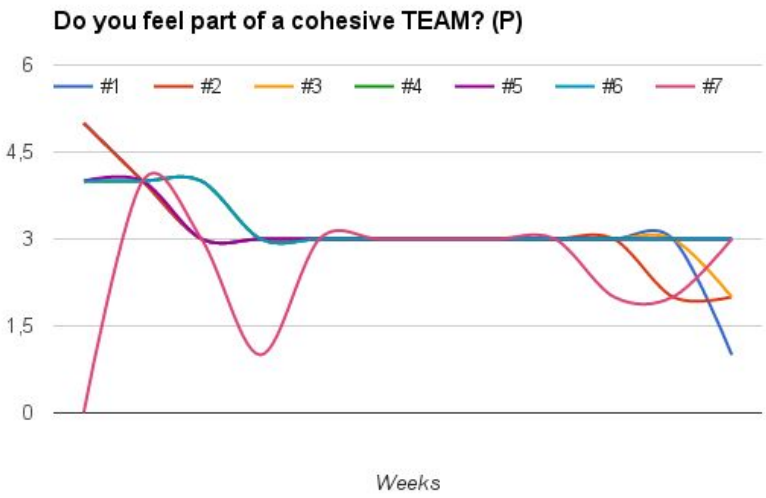
Looking at the calculated budget, even after we gave up on the steady 6 hours per week budget and started using our predicted workload as a metric, every team member was still below the *budget* line (except for member 7). <u>This alone does not mean the team didn't work as much as it should</u>:

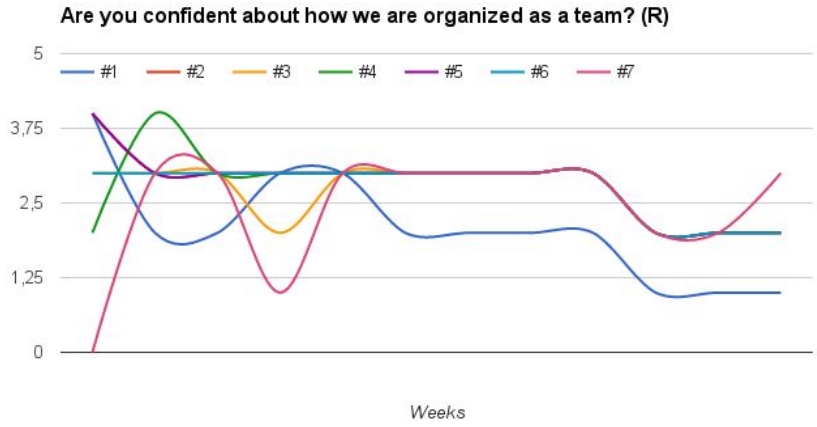
1. **[Ideal scenario]** It could mean that we basically overestimated the time we actually needed, performed better than expected, perfectly scheduled the tasks throughout the weeks and managed to finish everything on time.
2. **[Actual scenario]** Knowing the end state of the project and that many assigned tasks were left *OPEN* (of which 25% belonged to the *TEST* RUP category), it shows that the effort put into the project was just lacking.

5.5.3 Cohesion

Below are the graphs for the team members' team evaluation:







Looking at the above graphs *P* and *R* we can infer that there was a bit of a conflict at the beginning, although we guessed it would be normal for a newly assembled team. For the same graphs, these values then converged to more or less similar values at around week 5 or 6, which were upset a couple of weeks later by the introduction of new project goals.

As for graph *T*, the values kept oscillating between 3 and 4 until the introduction of a destabilizing factor (new project goals). Even then, the values still converged a couple of weeks later.

From this, it doesn't seem that there was any clear indicator of an overly-dissatisfied team member, or an overly-satisfied one. The whole team seemed to be in sync when agreeing that something was clearly getting worse but no one felt like their own work wasn't paying off.

5.5.4 Final thoughts

Even though some problems could have been detected and prevented early on, it seemed like the team's heartbeat was similar to each of the separate members'. There was a problem that everyone was aware, in all three categories (*R, T and P*), but the group didn't know how to handle it or to adapt to the circumstances, which eventually led to our final result. Nevertheless we think that, apart from these problems, there were still clear indicators of lack of work and effort.

Finally, one big improvement could simply be following *Section 3* more strictly. By slacking on this component, eventually, product's quality gets affected and team morals may go down with an increased number of bugs / unexpected behaviours.