

# TQS: Quality Assurance manual

## Grupo 105

v0000-00-00

1.1	Team and roles	1
1.2	Agile backlog management and work assignment	1
2.1	Guidelines for contributors (coding style)	2
2.2	Code quality metric	2
2.3	Git Standards	2
3.1	Development workflow	3
3.2	CI/CD pipeline and tools	3
3.3	Artifacts repository [Optional]	3
1.1	Overall strategy for testing	3
1.	Functional testing/acceptance	4
2.	Unit tests	4
3.	System and integration testing	4
4.	Performance testing [Optional]	4

## 1 Project management

### 1.1 Team and roles

**Team manager:** Tomás Costa

**DevOps master:** João Marques

**Product owner:** Francisco Jesus

**QA Engineer:** Miguel Matos

**Developer:** Everyone

**Backend:** João Marques, Francisco Jesus

**Frontend:** Tomás Costa, Miguel Matos

### 1.2 Agile backlog management and work assignment

For backlog management we are using GitLab Boards and Milestones, assigning each task to a specific developer. We are experiencing GitLab instead of Jira, to expand our technology stack, and see how well it works.

Renti > Milestones > Milestone 2

Open Milestone May 10, 2020–May 13, 2020 Edit Close milestone Delete

### Milestone 2

Milestone for the 2nd deliverable week

Issues 18 Merge Requests 0 Participants 4 Labels 0

Unstarted Issues (open and unassigned)	Ongoing Issues (open and assigned)	Completed Issues (closed)
0	11	7

**Frontend-Mobile** - Setup Mobile Pipeline  
#3

**Docs** - Presentation for class  
#13

**Docs** - Next iteration planning  
#12

**Frontend-Web** - Setup of the initial user stories  
#1

**Frontend-Mobile** - Setup of the initial user stories  
#1

**Docs** - Define core user stories with priority  
#11

**Docs** - Define SQE Strategy  
#10

**Docs** - Quality Assurance Manual  
#9

**Docs** - Update Repository  
#8

**Backend** - Setup Backend Initial Pipeline  
#2

**Frontend-Web** - Setup Web Pipeline  
#3

**Frontend-Mobile** - Create project structure  
#2

**Frontend-Web** - Creation of project structure  
#2

**Docs** - General Branding  
#7

**Docs** - Backlog management and top user stories  
#4

**Docs** - Create group and divide in repositories  
#3

## 2 Code quality management

### 2.1 Guidelines for contributors (coding style)

Still a work in progress, but heavily inspired in [AOS project](#) with standards like:

- o Dont ignore exceptions
- o Don't catch generic exceptions
- o Defining fields in standard places
- o Using TODO Comments
- o Logging instead of printing
- o Using standard bracket style

Also some standards from “[Clean Code](#)”:

- o Avoid duplication anywhere in code
- o [Law of Demeter](#)

### 2.2 Code quality metric

We are making use of static code analysis engines to evaluate our code quality and prevent errors.

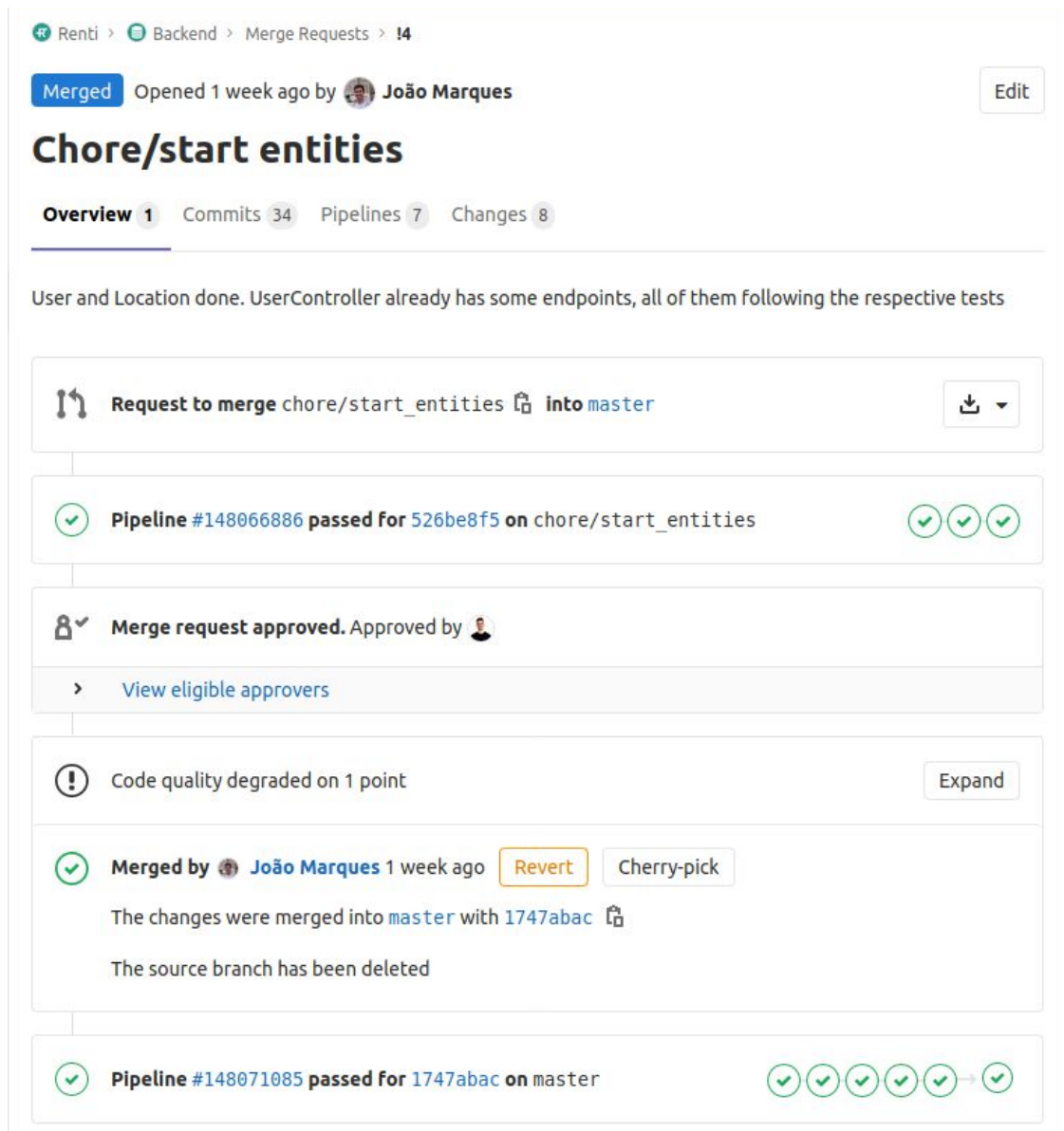
As of the specific code quality metrics, we opted by using [GitLab Code Quality](#).

This set of tests use [Code Climate Engines](#). This reports code errors, security problems, vulnerabilities, code smells, and other errors based on common standards, and easily integrates it with each commit and merge request in GitLab.

This code analysis is then integrated with the Gitlab CI pipeline for the project, as explained ahead.

Having access to this tool allows us to identify errors we probably would miss otherwise, and to learn how to write better code.

The fact that we use this specific static code analysis allows us to see problems in Code Quality associated with each Merge Request, as shown below:



The screenshot shows a GitLab Merge Request interface. At the top, it indicates the request is 'Merged' and was opened 1 week ago by João Marques. The title of the merge request is 'Chore/start entities'. Below the title, there are tabs for 'Overview' (selected), 'Commits' (34), 'Pipelines' (7), and 'Changes' (8). The description states: 'User and Location done. UserController already has some endpoints, all of them following the respective tests'. The main content area shows a 'Request to merge chore/start\_entities into master'. Below this, a pipeline status is shown: 'Pipeline #148066886 passed for 526be8f5 on chore/start\_entities', with three green checkmarks. A message indicates 'Merge request approved. Approved by [user]'. Below this is a link to 'View eligible approvers'. A warning icon indicates 'Code quality degraded on 1 point', with an 'Expand' button. The merge history shows it was 'Merged by João Marques 1 week ago' with a 'Revert' button and a 'Cherry-pick' button. It notes 'The changes were merged into master with 1747abac' and 'The source branch has been deleted'. At the bottom, another pipeline status is shown: 'Pipeline #148071085 passed for 1747abac on master', with six green checkmarks.

On the other hand, we also added a [SonarQube](#) static code analysis, in order to have more metrics and a more extensive project dashboard in terms of code quality. For this to work, we installed SonarQube in the provided server using docker, and programmed the GitLab CI pipeline to trigger an analysis in each commit.

## 2.3 Git Standards

GitLab was the obvious choice for the Git Platform since it has easier CI/CD Integration and our backlog management, which allows us to close tasks in commits. Some standards are:

- o Never merge directly, always make pull requests and identify at least one person to check (review) that pull request before merging the PR. (All repositories are configured to not accept a single person merge)

- o **New feature branch:** For each new feature create a branch following the standard: `feature/<feature_name>`.
- o **New Issue branch:** For each fix create a branch following the standard: `fix/<fix-name>`.
- o Closing issues/tasks can be done by writing in commit message: "this closes #<issue\_nr>"

### 3 Continuous delivery pipeline (CI/CD)

#### 3.1 Development workflow

[Clarify the workflow adopted [e.g.. [gitflow](#) workflow, [github flow](#) . How do they map to the user stories?]

[Description of the practices defined in the project for *code review* and associated resources.]

[What is your team "Definition of done" for a user story?]

We decided to adopt a standard Git Flow, which resides on the following:

1. Map each user story into one (or more) system feature(s).
2. Map each feature into a git branch.
3. Classify the feature as done when the expected system behavior is accomplished.
4. Merge the feature branch into the master/production branch.

#### 3.2 CI/CD pipeline and tools

[Description of the practices defined in the project for the continuous integration of increments and associated resources. Provide details on the tools setup and config.]

[Description of practices for continuous delivery, likely to be based on *containers*]

As in previous projects, we implemented a CI/CD workflow using Gitlab.

On each commit made to the code repository, a pipeline is triggered. This pipeline is responsible for conducting the previously mentioned **code quality static analysis**, and producing the respective report, as well as conducting **tests**, **building** the source code into **packages**, and continuously **deploying** the services.

#### 3.3 Artifacts repository [Optional]

[Description of the practices defined in the project for local management of Maven *artifacts* and associated resources. E.g.: [artifactory](#)]

The software elements are packaged into **docker containers**, used both to run the services locally, and in deployment. In the Gitlab pipeline, these docker images are automatically built and uploaded to a private **container registry**, making them accessible for our group and for the deployment environment.

This application packaging also allowed a **consistent development environment**. Using Docker Compose, when developing, we simply build the docker image corresponding to the current module we are working on (be it the Backend, Web Frontend or Mobile Frontend) and pull the rest of the necessary previously built images corresponding to the other modules from our container registry. This

way, we are not developing and testing using the production environment, but we also ensure the entire team has access to the same environment.

## 4 Software testing

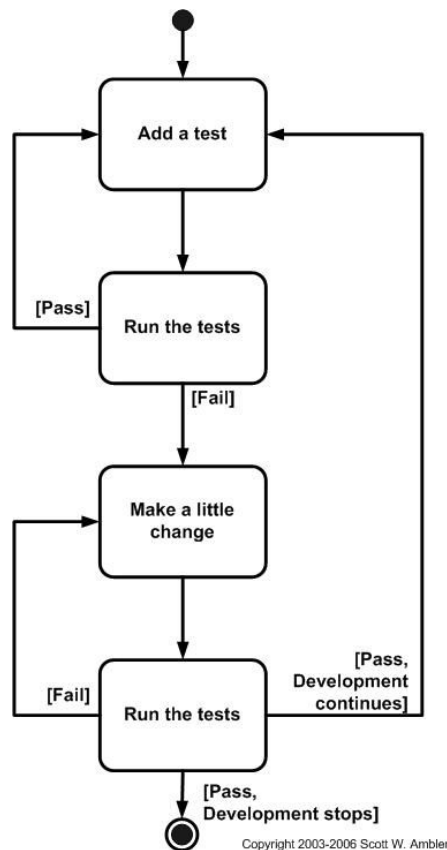
### 4.1 Overall strategy for testing

The strategy is based on these articles and “Clean Code” chapters for testing:

- o [Strategies](#)
- o [Testing overview](#)

[what was the overall test development strategy? E.g.: did you do TDD? Did you choose to use Cucumber and BDD? Did you mix different testing tools, like REST-Assured and Cucumber?...]

The overall strategy is very similar to Test Driven Development (TDD), so the overall workflow is as follows



Therefore, for each chunk of development, we create tests for it and only keep on developing when the tests pass, otherwise we will continue to adjust code until it passes the tests.

One of the biggest advantages of using TDD is that we are making small increments when writing code, and therefore issues are easier to fix, since they only address a small chunk of code.

#### Three Laws of TDD

- You may not write production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

- You may not write more production code than is sufficient to pass the currently failing test.

Another advantage we have encountered with TDD is that we always ensure a consistent and stable production environment. Given that we have Continuous Deployment set up, extensive tests help us make sure we are not “breaking” the production environment on a bad commit.

## 4.2 Functional testing/acceptance

[Project policy for writing functional tests (closed box, user perspective) and associated resources.]

As for functional tests, we opted by following a higher level approach, meaning we conduct the tests from the user’s perspective, basically only assuring the main features and user stories are working,

## 4.3 Unit tests

[Project policy for writing unit tests (open box, developer perspective) and associated resources.]

As for unit tests, we write them to assure that a given module works as expected, making use of Mocks and other test writing tools to abstract a specific module from it’s dependencies and test it extensively and effectively.

From the developer’s perspective, he only has to worry about the module he is working on.

## 4.4 System and integration testing

[Project policy for writing integration tests (open or closed box, developer perspective) and associated resources.]

As for a higher level system testing, we also wanted to ensure a full system module works as expected.

In the case of the backend API, we performed integration tests using MockMvc, fully testing the Spring Boot Web layer.