deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Grupo 105
v0000-00-00

1 Project management

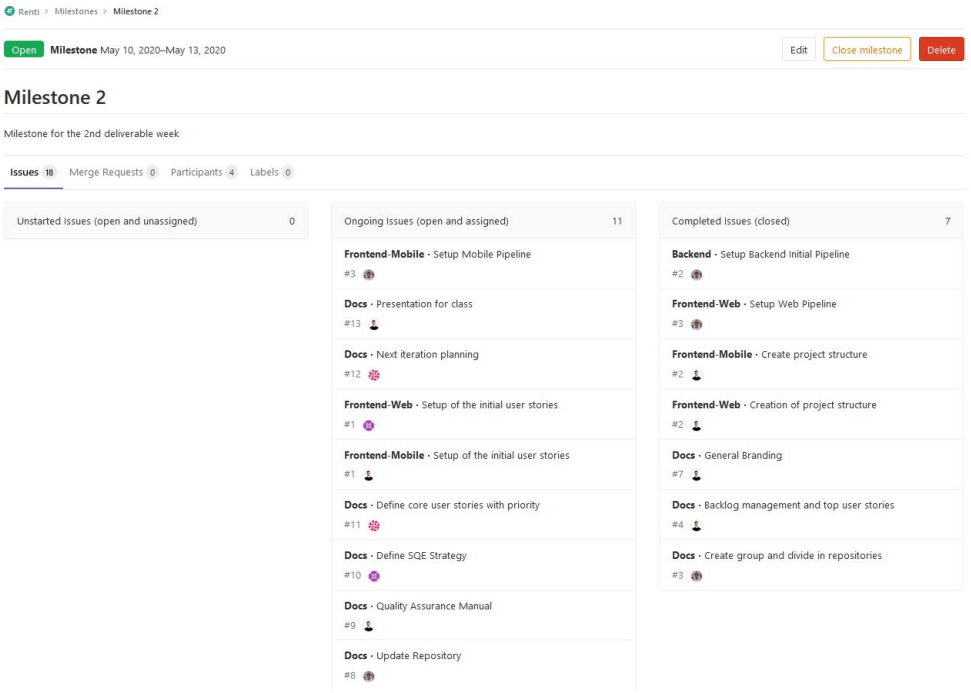
1.1 Team and roles

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.



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:

- Dont ignore exceptions
- Dont catch generic exceptions
- Defining fields in standard places
- Using TODO Comments
- Logging instead of printing
- Using standard bracket style

Also some standards from "Clean Code":

- Avoid duplication anywhere in code
- 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.

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:

- 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)
- **New feature branch:** For each new feature create a branch following the standard: `feature/<feature_name>`.
- **New Issue branch:** For each fix create a branch following the standard: `fix/<fix-name>`.
- 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 produce 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, this docker images are automatically build and uploaded to a private **container registry**, making them accessible for our group and for the deployment environment.

4 Software testing

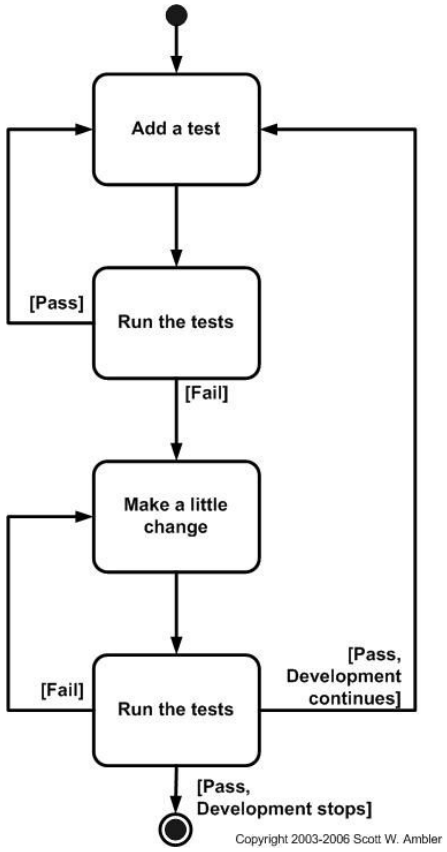
4.1 Overall strategy for testing

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

- Strategies
- 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 TDD or Test Driven Development, so the overall workflow is as follows



And 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.

4.2 Functional testing/acceptance

4.3 Unit tests

4.4 System and integration testing

4.5 Performance testing [Optional]

1. Functional testing/acceptance

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

2. Unit tests

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

3. System and integration testing

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

4. Performance testing [Optional]

[Project policy for writing performance tests and associated resources.]