universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Grupo 105
2020-05-28

1 Project management

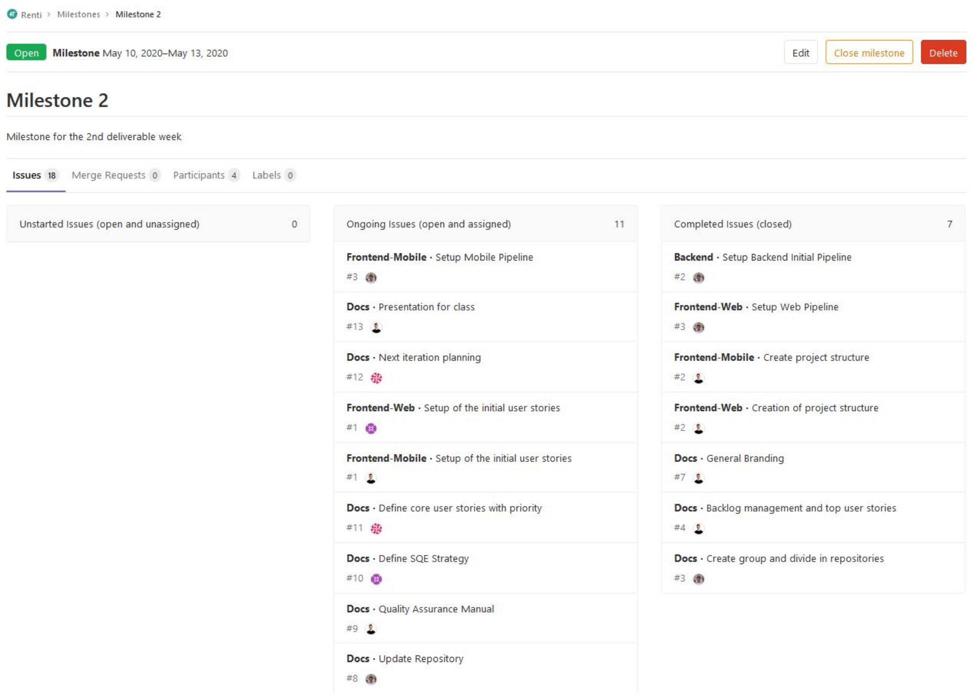
1.1 Team and roles

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) 0

Ongoing Issues (open and assigned) 11

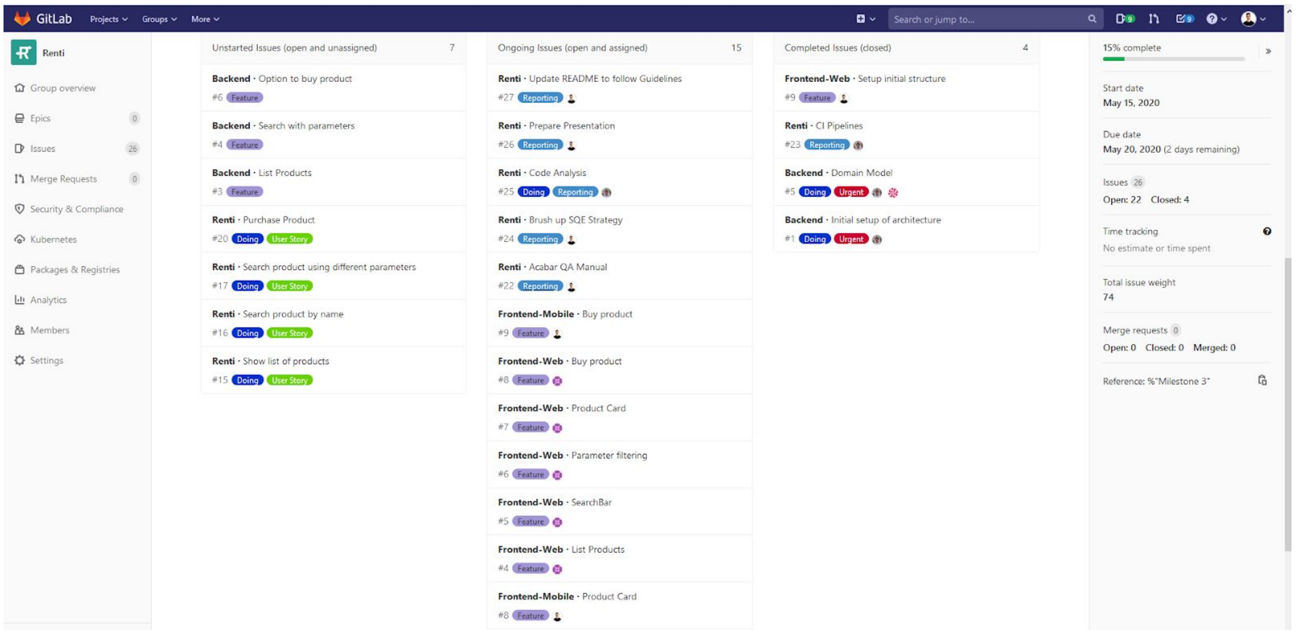
- **Frontend-Mobile** · Setup Mobile Pipeline #3
- **Docs** · Presentation for class #13
- **Docs** · Next iteration planning #12
- **Frontend-Web** · Setup of the initial user stories #1
- **Frontend-Mobile** · Setup of the initial user stories #1
- **Docs** · Define core user stories with priority #11
- **Docs** · Define SQE Strategy #10
- **Docs** · Quality Assurance Manual #9
- **Docs** · Update Repository #8

Completed Issues (closed) 7

- **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

(before GitLab Gold)

After a brief discussion with another group, we decided to upgrade our GitLab account and start using the features GitLab provides to help us with backlog management, specially **labels** and **weights**. This greatly improved our backlog visualization and was easier to prioritize tasks.



Unstarted Issues (open and unassigned) 7

- **Backend** · Option to buy product #6 Feature
- **Backend** · Search with parameters #4 Feature
- **Backend** · List Products #3 Feature
- **Renti** · Purchase Product #20 Doing User Story
- **Renti** · Search product using different parameters #17 Doing User Story
- **Renti** · Search product by name #16 Doing User Story
- **Renti** · Show list of products #15 Doing User Story

Ongoing Issues (open and assigned) 15

- **Renti** · Update README to follow Guidelines #27 Reporting
- **Renti** · Prepare Presentation #26 Reporting
- **Renti** · Code Analysis #25 Doing Reporting
- **Renti** · Brush up SQE Strategy #24 Reporting
- **Renti** · Acabar QA Manual #22 Reporting
- **Frontend-Mobile** · Buy product #9 Feature
- **Frontend-Web** · Buy product #8 Feature
- **Frontend-Web** · Product Card #7 Feature
- **Frontend-Web** · Parameter filtering #6 Feature
- **Frontend-Web** · SearchBar #5 Feature
- **Frontend-Web** · List Products #4 Feature
- **Frontend-Mobile** · Product Card #8 Feature

Completed Issues (closed) 4

- **Frontend-Web** · Setup initial structure #9 Feature
- **Renti** · CI Pipelines #23 Reporting
- **Backend** · Domain Model #5 Doing Urgent
- **Backend** · Initial setup of architecture #1 Doing Urgent

15% complete

Start date
May 15, 2020

Due date
May 20, 2020 (2 days remaining)

Issues 26
Open: 22 Closed: 4

Time tracking
No estimate or time spent

Total issue weight
74

Merge requests 0
Open: 0 Closed: 0 Merged: 0

Reference: %"Milestone 3"

(after GitLab Gold)

2 Code quality management

2.1 Guidelines for contributors (coding style)

Our baseline for writing code is heavily inspired in AOS project with standards like:

- Dont ignore exceptions
- Don't 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

As for specific Java code guidelines, we are trying to follow the REST API Naming Standards and the Google Java Style Guide.

For the React web code, we are using the new version of react and its stateful changes, including React Hooks and navigation 5. These hooks and functions are overtaking components, since they are soon to be deprecated. The quality standards for writing react hooks were based off this documentation.

2.2 Code quality metric

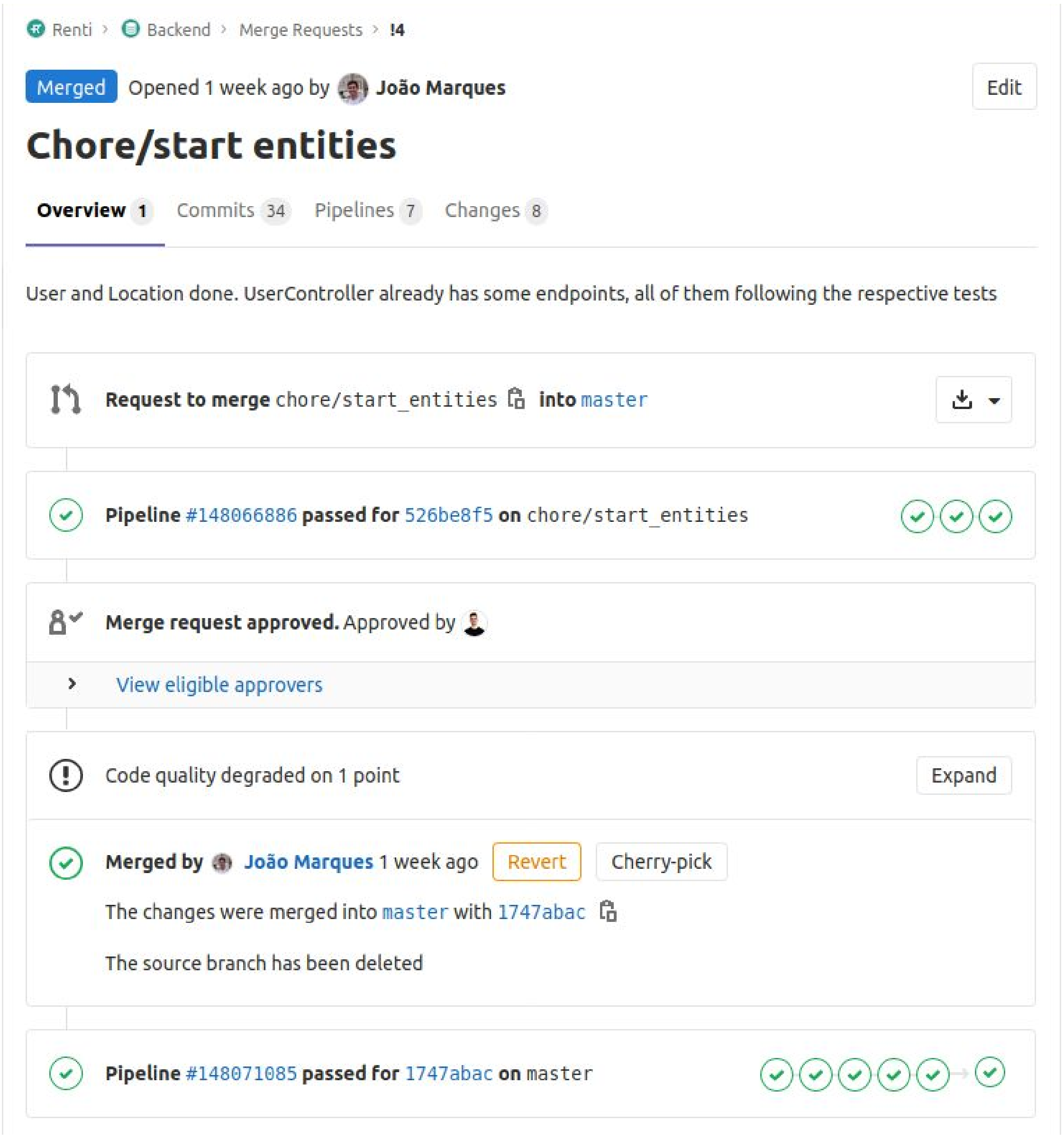
We are making use of static code analysis engines to evaluate our code quality and prevent errors.

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:


Renti › Backend › Merge Requests › !4

Merged Opened 1 week ago by João Marques

Edit

Chore/start entities

Overview 1 Commits 34 Pipelines 7 Changes 8

User and Location done. UserController already has some endpoints, all of them following the respective tests

Request to merge chore/start_entities into master

Pipeline #148066886 passed for 526be8f5 on chore/start_entities

Merge request approved. Approved by

View eligible approvers

Code quality degraded on 1 point

Expand

Merged by João Marques 1 week ago Revert Cherry-pick

The changes were merged into master with 1747abac

The source branch has been deleted

Pipeline #148071085 passed for 1747abac on master

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:

- 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

Following these standards for code reviews:

1. Every merge request required the approval of another member.
2. Medium/small code reviews, lesser than 300 lines of code
3. Light inspection rates, review lines carefully
4. 1-2 member approval on specific subject is good for our 4 man team.

On our backlog management we defined a User Story as a general feature the user could complete. Therefore, we would define the user story on a global scale on our “Renti” repository (for backlog purposes), and then replicate the feature extracted from that user story, to the respective repository: Backend, frontend-web, frontend-mobile. And we would finalize a User Story when all the features necessary for the user to complete the action, were done.

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

5. Map each user story into one (or more) system feature(s).
6. Map each feature into a git branch.
7. Classify the feature as done when the expected system behavior is accomplished.
8. Merge the feature branch into the master/production branch.

3.2 CI/CD pipeline and tools

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

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.

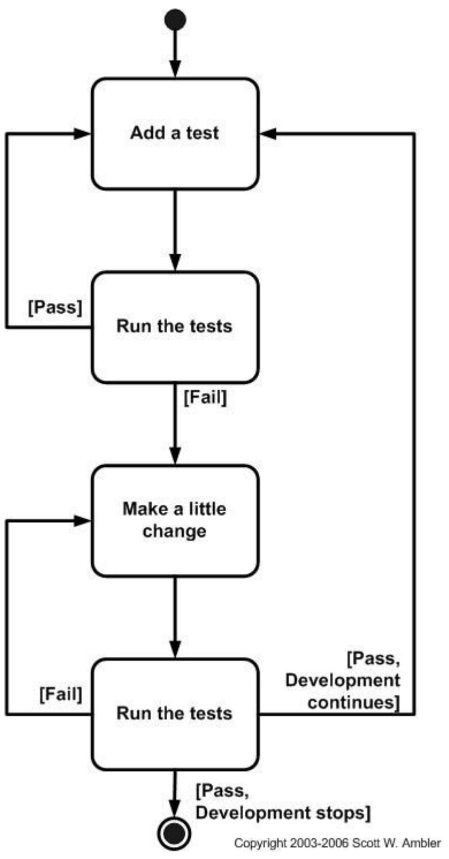
4 Software testing

4.1 Overall strategy for testing

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

- o Strategies
- o Testing overview

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


Add a test
Run the tests
[Pass]
[Fail]
Make a little change
Run the tests
[Fail]
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

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

As for functional tests, we opted by following a higher level approach, meaning we conduct the tests from the user's perspective, basically assuring the user could do what we expected, by only accepting a pass when the user story was fully complete.

4.3 Unit tests

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 its 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

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.

In the frontend, we developed JEST tests to cover system integration, using a similar approach to the backend, by using mock requests.

5 System Monitoring

As for the optional implementation of system monitoring, we opted by establishing several monitoring dashboards, using different technologies.

5.1 Getting metrics

The first step for setting up the monitoring system involved getting the appropriate metrics from different systems:

1. **Spring Boot**: We exposed a specific endpoint with Java system metrics.
2. **Gitlab CI Pipelines:** We set up a specific service to query the Gitlab API for pipeline statuses and expose the metrics in a API.
3. **System (Docker):** We also have a CAdvisor service responsible for querying the system for metrics, and also expose them in a API.

After having all these metrics available, he have a Prometheus service running, responsible for periodically querying all these API exposed metrics and making them available in a time series database, which will then be consumed by Grafana, the service responsible by taking all the information previously recorded and presenting it in useful dashboard through queries to the Prometheus database.

This way, we can constantly monitor both our internal system, as well as the production environment for business related inquiries (like the amount of visits to the website and API, number of database records), and also get useful performance metrics for the system.