TQS: Quality Assurance manual

***Alexandre Cotorobai [107849]*, *Bernardo Figueiredo [108073]*, Hugo Correia [108215], Joaquim Rosa [109089]**

v2023-04-18

1 Project management

1.1 Team and roles

Member	Role	Description
Hugo Correia	Team Coordinator	Responsible for scheduling team meetings, ensuring the tasks that should be done for each iteration considering its importance and effort needed to complete them.
Joaquim Rosa	QA Engineer	Responsible for guaranteeing the quality of the product, creating and executing tests to identify defects and ensure that the software meets the specified requirements.
Alexandre	Product Owner	Responsible for understanding the customers needs, and,

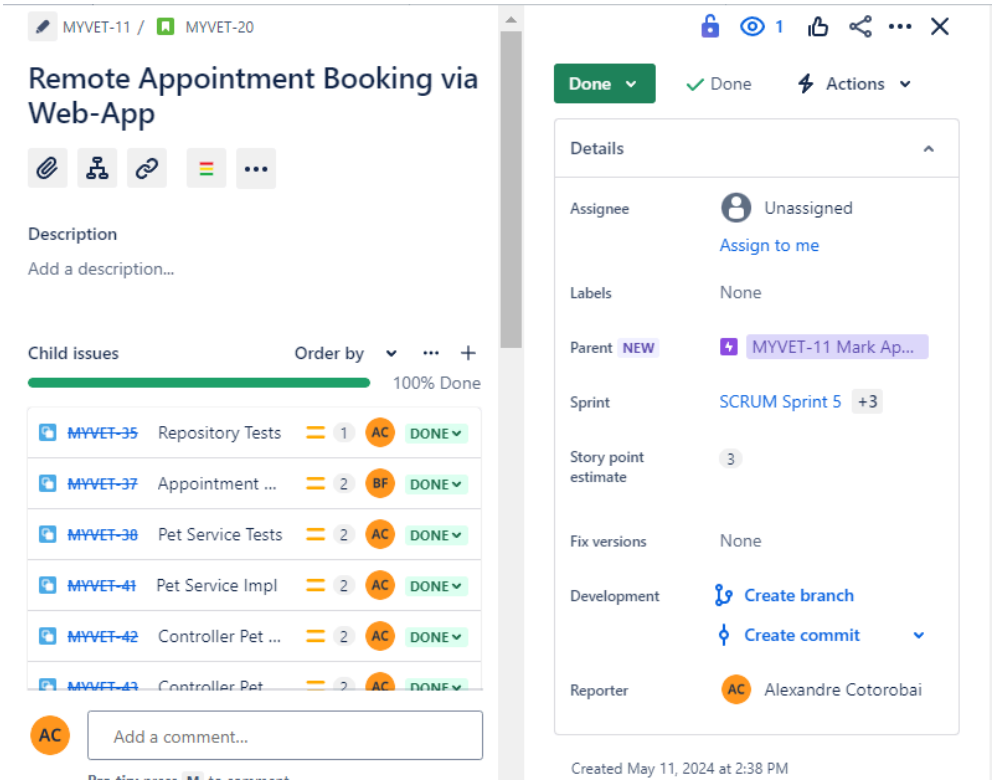
Cotorobai		consequently, gathering the requirements for our system.
Bernardo Figueiredo	DevOps Master	Responsible for integrating development and operations processes to achieve continuous delivery and deployment

1.2 Agile backlog management and work assignment

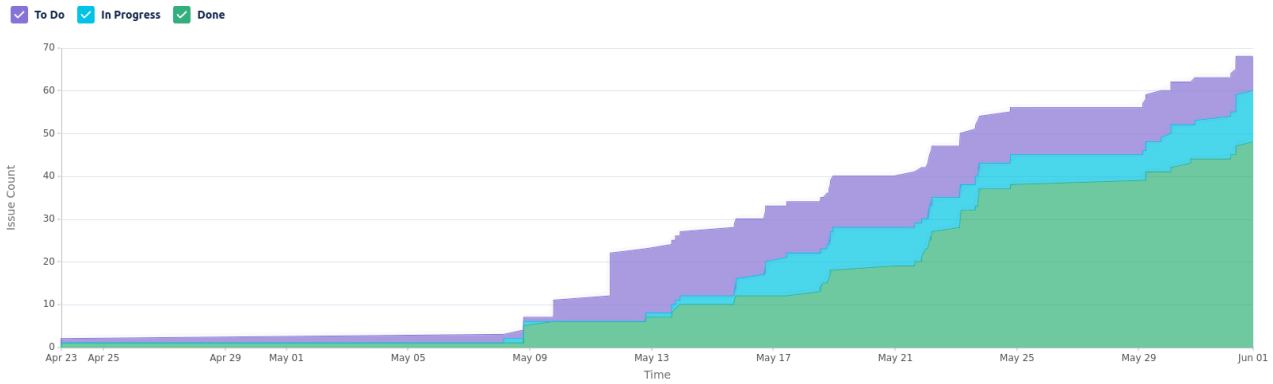
To set up our backlog management we decided to use Jira, because we have all already been in touch with Jira during the development of our Final Degree Project, and it has integration with GitHub.

Here is an explanation of how we used Jira to develop our project:

1. We defined the Epics and User Stories for our project and added them to Jira.
2. We prioritised each user story to focus on the most important features. However, instead of assigning story points to the user stories as we were supposed to, we initially assigned them to the subtasks of each user story. This was corrected midway through development, with each user story receiving story points according to its priority.



3. We selected a user story to develop, broke it down into subtasks as mentioned earlier, and then chose subtasks to work on. When we selected a task to work on, we created a branch for it from Jira and moved it to the "In Progress" column as we began working on it. After a subtask's development was completed and all tests passed, we made a pull request and marked the subtask as "Done."
4. Someone was responsible for accepting or rejecting the pull request after the CI build was approved. If the pull request was rejected, the reviewer moved the task back to the "In Progress" column. If the pull request was accepted and it was the last task remaining for the user story, the user story was moved to the "Done" column.
5. This process was repeated for all user stories being implemented.



2 Code quality management

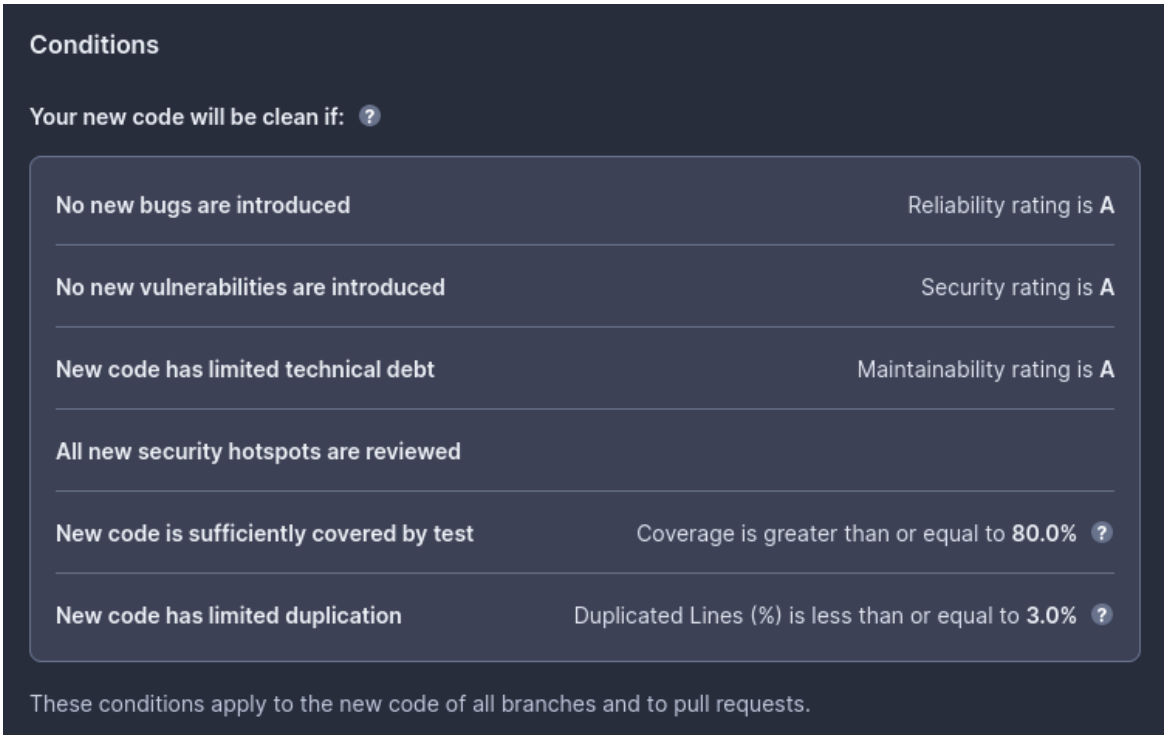
2.1 Guidelines for contributors (coding style)

To enhance the software quality, we adhered to the Google Java Style Guide, as the core of the project was in Java. This meant we applied the code style guide to the API, prioritising the quality of the backend business logic over the web app. The IDE we used facilitated the adoption of this code style through a plugin.

2.2 Code quality metrics and dashboards

The metrics established to assess code quality are based on an analysis that must be performed by the SonarCloud tool. We configured SonarCloud to integrate with our CI/CD pipeline, ensuring that each commit and pull request is automatically analysed.

We used the default code quality setting provided by the SonarCloud tool, Sonar Way. The key metrics monitored are the following:



These rules are a combination of industry best practices and principles established by the developer community.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The workflow adopted for this project was gitflow workflow. Each branch corresponds to a task of a user story. After the feature is done, the branch is merged into the “develop” branch, which is the main branch in our case. In order for the feature branches’ pull requests to be merged into the “develop”, another contributor needs to review the pull request and accept it. The main idea is for the contributors working at the same module(eg. frontend) to review their peer’s pull requests.
The user stories will be considered as “done”, everytime those have already been peer-reviewed, have met all acceptance criteria, have been deployed in the Production Environment as well as unit tested and integration-tested. Besides this, it will be necessary that the documentation has been written.

3.2 CI/CD pipeline and tools

The project has had a CI (using github actions and .yaml files) since the beginning, starting with the one given by SonarCloud on setup.
Over time however, the pipeline grew to accommodate CD as well, the overall CICD pipeline has the following setup:

Following the standard of creating branches and issues for development of new features or refactoring of old, the first step of the CI runs automatically on every Pull Request opened, and it runs all the unit tests and integration tests, if a test fails the Pull Request shouldn’t be merged into main.

The CD workflow is manually triggered when the person in charge of the product wants to update the running version. Due to the complexity of the project there is no easy way to automate the acceptance tests that have been written, so before initiating the workflow one should checkout the repository locally and run them.
The workflow consists of building the images based on the docker compose file, uploading them to the github container registry (ghcr.io), all of this on a github runner. The second step consists of telling our self hosted runner to pull the images previously built and run the docker compose, which will finalise the deploy.

3.3 System observability

We were unable to implement tools for system observability in our deployment, although in the future if more time was given, we would implement a system that monitors our resource consumption on deployed machines using the Prometheus Monitoring service to collect real time info on our containers in conjunction with grafana to visualise them, keeping us to date on the status of our deployment.

4 Software testing

4.1 Overall strategy for testing

For our project, we adopted a rigorous approach to testing to ensure the quality and reliability of our code.
For the back-end, we employed a Test-First approach for every module's unit testing, which means that we designed the tests before implementing the modules. For example, we first created tests for a service and then implemented the service, followed by creating tests for the controller and then implementing the controller. The frameworks used for unit testing were JUnit Jupiter, Hamcrest, Mockito, Spring Boot MockMvc, and Jacoco for code coverage analysis. After the successful execution of all unit tests in each isolated module, we created integration tests to ensure that different parts of the application worked together seamlessly. For the integration tests we used JUnit Jupiter, Hamcrest, and MockMvc. This process was repeated for each feature.

4.2 Functional testing/acceptance

For acceptance testing, we implemented Behavior-Driven Development (BDD) using the Cucumber framework. We first laid out scenarios in Gherkin language representing the expected behaviour of the web application, and then, using Cucumber, we created a series of automated tests to simulate user interactions and ensure the web application functions as expected.

4.3 Unit tests

As said previously, we used a First-Test approach, so every Unit test needed to be created before the feature itself. The main testing framework used was JUnit5 and the mocking framework used was Mockito. With both frameworks it was possible to mock the dependencies of a function and test the function behaviour based on the mocked values.
Tests were made to the repositories to ensure that the entities were being saved accordingly and that complex queries were providing us the expected data.
Service tests focused on checking if the returned values from the service were correct according to the provided data from the repositories.
Controller tests checked if the JSON responses were correct based on the service response.

4.4 System and integration testing

API testing

In our project, integration tests are designed to verify the proper interaction between various components of the system, ensuring they work together seamlessly. Our approach to integration testing combines both open box (developer perspective) and closed box (user perspective) methodologies. From the developer's perspective, integration tests are written with knowledge of the internal workings of the modules. These tests ensure that different components, such as services, repositories, and controllers, interact correctly with each other. From the user's perspective, integration tests focus on the system's behaviour as a whole, verifying that it meets the specified requirements without considering internal implementations.