

TQS: Quality Assurance manual

Daniel Gomes [93015], Mário Silva [93430], Bruno Bastos [93302] , Leandro Silva [93446]

v2020-05-03

Project management	1
Team and roles	1
Agile backlog management and work assignment	2
Code quality management	3
Guidelines for contributors (coding style)	3
Code quality metrics	3
Continuous delivery pipeline (CI/CD)	3
Development workflow	3
CI/CD pipeline and tools	4
Artifacts repository [Optional]	4
Software testing	4
Overall strategy for testing	4
Functional testing/acceptance	4
Unit tests	4
System and integration testing	4
Performance testing [Optional]	4

1 Project management

1.1 Team and roles

- **Team Leader:** For this position, we decided to assign **Daniel Gomes**, in which will have the tasks and functions of scheduling team meetings, ensuring the tasks that should be done for each iteration considering its importance and effort needed to complete them.
- **Product Owner:** As our Product Owner, we have chosen **Mário Silva**, being responsible to understand the customers needs, and, consequently, gathering the requirements for our system.
- **DevOps Master:** To complete the functions of this role we assigned **Bruno Bastos**, which will have the tasks of creating and managing the repository, as well as reviewing the Pull requests, and managing the CI/CD pipelines.
- **QA Engineer:** To this role, we settled that **Leandro Silva** would be fittable for it. His tasks will involve defining the code style standards, as well as ensuring that those are followed. Besides this, he will make sure that every feature is being tested correctly and well covered by the tests.

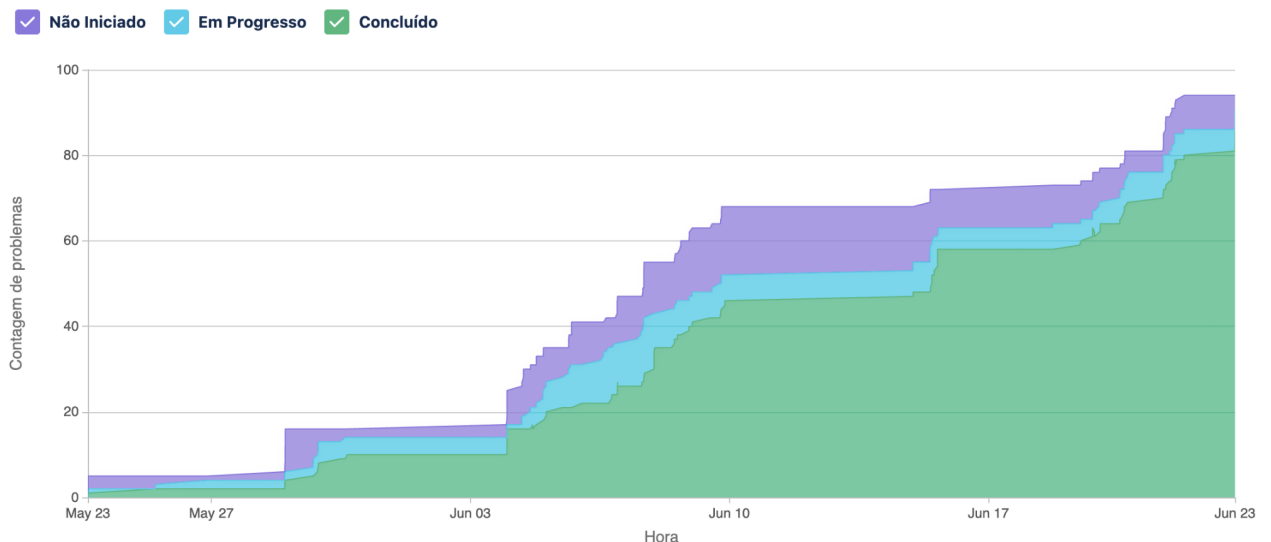
1.2 Agile backlog management and work assignment

To set up our backlog management we decided to use Jira for some reasons. First of all, we have all already been in touch with Jira. Besides this, Jira supports many functionalities that help us improve the quality of the backlog management, as for example the integration with GitHub, allowing us to create new Pull Requests related to the User Stories defined on Jira.

<https://crowdwire.atlassian.net/jira/software/projects/MED/boards/3/reports/burnup>

To increment new features to the project, we will guide ourselves through a story workflow, where we:

1. Assemble the project features by **creating user stories**.
2. **Estimate the story points** for each user story, so that we can **prioritize** the most important features. Add the users stories collected to the *For Development* column on the Jira's Scrum Board. These should be ordered by their priority.
3. **Choose a user story** to develop. **Divide the user story in tasks** (and subtasks, if needed) and also add them as issues to the Scrum Board *For Development* column. Then, **choose a task** and move it to the *In Progress* column, as well as the user story from which it belongs.
4. After the development of a certain task is completed, and if all tests have passed, make a **pull request** through the Jira's Github integration and move the respective task to the *Done* column. Jira will then notify someone to review the pull request.
5. As the responsible for the pull request, **accept or reject** the new feature **after the CI build has passed**. If the pull request is rejected, the reviewer should move the task back to the *For Development* column. If it is accepted and it happens to be the last task associated with a user story, the reviewer should move this user story to the *Done* column.
6. **Restart on point 3** till all user stories have been completed. Every time the sprint reaches the end, make a new increment.




2 Code quality management

2.1 Guidelines for contributors (coding style)



To enhance the software quality, we will follow Google's code style guide on Java [1], since the core of the project will be in Java. This means that we will only respect a code style guide on the API, as we are more concerned with the quality of the backend business logic, and not the web app. The IDE used provides an easy adoption of this code style using a plugin.

2.2 Code quality metrics















The following image shows the conditions to be fulfilled in order to pass on the quality gate that was defined.

Conditions 
Add Condition

Conditions on New Code
 Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value	Edit	Delete
Code Smells	is greater than	2		

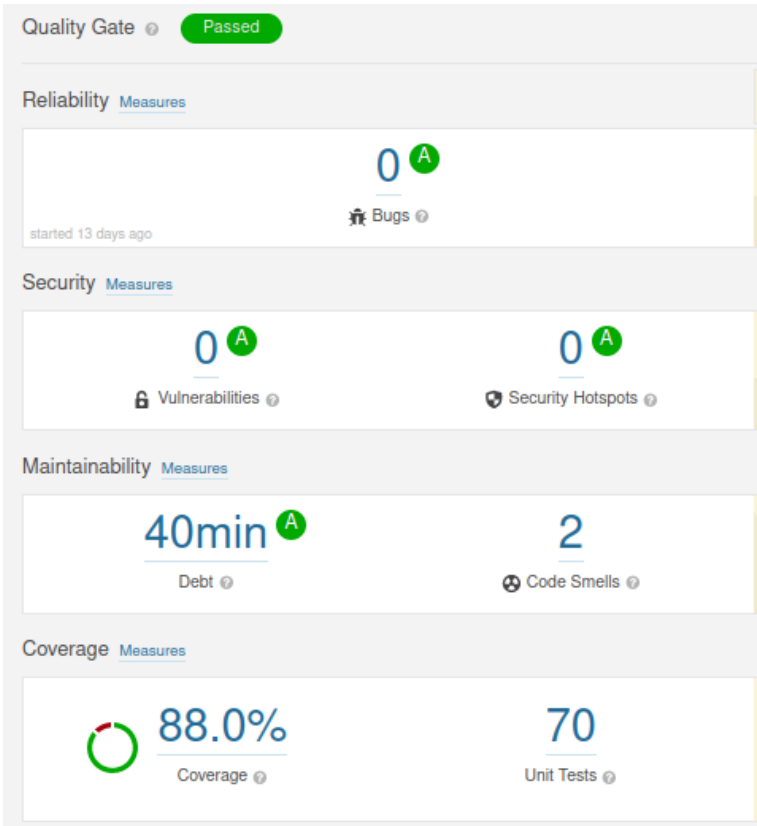
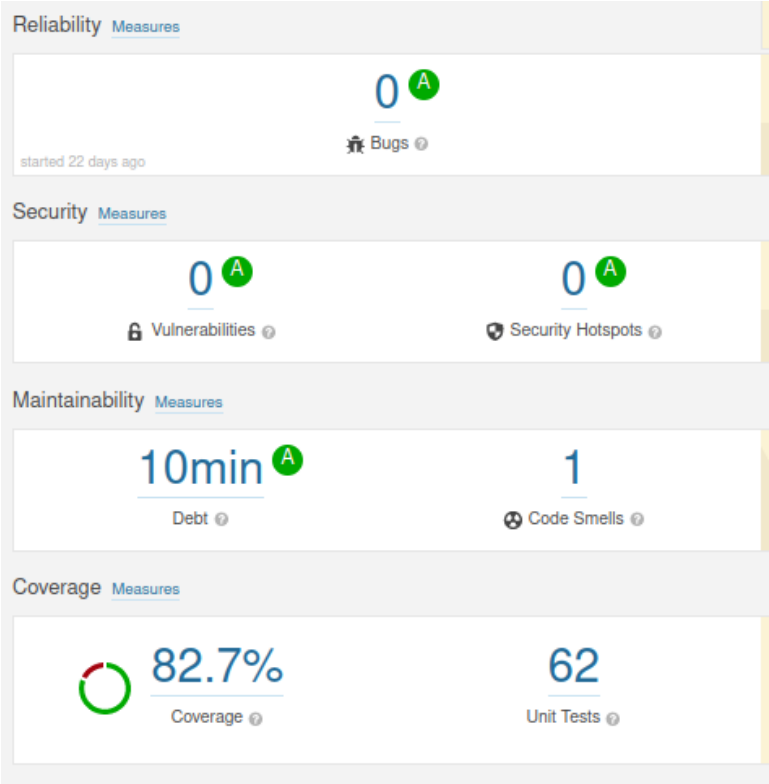
Conditions on Overall Code
 Conditions on Overall Code apply to long-lived branches only.

Metric	Operator	Value	Edit	Delete
Code Smells	is greater than	10		
Coverage	is less than	80.0%		
Duplicated Lines (%)	is greater than	3.0%		
Maintainability Rating	is worse than	A		
Reliability Rating	is worse than	A		
Security Hotspots Reviewed	is less than	100%		
Security Rating	is worse than	A		

This quality gate has a minimum of 80% in coverage, as we felt we should lean our coverage to the maximum reasonable coverage. We gave no tolerance in maintainability, reliability, and security, to make our application very assurant. We tolerated 10 code smells on the overall code, since we expected to get some code smells which we would find unreasonable. Besides, to prevent the code smells to increase rapidly, we restrict 2 code smells on new code. At last, we defined a maximum of 3% duplicated lines, which was the tolerant margin for the cases where avoiding duplicate lines would be unjustified.

We also used the IDE extension SonarLint, as it helps to detect issues as we write code, which reduces the existence of issues on the pull requests, makes them more likely to be passed, and, consequently, saves us many time.

In the end, this was the state of our quality gate metrics. The first one corresponds to ExDelivery, followed by MedEx. Both presented good results from which we were satisfied.



3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We used a *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. 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 peers’ 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.

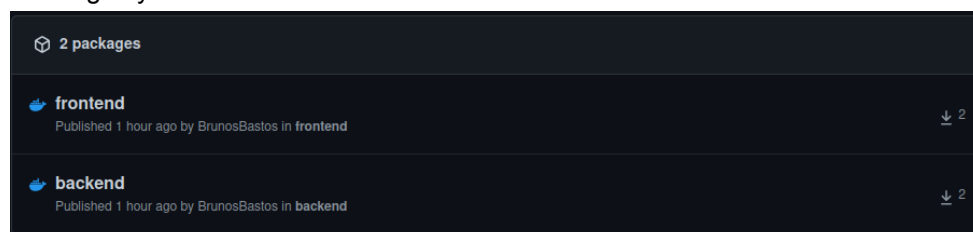
3.2 CI/CD pipeline and tools

In order to set up and configure our CI/CD pipelines, we decided to make use of Github Actions, since it is pretty straightforward to use, and we have already got in touch with it. Starting with the CI pipelines, our main workflow consisted of running the tests using the “*mvn verify*” command that runs up both unit and integration tests, integrated with sonarcloud for the static code analysis. Github would alert us if any of the tests failed or if the quality gate defined in the sonarcloud didn’t pass. The CI pipeline runs on every push to the repository.

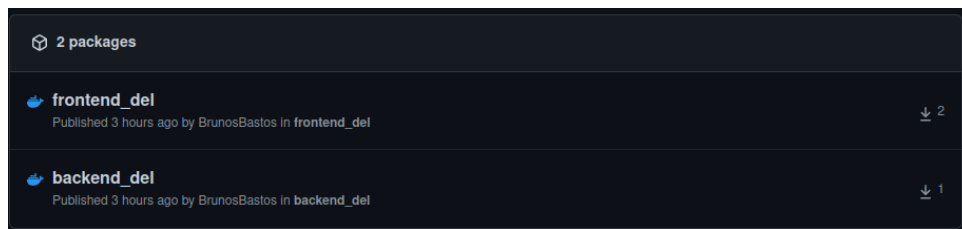
The CD pipeline only runs when a pull request is made to the master branch. This way we reduce the number of deployments made. The first step of the CD phase was building the registry of our modules. Then the images were pushed to the github packages integration with docker hub. If those steps succeed, another workflow is runned in the VM provided for the class. This workflow will pull, build and then run the images from both repositories.

Our continuous Delivery followed a container-based approach, therefore we made use of Docker and Docker Images. Every time a new release is triggered on the CI pipeline that builds our Docker images, and pushes those images to GitHub Packages, which works out as a Container Images Repository. This kind of workflow happens to be really important, since these images are able to be used on the CD pipeline, by being pulled from the repository, and, consequently deployed on our Production Environment.

The following image shows up the *packages* on our Repository, corresponding to each image pushed to the Github Registry:



MedEx Image Registry



ExDelivery Image Registry

4 Software testing

4.1 Overall strategy for testing

During the development of this project, we used Test Driven Development as our testing strategy, meaning that the tests were written before the actual development of the features.

For the tests made to the frontend it was used Selenium and Cucumber using a BDD approach. These tests were made for each page after it was integrated with the backend.

The tests to the API used *JUnit5* as the base framework and *RestAssured* for both the controller Unit tests and the Integration tests. To make assertions we explored the use of *AssertJ* and *HarmCrest*.

Tests are being run in the CI pipeline using Github Actions.

4.2 Functional testing/acceptance

For testing the functionality of our application we decided to use Cucumber tests combined with Selenium.

Whenever we finished a feature on the backend and then integrated it in the frontend app we would first map the feature to a cucumber feature. This way we adopted a BDD approach for testing the frontend.

We used the Web Driver Manager from <https://github.com/bonigarcia/webdrivermanager>, which was useful for avoiding linking directly the driver into the source code.

Developing the tests with Gherkin helped in testing and also documenting the features as well, since it has all the steps necessary to be able to execute and complete a feature, and also tests the feedback for the end user, for example:

```

Feature: Purchase medicine

Background:
  Given I am a client on the products list page
  When I add a few products to my Cart
  And I go to my Shopping Cart

Scenario: Make a valid purchase
  And I set the quantity of a product to 2
  And I press the purchase button
  And I insert the latitude for the delivery location as 40.34
  And I insert the longitude for the delivery location as 55.12
  And I finalize my purchase
  Then A successfully made a purchase message should appear

Scenario: Make a purchase requesting amounts that exceed stock
  And I set the quantity of a product to 99999
  And I press the purchase button
  And I insert the latitude for the delivery location as 40.34
  And I insert the longitude for the delivery location as 55.12
  And I finalize my purchase
  Then A purchase failed message should appear

Scenario: Make a purchase with an invalid delivery location
  And I press the purchase button
  And I insert the latitude for the delivery location as 250.01
  And I insert the longitude for the delivery location as 45.123
  And I finalize my purchase
  Then A purchase failed message should appear

```

To continue explaining the process let's use this specific feature for example, after writing the feature, we would create a `PurchaseSteps.java` class that references each step defined in the feature and then we would create the pages that were necessary for the feature, in this case `PurchaseDetailsPage.java`. This follows the Page Object Pattern and this way we re-used code, since some pages were necessary for other features, such as the Login Page that was necessary for almost all other features:

```

public AddSupplierSteps() {
    WebDriverManager.firefoxdriver().setup();
    driver = new FirefoxDriver();
    loginPage = new LoginPage(driver);
    addSupplierPage = new AddSupplierPage(driver);
    errorMessage = new ErrorMessage(driver);
}

@Given("I am the pharmacy owner on the add supplier page")
public void loggInAsOwner() {
    loginPage.loggInAs( email: "clara@gmail.com", password: "string");
    addSupplierPage.goTo();
}

```

After running the tests successfully, the feature would be now moved to the Done section in the backlog.

4.3 Unit tests

As said previously, it was used a TDD 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

Integration tests were made to the API by using RestAssured to call the endpoint. These tests used an in-memory database with preloaded data.

Some tests were made using an order so that the changes to the database made by the other tests were taken into consideration in the next tests. However, for some tests it was necessary that the database was not altered from test to test, so we used `@DirtiesContext` annotation that will reload the context of spring boot, thus loading the database again. Using this annotation slows down the tests by a lot, so it was only used when it was a must.

4.5 Performance testing

As our project was deployed inside the University of Aveiro Network, and our tests pipeline runs on the Github Actions environment, it was not possible to run performance tests dynamically on the pipeline, and, consequently, set up a good policy to write performance tests.

However, in order to test the performance of both API and frontend Web App in the *MedEx* and *ExDelivery* systems, we went forward to use *JMeter* for our performance testing. For each of those tests we simulated with 500 threads, which corresponds to 500 users, and for all of them we only did GET requests.

Service	Endpoint	Average Latency	Throughput	Standard Deviation
ExDelivery API	/api/v1/couriers?page=0	2969 ms	81.5/sec	1482.69
ExDelivery App	/	31 ms	238.4/sec	28.36
MedEx API	/api/v1/products	2379 ms	85.3/sec	862.02
MedEx App	/	9 ms	130.3/sec	14.04

As we may see from the APIs, as the deviation is way too high, the latency values are exaggerated from reality. On the other hand, the values for the frontend web apps are looking acceptable as the latency values are minimal, with a deviation of less than 30, in both cases.

[1] Google's Java Code Style
<https://google.github.io/styleguide/javaguide.html>