universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Diogo Carvalho [92969], Pedro Amaral [93283], Rafael Baptista [93367], Ricardo Cruz [93118]

1 Project Management

1.1 Team and roles

Members	Roles	Responsibilities
Diogo Carvalho	Team Coordinator	We decided that Diogo Carvalho should be the team coordinator given that he already has experience leading this team. He is responsible for deciding which tasks each member will work on and manage the weekly sprints.
Ricardo Cruz	Product Owner	We decided that Ricardo Cruz would be the fittest for the role of product owner since he is the one that came up with the idea of this project. He is responsible for understanding the product that is being built and answer any questions that may arise.
Pedro Amaral	QA Engineer	We decided that Pedro Amaral would be fit for the role of QA Engineer since he had the most experience with creating CI/CD pipelines. He is responsible for implementing the CI/CD pipelines and managing the quality assurance practices and tools.
Rafael Baptista	DevOps Master	Finally, we decided that Rafael Baptista should be the DevOps master since he is more experienced with working with docker containers. He is responsible for configuring the git repository, the development/production infrastructure, and the deployment containers.
All	Developer	Every member of the project is a developer.

1.2 Agile backlog management and work assignment

To manage our backlog, we will make use of the tool Jira. We will have our backlog divided into 4 sprints of one week each given the timeframe of the project. In each sprint there will be a set of user stories, tasks and bug fixes that will be selected to be implemented. These items will initially be on the "TO DO" column and when a developer starts working on a given item it should be moved into the "IN PROGRESS" column. When the item is finished it should be passed into the "TESTING" section if needed and finally into the "DONE" section. Besides the title and the description each item will have an assignee, zero or more co-assignees, one or more labels and a story point estimate from 1 to 4 depending on the effort required to perform the task, 1 being the easiest and 4 the hardest. The link to the project backlog can be found on the project's README.

2 Code quality management

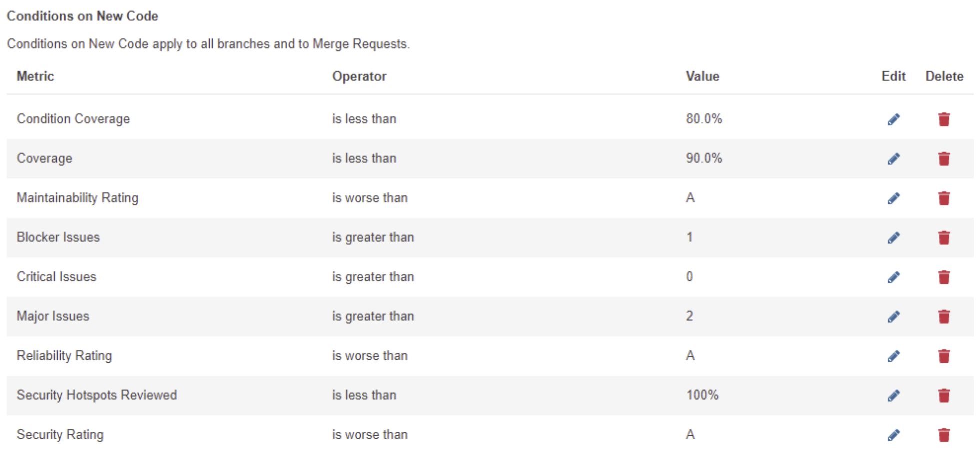
2.1 Guidelines for contributors (coding style)

In this project we decided to use the Mozilla coding style available in the following link: https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html.

2.2 Code quality metrics

To obtain a static code analysis of the project the tool sonarcloud was used. There we have four dashboards, two of them being the two Spring Boot APIs and the other two for the two React Web Applications. The link for all sonarcloud dashboards can be found on the project's README.

In this project we decided to use the following quality gates for the spring boot APIs. They were mostly based on sonarcloud's default configuration, but we decided to add more conditions to produce code with an even better quality. Also, to obtain better coverage percentages, we decided to exclude the Spring Boot models and Web Security Configuration classes from these conditions.

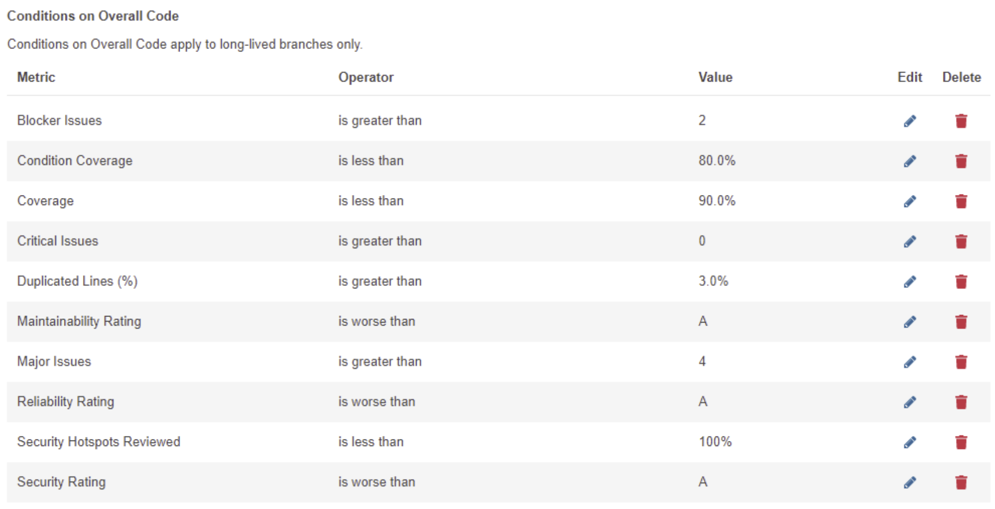


Conditions on New Code

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

Metric	Operator	Value	Edit	Delete
Condition Coverage	is less than	80.0%		
Coverage	is less than	90.0%		
Maintainability Rating	is worse than	A		
Blocker Issues	is greater than	1		
Critical Issues	is greater than	0		
Major Issues	is greater than	2		
Reliability Rating	is worse than	A		
Security Hotspots Reviewed	is less than	100%		
Security Rating	is worse than	A		

Figure 1 – Quality Gate Conditions for New Code



Conditions on Overall Code

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

Metric	Operator	Value	Edit	Delete
Blocker Issues	is greater than	2		
Condition Coverage	is less than	80.0%		
Coverage	is less than	90.0%		
Critical Issues	is greater than	0		
Duplicated Lines (%)	is greater than	3.0%		
Maintainability Rating	is worse than	A		
Major Issues	is greater than	4		
Reliability Rating	is worse than	A		
Security Hotspots Reviewed	is less than	100%		
Security Rating	is worse than	A		

Figure 2 – Quality Gate Conditions for Overall Code

Although we also configured sonarcloud to analyze the React projects, since this was not the focus of the work, we decided to prevent its results from failing the CI pipeline. Also, they use the sonarcloud's default quality gate.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

In this project the gitflow workflow was adopted resulting in the following branches:

- master – stores the official release history.
- develop – stores the in-development version of the product.
- release – stores a checkpoint of the develop branch and until it is merged to master it should only receive documentation and bug fixes.

- Feature-branches – branch where a developer develops a certain task which belongs to a user story. It can named in 4 different ways: new/(…) for new features, imp/(…) for improvement of an existing feature, bug/(…) for bug fixes and doc/(…) for documentation.

When a member of the team wants to merge his feature-branch into another branch, he must do a merge request on Gitlab. Then his changes must receive a code review by another member of the team who can approve of these changes or refuse leaving a comment about what must be changed before being reviewed again. Then before each sprint ends the new contents on the branch develop are merged to the branch release, where they can suffer additions of documentation or bug fixes before being merged to the branch master. Also, merge requests are configured on Gitlab so that they can only be accepted if the CI pipeline described in the next section is successful.

In this project we classified a User Story as done once it met all the acceptance criteria, had unit tests and integration tests written, executed, and passed, and was successfully deployed in a production environment.

3.2 CI/CD pipeline and tools

To implement a Continuous Integration and Delivery pipeline, Gitlab CI was used. This pipeline runs on Gitlab runners hosted on the VM provided by the teacher.

In terms of CI, for each commit pushed to the repository server a pipeline containing 2 stages will be run. In the first stage the spring boot unitary and integration tests are run and a static code analysis is done with sonarcloud both for the spring boot projects and for the react projects. Here the pipeline fails if one of the tests fails or one of the spring boot projects breaks its quality gate. Then both web applications will be built for production breaking the pipeline if an error happens or a warning appears.

In terms of CD, given that the CI pipeline is successful, and it was running on the branch master it will proceed to build 4 docker images, one for each one of our subprojects, saving them on our Gitlab Repository Container Registry. Then if the build was successful, it will proceed to run these docker containers on the VM provided by the teacher alongside 2 databases which are configured with volumes so that the data is not deleted.

3.3 Artifacts repository

To manage our docker images in the CD process we make use of the Gitlab Container Registry.

4 Software testing

4.1 Overall strategy for testing

To develop each API, a TDD (Test-driven development) strategy was adopted. First, a simple structure of the API was coded. Then for each new feature first the needed tests are implemented followed by the feature so that it satisfies all acceptance criteria. The tests are implemented using JUnit5 as the base, Mockito when we need to simulate dependencies and Spring MockMVC when we need to implemented rest controller tests.
To develop each web application, a BDD (Behavior-driven Development) strategy was adopted. First Cucumber was used to define the user stories that were needed. Then the feature in the webpage is developed having a user story in mind. Finally, that user story is recorded with Selenium IDE and the resulting code is adapted to implement the page object pattern and to support Cucumber.

4.2 Functional testing/acceptance

To implement functional testing in our project, we made use of a BDD (Behavior-driven Development) approach using the technologies Selenium and Cucumber. For each new user story or functionality, we create a new feature, defining in the background common steps between scenarios, and then implement different scenarios to that functionality. Finally, in cases where the test is very extensive, the user story is recorded using Selenium IDE to facilitate this work, which does not allow us to use a test-driven approach. However, in most cases, since we are splitting tests accordingly to features, we utilize a test-driven approach, starting by defining the steps of the scenario, and then creating the necessary methods from the pages we define utilizing Page Object Pattern. We decided to implement this pattern, since web tests tend to get very extensive and incomprehensible. With Page Object Pattern, defining the pages of the website, we define the necessary methods, and in the test area, we simply call those methods. This allows the tests to be comprehensible and easy to understand and prevents in many cases duplication of code.

4.3 Unit tests

In this project there are unit tests implemented for the repositories, services, and controllers of both Spring Boot APIs. The tests on the repositories verify if the operations with the database are working as intended. Then the tests on the services mock the repositories to verify that the business logic is correctly validating its input and returning the intended data if successful and nothing if not successful. Finally, the controller tests mock the services to make sure that its input is correctly mapped to a method of the correspondent service and that they return the correct information and status code if successful and only the error status code if not successful.

4.4 System and integration testing

In this project we implemented integration tests in both Spring Boot APIs, the tests are very similar to the controller unit tests but now there is no mocking having all classes fully instantiated. In these tests, we verify if the entire controller-service-repository connection is correctly done, which means that upon receiving a http request the server will proceed to return or save the correct data.

4.5 Performance testing

In thus project we performed performance testing of our APIs using the JMeter framework. We decided to test the most used endpoints of both APIs which are the GET notifications endpoint for the deliveries server and the GET orders for the orders server.

Endpoint	Latency	Elapsed Time
GET http://192.168.160.233:8080/api/private/notifications?id=2	31	40
GET http://192.168.160.233:8081/api/orders/42	30	37

As we can see the latency is small and very close to the elapsed time, meaning that the operation is very fast. However, the reason for it being fast can be the low amount of data it fetches so we concluded that in a real situation these tests should be run frequently.