deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Paulo Pereira [98430], Mariana Rosa [98390], Artur Romão [98470], Diogo Cruz [98595]

v2022-05-30

1 Project management

1.1 Team and roles

- Team Leader - Mariana Rosa
- DevOps Master - Artur Romão
- QA Engineer - Paulo Pereira
- Product Owner - Diogo Cruz

1.2 Agile backlog management and work assignment

For the workflow of this project we will use Jira Software to keep track of our sprints and assigned tasks.
We have weekly meetings on Mondays to check which epics and user stories need to be developed during the week. We also do *Pull Requests* for each feature on our meeting, therefore we can always have the code merged. For each feature on our applications, a new branch will be created. After the

feature is made we delete the branch. We previously defined our epics and user stories, therefore we have an idea on what to do.

2 Code quality management

2.1 Guidelines for contributors (coding style)

For this project we will follow the AOSP Java Code Style. We decided on using concrete exception names, to identify problems quickly and maintain a clean coding style, to make sure it's easily readable by other developers.

2.2 Code quality metrics

[Description of practices defined in the project for *static code analysis* and associated resources.]
[Which quality gates were defined? What was the r[ationale?]

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

In order to distribute the work among the group elements, several tasks were assigned in Jira Software as said above in [1.2]. Each one of those tasks is focused on the development of a certain feature or *user story*. In terms of development, a branch was created for each Jira task so that the work remains organized, easy to follow and to track. Whenever both development and deployment of a task are completed, a *Pull Request* to the *"development"* branch is done regarding the good practices of **CI/CD**.
We adopted three possible states for the Jira tasks as learned in IES course in the last semester:

- To do - whenever a task was merely discussed and defined by the team and no work has been done yet;
- In progress - during the development of the task or when it has been finished but is waiting deployment (this requires team communication);
- Done - when all team members agreed that the task is working as expected, which means that the functionality is operating, all tests are passing and the deployment of the *user story* was successfully done.

When it comes to *code review*, we define that for each *Pull Request*, there should be at least one reviewer apart from the element that started the *Pull Request*.
About *branch protection rules*, we decided to lock *Pushes* for both *"develop"* and *"main"* branches in order to prevent unwanted errors/problems (in case someone forgot to create a branch to develop a feature).

The last thing to do was a *Pull Request* from the *"develop"* branch to the *"main"* branch when everything was working just fine in the former.

3.2 CI/CD pipeline and tools

Both Continuous Integration and Continuous Delivery are assured with GitHub Actions.
We have two *maven.yml* files, one for each GitHub repository (Generic Deliveries Engine, **OnYourWay**, and Specific Market Application, **24Meds**).

Continuous Integration

Everytime a *Push* is done to an opened *Pull Request* in which *"develop"* branch or *"main"* branch is the base, CI pipeline tests Spring Boot services in both repositories and analyzes the code with SonarCloud. All Spring Boot tests must pass as well as the SonarCloud Quality Gate rules for all jobs to succeed.

Continuous Delivery

Following the same logic of CI, when a *Push* is done to an opened *Pull Request* for the main branches (*"dev"* and *"main"*), if every CI test passes, an *ssh* file containing commands to build the docker images and deploy the project to a virtual machine is executed.

4 Software testing

4.1 Overall strategy for testing

Regarding our testing strategies we will implement two different options.

For the **Backend,** a **TDD** *(Test-Driven Development)* approach will be used, even though it may cause some difficulties to implement. This implies that we create unit test cases before developing the actual code. These unit tests are great to learn whether or not individual parts of an application work, before actually trying to tie everything together.

For the **Frontend**, a **BDD** *(Behavior-Driven Development)* approach will be used. This creates a human-readable description of software user requirements as the basis for software tests. For this we will integrate **Selenium** with **Cucumber**.

All tests will be developed using **JUnit5** as the base framework

4.2 Functional testing/acceptance

The objective of these tests is to exercise if a component is acting in accordance with predetermined requirements.

To test the user interface of the system we opted for **Selenium** that can be used in both Chrome, Internet Explorer, Safari, Opera and Firefox with their specific drivers. This way we can test web browser components of the application.

Cucumber will also be used for the purpose of BDD, to develop test cases for the behavior of software's functionalities from an end-user's perspective. To make the tests more maintainable and readable we will try to implement **PageObject Pattern**, separating navigation from verification. For each page of our application we create a class and the tests themselves will use those classes.

4.3 Unit tests

These tests are used for testing isolated components of specific methods to verify functionality, ensuring the behavior is as expected.

For good testing practices, there should be tests for valid and invalid input, to make sure there is no unexpected behavior.

Mock is going to be used to isolate a component and focus on the code being tested and not on the behavior or state of external dependencies. For this we will use **Mockito** to test *Service* components and **MockMVC** for *Controller* components

4.4 System and integration testing

These tests are performed after unit tests, to make sure all the different units, modules or components previously tested in isolation work together in the overall application. Here, there is no mocking of the *Services*, relying on the actual implementation.

They will be developed for the *Controller* endpoints, making sure they work correctly when interacting with the *Rest API*. For this we will use **Test Containers.**

To perform these tests we will use an in-memory database to make sure they run safely and don't interfere with the actual database.

4.5 Performance testing

For Performance Testing/Static Code Analysis the **SonarCloud** tool will be used, to collect and analyze the source code and to report the code quality and the occurrence of known weaknesses.