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

[This report should be written for new members coming to the project and needing to learn what are the QA practices defined. Provide concise, but informative content, allowing other software engineers to understand the practices and quickly access the resources.
Tips on the expected content, along the document, are meant to be removed.
You may use English or Portuguese; do not mix.]

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

[Clarify the workflow adopted [e.g.. gitflow workflow, github flow . How do they map to the user stories?]

[Description of the practices defined in the project for *code review* and associated resources.]

[What is your team "Definition of done" for a user story?]

3.2 CI/CD pipeline and tools

[Description of the practices defined in the project for the continuous integration of increments and associated resources. Provide details on the tools setup and config.]

[Description of practices for continuous delivery, likely to be based on *containers*]

3.3 Artifacts repository [Optional]

[Description of the practices defined in the project for local management of Maven *artifacts* and associated resources. E.g.: artifactory]

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.