deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Diogo Oliveira Magalhães [102470]
Rafael Fernandes Gonçalves [102534]
Leonardo Almeida [102536]
Pedro Henrique Figueiredo Rodrigues [102778]
v2023-06-05

1 Project management

1.1 Team and roles

For this project, we have a team of four members, each one with a specific role and responsibilities. The team roles were assigned based on the skills and experience of each member and the needs of the project. These roles follow the OpenUP methodology which allows a software development process focused on people interaction and collaboration since "Nobody does great software alone but a team working together can do extraordinary things”.

The team roles are:

Team Coordinator - Leonardo Almeida - Responsible for ensuring that there is a fair distribution of tasks to each member of the team, for certifying that the project outcomes are delivered in time, for making sure that the team is collaborating and facilitating team meetings, and for addressing problems that may occur and supporting team members.

Product Owner - Pedro Rodrigues - Responsible for defining the product vision and strategy and having a deep understanding of the application domain, responsible for keeping the backlog updated, ensuring that the product meets the needs of its customers and stakeholders, and for providing feedback to the development team.

QA Engineer - Diogo Magalhães - Responsible for ensuring that the software or product being developed meets the expected quality standards by defining, in collaboration with the other elements of the team, the best quality assurance practices, monitoring that the team follows them, and by creating the instruments to measure the quality of the deployed application.

DevOps Master - Rafael Gonçalves - Responsible for managing the infrastructures and environments necessary for the deployment and production of the applications leading their preparation, for trying to automate processes and ensuring that the development frameworks work properly, and for ensuring continuous integration and delivery pipelines and practices.

In addition to the specific roles mentioned above, all team members are also developers. We collaborated to develop software, write code, and implement features and functionality. We also work together to ensure that the software meets the requirements and is delivered on time.

1.2 Agile backlog management and work assignment

In this project, we followed an Agile methodology, with Jira as the main tool for backlog management and work assignment. We used user stories as a unit of work and we did feature-driven development for a total of 4 sprints. Each sprint had epics which were the main goals for the sprint and were composed of user stories and tasks.

Each user story had an acceptance criteria, and a definition of done. The definition of done was used to determine if the user story was completed, more on the definition of done later.

The user stories and tasks were mainly assigned to the corresponding role and prioritised by the Team manager.

2 Code quality management

2.1 Guidelines for contributors (coding style)

For the development of this project, we adopted the coding style of **AOSP (Android Open Source Project)**. The AOSP refers to a set of guidelines and conventions for writing Java code, which is widely recognized and adopted by the developer community, allowing us to maintain a consistent standard throughout the source code. Following a specific guideline like this promotes readability, facilitates maintenance, and encourages collaboration within the team, demonstrating a commitment to producing clean, organised, and high-quality code.
In our case, since the majority of our team was already used to and had a similar coding style, this looked like the best option to adopt.

2.2 Code quality metrics

Static code analysis plays a very important role in the regard of quality assurance best practices. This type of analysis can be performed during the development and writing of the code and its main goal is to identify programming errors, coding standard violations, security vulnerabilities, and potential bugs and errors in the code before running and executing the program. This is usually done by analysing the code against a set of coding rules and standards.

In this project, we used two different static code analysis tools, SonarLint and SonarCloud.
SonarLint was used to analyse the code locally, everyone in the team had it installed in their IDE and was committed to fix the issues that were found by the tool. This was useful because it allowed us to fix the issues before pushing the code to the repository saving us time and effort.
SonarCloud was used to analyse the code in the repository, this was useful because it allowed us to see the issues that were found by the tool in the code that was already pushed to the repository and to see the evolution of the code quality over time. This tool was the one that we had to follow more closely because it was the one that was used to evaluate the quality of the code when pushing some changes to the repository.
Using **SonarCloud** integrated with GitHub, we created a **Quality Gate** that was used to evaluate the quality of the code when pushing some changes to the repository.

Metric	Operator	Value
Coverage	is less than	85.0%
Duplicated Lines (%)	is greater than	1.0%
Maintainability Rating	is worse than	A
Bugs	is greater than	0
Vulnerabilities	is greater than	0
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

Fig.1 - Defined Quality Gates

For the Quality Gates, we opted to use more strict rules than the default ones, this was done to ensure that the code quality was always good and that the code was always ready to be deployed, for example we increased the mandatory percentage of code coverage from 80% to 85%, reduced the maximum percentage of duplicated lines from 3% to 1% and made the maximum quantity of bugs and vulnerabilities to be equal to 0.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Having a fixed code management approach is very important because it allows us to have a clear and defined workflow that everyone in the team can follow and that allows us to have a better control over the code that is being pushed to the repository.
In this project we implemented a **Feature-Branch development** workflow as our code management approach. This kind of workflow is based on the idea that all the new features should be developed in a separate branch and only merged to the main branch when they are ready to be deployed, allowing us to have a better control over the code, a more stable main branch without broken code and allowing developers to work better in parallel without interfering with each other.

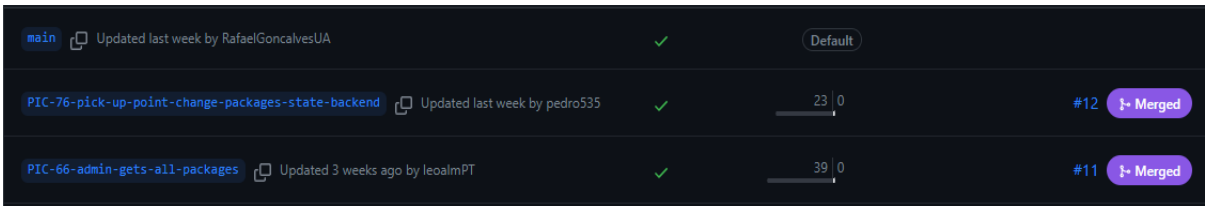


Fig.2 - Feature Branch Workflow

Additionally, we established clear guidelines for the coding review process on a pull request. This process was done by a different team member than the one who created the pull request and it was done to ensure quality, readability, and adherence to best practices in the code that was being merged into the main branch. This new code had to follow and accomplish the coding standards and guidelines that were established by the team, described in the Definition of Done. This process took advantage of GitHub Pipelines, which allowed us to automatically run the tests and the static code analysis tools when a pull request was created, helping the reviewer to have a better understanding of the code that was being merged, since it generated a report with the most important information about the code quality. It was up to the reviewer to decide if the code was ready to be merged or if it needed some changes before being merged, accepting or rejecting the pull request accordingly.

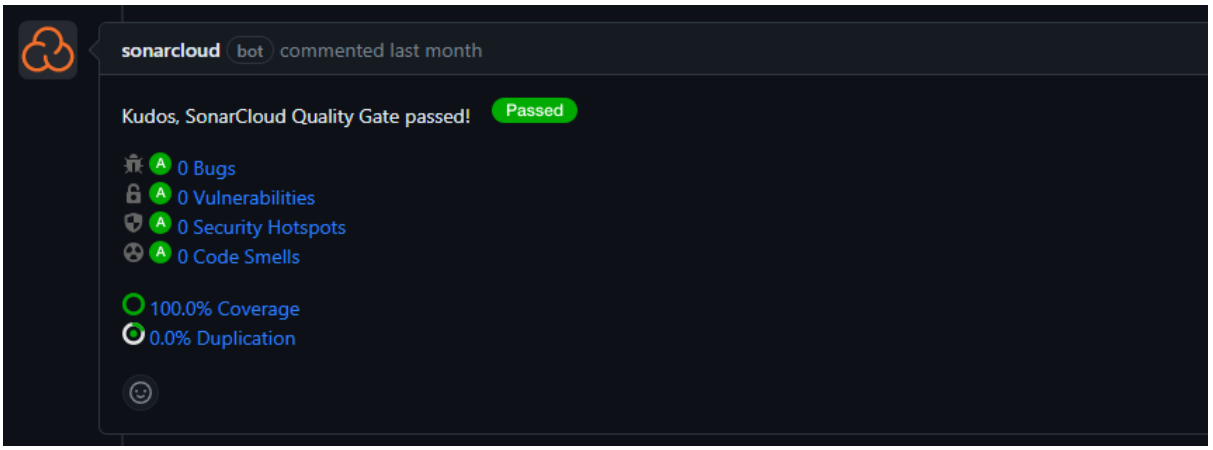


Fig.3 - Reviewer Automatic Quality Gate Report

The definition of done is when all conditions that a software product must satisfy are met and ready to be accepted. For our **Definition of Done**, we defined a clear delineation of what it means for a user story to be completely implemented. We defined that a feature was only completely implemented when the Acceptance criteria was already established, it was fully tested with all the tests passing, the Sonarcloud Quality Gate was also passing, the code was properly documented and the code was reviewed by another team member.

3.2 CI/CD pipeline and tools

Using **GitHub Actions**, we defined a CI/CD pipeline for the 4 repositories: PrintPlate, eStore_Frontend, eStore_Backend and DPP_Backend. Whenever a pull request (PR) is opened, the unit and integration tests are executed. Then, SonarCloud checks the quality gate. If the workflow succeeds, the PR is accepted with a review. After the code increment is merged into the main branch, the new version is deployed on the Google VM through SSH.

We also defined a CI pipeline for repository DPP_BDD, where we do functional testing in the deployed version, but to keep things organised in separated repositories we needed to connect 2 repositories, this one and DPP_Backend and everytime there is a new deployment we wanted to trigger the functional tests.
To achieve this we used repository dispatch shown in the figure below:

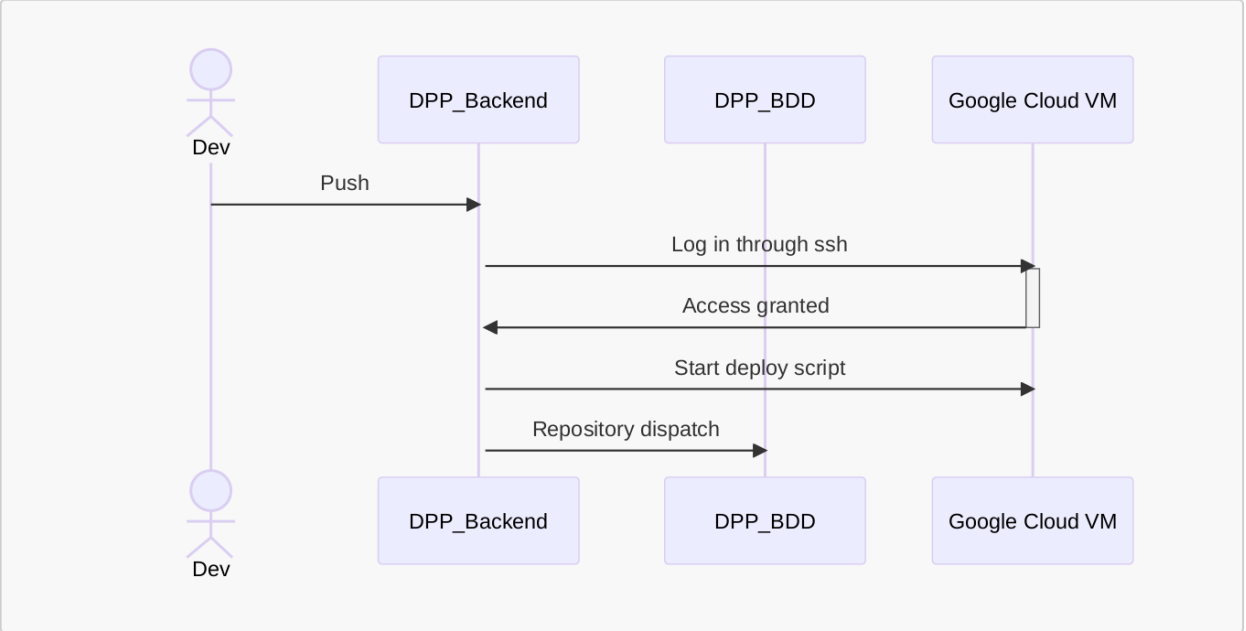


Fig.4 - Repository dispatch - sequence diagram.

3.3 System observability

We have installed **Nagios** on our virtual machine and run its web application, which allows us to remotely monitor our infrastructure.
On the dashboard we can see, for example:

- The processor load
- The number of open sessions
- The latency in HTTP requests or PINGs
- The disk occupation
- If ssh is active or not
- Swap usage
- Total no. of processes

When the host is down, the client in

http://34.175.190.59/nagios/
username: nagiosadmin
password: linux

4 Software testing

4.1 Overall strategy for testing

For the backend we opted to use a **Test-Driven Development** for the unit tests in each component, meaning that we first create the test for the component, then we test to check if the test fails, and only then we implement and develop the code for the component until all the tests pass.
After all unit tests pass in each individual component, we develop the integration tests and check if they pass, if they do not we fix and try again.
For the unit test, we chose to use the following frameworks: JUnit Jupiter, Hamcrest, Mockito, Spring Boot MockMvc, and Jacoco and SonarCloud to check code coverage by the tests.

For the integration tests, we chose to use the following frameworks: JUnit Jupiter, Hamcrest, and REST-Assured.

For the front-end tests, we used Behaviour-Driven Development by first defining some scenarios that reflect the expected behaviour of the Web Application in a human-readable format using the Gherkin Language. From these scenarios and making use of the Cucumber framework, a set of automated tests are developed in order to simulate user interactions and verify that the web application is working and behaving as expected. For developing those automated tests, we chose to use JUnit Jupiter and Hamcrest for the assertion.

To improve the code quality every member of the group was committed to using **SonarLint** to check in real-time quality issues with the code that is being developed.
SonarCloud is being used to check if the code is following the quality gates defined by the team.

4.2 Functional testing/acceptance

Functional testing is a critical aspect of software development that is used to check and ensure that specific features of the application are working correctly and meet the expected requirements.
Through the **Selenium** framework, we are able to simulate user inputs and experiment if the application is reacting as our tests are expecting them to do.

Since some of the test cases were developed even before the start of the development of the application, without the knowledge of how the internal part of the website works, we ended up using closed-box functional testing.

To improve the application and make sure that all the functionalities and specific cases were being tested we, as developers, put ourselves in the position of the user and checked if the application was behaving as we wanted to, so we also used user perspective functional testing.

To make tests look cleaner, avoid duplication and make the code more readable, we are using **Page Object Model pattern**.
As we followed a Behavior-Driven Development, the acceptance criteria of each feature were written using Gherkin Language and converted to tests using **Cucumber** expressions.

4.3 Unit tests

Open box testing involves testing the internal works of the software by analising the source code directly. That way testers can create Unit tests that target these special issues of logic, programming errors, and incorrect data structures.

Developer perspective Unit testing is when developers themselves perform and write the unit tests according to their expected results and requirements. The main objective of this type of testing is to have a high percentage of code coverage and to avoid finding issues on a component too late in the development of the software.

In our Unit tests, what we did was basically a combination of both worlds. Firstly we implemented a Test-Driven Development of the Unit tests where the developers write their expectations of what should the outcome of a determined component be and then, after checking that the tests were good and that they failed before the component implementation, write the component code until all the Unit tests pass. After all the developer Unit tests passed, on the pull request, the source code was analised trying to find some issues that were not being tested and, if needed, created some extra tests to fulfil those special issues with the code.

Unit Tests were mostly used for testing the controller layer by mocking the service layer, for testing the service layer by mocking the Repository layer and Utils layer, and for testing the Utils layer.

For these tests, the frameworks used are JUnit Jupiter, Hamcrest, Mockito, and Spring Boot MockMvc. For checking code quality and test coverage the frameworks used were Jacoco and SonarCloud.

4.4 System and integration testing

After each component is completed with all Unit tests passing, we need to integrate it into our system. For that, Integration tests are really useful because they put into action how each unit interacts with each other in real-world and real-time applications.

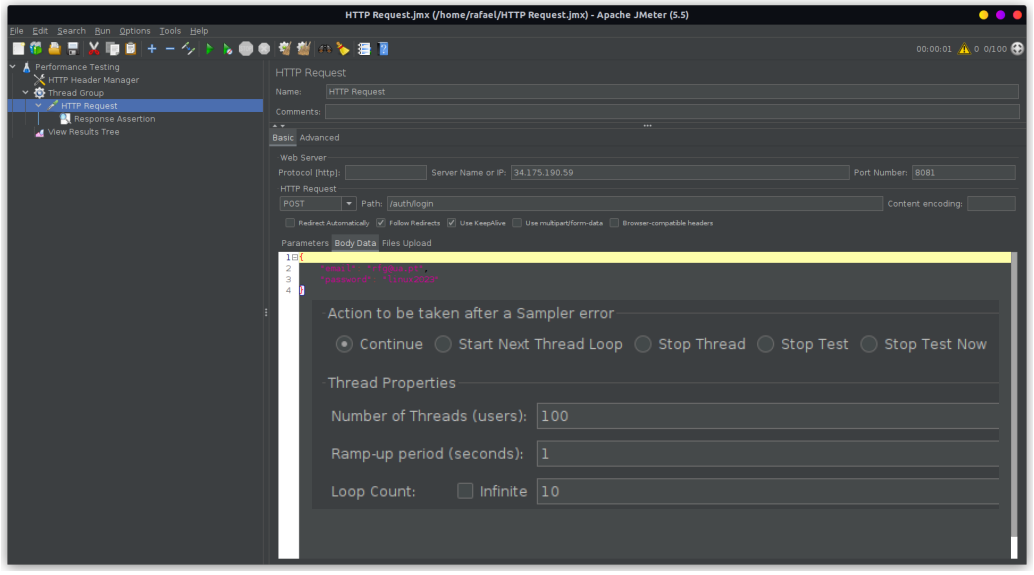
Open box and developer perspective integration tests are basically the same as described before, being used when we create tests while analising the code, and when developers write the tests according to their expectations and features requirements, respectively. On the other hand, closed box integration tests don't rely on understanding the code and main software aspects of the system and are mostly used for measuring system response time, testing if the results and outcomes of a function are the expected, evaluating usability, and troubleshooting issues.

Integration tests were mostly used for testing the interactions between the controller, service, and repository layers. For this, we ended up testing controller-service interactions while mocking the repository, testing service-repository interactions, and testing the interaction between controller-service-repository layers interactions all together.

For these integration tests, the most used frameworks were JUnit Jupiter, Hamcrest, and REST-Assured.

4.5 Performance testing [Optional]

In order to see if our Google infrastructure could handle a large number of HTTP requests, we ran a quick performance test using **JMeter**. First, we injected the necessary fields, like content type, into the request headers. We created a group of 100 threads, which simulated 100 users. Each thread performed 10 POSTs to the login endpoint and JMeter asserted that the response code was always 200. At the end, we presented a table with the results. Note that although the test was created in the GUI, it was executed from the command line, to get more reliable results.



Performance test plan and results can be found in PerfomanceTests folder.