deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Vânia Morais [102383], João Sousa [103415], Catarina Costa [103696]

1 Project management

1.1 Team and roles

Vânia Morais is our team manager, she is responsible for the distribution of the work as well as making sure everyone is doing their job correctly and on time. She is responsible for preparing our eStore frontend.
João Sousa is our DevOps master. He is responsible for preparing our delivery engine and deploying the environments of each component, assuring CI/CD principals
Catarina Costa is our product owner, she is making sure we are making the product she devised. She is responsible for preparing our admin and acp frontend.
Since there is no 4th element in the group, we all have the role of Developer and QA Engineer.

1.2 Agile backlog management and work assignment

Given that this work's development is user story oriented, there are multiple steps that can be listed in order to create and manage the backlog:

1. **Creation of the User Stories**: As a first step in our project, our Team Coordinator created personas and user stories for each persona, and decided on which user stories had priority over each other. These were added to the project's backlog where the other team members get to see, and in case there's something we don't agree with, discuss it as a team. The backlog management tool used was Jira;
2. **Development Time for each User Story**: After the first step was complete, the team had a small meeting in which we focused primarily on trying to estimate the time it would take to have all the user stories developed;
3. **Work Distribution**: With the user stories layed out, the Team Coordinator assigned different tasks to each one of the Developers. In Jira, we could keep track of each others' progress by checking if the task was: "**To Do**", "**In Progress**", or "**Done**". These are all the states we defined for our tasks.
4. **Work Prioritization:** Once the user stories are in the backlog, the team collaboratively prioritizes them based on their value, dependencies, and project goals. This ensures that the most important and valuable user stories are addressed first.
5. **Sprint Planning:** Before the start of each sprint, the team conducts a sprint planning meeting to select a set of user stories from the backlog to work on. The team considers the capacity, estimated effort, and dependencies to determine which user stories can be completed within the sprint.
6. **Task Breakdown and Estimation:** Once the user stories are selected for the sprint, the team breaks them down into smaller tasks or sub-tasks. These tasks are then estimated for effort and complexity. Task breakdown and estimation help in better planning, tracking progress, and identifying potential challenges.
7. **Collaborative Decision-Making: I**mportant decisions regarding user stories, priorities, and technical approaches are made collaboratively within the team. Discussions and consensus-building allow for a shared understanding and ownership of the project, fostering a collaborative and empowered team environment.
8. **Continuous Improvement:** The team embraces a culture of continuous improvement, regularly reflecting on their processes, practices, and outcomes. They seek opportunities to enhance efficiency, productivity, and quality through retrospectives, feedback loops, and adopting best practices.

By incorporating these practices into the agile backlog management and work assignment, the team can effectively plan, prioritize, and execute the development work while maintaining transparency, collaboration,

and flexibility throughout the project. Is important to refer that we used Jira to help us in having a consistent workflow and help us distribute all the work.

2 Code quality management

2.1 Guidelines for contributors (coding style)

In order to make sure our code is concise and error free, we decided on following some different programming style guidelines: Standard Git Flow, Standard Feature Branching and Standard Naming Conventions.
We decided to follow these conventions because it is easier and simple to keep the code consistent.

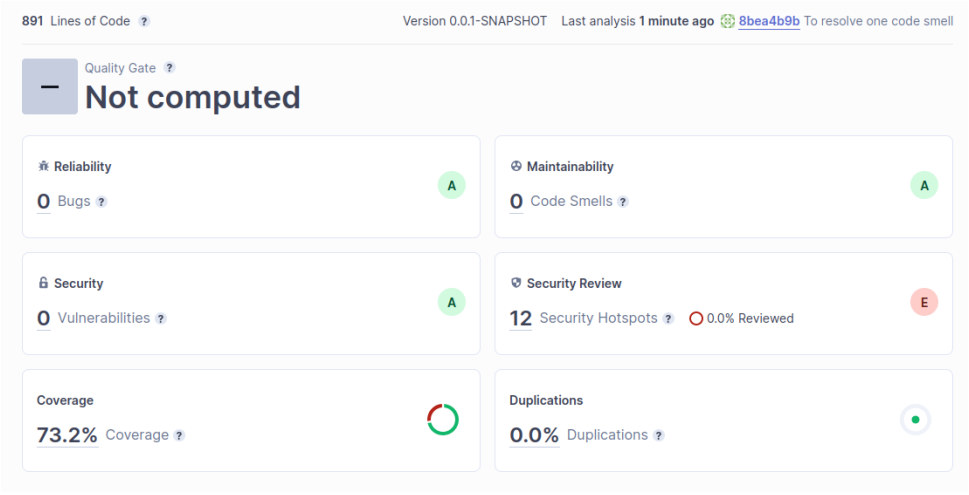
2.2 Code quality metrics

Code quality metrics play a crucial role in ensuring the maintainability, reliability, and overall quality of the codebase. In our project, we adopted several practices for static code analysis to assess the code quality. These practices included utilizing SonarCloud: SonarCloud is a widely used platform for continuous code quality inspection. It analyzes code for bugs, vulnerabilities, code smells, security hotspots, duplications, and test coverage. By integrating SonarCloud into our development workflow, we aimed to identify and address potential code issues early on.

In terms of the associated resources, we leveraged SonarCloud's capabilities to perform automated code analysis. The analysis results provided valuable insights into the codebase, highlighting areas that required attention and improvement.

Regarding the specific metrics for SonarCloud, here are the results for the last analysis:

- Bugs: 0
- Vulnerabilities: 0
- Code smells: 0
- Security hotspots: 12 (attributed to the IP of the VM for the API)
- Duplications: 0%
- Test coverage: 73.2%



Additionally, we established quality gates to ensure the adherence to specific code quality thresholds before accepting merge requests. These thresholds were set based on our project's requirements and

standards, and they encompassed various code quality metrics, including bugs, vulnerabilities, code smells, security hotspots, duplications, and test coverage.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

For this project, we use GitFlow as our branching model. Each person working on the project will have their own repository, including:

- Readly-eStore: Repository for the eStore frontend.
- PackagePal-backend: Repository for the backend.
- PackagePal-frontend: Repository for the admin and ACP frontend.
- Docs: Repository for all project documentation.

The main branches of each repository are protected. When a new feature is completed, a pull request will be created, and another team member, who is not the author of the pull request, will review and approve it. This workflow ensures that all changes go through a review process, contributing to code quality and collaboration within the team.

A user story will be considered complete when all the related features are functioning as intended and have been thoroughly tested.

3.2 CI/CD pipeline and tools

In our CI/CD pipeline, we utilize GitHub Actions for automating various tasks. Here are some of the practices and tools used:

- Code Analysis: SonarCloud is integrated into the pipeline to ensure that code submitted to the main branches meets quality standards. SonarCloud provides quality gate checks, code coverage checks, and security vulnerability analysis to ensure code reliability and maintainability.
- Continuous Delivery with Containers: Our continuous delivery process is based on containers. We leverage containerization technologies like Docker to package and deploy our applications consistently across different environments. We also tried to deploy via GitHub Actions but it didn't work properly.

3.3 System observability

System observability is an essential aspect of our project. We employ various tools and practices to monitor and gain insights into the system's behavior, including:

- Logging: We implement logging mechanisms to capture relevant information about the application's execution, errors, and events.

4 Software testing

4.1 Overall strategy for testing

The overall test development strategy in your project involved a combination of different testing approaches to ensure comprehensive coverage and quality assurance. The following strategies were adopted:

- Test-Driven Development (TDD): Tests were written before implementing the source code, ensuring that the functionality is correctly implemented from the beginning and facilitating code improvement.
- Continuous Testing: Tests were conducted throughout the development process, allowing for early detection of bugs and ensuring that the code is continuously tested and improved.
- Code Reviews: The code was reviewed by peers or senior developers, providing an additional layer of quality assurance and identifying potential issues or improvements.

4.2 Functional testing/acceptance

In our project, functional testing and acceptance testing were conducted from a closed box perspective, focusing on the user's perspective. The policies and practices adopted for writing functional tests included:

- Writing test cases that simulate user interactions with the system, covering different use cases and scenarios.
- Validating the system's behavior and ensuring that it meets the functional requirements specified in the project.
- Utilizing testing frameworks or libraries specific to the programming language or framework used in the project like:
    - Hamcrest: A matching library used in tests to express expectations in a more readable way. It provides methods like `is()`, `equalTo()`, `hasSize()`, among others, which are used to verify the expected conditions in test results.

Associated resources for functional testing/acceptance may include test cases, test data, and any necessary tools or frameworks used for executing the tests.

4.3 Unit tests

Unit testing in your project followed an open box perspective, focusing on the developer's viewpoint. The project policy for writing unit tests included:

- Writing tests for individual units of code, such as methods or classes, to ensure their correctness and isolate any potential issues.
- Using testing frameworks or libraries specific to the programming language or framework employed in the project like:
    - JUnit 5: A widely used testing framework for unit testing in Java. It provides annotations and assertions to facilitate the writing and execution of tests.

    - Mockito: A Java mocking library used to create simulated objects (mocks) of dependencies in unit tests. It allows simulating the behavior of external classes and objects, facilitating the isolation of components during testing.
- Testing various scenarios, including edge cases and input/output validations.

Associated resources for unit testing may include test classes, test data, and any necessary testing frameworks or libraries.

4.4 System and integration testing

System and integration testing in your project were conducted from a developer's perspective, combining open and closed box approaches. The policy for writing integration tests involved:

- Verifying the interaction between different components of the system, such as APIs, databases, and external services.
- Testing the system as a whole to ensure the proper functioning of integrated modules.
- Utilizing specific testing frameworks or libraries suitable for integration testing like:
    - Spring Boot Test: An extension of the Spring Framework for integration testing and unit testing in Spring Boot applications. It provides features for configuring and running tests in Spring Boot application contexts, as well as for making simulated HTTP requests to REST controllers.
    - TestEntityManager: A class provided by Spring Boot for testing data access with Java Persistence API (JPA). It allows performing persistence operations on an in-memory database during tests.

API testing was performed as part of system and integration testing, focusing on testing the API endpoints, data validation, and ensuring the API functions according to the specified requirements.

Associated resources for system and integration testing may include test suites, test scripts, test data, and any required testing tools or frameworks.