TQS: Quality Assurance manual

Daniel Ferreira [102442], Guilherme Antunes [103600], Mariana Andrade [103823], Vicente Barros [97787]
v2023-06-05

1.1 Team and roles 1
1.2 Agile backlog management and work assignment 2
2.1 Guidelines for contributors (coding style) 2
2.2 Code quality metrics 2
3.1 Development workflow 2
3.2 CI/CD pipeline and tools 3
3.3 System observability 5
4.1 Overall strategy for testing 5
4.2 Functional testing/acceptance 5
4.3 Unit tests 6
4.4 System and integration testing 6
4.5 Performance testing 6

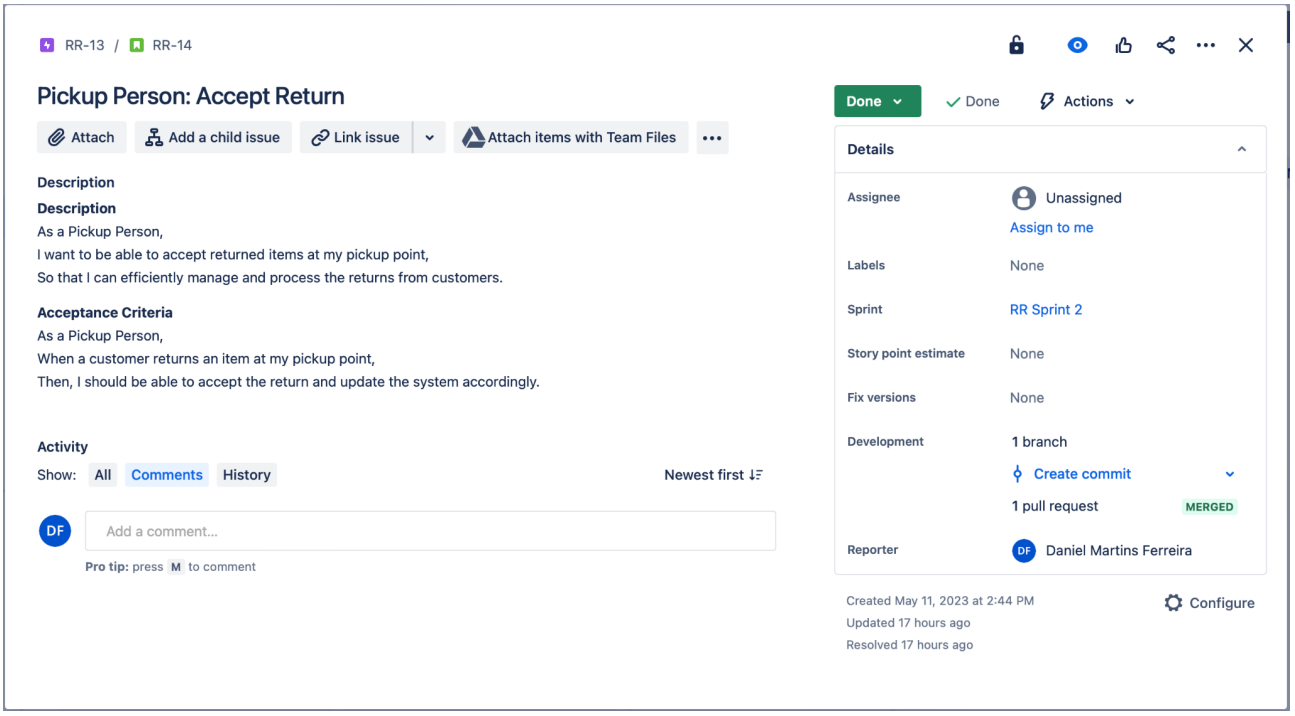
1 Project management

1.1 Team and roles

Member	Role	Description
Daniel	Team Coordinator	Responsible for coordinating team activities and ensuring effective communication within the development team.
Guilherme	QA Engineer	Responsible for ensuring software quality by designing and executing tests and identifying defects.
Mariana	Product Owner	Responsible for defining and prioritizing product features and functionalities in collaboration with the development team.
Vicente	DevOps Master	Responsible for integrating development and operations processes to achieve continuous delivery and deployment.

1.2 Agile backlog management and work assignment

This project was organized with an Agile methodology in mind. All of the development was mapped to user stories in our backlog management: Jira.



Each story consists of a description that enumerates the respective actor's motivation and a acc

2 Code quality management

2.1 Guidelines for contributors (coding style)

The coding style adopted in this project is based on the camelCase convention, applied in both the backend code development and Java tests, as well as in the frontend with TypeScript. Variable names should be descriptive and clearly reflect their purpose in the code. Similarly, all functions, classes, interfaces, and other named elements should be clear and self-explanatory.

It is recommended to develop the code in a modular way, aiming for clean, organized, and readable code. The use of modules should clearly separate different functionalities, favoring reusability whenever possible and necessary.

It is important for the code to be accompanied by documentation, whenever necessary, by including comments that help explain its functionality, thereby avoiding possible doubts. The main objective is to complement maintenance strategies, ensuring code readability and clarity.

2.2 Code quality metrics

The metrics established to assess code quality are based on an analysis that must be performed by the SonarCloud tool. Through its integration with GitHub Actions, the "main" branch will receive automatic analyses whenever there are code changes, ensuring a continuous and systematic analysis.

The acceptance criteria defined as quality gates include a minimum test coverage of 80%, absence of vulnerabilities, a maximum computational complexity of O(2n), preferably no code duplication, and no bugs or security risks.

These quality gates aim to ensure high quality and reliability of the software. Each measure aims to achieve comprehensive test coverage to avoid unnoticed code errors, prioritize code and data security, reflecting a zero tolerance policy towards vulnerabilities, bugs, and security risks. Additionally, the goal is to maintain readable, easily maintainable, and efficient code by adhering to the policy of avoiding code duplication and keeping computational complexity low.

All these measures result in time and effort savings in problem resolution, with the aim of minimizing setbacks.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The Git workflow we have adopted for our project is an adapted version of the standard GitFlow. We have two primary branches, the 'main' branch, and the 'dev' branch. The 'main' branch is used for our production releases, while the 'dev' branch is where all the developmental work happens.
When a catastrophic bug is discovered, a 'hotfix' branch is created directly from the 'main' branch. After the bug is fixed, the changes are merged back into both the 'main' and 'dev' branches. There's no need for a separate 'release' branch in this case.
For new features, a unique branch is created from the 'dev' branch. Once the feature development is complete, it's first merged back into the 'dev' branch, then into the 'main' branch when ready for a production release.
For our branch naming conventions, we use the following categories: 'new' for adding new features, 'bug' for fixing bugs, 'imp' for improving existing features, 'wip' for long-lasting tasks, 'junk' for experimental branches, and 'release' for new releases prior to merging with the main branch.

For pull requests, we require a summary in the title and detailed descriptions. The description should include what was done, what was fixed if relevant, and if applicable, how the implementation was tested. Any other pertinent information should also be included. Additionally, each pull request is associated with a user story, identified by its unique ID from our backlog management system. This link between the user story and the pull request ensures that every code change can be traced back to its originating user story.

As part of our code review process, we have implemented a policy where code changes are reviewed by at least one person who was not involved in the development. This helps ensure a fresh perspective and provides valuable feedback for improvement. To further enhance our code quality efforts, we leverage SonarQube, a powerful static code analysis tool.
SonarQube is integrated into our code review process to automate the analysis of our codebase. It offers a comprehensive range of features and insights, helping us identify potential issues such as code quality problems, security vulnerabilities, and compliance violations. By utilizing SonarQube, we proactively address these concerns and maintain a high standard of code quality throughout our projects.
The combination of human code review and SonarQube's automated analysis empowers us to identify and resolve code-related issues efficiently. This approach enhances the reliability, maintainability, and

overall quality of our software, reducing risks and ensuring the delivery of robust solutions to our users.
By incorporating SonarQube into our code review process, we demonstrate our commitment to continuous improvement and the delivery of high-quality code. It provides us with valuable insights that help us deliver reliable software solutions to meet our users' needs.

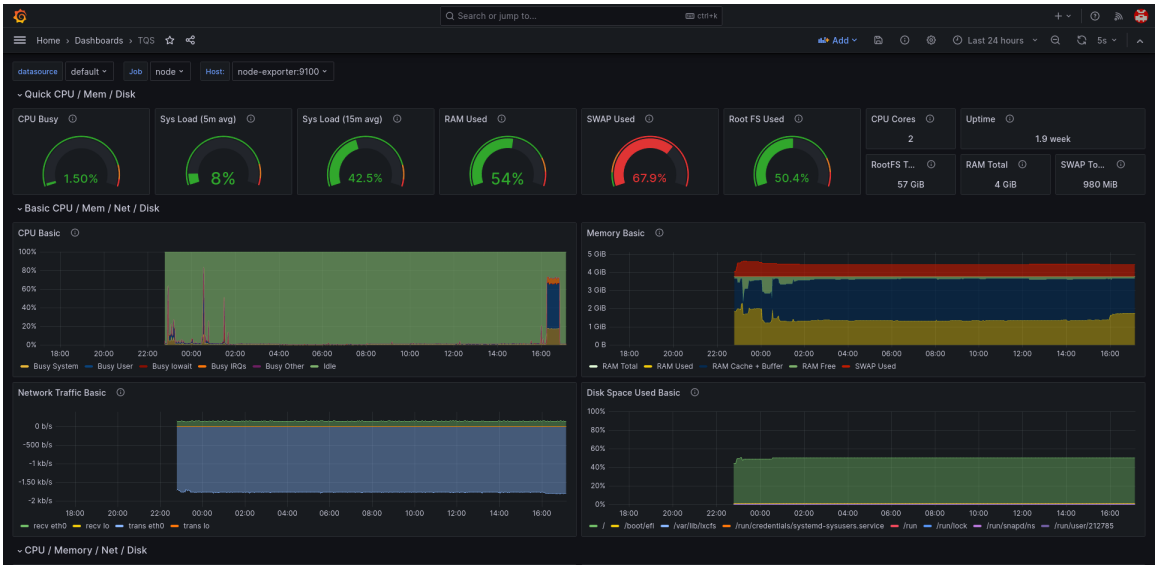
A user story is considered complete within the Quality Assurance process when the described functionality has been successfully implemented in the system's code. This includes the implementation of thorough unit and integration tests to ensure the correct functionality of the implemented features.
In addition, the user story must meet the acceptance criteria defined in its description within the backlog management system. It should be properly documented, providing clear and concise information regarding its purpose, functionality, and any specific requirements or constraints.
Ultimately, the user story's completion is marked when the corresponding features undergo a pull request, and the changes are successfully reviewed and accepted by the relevant stakeholders in the development team.

3.2 CI/CD pipeline and tools

From the beginning it was implemented a CI pipeline with the help of Github tools. The first default configuration to be changed was the need for a reviewer on every pull request, where the reviewer is not the same person who purposed the changes.
Secondly, in the development repositories, with the help of the Github Actions, were set up routines that on every pull request checked if the tests were still being passed with the new code to assure the quality of it. In addition, it was set up on every development repository a sonar cloud analysis to keep the code free from bugs, and code smells and check if the new code was tested. For the pull request to be accepted, all the previous obligations were necessary to be approved. In the Sonar Cloud, some changes were made to the Quality Gates of the repositories of the Shop, because just the simpler tests were made. In the main backend repository, the quality was set to default as it was the main part of the project.
For the delivery part, all the repositories have a Dockerfile that allows their containerization of them. In the backend ones, the Dockerfiles have the obligation to run the maven tests and if the tests fail the process end without affecting the current deployment. For the front end, a similar routine is made because of the usage of Typescript. The Typescript needs to be transpiled into vanilla Javascript and in that process is made type checking analysis and if it fails, the process will stop and the current deployment is not affected.
The deployment is set up in a VM from the University and it was implemented by running a Github Actions runner in the VM that on every change on the main branch of the Roadrunner backend will update the container in the deployment with the help of the Watchtower container.
To orchestrate all these interactions, in the main repository exists a docker-compose file that deals with all the parts of launching and managing the container, with the help of a .env file.

3.3 System observability

To check the status of the deployment, a monitoring system was created with the help of a Grafana Dashboard (using the cloud version offered by Grafana), and a Prometheus Database that checks all the information of the system such as active threads, memory and CPU usage.



4 Software testing

4.1 Overall strategy for testing

In shaping the back-end, we employed a Test-First approach for every module's unit testing, which signifies that we initially design the test for the module, assess if the test falls short, and only afterwards do we devise and evolve the code for the module until every test is successful.
Upon the successful execution of all unit tests in each isolated module, we forge the integration tests and verify their success. If they fall short, we amend and reattempt. For the unit test, our selection of frameworks consisted of: JUnit Jupiter, Hamcrest, Mockito, Spring Boot MockMvc, and for code coverage analysis, we utilized Jacoco and SonarCloud. For the integration tests, we opted for the following frameworks: JUnit Jupiter, Hamcrest.
Regarding the front-end evaluations, we implemented Behaviour-Focused Development by initially laying out scenarios that represent the anticipated operation of the Web Application in a format comprehensible to humans using the Gherkin Language. From these situations and with the assistance of the Cucumber framework, a sequence of automated assessments are devised to replicate user interactions and affirm that the web application functions and behaves as intended. For developing these automated tests, we selected JUnit Jupiter and Hamcrest for the assertion.
SonarCloud is utilized to verify adherence to the quality gates defined by the team. To ensure these practices are adhered to, a Continuous Integration (CI) pipeline is employed to execute all tests, and review code coverage, bugs, vulnerabilities, and high-risk code smells. Consequently, we guarantee that pull requests are only endorsed if all these facets successfully pass their tests.

4.2 Functional testing/acceptance

Functional testing forms a crucial component of software development utilized to validate and confirm that specific features of the application function properly and satisfy the expected requirements. With the Selenium framework, we can mimic user inputs and gauge if the application responds as our tests anticipate. As some of the test cases were devised even prior to the commencement of the

application development, without insights into the internal operation of the website, we resorted to black-box functional testing. To enhance the application and confirm that all functionalities and specific scenarios were being evaluated, we, as developers, placed ourselves in the shoes of the user and assessed if the application behaved as per our expectations. Hence, we also adopted user viewpoint functional testing. To present the tests in a cleaner format, prevent repetition, and enhance code readability, we have adopted the Page Object Model pattern. As we are implementing Behavior-Driven Development, the features were composed using Gherkin Language and converted into tests employing Cucumber expressions.

4.3 Unit tests

White box testing encompasses the examination of the software's inner workings through direct scrutiny of the source code, thereby enabling testers to formulate Unit tests specifically tailored to address intricate issues of logic, programming inconsistencies, and flawed data structures. The approach of developer-perspective Unit testing involves developers themselves authoring and conducting the unit tests, based on their anticipated outcomes and requirements. The chief objective of this testing modality is to achieve an expansive coverage of code and to preclude the discovery of component issues at a late stage in the software development lifecycle. Our Unit testing strategy effectively amalgamated the best of both methodologies. Initially, we adopted a Test-Driven Development paradigm for Unit tests, wherein developers articulated their expected results for a specific component and subsequently, upon verifying the robustness and initial failure of the tests prior to component implementation, proceeded to develop the component code until all Unit tests were successfully passed. Once all developer-authored Unit tests achieved success, during the pull request phase, the QA engineer conducted a meticulous analysis of the source code to uncover any untested potential issues and, if necessary, authored additional tests to address these specific code anomalies. Unit tests were predominantly employed to evaluate the controller layer through the simulation of the service layer, and to test the service layer by mocking the Repository layer. For these assessments, we utilized a suite of frameworks including JUnit Jupiter, Hamcrest, Mockito, and Spring Boot MockMvc. Code quality and test coverage were evaluated through the application of the Jacoco and SonarCloud frameworks.

4.4 System and integration testing

Upon the successful execution of all Unit tests and the completion of each component, the necessity arises to incorporate it into our overarching system. For this purpose, Integration tests prove to be invaluable as they simulate the interaction between units within the context of real-world and real-time applications. The principles of open box and developer-perspective integration testing mirror those elucidated earlier, coming into play when we construct tests by examining the code, and when developers script the tests in alignment with their anticipations and feature prerequisites, respectively. Conversely, black box integration tests don't necessitate a comprehension of the code and the core software facets of the system. They are primarily employed to gauge system response time, verify if the results and outcomes of a function align with expectations, appraise usability, and troubleshoot issues. Integration tests were predominantly utilized to examine the interplay between the controller, service, and repository layers. Consequently, we tested controller-service interactions while simulating the repository, scrutinized service-repository interactions, and assessed the interaction between controller-service-repository layers in a holistic manner. For these integration tests, the frameworks we chiefly relied on were JUnit Jupiter, Hamcrest, and RestAssured.

4.5 Performance testing

To ensure the quality and reliability of the backend, we used performance testing with the tool JMeter, testing the quality of responses of the Roadrunner and Shop API responses. For that, several threads are making requests to the APIs to simulate user requests. When all the jobs are done, a report is generated with a detailed analysis of the test