TQS: Quality Assurance manual

Miguel Alexandre Moura Neto [119302]
Alexandre Filipe da Paula Andrade [119279]
Thiago Aurélio Pires Barbosa Vicente [121497]
João Pedro Silva Pereira [120010]

v2025-11-25

Contents

[This report should be written with the following objectives in mind:

- Provide **clear guidelines** for project members needing to learn what the QA practices are. Use concise and informative content, allowing other software engineers to understand and comply with the practices.

- Provide context and **rationale** to the practices defined and **offer evidence of the tools/setups** used by the team. **Hyperlink team resources** in the text so it is possible to directly access the resources cited in context.

Tips on the expected content (marked in colored text) along with the document are meant to be removed.
You may use English or Portuguese; do not mix.]

1 Project management

1.1 Assigned roles

Team Coordinator: Alexandre Andrade

Product owner : Miguel Neto

QA Engineer: João Pereira

DevOps master : Thiago Vicente

1.2 Backlog grooming and progress monitoring

What are the practices to organize the work (in JIRA)? Don't need to explain the concepts, but how your team works in routine (adopting agile project management practices, based on user-stories).
How is the progress tracked in a regular basis? E.g.: story points, burndown charts,…
Is there a proactive monitoring of requirements-level coverage? (with test management tools integrated in JIRA, like X-Ray)
We use JIRA as our organization tool.
We are following agile project management practices, so we use an incremental method, using sprints to manage work. The progress is tracked using story points, and by regularly checking in on our sprint to see if we are following the calendar correctly.
We are using X-RAY on Jira to monitor the tests and integrate the testing on the organizing part of our project.
This is our Jira project: Jira Project

2 Code quality management

2.1 Team policy for the use of generative AI

Clarify the team position on the use of AI-assistants, for production and test code
Give practical advice for newcomers.
Be clear about "do"s and "Don't"s
AI-assistants might be used to generate code for production , as it will be tested.
For the tests, AI-assistants will not be used, because the tests ensure that the code is working

and that it has quality, so it needs to be done by hand, as AI can mess up those tests and it would be hard to find out.

2.2 Guidelines for contributors

Coding style

[Definition of coding style adopted. You don't need to be exhaustive; rather highlight some key concepts/options and refer to a more comprehensive resource (that you may have adopted) for details. → e.g.: AOS project]

Code reviewing

Instructions for effective code reviewing. When to do? Integrate AI tools?...

Feel free to add more section as needed.

Code reviews are done by the person responsible for each user story.We 'll use copilot to help with the reviews but he won't be enough to accept the pr. All pull requests must also not have a single syntax error, in that case it'll be automatically rejected.

2.3 Code quality metrics and dashboards

[Description of practices defined in the project for *static code analysis* and associated resources.]
[Which quality gates were defined? What was the rationale?]
For code quality metrics, we are using SonarQube. As the quality gate we'll be using the default from sonar.
This is our SonarQube Dashboard: SonarQube Dashboard

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

[Explain, for a newcomer, what is the team coding workflow: how does a developer get a story to work on? Etc…
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.]
First of all, we choose the tasks/user stories for the next sprint, after that we assign each task/user story to one developer.
We will use GitFlow workflow.
Each user story will have a feature created on the git repository and after it is done, a pull-request will be done and after a review it will (or not) be merged into the dev branch. When it's ready for production it will be put on the main branch.
When a commit is pushed onto github a SonarQube code analysis is triggered and the results are displayed on the dashboard.

Definition of done

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

A user story is considered done, when its features are done, passing all of the tests and passing the static code analysis review.

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 tool's setup and config.]
[Description of practices for continuous delivery, likely to be based on *containers*]
[Which different environments are included in CD? Staging, production…]
For continuous integration, we have github actions running tests on each new code that is committed, that way we can address bugs and errors early, before they cause damage. Each commit is also automatically analysed by SonarQube to check for code quality.
For continuous delivery … (???).

3.3 System observability

What was prepared to ensure proactive monitoring of the system operational conditions? Which events/alarms are triggered? Which data is collected for assessment?...

3.4 Artifacts repository [Optional]

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

4 Continuous testing

4.1 Overall testing strategy

[what was the overall test development strategy? How are you **aligning testing with CI/CD**? E.g.: did you do TDD? Did you choose to use Cucumber and BDD? Did you mix different testing tools, like REST-Assured and Cucumber?...]
[do not write here the contents of the tests, but to explain the policies/practices adopted and generate evidence that the test results are being considered in the CI process.]
For testing, we will be doing TDD to better align with CI (first we make the tests and they don't pass, and only after the code is done the tests will start to pass). Each component that is created must have unit tests. If a component is modified and a new feature is added, unit tests should be modified to account for that, but if a component is modified, and the modifications just changed the implementation and didn't add new features, the current tests are enough.
[o resto dos testes temos que discutir melhor]

4.2 Acceptance testing and ATDD

[Project policy for writing acceptance tests (closed box, user-facing perspective) and associated resources. when does a developer need to develop these?
For writing acceptance tests, we will be using BDD with Cucumber and using Playwright for UI. The acceptance tests will test the behavior of the system in a user-facing perspective way,

through the UI and also use a black-box approach since it won't care about the system architecture, just the behavior.
The developer needs to write these each time a user story is completed, or code changes to add a new feature, or fixing a bug that affects user experience.

4.3 Developer facing tests (unit, integration)

[Project policy for writing unit tests (open box, developer perspective) and associated resources: when does a developer need to write unit test?
What are the most relevant unit tests used in the project?]

[Project policy for writing integration tests (open or closed box, developer perspective) and associated resources.]
API testing
For unit tests, they should be open box and developer perspective, as they should validate classes and internal logic while isolated. We will use Junit with Assertj for unit testing. Unit tests shall be written for each class that has no trivial code (ex: only getters and setters dont need to be tested, when code is refactored unit tests need to be modified to take that into account, a new feature is added on a class. The most relevant unit tests are the ones that cover the business logic, as they validate how the system core works.

For integration tests, we will use a closed box developer perspective, but for some Spring Boot integration tests, they might be open-box (ex: testing how the repository interacts with the DB). Integration tests shall be written when the API changes, when the connection between layers changes, when one layer changes its interfaces or when a external service is implemented.

4.4 Exploratory testing

[strategy for non-scripted tests, if any]

4.5 Non-function and architecture attributes testing

[Project policy for writing performance tests and associated resources.]