TQS: Quality Assurance manual

Pedro Ponte [98059], Alexandre Regalado [124572], Miguel Soares Francisco [108304], Inês Ferreira [104415]
v2025-04-30

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

1 Project management

1.1 Assigned roles

- Team Leader - Pedro Ponte

- QA Engineer - Alexandre Regalado
- DevOps Master - Miguel Soares Francisco
- Product Owner - Inês Ferreira
- Developer - Everyone

1.2 Backlog grooming and progress monitoring

What are the practices to organize the work in JIRA?
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)

In this project, we use JIRA to organize and monitor the backlog in an agile workflow, this helps ensure tasks are distributed among the team members and progress is tracked.

Story points are used as the primary measure of effort, where 1 story point equals 1 hour of work. Tasks are estimated based on this ratio, making it easier to plan workloads evenly between team members. To improve organization, tags are applied to classify tasks by categories such as *frontend*, *backend*, or *QA*, allowing for quick filtering. Work is distributed based on team roles, although in some iterations some roles have a bigger workload than others, so other members may balance this out by helping them. Epics represent the big picture work, like *Reports* for all work related to elaborating this report and the Specification Report, and complex stories are divided into subtasks, as some of them require more work and Jira does not allow multiple assignees in one unique task. The workflow follows the usual structure: **To Do -> In Progress -> Done** for tracking progress.

To track work progress, we rely on JIRA's integration with GitHub. Branches are created directly from JIRA issues or by using JIRA notations, such as *CH-19-Create-use-case-diagram*, where *CH-19* is the issue key and *Create use case diagram* is the task name. Commits are associated with their corresponding issues by including the issue key in commit messages, making it easier to track development work and tasks. Pull requests (PRs) are also created through JIRA, linking them directly to the associated work and providing an easy way to review the implementation of a task.

Progress is monitored using tools like burnup charts, which display the completed work in the sprint compared to work left in the scope of a single sprint.

É PRECISO IMPLEMENTAR ISTO Test coverage and requirements-level monitoring are handled through test management tools integrated into JIRA, such as Xray or Zephyr. These tools link test cases directly to JIRA issues, ensuring every requirement is covered by at least one test. Automated and manual test results are also tracked in JIRA, making it easy to monitor coverage and identify gaps. Dashboards provide an overview of key metrics, such as unresolved defects, test coverage, and overall progress. This proactive approach ensures all requirements are tested and risks are minimized.

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

For this project, everyone is allowed to use AI tools as they see fit, whether it's for generating functional code, API tests, asking for help regarding deployment tools or help in writing the reports. However, this usage comes with great responsibility and all members should **always** review the work done before submitting to prevent unknown issues, as the submission is not made by AI, but actual developers who will take the blame if a problem arises.

For technical questions and how to implement certain features using external tools like using a weather API or creating workflows, AI can be used to create such implementations, but the prompts should when possible include examples provided in the updated official documentation, so that the model can use the most recent practices and everyone as an easier time when trying to understand it.

For documental text like reports it shouldn't be used to generate all text based on title only, we should rather provide the section title, provided instructions from the template and a brief explanation of our situation, trying to write as much information as possible and using AI tools only for grammatical/lexical improvements and better explainability of the general problem.

Jira also provides an AI tool for helping with writing task descriptions, the tool will be applied after the description is manually written, this way we can have a standard format in all tasks making them more readable and complete.

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 a more comprehensive resource for details. → e.g.: AOS project]

For better consistency and code readability we'll be following the Google Java Style Guide guidelines, this includes:

- Naming - UpperCamelCase for class, lowerCamelCase for methods and non-constant field names
- Formatting - one statement per line, vertical whitespace, block comment style (/* … */), TODO comments

We also have a github-actions workflow using google-java-format to automatically reformat committed code to abide by the aforementioned guidelines.

For the *frontend* we will follow Airbnb JavaScript Style Guide with principles like:

- Naming - PascalCase for files, camelCase for instances
- Spacing
- Props

Code reviewing

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

Feel free to add more section as needed.

When code reviewing we use GitHub's AI tool Copilot that can be used in pull requests to review code and give feedback. It explains the implemented logic and tries to provide detailed feedback on features that may be skipped by human reviewers like some dead code and/or duplicate code. After the AI review, at least two other reviewers must look at the changes and approve them so they can be merged to our *development* branch.

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?]

ESTA PARTE É DO ALEXANDRE, TEM A VER COM O INTEGRAR O SONARQUBE NAS WORKFLOWS

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.]

For our workflow we follow the gitflow workflow. Our repository has a *main* branch and a *develop* branch.

Develop is used to keep the most recent working code, this branch is updated regularly with work from other branches being merged into it after an approved pull request, it is impossible to directly commit anything to this branch. These other branches are created every time new work is done and must be associated with a relevant task already defined in the Jira backlog by following the naming rules. The tasks are created and assigned by the Team Leader, but anyone can create tasks they find relevant.

Main can only be updated with a pull request accepted by three members, however to contribute to it, the work must come either from a *release* branch, which happen at the end of every sprint and are created based on *develop*, or from a *hotfix* branch, used for introducing fixes of **critical** problems found only after *main* was merged.

Definition of done

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

A user story can be defined as **Done** when the Acceptance Criteria are met and be confirmed through tests defined by us.

These tests are both functional, using tools for testing the User Story through a user interface and following the Acceptance Criteria and non-functional, by running a series of tests that will check our business logic, services and the API making sure everything is in order.

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*]

We use a CI/CD pipeline configured with **GitHub Actions** to automate our workflow. Each push or merge request to the `develop` branch triggers the following jobs:

- Build validation using Maven

- Unit tests execution with JUnit and Mockito
- Code coverage analysis with JaCoCo
- Static code analysis via SonarQube
- Integration tests using REST Assured and Spring Boot Test
- Linting and formatting
- CI status badges included in the README

For **continuous delivery**, the application is containerized using **Docker**, and deployment to a virtual machine provided by the professor is triggered manually or through the release pipeline. The final release is pushed to the `main` branch through a pull request, ensuring all checks are passed and peer reviews are approved. **DEVOPS tem de rever esta parte!!!!**

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?...

The project includes basic observability features to monitor the system's health. The following data is tracked:

- HTTP status codes and requests logs for all REST endpoints
- Error rates and response times using **Spring Boot Actuator**
- Test pass/fail status and code coverage via CI pipeline
- Static analysis results in SonarQube

Build failures, quality gate violations, or broken tests in GitHub Actions trigger email or platform-based notifications to the development team. Additional alerts are reviewed manually during code reviews and QA validation steps.

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]

The project currently uses public Maven repositories for dependency management and does not maintain a local artifacts repository.

4 Software testing

4.1 Overall testing strategy

[what was the overall test development strategy? 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.]

Our testing strategy follows multiple approaches, combining:

- **Test-Driven Development (TDD)** for core backend services
- **Behaviour-Driven Development (BDD)** using **Cucumber**, with files written in Gherkin
- Integration testing using **REST Assured**
- **Code coverage enforcement** via **JaCoCo**
- All automated tests are run as part of the GitHub Actions pipeline, and their status is reviewed before merging pull requests.

4.2 Functional testing and ATDD

[Project policy for writing functional tests (closed box, user perspective) and associated resources. when does a developer need to develop these?

Functional tests are written using the Acceptance Test-Driven Development (ATTD) methodology.

Acceptance criteria from each user story are translated into test cases, which are then implemented using Cucumber.

These tests validate that the system behaviour matches the expectations from a user's point of view. Developers must implement and pass these tests before a story can be marked "done."

4.3 Unit tests

[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?]

Unit testing is required for all core components. The tem uses JUnit for writing unit tests and Mockito for mocking dependencies. Unit tests are required when:

- A new service or controller is introduced
- A method includes logic or branching
- Bug fixes are implemented

Unit tests are executed in every CI run, and the project enforces a minimum of 80% test coverage using JaCoCo.

4.4 System and integration testing

[Project policy for writing integration tests (open or closed box, developer perspective) and associated resources.]
API testing

Integration testing ensures that system components interact correctly, especially across layers (controller -> service -> repository)

We use:

- **Spring Boot Test** for application context tests
- **REST Assured** for validating HTTP endpoints and API contracts
- Test cases include both success and error scenarios, database access, and service coordination

These are triggered after unit tests in the CI pipeline.

4.5 Non-function and architecture attributes testing

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

Performance and load testing are conducted using **K6**. Simulations include:

- Multiple users booking slots simultaneously
- Charging session initiation and termination

These scripts are run periodically and results are stored in the `/performance-docs/` folder. Performance regressions are monitored manually, and test failures are flagged before production deployment.