TQS: Quality Assurance manual

Gabriel Santos [113682], Guilherme Santos [113893],

João Gaspar [114514], Shelton Agostinho [115697]

v2025-05-14

Contents

[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

Gabriel Santos - Product Owner
Guilherme Santos - QA Engineer
João Gaspar - DevOps master
Shelton Agostinho - Team Coordinator

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)

Organizing Work in JIRA

- **Epics → Stories → Sub-tasks**
  We structure all work around **Epics**, each representing a major feature (e.g. Station Discovery, Booking). Under each Epic we create **User Stories** (as defined in our product backlog) and break those Stories into **sub-tasks** for implementation, testing or documentation.
- **Backlog Refinement**
  At the start of each sprint we hold a **Backlog Grooming** session: the Product Owner and team review upcoming Stories, confirm acceptance criteria, and ensure each Story is "Ready" (clear description, estimate, dependencies resolved).

Tracking Progress

- **Story Point Estimation**
  In our Sprint Planning we assign **story points** to each Story, reflecting relative effort. This helps the team commit to a realistic amount of work.
- **Sprint Board**
  Our Scrum board has columns **To Do**, **In Progress**, **Done** to ensure Stories flow smoothly to completion.

Requirement-Level Coverage Monitoring

- **Manual Traceability via Sub-tasks**
  We do not yet integrate a dedicated test-management plugin in JIRA. Instead, each Story contains sub-tasks. When all sub-tasks are complete, the Story is completed
- **Linking Code & Tests**
  Developers link their Pull Requests and CI build results back to the corresponding JIRA issue. This provides a manual but clear traceabilit.

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

We allow the use of generative AI tools to **support** code writing, testing, and documentation, as long as the output is:

1. Understood by the developer;
2. Reviewed critically before insertion;
3. Compliant with the team's style.

DO's:

1. Use AI to suggest **simple** code or test templates.
2. Use it for improving readability or formatting code into the team's style.
3. Use it for catching and solve **simple** errors/malpractices

DON'Ts:

1. Don't copy-paste AI-generated code **without review** or adaptation.
2. Don't use AI for **critical logic** or configurations without validation.
3. Don't use AI to bypass code review without personal evaluation.

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]

Our coding style is based on the **Google's Java Style Guide** with the help of some tools for automatic formatting:

1. We use **camelCase** for variables and methods;
2. We use **PascalCase** for classes and interfaces;
3. All other formatting (indentation, braces, spacing, etc.) is automatically handled using **Prettier** in every commit, which runs in the CI pipeline.

Code reviewing

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

Feel free to add more section as needed

Code review must be done **every time** that is made a Pull Request (PR) to the **dev** branch

1. All code must be submitted **via PRs**;
2. At **least one peer** must approve the PR;
3. AI tools (like GithubCopilot) may assist in reviews, but human review is **mandatory**.

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?]

Backend (BE): We use **SonarQube** to perform automated static code analysis on all backend commits. SonarQube reports on issues such as bugs, vulnerabilities, code smells, duplication, and also aggregates code coverage data from our test suites although we have not yet defined explicit coverage thresholds (quality gates) for blocking merges.

Frontend (FE):

1. **Prettier** is integrated into the pipeline as an opinionated code formatter that automatically enforces consistent styling (indentation, braces, spacing) on every save. It provides a "format-on-save" safety net but offers limited configurability;

2. **ESLint** runs after Prettier to catch linting errors and enforce code-quality rules. By default it applies the community-recommended rule set, but it can be extended or tuned.

Dashboard & Quality Gates:
The SonarQube dashboard consolidates **BE** and **FE** analysis results for an overall view of code health.
In future iterations, we will define and enforce minimum coverage and issue thresholds as part of our GitHub pull-request quality gates.

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.]

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

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*]

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 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.]

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?

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

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.]