TQS: Quality Assurance manual

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

João Gaspar [114514], Shelton Agostinho [115697]
v2025-05-14

Contents

TQS: Quality Assurance manual **1**
1 Project management **1**
1.1 Assigned roles 1
1.2 Backlog grooming and progress monitoring 1

2 Code quality management **2**
2.1 Team policy for the use of generative AI 2
2.2 Guidelines for contributors 2
2.3 Code quality metrics and dashboards 2

3 Continuous delivery pipeline (CI/CD) **2**
3.1 Development workflow 2
3.2 CI/CD pipeline and tools 2
3.3 System observability 3
3.4 Artifacts repository [Optional] 3

4 Software testing **3**
4.1 Overall testing strategy 3
4.2 Functional testing and ATDD 3
4.3 Developer facing testes (unit, integration) 3
4.4 Exploratory testing 3
4.5 Non-function and architecture attributes testing 3

[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

- **Hierarchy logic**
  We structure all work around Epics, each representing a major feature or functional area (e.g., Station Discovery, Booking, Charging Session). Under each Epic, we create the User Stories defined in our product backlog. These describe the user-facing goals and acceptance criteria. Each User Story is then implemented through its subtasks that break down the user story into pieces. This allows us to track technical progress toward the completion of each functional requirement. These sub-tasks are used to manage the step-by-step work of individual developers and ensure visibility into the development process.
- **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**, **In Review** and **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 traceability.

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.]
Work flows directly from JIRA into our GitHub repository following a GitHub-Flow variant, but with branch creation automated by JIRA. For each Issue, JIRA provides a "Create branch" button that generates a properly named feature branch (e.g. `EDISON-123-search-stations`) off the shared `dev` branch. Developers then simply pull and checkout that branch locally:

1. Select a Story/Issue in JIRA in the **Board**;
2. **Click "Create branch"** in the JIRA UI selecting dev in the "Branch from" section
3. Select a Sub-task or Major Task related to the Story
4. Click "Create branch" now selecting the **Issue's** branch that was created in the "Branch from" section
5. Now just git pull and git checkout <sub-task-branch-name> and start coding
6. When a sub-task is done, make a PR to merge into the Task's branch in Github and merge the branch.
7. After all the sub-tasks of a task are done, make a PR to the dev, make sure that there are no merging errors by doing git merge dev and fixing any conflicts and **assign a peer for Review.**

Definition of done
[What is your team "Definition of done" for a user story?]
A User Story is only marked **Done** in JIRA when **all** the following conditions are satisfied:

- **Code implemented** meets every acceptance criteria.
- **Sub-tasks** are all resolved and closed.
- **Automated tests** (unit and integration) execute successfully in CI.
- **Static analysis** (Prettier, ESLint, SonarQube) reports no new blocking issues.
- **Pull Requests** for sub-tasks and story branches have all approving reviews (those who have reviews).
- **Documentation** is updated to reflect any new APIs or configuration.
- **Merged into `dev`** without conflicts, ensuring the integration branch remains stable.
- **Demo-ready**: the feature can be deployed to our staging environment for validation and sprint 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 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.]
Our team adopted a **mixed testing strategy**, combining different approaches and tools to ensure that functional correctness, integration stability, and business behavior are all well covered.

We follow a **Test-Driven Development (TDD)** methodology for new features: tests are written first, fail initially, and are then used to drive the implementation of the necessary classes and logic. Once the implementation is complete, the tests pass, providing immediate feedback on correctness.

In parallel, we also apply **Behavior-Driven Development (BDD)** practices using **Cucumber**, which allows us to express user-level behavior in business-readable scenarios. These scenarios help validate not only the technical logic but also ensure alignment with the functional requirements defined in the user stories.

Testing Tools and Frameworks

- Backend: We use JUnit, Testcontainers, and H2 in-memory database for isolated and repeatable tests. Flyway is used to apply migrations in test environments to mirror real database behavior.

- Frontend: Linting and formatting are enforced using ESLint and Prettier, and the testing workflow is being incrementally expanded.

- Xray (JIRA Plugin): Test cases and execution results are tracked and documented within JIRA using Xray, allowing us to map coverage back to requirements and user stories.

CI Integration

All tests are automatically executed as part of the **CI pipeline**:

- **Backend CI**: Integrated with **SonarCloud** to analyze code quality and coverage.
- **Frontend CI**: Validates formatting and code quality using **Prettier** and **ESLint**.

The team collectively ensures that all scenarios (normal flows, edge cases, and exceptions) are covered with meaningful tests. Pull requests are only approved when test coverage and behavior are considered sufficient by peers.

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?

Our project adopts **Acceptance Test-Driven Development (ATDD)** to validate system behavior from the user's perspective. Functional tests are written as **closed-box tests**, focusing on verifying that the system meets its expected behavior based on user stories and acceptance criteria.

When are they written?

Functional tests are created during the Sprint, before the implementation of the related feature. Developers are responsible for writing them in parallel with the development of the user story.

Tools and Practices

- **Cucumber** is used to define acceptance tests in Gherkin syntax, directly reflecting user story criteria.
- Functional scenarios are also tracked in **Xray (JIRA)**, ensuring traceability between requirements and test coverage.

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.]

Unit Testing Policy

We adopt an **open-box** approach for unit testing. Developers are required to write unit tests for all new classes and methods with business logic. These tests are written during or immediately after implementation and follow the principles of **Test-Driven Development (TDD)**.

- **Technologies Used**: JUnit 5

Integration Testing Policy

We apply a **mixed open/closed box** strategy for integration testing, depending on context. Tests validate interactions between application components (e.g., REST controllers, services, database).

- **Technologies Used**: Spring Boot Test, Testcontainers, H2, Flyway

4.4 Exploratory testing

[strategy for non-scripted tests, if any]
Although our primary testing strategy is based on scripted and automated tests, we also incorporate **exploratory testing** as a complementary practice, particularly during the final stages of each sprint or before major merges into main.

Strategy

Exploratory testing is conducted **informally and without predefined test cases**, focusing on identifying unexpected behavior, usability issues, or edge cases that may not be captured by automated tests.

- **Who performs it**: Any team member can carry out exploratory sessions.
- **When it's done**: Typically at the end of a sprint, during review or demo preparation, or after merging a larger feature into dev.

Reporting

Findings from exploratory testing are documented directly in JIRA as:

- **Bug reports** (with steps to reproduce if applicable);
- **Improvement** suggestions.

4.5 Non-function and architecture attributes testing

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