TQS: Quality Assurance manual

Guilherme Rosa [113968], Maria Linhares [113534], Martim Santos[114614], Rui Machado[113765]
v2025-05-18

Contents

1 Project management

1.1 Assigned roles

Each team member fulfills a dual role: while focusing on their area of specialization, they also actively contribute to the development of the solution. Below is an overview of our team structure, roles, and specific responsibilities.

Team Member	Assigned Role
Guilherme Rosa	DevOps Master
Maria Linhares	Product Owner
Martim Santos	Team Coordinator
Rui Machado	QA Engineer

Roles' Description

- Team Coordinator (Martim Santos)

  Oversees fair task distribution and ensures alignment with the project plan;
  Encourages team collaboration and proactively resolves any emerging issues:
  Ensures timely delivery of project milestones and outcomes.

- Product Owner (Maria Linhares)

  Acts as the primary liaison for stakeholders, representing their interests;
  Provides in-depth knowledge of the product and domain context;
  Clarifies feature expectations and requirements to the development team;
  Participates in reviewing and accepting completed solution increments.

- QA Engineer (Rui Machado)

  Leads quality assurance initiatives and implements quality measurement practices;
  Ensures adherence to agreed QA standards and processes;
  Works closely with other team members to maintain quality across deployments.

- DevOps Master (Guilherme Rosa)

  Manages infrastructure and configuration for both development and production environments;
  Ensures stability and efficiency of the development framework;
  Oversees setup and maintenance of deployment environments, version control systems, cloud resources, databases, and related tools.

- Developer Role (All Team Members)

Contribute directly to development tasks, including implementation of features, bug fixes, and code reviews.

Document contributions through commits and pull requests in the team repository.

1.2 Backlog grooming and progress monitoring

In JIRA, we organize work using a combination of Epics, Stories, Tasks, and Sub-tasks. We follow the Scrum framework to manage our workflow, operating in weekly sprints. Our JIRA instance is integrated with GitHub, and we use issue automation to streamline updates between commits and tasks.

Progress is tracked using story points to estimate effort and complexity. Sprint progress is monitored through standard Agile reports such as burndown charts and velocity charts. Our project is structured into five sprints: one for inception, one for elaboration, two for construction, and one for transition.

For test management, we use Xray, which is fully integrated with JIRA. This enables us to link test cases to user stories, monitor test coverage, and ensure end-to-end traceability from requirements to validation.

2 Code quality management

2.1 Team policy for the use of generative AI

We treat AI assistants as productivity boosters not decision-makers.

1. Allowed: Generate boilerplate and execute well defined tasks with a small scope;
2. Forbidden: Committing AI code without review, exposing sensitive data, or letting AI decide architecture;
3. Musts: Every AI-generated change is peer-reviewed.

2.2 Guidelines for contributors

Coding style

For our project's coding style, we followed the AOS Java guidelines. Below is a concise overview of the conventions we applied:

Indentation and spacing:

<u>Indentation:</u>

Use four spaces for each level of indentation. Tabs are not allowed.

Braces:
Place opening braces on the same line as the corresponding control statement or method declaration. Closing braces should appear on a new line.

Spacing:
Include a single space around operators (e.g., =, +, ==). Do not include spaces inside parentheses (e.g., `if(condition)` instead of `if( condition )`).

Documentation:
Provide comprehensive JavaDoc comments for all classes and methods to ensure clarity and maintainability.

Code Structure:

Imports:
Grouped by standard library, third-party, and local application imports.

Methods and Classes:
Focus methods on a single responsibility and organize them logically.

Error Handling:
Use specific exceptions with meaningful error messages.

Testing:

Test Naming:
Use descriptive test method names that clearly indicate the functionality being tested. (e.g., testUserLoginWithCorrectCredentials).

Test Coverage:
Minimum of 80% coverage, with unit and integration tests

Naming Conventions:

- Variables and Methods: camelCase (e.g. updateUserProfile, isAvailable()).
- Classes and Interfaces: PascalCase (e.g. UserService, InventoryManager)
- Constants: All uppercase with underscores (e.g. MAX_RETRY_COUNT, DEFAULT_TIMEOUT)

Code reviewing

Code reviews are a **mandatory** part of our development workflow and are conducted for every pull request (PR) before merging into the main/dev branch. A designated reviewer is assigned to each PR to ensure the code meets our quality standards.

2.3 Code quality metrics and dashboards

To maintain high code quality and consistency, we integrated SonarQube Cloud into our development workflow for static code analysis. This tool continuously monitors and evaluates our code against a comprehensive set of predefined metrics.

Static Code Analysis with SonarQube

We use SonarQube’s default quality gate “Sonar Way” to assess code quality on all pull requests and dev and main branches. This quality gate enforces best practices under the Clean as You Code principle, helping to prevent technical debt and ensure code maintainability from the start.

Quality Gate: Sonar Way

The “Sonar Way” quality gate includes the following conditions applied to all new code:

- No new bugs introduced – Ensures a Reliability rating of A
- No new vulnerabilities – Ensures a Security rating of A
- Limited technical debt – Ensures a Maintainability rating of A
- All new security hotspots are reviewed – Must be 100% reviewed
- Adequate test coverage – Test coverage ≥ 80%
- Low code duplication – Duplicated lines ≤ 3%

Rationale for using "Sonar Way"

- It provides a solid balance between strict quality controls and developer agility.
- It enforces essential quality gates without requiring heavy configuration.
- It is well-integrated with CI/CD pipelines, enabling real-time feedback on code health.
- It helps prevent new issues rather than addressing them retroactively.

Dashboards and Reporting

SonarQube’s dashboard gives us a clear overview of the codebase’s health, displaying metrics such as: Code smells; Bugs and vulnerabilities; Test coverage; Code duplication; Technical debt.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow
The team follows the GitHub Flow with branches mapped to Jira issues.

1. Backlog analysis → Sprint planning → Story in "To Do";
2. Developer creates branch from issue from "`dev`";
3. Commits reference include unit tests;
4. Pull request to "`dev`" triggers CI - at least 1 reviews and all tests passed;

Definition of done (A story is done when)

1. All acceptance criteria are satisfied;
2. Unit and integration tests are passed;
3. CI pipeline passed as well as the quality gate;
4. Code reviewed and approved by at least 1 peer;

3.2 CI/CD pipeline and tools

Continuous Integration – GitHub Actions builds the project, executes all the tests, and performs static.

SonarQube (build.yml):

- Triggers**:** This pipeline is triggered on any push to the "`main`" or "`dev`" branches, and on pull request events such as opened, synchronized, or reopened.

- Environment Setup: An Ubuntu virtual machine ("`ubuntu-latest`") is used as the runner. The code is cloned with "`actions/checkout@v4`", using "`fetch-depth: 0`" to ensure a full clone for more accurate SonarQube analysis.

- JDK Setup: JDK 21 is installed via "`actions/setup-java@v4`", using the Zulu distribution.

- Cache: Caching is used to optimize performance:

  - SonarQube Cache: Stored in "`~/.sonar/cache`" with a static key.

  - Maven Cache: Stored in "`~/.m2`", using a key generated from the hash of all "`pom.xml`" files to detect changes in dependencies.

- Build and Analyze: The Maven build is executed with tests and SonarQube analysis using the Sonar Maven plugin. The "`SONAR_TOKEN`" secret is passed for authentication, and Jacoco code coverage reports are specified via the "`-Dsonar.coverage.jacoco.xmlReportPaths`" parameter.

3.3 System observability

We used SonarCloud for system observability to maintain high code quality through detailed static

analysis. It allowed us to quickly identify and fix bugs, security vulnerabilities, and maintenance issues, seamlessly integrating into our CI/CD pipeline for automated checks.
https://sonarcloud.io/summary/new_code?id=elytra-tqs_stations-management&branch=main

Data Collected for Assessment:
Bugs; Code Smells; Security Vulnerabilities; Security Hotspots; Coverage; Duplications; Code Complexity; Technical Debt.

3.4 Artifacts repository [Optional]

The project does not currently use a dedicated artifacts repository such as GitHub Packages, Nexus, or JFrog Artifactory. Maven dependencies and build outputs are managed locally within the CI environment. Caching mechanisms are in place to improve build performance:

Maven artifacts are cached in the GitHub Actions runner using the “`~/.m2`” directory to avoid re-downloading dependencies.

SonarQube packages are cached in “`~/.sonar/cache`” to speed up static code analysis.

All build and test processes are executed within GitHub Actions, and no additional artifact storage or distribution is configured at this stage.

4 Software testing

4.1 Overall testing strategy

Our overall strategy focused on achieving comprehensive test coverage by utilizing a variety of testing methodologies and tools. The following practices were implemented:

- Test-Driven Development (TDD): The core development methodology, writing tests before implementing functionality. This approach led to a more reliable and well-tested codebase.

- Behavior-Driven Development (BDD)**:** Used Cucumber to define test scenarios in an easily understandable format.

- Tool Integration: REST-Assured was chosen for API testing, allowing us to cover different layers of the application effectively.

- Continuous Integration (CI): Our testing strategy was fully integrated into the CI pipeline, ensuring that tests were automatically executed with each project build to maintain code quality and prevent regressions, using SonarCloud and Xray.

4.2 Functional testing and ATDD

The project adopts functional testing from a user's perspective (black-box testing) to ensure that the system behaves according to the defined functional requirements. These tests simulate real-world scenarios and validate the application's behavior as a whole.

The use of ATDD (Acceptance Test-Driven Development) is encouraged, where acceptance criteria are defined before development starts. These criteria guide the creation of automated acceptance tests that serve as validation for feature completeness.

When developers are expected to write these tests:

- When implementing new features with defined acceptance criteria.
- When there is a user story or functional requirement with expected system behavior.
- When fixing bugs that affect functional behavior.
- Before delivering a feature, to ensure it meets the expected user behavior and business requirements.

4.3 Developer facing tests (unit, integration)

Unit tests are written from a developer's perspective (open-box testing) to verify the correctness of individual components or methods in isolation. The goal is to ensure that each class or method works as expected under various conditions.

When developers must write unit tests:

When developing new classes, methods, or services.
When fixing bugs or refactoring existing code.
During test-driven development (TDD), where unit tests are written before implementation.
Before merging code into shared branches to maintain code quality and prevent regressions.

Tools and Frameworks:

Unit tests are written using JUnit 5.
Mockito is used for mocking dependencies and isolating units during tests.

Integration tests are used to validate interactions between components such as services, repositories, and controllers. These tests can be both open-box and closed-box, depending on the scope, and aim to ensure that the system behaves correctly as a whole when its components are combined.

When developers must write integration tests:

When implementing features that involve multiple layers of the application (e.g., controller → service → repository). When integrating with external APIs or services.
After refactoring or restructuring components that interact with one another.

Tools and Frameworks:

Integration tests are implemented using Spring Boot Test.
Common annotations include “`@SpringBootTest`” for full context loading, “`@DataJpaTest`” for persistence layer testing, and “`@WebMvcTest`” for controller testing.

Testcontainers or H2 in-memory databases may be used to simulate external dependencies like databases in isolated environments.

API testing ensures that REST endpoints behave as expected, return the correct status codes, and comply with business logic. MockMvc is used to simulate HTTP requests in Spring Boot and validate responses. These tests are often included in integration test suites and focus on request/response validation, input validation, and error handling.

4.4 Exploratory testing

Exploratory testing is performed regularly before major releases, especially after implementing complex features or refactoring. The project encourages this type of testing as a complementary strategy to automated and scripted tests. It is conducted manually by team members without predefined test cases, allowing them to freely explore the application to uncover unexpected behaviors, usability issues, or edge cases.

4.5 Non-function and architecture attributes testing

The project acknowledges the importance of validating non-functional requirements such as performance, scalability, and reliability. Performance tests are designed to assess system behavior under various loads and ensure that the application meets expected response times and throughput standards. These tests are conducted for critical endpoints and operations that handle high volumes of data or user requests and are typically executed before major releases or after significant backend optimizations to ensure that no performance regressions are introduced.