TQS: Quality Assurance manual

Gabriel Silva [113786], Henrique Teixeira [114588], João Roldão [113920], Sebastião Teixeira [114624]

v2025-06-09

1 Project management

1.1 Assigned roles

Team Coordinator	Sebastião Teixeira
Product Owner	João Roldão
QA Engineer	Gabriel Silva
DevOps Master	Henrique Teixeira
Developer	Gabriel, Henrique, Roldão e Sebastião

1.2 Backlog grooming and progress monitoring

- The team organizes work using JIRA, with Epics representing major features, and Stories/Tasks for individual work items.
- Backlog grooming occurs every Monday, led by the Product Owner, where new stories are refined and prioritized.
- Progress is tracked using Story Points and visualized in JIRA Sprint Boards and Burndown Charts, reviewed in weekly standups.
- We use the **Xray test management plugin** for JIRA to maintain traceability between requirements and test cases. Automated test results are uploaded to Xray via the CI pipeline, ensuring that requirements coverage can be tracked and reported directly within JIRA.

2 Code quality management

2.1 Team policy for the use of generative AI

Team policy:

- AI assistants (e.g., GitHub Copilot, ChatGPT) are allowed **only** to support development, never to generate production code or tests autonomously.
- Code produced with AI must be reviewed critically, refactored, and understood by the developer.

Do's:

- Use AI for generating boilerplate, exploring APIs, or refactoring suggestions.
- Validate and test any AI-generated snippet thoroughly.

Don'ts:

- Don't commit AI-generated code without review.
- Don't copy-paste from unknown sources without checking licenses.

2.2 Guidelines for contributors

Coding style:

- Follow language-specific conventions:
  - For Java: Google Java Style (enforced via IDE/formatter)
- Format code with automated tools like Prettier, ESLint, or IDE configurations.

Key practices:

- Keep methods short and focused.
- Name variables and methods clearly.
- Comment only when necessary, write self-explanatory code.

Code reviewing:

When to review:

- All code must go through **peer code reviews via pull requests** before being merged.

Review policy:

- **Main branch**: Requires **2 reviewers**.
- **Dev / Release branches**: Requires **1 reviewer**.

Pull request requirements:

- Clear and descriptive title.
- Link to JIRA issue.
- Include evidence of testing: screenshots, test results, or logs.

Checklist during review:

- Code correctness
- Readability and style adherence
- Performance impacts
- Security concerns
- Documentation updates (if applicable)

All team members are expected to:

- Provide **constructive feedback** on PRs.
- **Resolve merge conflicts** proactively.
- **Update relevant documentation** if changes affect behavior or APIs.

Tools:

- **SonarQube** for quality reporting and enforcing gates.
- **JaCoCo** for code coverage (reports in CI).
- **GitHub PR templates** for consistency.

2.3 Code quality metrics and dashboards

Quality Gates:

- Minimum 80% test coverage.
- No critical or blocker issues in SonarQube.
- Maintain code smells and duplications under defined thresholds.

Tools used:

- **JaCoCo**: For code coverage.
- **SonarQube**: For quality reporting and enforcing gates.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow:

- The team follows **Gitflow** for branch management:
- All new features, improvements, or bugfixes are developed in a separate branch, created from dev.
- Branches are named after the JIRA issue or feature (e.g., feature/STATION-123-new-endpoint).

To start work:

1. Select a JIRA story or task assigned during backlog grooming or sprint planning.
2. Create a feature branch from dev.
3. Implement changes, ensuring code follows the team's coding standards.
4. Write or update unit, integration, and/or acceptance tests as required.

When the feature is complete:

1. Push the branch to GitHub.
2. Open a Pull Request (PR) against the dev branch.
3. Link the PR to the relevant JIRA issue and include evidence of testing.
4. PRs must be peer-reviewed (1 reviewer for dev, 2 for main/release).
5. All checks in the CI pipeline (build, tests, static analysis) must pass.

- After review and approval, the PR is merged into dev.
- Release branches are created from dev and merged into main for production releases. Releases are tagged from main.

Definition of done for any story:

- Code committed and pushed.
- Code reviewed and approved.
- Tests implemented and passing.
- CI pipeline passed.
- Documentation updated if applicable.

3.2 CI/CD pipeline and tools

CI pipeline

- GitHub Actions is used for continuous integration.
- Static analysis with SonarQube is performed on every PR.
- The CI pipeline runs **unit, integration, and Cucumber/ATDD tests on every PR** using GitHub Actions.
- Test results are uploaded to **Xray on Jira Cloud** for requirement traceability.

- All test types are run in the main CI pipeline.

CD pipeline:
- **Containerization with Docker** is used for deployments.
- The CD pipeline is triggered on **main branch merges.**

3.3 System observability

Tools:
Prometheus and Grafana are used to monitor:
- user-service
- station-service
- frontend
- external station API

Alert triggers:
- System availability (service stops responding): monitored for all services
- CPU usage (warning at 60%, critical at 80%)
- Memory usage (warning at 70%, critical at 90%)

Other metrics (no direct alert):
- Global network utilization (req/s on proxy, both up and down)
- Per-service network usage (Bytes/s, up and down)

4 Software testing

4.1 Overall testing strategy

The project adopts a **comprehensive, layered testing strategy**, combining best practices and automation to ensure quality across the development lifecycle:

- **Test-Driven Development (TDD)**: Applied in Spring Boot projects to promote test-first implementation.

- **Acceptance Test-Driven Development (ATDD):** Used to validate functionality based on user stories and acceptance criteria, primarily using Cucumber.

- **Automated testing tools**:
  - **JUnit 5:** Core unit and integration tests
  - **Mockito:** Mocking dependencies
  - **Spring Boot Test:** Full application context
  - **TestContainers:** Isolated databases for integration tests
  - **Cucumber:** BDD/ATDD for features
  - **REST-Assured:** REST API testing

- **CI Integration:**

    - All tests (unit, integration, Cucumber/ATDD) are run in the main CI pipeline using GitHub Actions.
    - Test coverage (JaCoCo) is enforced, with a minimum pass rate as a merge requirement.
    - Results are pushed to Xray on Jira Cloud for traceability.

4.2 Functional testing and ATDD

Functional tests are written from a **black-box perspective**, focused on verifying system behavior against business requirements without reference to internal implementation. These are typically written:

- Based on the **acceptance criteria** provided in the user stories.
- Using **Cucumber syntax** for clarity and readability, enabling collaboration between developers, testers, and non-technical stakeholders.
- For features involving **user interaction, complex workflows, or external integrations**.

All functional tests are:

- Versioned alongside their related stories or features.
- Integrated into the CI pipeline.
- Implemented using Cucumber
- Maintained as part of the regression test suite to validate ongoing stability.

4.3 Developer facing tests (unit, integration)

Developer-facing tests are written from an **open-box perspective**, validating components and their interactions at a technical level. These tests are essential to ensure the reliability of the codebase and catch defects early.

Unit Testing Policy

Project policy:

- **Unit tests are mandatory** for all business-critical logic, utility functions, and non-trivial methods.
- Tests must be written by the **developer implementing the feature**.
- Unit tests are expected for:

    - Data processing and transformation
    - Business rules and validations
    - Exception handling and boundary conditions

Most relevant unit tests in this project:

- Service layer methods that implement decision logic
- Utility functions with calculation or formatting rules
- Repository methods with custom queries (mocked DB behavior)

Tools used:

- **JUnit 5**: Core unit testing framework
- **Mockito**: Mocking dependencies such as services or repositories
- **AssertJ**: Fluent assertions for improved test readability

These tests are enforced by the CI pipeline and contribute to code coverage metrics in SonarQube.

Integration Testing Policy

Project policy:

- Integration tests are required for validating the interaction between components, such as:
    - Controllers and service layers
    - Services and repositories
    - Internal and external APIs
- Developers must write integration tests when:
    - A feature crosses multiple architectural layers
    - New external services or dependencies are introduced
    - Database or transactional logic is involved

Execution:

- Integration tests are run as part of the main CI pipeline.

Tools used:

- **Spring Boot Test**: Full application context setup
- **MockMvc**: Testing REST controllers and request/response flows
- **TestContainers**: For spinning up isolated databases or message brokers

4.4 Exploratory testing

Exploratory testing is used as a **complement to scripted test cases**, focusing on discovering issues through unscripted, scenario-based investigation. It helps uncover edge cases, usability problems, and integration gaps that automated tests may miss.

Strategy:

- Conducted by developers and QA engineers during feature testing or sprint reviews.
- Applied especially to:
  - New or complex user flows
  - Features with multiple input combinations
  - Areas with recent changes or regressions
- Sessions are **time-boxed** and documented with observed behaviors and issues.

Approach:

- Use **pairwise testing** techniques for scenarios involving multiple input combinations.
- Apply **error guessing**, **boundary exploration**, and **scenario-based walkthroughs**.

Tools used:

- Manual tools (e.g., Postman, browser dev tools, curl) depending on the feature.

Results from exploratory sessions are logged in JIRA as observations, bugs, or improvement suggestions, and are reviewed with the team in retrospectives when relevant.

4.5 Non-function and architecture attributes testing

Non-functional testing focuses on validating how the system performs under expected and extreme conditions, rather than what the system does. This includes performance, scalability, reliability, and availability assessments.

Service Level Objectives (SLOs) defined for the project:

- **API response time**: 99% of requests should complete in **under 500ms**
- **System availability**: Minimum **99% uptime**

Performance Testing Strategy:

- Load and stress tests simulate real-world scenarios and volume.
- Tests are designed to measure:
  - Response time under load
  - Throughput and concurrency
  - Resource usage (CPU, memory, network)

Tools used:

- **K6**: For scripting and executing performance/load tests

Execution:

- Performance tests are not part of every CI run.
- They are executed in **scheduled pipelines**, before production releases, or during architectural reviews.
- Results are reviewed with DevOps and QA.