TQS: Quality Assurance manual

Pedro Fonseca[119264], Dinis Cunha[119316], Rafael Fernandes[118956], Francisco Silva[]
v2025-11-30

Contents

1 Project management

1.1 Assigned roles

All four members are responsible for the production of the code.

Dinis (Team Coordinator) - Ensure that there is a fair distribution of tasks
Francisco (Product owner) - Represents the interests of the stakeholders
Pedro (QA Engineer) - Responsible to promote the quality assurance practices
Rafael (DevOps master) - Responsible for the development and production

1.2 Backlog grooming and progress monitoring

The team organizes its work using Scrum-based agile practices, with one-week fixed sprints. JIRA is used as the main tool to manage all development activities and to keep visibility over the current state of the project. User stories represent the core unit of work and are defined and discussed collectively by the whole team, which helps ensure that everyone has a clear understanding of the requirements from the beginning. In general, user stories are handled as complete work items.

User stories are estimated using story points, which reflect both the complexity and relevance of the work to be done. These estimations are performed in a collaborative manner, based on team discussion. During the sprint, progress is followed mainly by checking the JIRA board, where the state of each user story is visible at all times. In addition to the board, team members maintain regular communication to understand the current status of development and address any blocking issues.

Development lifecycle:

- To Do: work that has not yet started
- In Progress: work currently being implemented
- In Review: functionality under testing and validation
- Done: work that has passed testing and review

To promote code quality, pull requests are always reviewed and approved by a different team member than the one who implemented the feature. Each user story is associated with its own dedicated Git branch, which allows the team to clearly relate code changes to specific requirements. Updates to the JIRA board are performed manually, ensuring that the status of each user story accurately reflects its real development stage.

Testing activities are managed using X-Ray integrated with JIRA. The team uses both manual and automated test cases, which are explicitly linked to user stories to ensure traceability between requirements and validation. Test progress and results are tracked through Test Executions, allowing the team to verify which requirements have already been tested. Test cases are created both before implementation, to guide development, and afterwards, to confirm correct behavior and full coverage of the acceptance criteria. We only accept pull requests if they have more than 80% test coverage.

2 Code quality management

2.1 Team policy for the use of generative AI

AI tools are allowed for both production code and test code, provided that their usage follows strict quality and accountability rules. All code introduced into the project must be fully understood, reviewed, and validated by the developer who submits it. AI-generated suggestions are treated as proposals, not final solutions. The team remains fully responsible for correctness, security, maintainability, and compliance with project requirements.

Do's:

- Use AI assistants to clarify concepts, explore alternative implementations, or generate initial drafts.
- Use AI to help write boilerplate code, test templates, or documentation.
- Always review, refactor, and adapt AI-generated code to fit the project's architecture and standards.

Don'ts:

- Do not submit AI-generated code without understanding how it works.
- Do not blindly copy and paste code from AI tools into the codebase.
- Do not rely on AI to make design decisions or architectural choices.
- Do not bypass testing, code review, or quality checks because the code was "generated by AI"

2.2 Guidelines for contributors

Coding style

The project follows a consistent and readable coding style to ensure that the codebase is easy to understand and maintain by all team members. Contributors are expected to prioritize clarity and maintainability, avoiding unnecessarily complex solutions. The backend is implemented using Spring Boot (Java) and follows a layered architecture with a clear separation between controllers, services, repositories, DTOs, and entities, which helps enforce separation of concerns and improves testability. Naming conventions are applied consistently across the codebase.

The team follows the official best practices and community standards of the technologies used, namely Java and TypeScript. Code is written with clear structure, consistent formatting, and a focus on small, well-defined methods. Duplicated logic and unnecessary complexity are avoided whenever possible. Automated tools such as SonarQube are used to continuously monitor code quality, and more detailed coding rules are documented in the project repository.

Code reviewing

All code changes must go through a mandatory code review process before being merged into the development branch. A review is performed after a pull request is opened and only after the

CI pipeline has passed successfully. A user story can only be marked as Done once the corresponding pull request has been approved, and the author of the pull request is not allowed to approve their own changes.

During the review, reviewers verify the correctness of the implementation, compliance with acceptance criteria, adherence to the defined coding style, and the presence of relevant tests. Test execution and a minimum test coverage of 80% are required. Reviewers may also perform local testing to validate behavior. The review process is collaborative and focused on improving code quality and maintainability.

2.3 Code quality metrics and dashboards

The project uses static code analysis to continuously monitor and improve code quality, supported by SonarQube as the main analysis tool. SonarQube is integrated into the CI pipeline and provides dashboards that give visibility over key quality metrics such as bugs, code smells, security issues, duplicated code, and test coverage. These dashboards allow the team to track code quality evolution and quickly identify areas that require attention.

Basic quality gates are enforced to prevent low-quality code from being merged. In particular, pull requests must meet the defined test coverage threshold and must not introduce critical issues. These gates act as a safeguard to ensure a consistent level of code quality throughout the project.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

The team follows a structured development workflow aligned with Scrum practices and managed through JIRA. Developers select user stories from the sprint backlog once they are refined and have clearly defined acceptance criteria. For each user story, a dedicated Git branch is created to ensure traceability between requirements and code changes. The project follows a GitHub Flow–like approach, where short-lived feature branches are merged into the development branch through pull requests. After completing the implementation, a pull request is opened and must successfully pass the CI pipeline before being reviewed. Code reviews are mandatory and are conducted by a different team member than the author, with reviewers focusing on correctness, adherence to coding standards, test execution, and compliance with acceptance criteria. Local testing may also be performed during the review process.

Definition of done

A user story is considered Done only when the implemented functionality fully satisfies the defined acceptance criteria, the corresponding pull request has been reviewed and approved by another team member, all automated tests pass successfully, and the minimum required test coverage is met.

3.2 CI/CD pipeline and tools

The project implements CI/CD practices using GitHub Actions to automatically validate, test, and deploy code changes. Continuous integration runs on GitHub-hosted runners and is triggered on pull requests and pushes to the development branch, with the option for manual execution. The CI pipeline performs backend and frontend builds, executes automated tests, generates test coverage reports, and runs static code analysis using SonarCloud. The pipeline fails if the build fails, tests do not pass, coverage requirements are not met, or critical quality issues are detected.

Continuous delivery is fully automated and based on containerized deployment. Both backend and frontend are packaged using Docker and deployed with Docker Compose on a self-hosted runner. The project defines three environments: development (feature branches), staging (development branch), and production (main branch). Deployments to staging and production are triggered automatically, with production deployment occurring upon merge into the main branch. Secrets are managed securely, and environment variables are configured on the server using .env files.

3.3 System observability

The project follows a lightweight observability approach focused on availability and runtime diagnostics. Observability is mainly achieved through application- and container-level logging, using Spring Boot's default logging configuration. Logs are written to standard output and can be inspected through Docker when the application is deployed with Docker Compose, allowing developers to analyze errors, warnings, and runtime behavior.

Operational monitoring is partially supported through container health checks. The PostgreSQL service includes a health check that ensures proper startup ordering, while the backend application itself does not currently expose dedicated health or metrics endpoints. No external log aggregation, metrics collection, or alerting tools are configured in the repository.

Performance and resilience are validated through load testing with k6, which is used to simulate concurrent users and assess response times, error rates, and system behavior under load. While k6 does not provide continuous observability, it complements runtime diagnostics by identifying performance bottlenecks and scalability limits during testing phases.

System issues are primarily detected through logs and deployment feedback, including failed deployments, container startup failures, restarts, and application errors. This setup enables basic operational visibility and supports post-deployment validation, while leaving room for future improvements if the project scope is extended.

4 Continuous testing

4.1 Overall testing strategy

The project follows a layered and continuous testing strategy to ensure software quality, reliability, and regression prevention. Automated tests are developed before, during, and after

implementation, and all critical functionalities are required to have automated test coverage before being accepted.

Testing is performed at multiple levels: unit tests (JUnit 5 and Mockito) validate business logic in isolation; integration tests (Spring Boot with H2) verify interactions between application layers; end-to-end tests (Playwright) cover critical user flows; and load tests (k6) assess performance and validate expected behavior under load. Acceptance tests are automated using Cucumber and executed via FlowRun, enabling structured execution of Gherkin-based scenarios and ensuring that functional requirements are validated against real application behavior. Cucumber is used to automate acceptance criteria through Gherkin scenarios, improving traceability between requirements and tests. While strict TDD was not systematically applied, a test-first or test-alongside approach was used whenever feasible.

Testing is fully enforced through the CI/CD pipeline, which runs on every pull request to any branch. Merges are automatically blocked if tests fail, code quality checks do not pass, or if the minimum coverage threshold of 80% is not met. As part of the CI process, a Lighthouse snapshot is executed to capture performance, accessibility, and best-practice metrics, allowing regressions in key web quality indicators to be detected early. This ensures that test results and quality metrics directly influence merge decisions and prevent regressions from being introduced into shared branches.

4.2 Acceptance testing and ATDD

The project adopts a closed-box, user-facing acceptance testing approach, focusing on validating system behavior from the perspective of end users rather than internal implementation details. Acceptance tests are derived directly from user story acceptance criteria and describe observable system behavior and expected outcomes.

Acceptance testing is implemented using Cucumber with Gherkin scenarios, complemented by end-to-end tests (Playwright) for critical flows. These tests validate that implemented features meet business expectations and functional requirements across system boundaries.

Developers are required to create or update acceptance tests whenever a user story introduces or modifies critical functionality. Acceptance tests are considered part of the Definition of Done. This ensures continuous alignment between requirements, implementation, and delivered behavior.

4.3 Developer facing tests (unit, integration)

The project adopts a developer-facing testing approach focused on validating internal logic, component interactions, and API behavior from an open-box perspective. These tests are an integral part of development and are required whenever new functionality is introduced or existing logic is modified

.

Unit tests are mandatory for all business-critical logic, validation rules, and authorization checks. Developers are required to write or update unit tests whenever service-level behavior changes. Unit tests are implemented using JUnit 5 and Mockito, with dependencies mocked to ensure isolation, determinism, and fast execution. The most relevant unit tests in the project focus on service logic, input validation, error handling, and permission enforcement.

Integration tests are required when functionality involves interaction between multiple layers, such as services, repositories, and persistence. These tests validate component integration using Spring Boot with an H2 in-memory database, providing a controlled environment suitable for CI execution. Integration tests ensure correct data flow, configuration, and transactional behavior.

API testing is performed as part of integration and end-to-end testing, validating REST endpoints, request/response contracts, and authorization behavior. API tests are executed automatically in the CI pipeline and contribute to overall quality gates, ensuring that internal changes do not break external interfaces.

4.4 Exploratory testing

Exploratory testing was informal and ad-hoc, performed by developers when running the application locally. It focused on manually validating basic functional flows and interactions. This activity complemented automated testing but was not scripted or formally documented.

4.5 Non-function and architecture attributes testing

Non-functional testing in the project focuses primarily on performance and frontend quality attributes. Performance and load testing are performed using k6, while Lighthouse is used to evaluate frontend performance, accessibility, and best practices. These tests are executed before releases, serving as a validation step to assess system behavior under expected conditions.

Performance testing evaluates response time, throughput, and behavior under load, aiming to identify potential bottlenecks or degradations. Frontend testing with Lighthouse complements this by providing insights into performance metrics, accessibility compliance, and general quality best practices. The project does not define formal SLO thresholds; instead, results are assessed implicitly, focusing on detecting regressions or abnormal behavior.

From an architectural perspective, system attributes are validated through a combination of code reviews, integration tests, and CI quality gates. This ensures that architectural decisions, component interactions, and quality attributes are continuously assessed as part of the development and integration process.