deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Pedro Rafael Lemos Rodrigues [102432]
v2026/01/30

Contents

1 Project management

1.1 Assigned roles

This project was developed individually. Therefore, all roles typically associated with a software development team (project management, backend development, frontend development, quality assurance, and DevOps/CI management) were assumed by the same team member.

This ensured full ownership of the codebase and quality processes, while also requiring disciplined self-review and adherence to defined quality practices.

1.2 Backlog grooming and progress monitoring

The project backlog was managed using **Jira**, following an agile, user-story–based approach.

Work was organized using:

- **User stories** to describe functional requirements from the user perspective
- **Story points** to estimate effort and complexity
- **Epics** to group related user stories by major functional topics

Backlog grooming was performed regularly by reviewing and refining user stories, clarifying acceptance criteria, and adjusting priorities.

Progress was monitored using:

- Jira Scrum/Kanban boards
- Story status tracking (To Do, In Progress, Done)
- Story points to assess workload and completion progress

Although no dedicated test management plugin (e.g., Xray) was used, traceability between requirements and tests was maintained by linking test cases and commits to the corresponding Jira user stories.

2 Code quality management

2.1 Team policy for the use of generative AI

Generative AI tools, including ChatGPT, were used as **assistive tools** to support development activities.

The approved use cases for AI included:

- Clarifying technical doubts and framework usage
- Supporting frontend development tasks
- Assisting in the creation of a limited number of unit tests to improve coverage
- Generating code suggestions for review and adaptation

The following policies were enforced:

Do:

- Use AI to clarify concepts and accelerate learning
- Review and fully understand all AI-generated code
- Adapt AI-generated code to project-specific requirements
- Validate all AI-generated code through tests and code review

Don't:

- Copy-paste AI-generated code without understanding it

- Use AI to bypass quality standards or testing requirements
- Rely on AI-generated code without manual verification

All AI-generated contributions were treated as draft suggestions and subject to the same quality gates and reviews as human-written code.

2.2 Guidelines for contributors

Coding style

The project uses:

- **Java with Spring Boot** for the backend
- **React** for the frontend

Coding style guidelines include:

- Consistent formatting and naming conventions
- Clear package and component structure
- Meaningful class, method, and variable names
- Use of annotations and configuration following Spring Boot best practices

For Java code, widely adopted Java conventions were followed. For frontend code, standard React and JavaScript conventions were applied, with consistent formatting.

Code reviewing

Code review practices include:

- All PRs required approval before merging
- Merging was blocked until a review was completed
- Reviews were performed to check:
    - Code correctness
    - Adherence to coding standards
    - Test coverage
    - Potential bugs and code smells

Although the project was developed individually, peer review was simulated using a secondary account to ensure that the review workflow and quality gates were enforced and exercised.

2.3 Code quality metrics and dashboards

Static code analysis and coverage monitoring were implemented using:

- **SonarCloud** for static analysis and quality gates
- **JaCoCo** for code coverage measurement

The following quality metrics were monitored:

- Code coverage
- Bugs and vulnerabilities
- Code smells
- Maintainability issues

Quality gates were defined as follows:

- **Minimum coverage on new code: 80%**
- No critical bugs or vulnerabilities allowed
- No blocking quality gate failures permitted for merging

The rationale for these gates was to ensure that new functionality is adequately tested and that technical debt is controlled throughout development.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

The project source code is hosted on **GitHub**.

The development workflow follows a feature-branch model:

- A **main** branch is used as the stable integration branch
- **Feature branches** are created for each user story or task
- Development is performed on feature branches
- Changes are merged into main via Pull Requests

Pull Requests are linked to user stories and reviewed before merging, ensuring traceability between requirements and code changes.

Definition of done

A user story is considered **Done** when:

- The required functionality is fully implemented
- Unit and integration tests are written and passing
- The project builds successfully
- Code is reviewed and approved

- SonarCloud quality gate passes
- No critical bugs or code smells are introduced

3.2 CI/CD pipeline and tools

Continuous Integration is implemented using **GitHub Actions**.

The CI pipeline includes:

- Automatic build on Pull Requests and main branch
- Execution of:
  - `mvn test`
  - `mvn verify`
- Execution of unit and integration tests
- Code coverage analysis with JaCoCo
- Static analysis and quality gates via SonarCloud

Docker was also used locally to support containerized services and infrastructure components, although full continuous deployment to production was not implemented.

3.3 System observability

No dedicated system observability or monitoring tools (e.g., metrics dashboards, alerts, or distributed tracing) were implemented.

Basic observability was ensured through:

- Application logs
- Docker container logs
- Spring Boot default logging mechanisms

These were used for manual diagnosis and troubleshooting during development and testing.

3.4 Artifacts repository [Optional]

The source code and project artifacts are stored in a **GitHub repository**.

Maven dependencies are managed using the standard **local Maven repository**. No dedicated remote artifact repository (such as Nexus or GitHub Packages) was configured.

4 Continuous testing

4.1 Overall testing strategy

The project follows a test-after-development approach, aligned with CI/CD practices.

The main testing tools used were:

- **JUnit** for unit and integration testing
- **Mockito** for mocking dependencies
- **JaCoCo** for coverage reporting

Testing is integrated into the CI pipeline to ensure that all code changes are automatically validated.

While Test-Driven Development (TDD) and Behavior-Driven Development (BDD) were considered, the project primarily focused on traditional unit and integration testing due to time and complexity constraints.

4.2 Acceptance testing and ATDD

End-to-end (E2E) and acceptance testing using **Selenium** was attempted in order to validate user-facing workflows.

However, due to technical complexity, integration issues, and time constraints, Selenium-based acceptance tests were not fully completed and were ultimately removed from the final testing strategy.

Acceptance testing was therefore partially validated through manual testing and verification against user stories.

4.3 Developer facing tests (unit, integration)

Developer-facing tests include:

- **Unit tests** for service and business logic layers
- **Integration tests** using Spring Boot test support

Mockito was used to isolate components and mock external dependencies.

The project achieved approximately **93% code coverage**, exceeding the defined quality gate thresholds.

API documentation and validation were supported using **Swagger**, which also facilitated manual API verification.

4.4 Exploratory testing

Exploratory testing was conducted manually by the developer.

This included:

- Manual UI testing of main user workflows
- Ad-hoc validation of edge cases
- Manual verification of error handling and user interactions

This exploratory testing complemented the automated test suite and helped identify usability and integration issues.

4.5 Non-function and architecture attributes testing

No formal non-functional testing (e.g., performance, load, or stress testing) was implemented.

Due to project scope and time constraints, the focus was placed on:

- Functional correctness
- Code quality
- Maintainability

Non-functional attributes such as performance and scalability were considered out of scope for this iteration.