TQS: Quality Assurance manual

Margarida Ribeiro 119876, Gonçalo Almeida 119792, João Barreira 120054, Rodrigo Santos xxxxxx
v2025-11-29

Contents

1 Project management

1.1 Assigned roles

For this project, all four listed team members are developers who also carry one additional, assigned project management role.

João Barreira - Team Coordinator

- Ensures fair distribution of tasks among team members.
- Ensures that members work according to the project plan.

Rodrigo Santos - Product Owner

- Represents the interests of the stakeholders.

Margarida Ribeiro - QA Engineer

- Promotes quality-assurance practices within the project.
- Implements and applies instruments to measure the quality of deployments.

Gonçalo Almeida is our DevOps master

- Responsible for development and production infrastructure
- Responsible for required infrastructure configurations

1.2 Backlog grooming and progress monitoring

Our team adopts **agile project management** practices centered around **user stories** as the fundamental unit of work. All development activities are managed through JIRA, which serves as the central repository for requirements, tasks, and progress tracking.

User Stories are formulated following the standard template: "As an Admin, I want to suspend or reactivate user accounts when they violate platform rules so that I can maintain platform safety.". This structure ensures that each requirement clearly identifies the target user, the desired functionality, and the business value it delivers. Each user story is accompanied by detailed acceptance criteria that define the conditions of satisfaction for the feature. These acceptance criteria serve as the foundation for test case development and validation.

We, as a team, also employ story points as the estimation method for user stories. The estimation sessions involve the entire development team to leverage collective knowledge and promote shared understanding of requirements.

To facilitate high-level planning, related user stories are grouped into epics. The project currently maintains three epics that represent major functional areas for each persona. This hierarchical organization enables us to track progress toward broader project goals while maintaining focus on granular, deliverable user stories within each sprint.

Backlog grooming sessions are conducted one to two times per week, depending on the team's needs and the evolving requirements. During these sessions, the team reviews upcoming items in the backlog, estimates story points and ensures that the acceptance criteria are well-defined and testable. This way, we can always have a pipeline of ready user stories for subsequent sprints.

JIRA workflow

We follow a standardized workflow within JIRA that reflects the lifecycle of each user story. The workflow consists of four distinct states:

- **To Do**: User stories that have been refined and are ready for development but have not yet been started
- **In Progress**: User stories currently being implemented by a developer
- **In review**: User stories for which development is complete and testing is underway
- **Done**: User stories that have satisfied all acceptance criteria and have been integrated into the main codebase

Jira is integrated with the project's GitHub repository, creating seamless traceability between user stories and code changes. This integration enables automatic synchronization of development activities with project tracking artifacts. When the developers create branches, commit code, or submit pull requests that reference JIRA ticket identifiers, the corresponding user stories are automatically updated into development progress. The integration ensures that all code changes are traceable to specific user stories, supporting both progress monitoring and quality assurance activities.

We operate on a one week sprint cycle, which provides rapid feedback loops from our client and frequent opportunities for course correction. Progress is monitored continuously throughout the week through regular team communication and JIRA board updates.
The team has recognized the value of burndown charts as a visual tool for tracking sprint progress and intends to incorporate them into routing practice.

X-Ray in JIRA

Each user story in JIRA is linked to corresponding test cases in X-Ray. This linkage creates a bidirectional relationship that allows the team to verify that all requirements have associated tests and that all tests trace back to specific requirements. Tests are created before development begins, in alignment with **test-driven development principles**, ensuring that acceptance criteria are translated into executable validation early in the development process.
What should be generated when using X-Ray?, What are the expected results for this project?

2 Code quality management

2.1 Team policy for the use of generative AI

The team recognizes generative AI assistants as valuable tools that can enhance productivity, support learning and improve code quality when used responsibly.
The fundamental principle governing our group is **understanding must always precede acceptance**. We, as developers, remain fully responsible for all code committed, regardless of its origin.
Developers may use AI assistants to generate production code, subject to the following requirements and restrictions:

- **All AI generated code must be thoroughly reviewed and understood by the developer before being committed.** Developers must be able to explain the purpose and logic behind every line of code they introduce into the codebase, whether written manually or generated by AI. Blindly accepting AI suggestions without comprehension is strictly prohibited.
- **Service layer code should be avoided when using AI generation**. The service layer contains core business logic and represents the heart of the application. While AI may be consulted for debugging or exploring approaches, developers should write service layer code manually to ensure deep understanding of business requirements.
- **Frontend code may use AI more freely.** Since frontend development is not the primary focus of this project, AI assistants may be leveraged more extensively for user interface implementation. However, the requirement for review and understanding still applies.
- **All code, including AI generated code, must pass SonarQube quality gates.** Regardless of origin, all committed code is subject to the project's static analysis requirements.

Testing is a core learning objective of this course. As such, we adopt a more conservative approach to AI usage in test development to ensure that all team members gain hands-on experience with testing principles and practices.

- **AI should be used sparingly for test code generation.** While AI can be a useful tool for achieving continuous delivery and maintaining comprehensive test coverage, developers should prioritize manual test development to deepen their understanding on this subject.
- **AI may be used for simple tests, but complex tests must be written manually.** Straightforward unit tests with obvious assertions may be generated with AI assistance. However, tests that validate complex business logic, interaction patterns, or integration scenarios should be written manually.

Prohibited Practices:

- **Copying AI generated code without understanding it.** Developers who cannot explain the code they commit violate the learning objectives of this project.
- **Blindly accepting AI suggestions.** All AI output must be critically evaluated. AI models can produce plausible but incorrect code.

Summary of Guidelines

Practice	Guideline
Production Code Generation	**DO** use AI with thorough review and understanding
Service Layer Code	**DON'T** use AI for service layer implementation
Frontend Code	**DO** use AI freely, with review
Code Autocomplete	**DO** use selectively; **DON'T** accept blindly
Debugging Assistance	**DO** consult AI for troubleshooting guidance
Refactoring Suggestions	**DO** evaluate AI proposals critically
Simple Test Generation	**DO** use AI moderately with full understanding

Practice	Guideline
Complex Test Generation	**DON'T** rely on AI; write manually
Understanding AI Output	**DO** ensure comprehension before committing
SonarQube Compliance	**DO** ensure all code passes quality gates

2.2 Guidelines for contributors

Coding style

The PartyShare project adopts the **Google Java Style Guide** as the foundational coding standard for all Java and Spring Boot code. We are expected to familiarize ourselves with the complete guide, available at https://google.github.io/styleguide/javaguide.html .
While the Google Java Style Guide provides comprehensive coverage of formatting, naming and structural conventions, several key principles warrant explicit emphasis:

- **Naming Conventions** - class names should use 'UpperCamelCade', method and variable names should use 'lowerCamelCase', and constants should use 'UPPER_SNAKE_CASE'. Names should be descriptive and convey intent without requiring extensive comments. Avoid abbreviations except for widely understood terms.
- **Code Organization** - classes should follow a logical ordering; group related methods together and order them from public to private; each class should have a single, well defined responsibility.

Code reviewing

Code review is a mandatory quality gate for all code changes. No code may be merged to the main branch without approval from at least one team member. The code review process serves multiple purposes: identifying defects, ensuring adherence to coding standards, sharing knowledge across the team and maintaining consistency across the project.
All pull requests must be reviewed and approved by one team member before merging. The reviewer must be someone other than the author developers, who cannot approve their own pull requests. Given that the team members hold multiple roles, the approval should come from someone acting in the **QA Engineer** capacity when possible, as quality assurance is their primary responsibility. However, any team member with appropriate technical expertise may serve as a reviewer.
Reviews must be completed within the same spring in which the pull request is submitted. This ensures that work progresses smoothly and that context remains fresh for both author and reviewer.

What to review

Reviewers should evaluate pull requests across multiple dimensions:

- **Tests** - Verify that appropriate tests have been written and that they adequately cover the new or modified functionality. Check that tests are meaningful, not merely achieving coverage metrics. Ensure that edge cases and error conditions are tested.

- **Business Logic** - Confirm that the implementation correctly satisfies the requirements specified in the acceptance criteria. Validate that the code handles expected inputs appropriately and degrades gracefully for unexpected inputs
- **Quality Gates** - Confirm that SonarQube quality gates have passed. Review any warnings or issues flagged by static analysis and ensure that they have been addressed or appropriately justified. Verify that code coverage metrics meet project thresholds.

Use of AI tools in Code Review

Reviewers are encouraged to use AI tools as assistants during the code review process. **GitHub Copilot** and **SonarQube** can provide valuable insights that augment human judgment. SonarQube automatically analyzes all pull requests and flags potential issues related to code quality, security vulnerabilities and maintainability concerns. GitHub Copilot can assist reviewers in understanding unfamiliar code patterns, suggesting alternative implementations or identifying potential edge cases.

AI tools should enhance, not replace, thoughtful human review. Reviewers must apply their own expertise, contextual knowledge, and professional judgment to evaluate code quality and correctness.

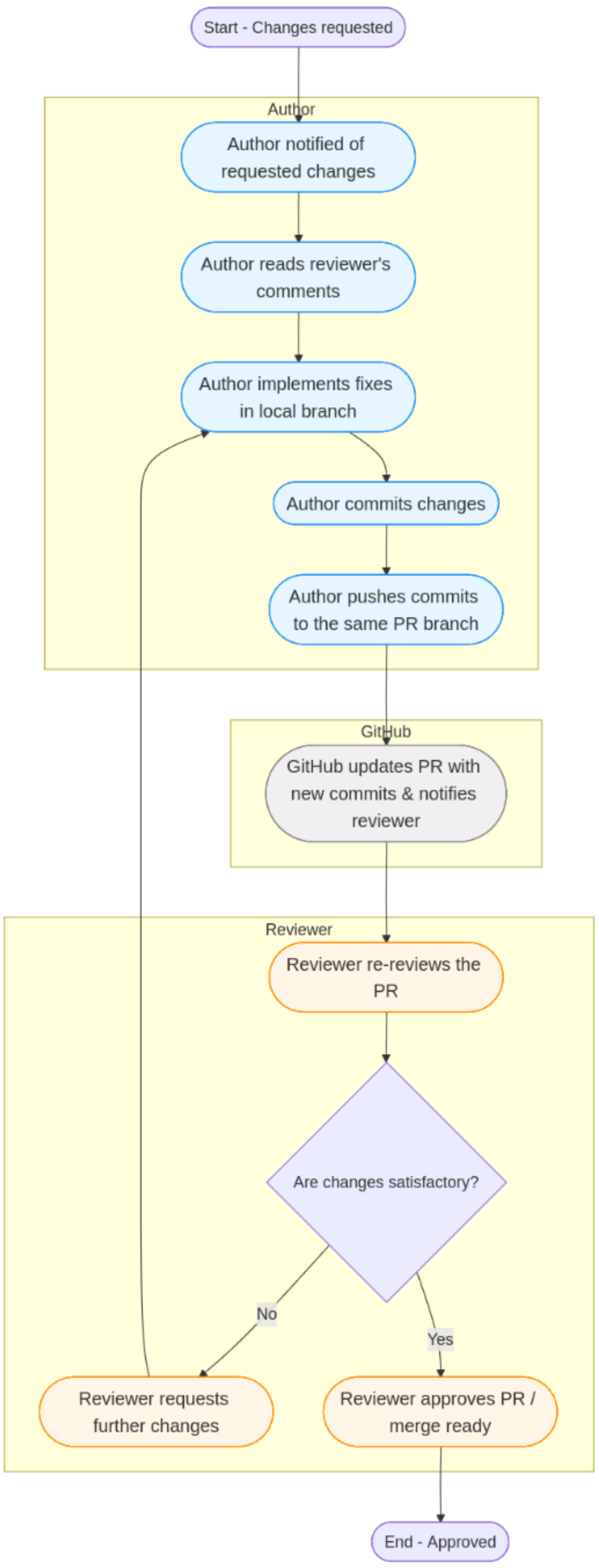
Review Process and Workflow

Code reviews are conducted **asynchronously**. When a pull request is submitted, the author notifies potential reviewers. Reviewers examine the code and provide feedback through GitHub's review interface.

GitHub provides three types of review outcomes:

- **Comment** - Provide feedback without explicit approval or rejection
- **Request Changes** - Indicate that specific issues must be addressed before the PR can be merged
- **Approve** - Confirm that the code meets quality standards and is ready for integration

When changes are requested, the following workflow applies:


Start - Changes requested
Author
Author notified of requested changes
Author reads reviewer's comments
Author implements fixes in local branch
Author commits changes
Author pushes commits to the same PR branch
GitHub
GitHub updates PR with new commits & notifies reviewer
Reviewer
Reviewer re-reviews the PR
Are changes satisfactory?
No
Yes
Reviewer requests further changes
Reviewer approves PR / merge ready
End - Approved

Once final approval is granted, the **author** is responsible for merging the pull request. The project uses a **merge commit** strategy, which preserves the complete commit history of the feature branch and creates an explicit merge commit in the main branch. After merging, the feature branch may be deleted to keep the repository clean.

2.3 Code quality metrics and dashboards

The project employs **SonarCloud** as the primary platform for static code analysis, code quality monitoring, and technical debt management. Static analysis is fully integrated into the project's continuous integration pipeline through GitHub Actions. Analysis is triggered automatically on every push to the **main** and **develop** branches, as well as on all pull requests targeting these branches. This automation ensures that code quality is evaluated consistently and that quality issues are identified early in the development lifecycle.
While continuous integration provides automated quality checks, developers are encouraged to run SonarQube analysis locally during development. Local analysis enables developers to identify and resolve quality issues proactively before pushing code, reducing the iteration cycle during code review.

This project employs the **default SonarCloud** quality gate, which represents industry best practices for code quality thresholds.
The Sonar way quality gate enforces the following conditions on new code:

- **Coverage** - Code coverage on new code must be at least 80%
- **Duplicated Lines** - Duplicated lines in new code must not exceed 3%
- **Maintainability Rating** - New code must achieve a maintainability of A
- **Reliability Rating** - New code must achieve a security rating of A

Code coverage is measured using **Jacoco**, which is integrated into the Maven build process. The project has configured a code coverage target of **80%** in the pom.xml configuration. This threshold ensures that the majority of the codebase is validated through automated tests. The 80% balances thoroughness with practicality, acknowledging that certain code may not warrant exhaustive testing.
Coverage reports are generated during the \`verify\` phase of the Maven build and are automatically uploaded to SonarCloud for visualization and trend tracking. Developers can also view detailed coverage reports locally by examining the HTML reports generated in the \`target/site/jacoco\` directory.
How specific do I have to be with rationale? What I have is enough or need to go deeper?

Beyond the quality gate thresholds, we also actively monitor several additional metrics to maintain awareness of code health and identify areas of improvement:

- Code Smells
- Bugs
- Duplicated Code

These metrics are accessible through the SonarCloud dashboard and are reviewed as part of the team's quality monitoring activities.

Handling Quality Gate Failures

Quality gate failures represent a mandatory quality checkpoint that must be resolved before code can be merged. Our policy is that **all code must pass the SonarCloud quality gate prior to merge**. This non-negotiable standard ensures that quality does not erode incrementally and that the main branch always represents a known, acceptable level of quality.
When a pull request fails the quality gate, the author must investigate the failing conditions, address the underlying issues, and push updated commits that satisfy the quality requirements. The reviewer is responsible for verifying that the quality gate has passed as part of their review checklist.

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.]

This project follows a structured development workflow that ensures quality, traceability and collaboration at every stage. For developers new to the team, understanding this workflow is essential to contributing effectively and maintaining the project's quality standards.

The development process centers on **user stories** as the fundamental unit of work. Each user story represents a "slice" of functionality with clear acceptance criteria. All development activities are organized around delivering these user stories in short, iterative cycles.

We adopt a **feature branch** workflow based on **GitHub Flow** principles, in order to support continuous integration and structured quality gates. The workflow uses two primary branches:

- **develop** - the main development branch that serves as the integration point for all feature work. This branch represents the current state of ongoing development and is the source for creating feature branches
- **main** - the stable production branch that contains only thoroughly tested, reviewed and approved code. Code is merged into 'main' after passing all quality gates and receiving approval.

Branching Model

1. **Feature branches are created from 'develop'** - When a developer begins work on a user story, they create a new feature branch based on the current state of the 'develop' branch
2. **Development occurs in isolation** - All implementation, testing and refinement happens within the feature branch
3. **Pull requests (PR's) merge into 'main'** - When the feature is complete and ready for integration, the developer creates a PR that merges the feature branch into 'main'
4. **'develop' is kept synchronized** - After a successful merge into 'main', the 'develop' branch is updated to incorporate the new feature

User Story to Production Code

For a developer joining the team, the typical workflow for implementing a user story proceeds through the following stages.

At the beginning of each sprint, the **Team Coordinator** facilitates sprint planning. During this session, the team reviews the refined backlog, discusses upcoming user stories and distributes work for the week. The Team Coordinator is the one who assigns stories to developers.

Each developer leaves sprint planning with a clear understanding of which user stories they are responsible for and what the acceptance criteria require.

1. Before beginning development, the developer creates a **feature branch** from the \`develop\` branch. The branch name must follow the format.

   Format: <prefix>/<ISSUE-KEY>-<description>.

2. With the feature branch created, the developer implements the functionality specified in the user story's acceptance criteria. Development proceeds iteratively, with frequent commits that capture logical increments of progress. Commits should be atomic and logically coherent, representing discrete units of progress. The commit history should tell a clear story of how the feature was developed.
   Format: "<ISSUE-KEY> <type>: <message summary>"
   The type can be:
   - feat
   - fix
   - refactor
   - test
   - docs
   - chore

3. Before pushing code to the remote repository, the developer runs a comprehensive set of local quality checks.
   Command: ./mvn clean test;
   Local SonarQube analysis should be performed before committing any new code.
   The developer can also examine detailed coverage reports. If local checks reveal issues, failing tests, insufficient coverage or quality gate failures the developer **must** address them before pushing the code.
4. When the feature is complete and all acceptance criteria are satisfied, the developer creates a pull request that proposes merging the feature branch into `main`.
   Title Format: <ISSUE-KEY>: <Descriptive Title>
   The pull request description provides comprehensive context for reviewers and should include:
   - A link to the JIRA issue
   - A summary of what was implemented
   - A description of key changes
   - Confirmation that tests were written and pass
   - Confirmation that SonarQUbe quality gates pass

   Creating the pull request automatically triggers the GitHub Actions CI pipeline, which runs the full test suite and SonarCloud analysis.

Once the pull request is created, the workflow is the one described on section **2.2 Guidelines for Contributors** under **Code Review.**

Note: Every branch name, commit message, and pull request title **must** include the JIRA Issue Key

Definition of done

We maintain a clear, unambiguous **Definition of Done (DoD)** that specifies when a user story is considered complete and ready for production. All user stories must satisfy every criterion in the DoD before they can be moved to "Done" status in JIRA.
A user story is considered **DONE** when:

- ☐ **All acceptance criteria is met** - The implemented functionality satisfies every condition specified in the user story's acceptance criteria.
- ☐ **Code is implemented and reviewed** - The feature has been **fully** implemented and the code has undergone thorough review by at least one team member who has approved the pull request.

- ☐ **Unit tests are written and passing** - Comprehensive unit tests have been created that validate the functionality at the component level.
- ☐ **Code coverage meets the 80% threshold**
- ☐ **SonarCloud quality gate has passed**
- ☐ **Code is merged to the main branch**
- ☐ **JIRA issue is moved to "Done" status**

This DoD ensures that "Done" has consistent and objective meaning across all user stories, preventing premature closure of incomplete work.

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 tool's setup and config.]
[Description of practices for continuous delivery, likely to be based on *containers*]
[Which different environments are included in CD? Staging, production…]

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 Continuous testing

4.1 Overall testing strategy

[what was the overall test development strategy? How are you **aligning testing with CI/CD**?
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.]

4.2 Acceptance testing and ATDD

[Project policy for writing acceptance tests (closed box, user-facing perspective) and associated resources. when does a developer need to develop these?

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.]
API testing

4.4 Exploratory testing

[strategy for non-scripted tests, if any]

4.5 Non-function and architecture attributes testing

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