TQS: Quality Assurance manual

Pedro Ponte [98059], Alexandre Regalado [124572], Miguel Soares Francisco [108304], Inês Ferreira [104415]
v2025-04-30

1 Project management

1.1 Assigned roles

- Team Leader - Pedro Ponte
- QA Engineer - Alexandre Regalado
- DevOps Master - Miguel Soares Francisco
- Product Owner - Inês Ferreira
- Developer - Everyone

1.2 Backlog grooming and progress monitoring

In this project, we use JIRA to organize and monitor the backlog in an agile workflow, this helps ensure tasks are distributed among the team members and progress is tracked.

Story points are used as the primary measure of effort, where 1 story point equals 1 hour of work. Tasks are estimated based on this ratio, making it easier to plan workloads evenly between team members. To improve organization, tags are applied to classify tasks by categories such as *frontend*, *backend*, or *QA*, allowing for quick filtering. Work is distributed based on team roles, although in some iterations some roles have a bigger workload than others, so other members may balance this out by helping them. Epics represent the big picture work, like *Reports* for all work related to elaborating this report and the Specification Report, and complex stories are divided into subtasks, as some of them require more work and Jira does not allow multiple assignees in one unique task. The workflow follows the usual structure: **To Do -> In Progress -> Done** for tracking progress.

To track work progress, we rely on JIRA's integration with GitHub. Branches are created directly from JIRA issues or by using JIRA notations, such as *CH-19-Create-use-case-diagram*, where *CH-19* is the issue key and *Create use case diagram* is the task name. Commits are associated with their corresponding issues by including the issue key in commit messages, making it easier to track development work and tasks. Pull requests (PRs) are also created through JIRA, linking them directly to the associated work and providing an easy way to review the implementation of a task.

Progress is monitored using tools like burnup charts, which display the completed work in the sprint compared to work left in the scope of a single sprint.

Test coverage and requirements-level monitoring are handled through test management tools integrated into JIRA, such as Xray. These tools link test cases directly to JIRA issues, ensuring every requirement is covered by at least one test. Automated and manual test results are also tracked in JIRA, making it easy to monitor coverage and identify gaps. Dashboards provide an overview of key metrics, such as unresolved defects, test coverage, and overall progress. This proactive approach ensures all requirements are tested and risks are minimized.

2 Code quality management

2.1 Team policy for the use of generative AI

For this project, everyone is allowed to use AI tools as they see fit, whether it's for generating functional code, API tests, asking for help regarding deployment tools or help in writing the reports. However, this usage comes with great responsibility and all members should **always** review the work done before submitting to prevent unknown issues, as the submission is not made by AI, but actual developers who will take the blame if a problem arises.

For technical questions and how to implement certain features using external tools like using a weather API or creating workflows, AI can be used to create such implementations, but the prompts should when possible include examples provided in the updated official documentation, so that the model can use the most recent practices and everyone as an easier time when trying to understand it.

For documental text like reports it shouldn't be used to generate all text based on title only, we should rather provide the section title, provided instructions from the template and a brief explanation of our situation, trying to write as much information as possible and using AI tools only for grammatical/lexical improvements and better explainability of the general problem.

Jira also provides an AI tool for helping with writing task descriptions, the tool will be applied after the description is manually written, this way we can have a standard format in all tasks making them more readable and complete.

2.2 Guidelines for contributors

Coding style

For better consistency and code readability we'll be following the Google Java Style Guide guidelines, this includes:

- Naming - UpperCamelCase for class, lowerCamelCase for methods and non-constant field names
- Formatting - one statement per line, vertical whitespace, block comment style (/* … */), TODO comments

We also have a github-actions workflow using google-java-format to automatically reformat committed code to abide by the aforementioned guidelines.

For the *frontend* we will follow Airbnb JavaScript Style Guide with principles like:

- Naming - PascalCase for files, camelCase for instances
- Spacing
- Props

Code reviewing

When code reviewing we use GitHub's AI tool Copilot that can be used in pull requests to review code and give feedback. It explains the implemented logic and tries to provide detailed feedback on features that may be skipped by human reviewers like some dead code and/or duplicate code. After the AI review, at least one other reviewer must look at the changes and approve them so they can be merged to our *development* branch (two reviewers for *main*).

2.3 Code quality metrics and dashboards

We use **SonarCloud** for continuous static code analysis integrated directly into our GitHub Actions pipeline. Every push and pull request triggers an automatic scan that checks the codebase for issues related to:

- Code coverage
- Code duplication
- Readability
- Security Vulnerabilities
- Maintainability (technical debt)

We have defined the following **quality gates**:

- **Minimum 80% code coverage**
- **Maximum 3% code duplication**
- **Readability rating: A**
- **Security rating: A**
- **Maintainability rating: A**

These thresholds ensure a consistent level of code quality. The rationale for these metrics is based on industry best practices and limitations of the **free tier** provided by SonarCloud.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

For our workflow we follow the gitflow workflow. Our repository has a *main* branch and a *develop* branch.

Develop is used to keep the most recent working code, this branch is updated regularly with work from other branches being merged into it after an approved pull request, it is impossible to directly commit anything to this branch. These other branches are created every time new work is done and must be associated with a relevant task already defined in the Jira backlog by following the naming rules. The tasks are created and assigned by the Team Leader, but anyone can create tasks they find relevant.

Main can only be updated with a pull request accepted by three members, however to contribute to it, the work must come either from a *release* branch, which happen at the end of every sprint and are created based on *develop*, or from a *hotfix* branch, used for introducing fixes of **critical** problems found only after *main* was merged.

Definition of done

A user story can be defined as **Done** when the Acceptance Criteria are met and be confirmed through tests defined by us.

These tests are both functional, using tools for testing the User Story through a user interface and following the Acceptance Criteria and non-functional, by running a series of tests that will check our business logic, services and the API making sure everything is in order.

3.2 CI/CD pipeline and tools

We use a CI/CD pipeline configured with **GitHub Actions** to automate our workflow. Each push or merge request to the `develop` branch triggers the following jobs:

- Build validation using Maven
- Unit tests execution with JUnit and Mockito
- Code coverage analysis with JaCoCo
- Static code analysis via SonarQube
- Integration tests using REST Assured and Spring Boot Test
- Linting and formatting
- CI status badges included in the README

For **continuous delivery**, the application is containerized using **Docker**, and deployment to a virtual machine provided by the professor is triggered manually or through the release pipeline. The final release is pushed to the `main` branch through a pull request, ensuring all checks are passed and peer reviews are approved.

Whenever code is pushed to the `main` branch, a **deployment workflow** is automatically triggered to deploy **ChargeUnity**. The process uses **GitHub Actions** with a **self-hosted runner** (installed on a university virtual machine or a local server).

How the Continuous Delivery process works:

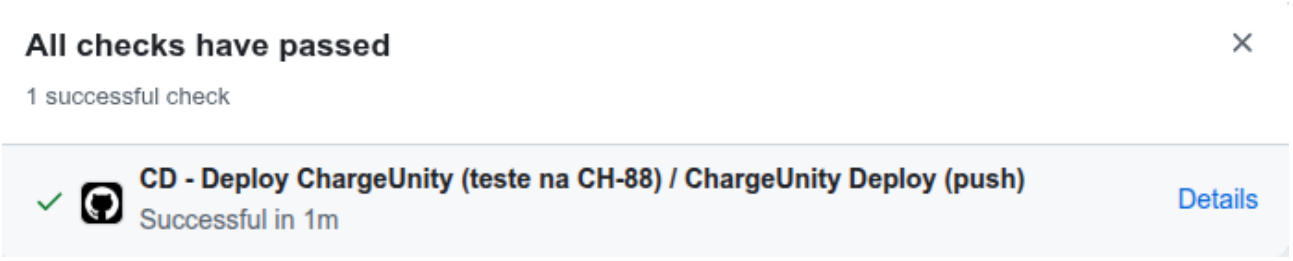
1. **Push to the main branch**
   Code is pushed to the **`main`** branch, which typically happens through a pull request.

2. **GitHub Actions triggers the runner**
   As soon as a push is made to **`main`**, GitHub Actions notifies the self-hosted runner. This runner is configured on a virtual machine and listens for deployment events in the background.

3. **Runner handles the deployment with Docker**
   The runner:

   - Dynamically generates a **`.env`** file using production secrets.
   - Stops previous containers (**`docker compose down`**) while **keeping named volumes intact**.
   - Cleans up old Docker images with `docker builder prune`.
   - Builds the backend using Maven (**`mvn clean package`**).
   - Restarts all services with **`docker compose up -d --build`**, exposing the necessary ports to the outside world.

4. **Application is ready**
   ChargeUnity is automatically deployed and made available to users.

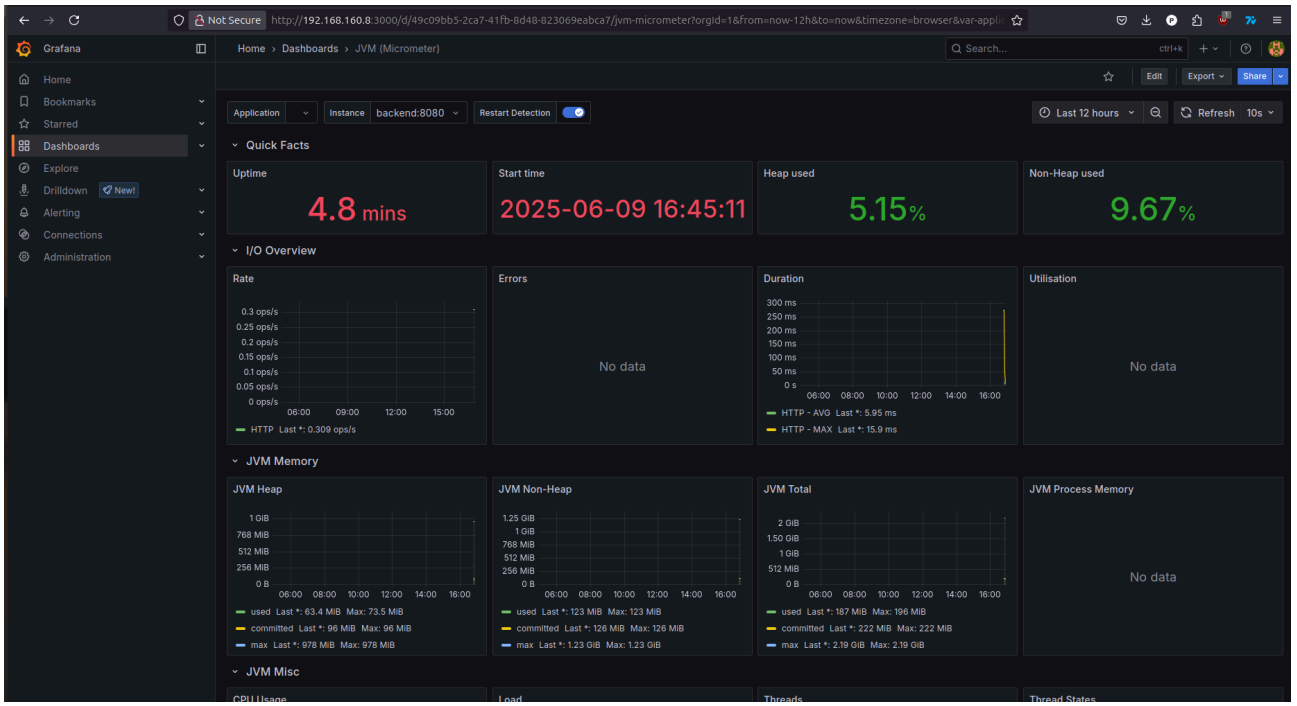


3.3 System observability

The project includes basic observability features to monitor the system's health. The following data is tracked:

- HTTP status codes and requests logs for all REST endpoints
- Error rates and response times using **Spring Boot Actuator**
- Test pass/fail status and code coverage via CI pipeline
- Static analysis results in SonarQube

Build failures, quality gate violations, or broken tests in GitHub Actions trigger email or platform-based notifications to the development team. Additional alerts are reviewed manually during code reviews and QA validation steps.



3.4 Artifacts repository [Optional]

The project currently uses public Maven repositories for dependency management and does not maintain a local artifacts repository.

4 Software testing

4.1 Overall testing strategy

Our testing strategy follows multiple approaches, combining:

- **Test-Driven Development (TDD)** for core backend services
- **Behaviour-Driven Development (BDD)** using **Cucumber**, with files written in Gherkin
- Integration testing using **REST Assured**
- **Code coverage enforcement** via **JaCoCo**
- All automated tests are run as part of the GitHub Actions pipeline, and their status is reviewed before merging pull requests.

4.2 Functional testing and ATDD

Functional tests are written using the Acceptance Test-Driven Development (ATTD) methodology.

Acceptance criteria from each user story are translated into test cases, which are then implemented using Cucumber.

These tests validate that the system behaviour matches the expectations from a user's point of view. Developers must implement and pass these tests before a story can be marked "done."

4.3 Unit tests

Unit testing is required for all core components. The tem uses JUnit for writing unit tests and Mockito for mocking dependencies. Unit tests are required when:

- A new service or controller is introduced
- A method includes logic or branching
- Bug fixes are implemented

Unit tests are executed in every CI run, and the project enforces a minimum of 80% test coverage using JaCoCo.

4.4 System and integration testing

Integration testing ensures that system components interact correctly, especially across layers (controller -> service -> repository)

We use:

- **Spring Boot Test** for application context tests
- **REST Assured** for validating HTTP endpoints and API contracts
- Test cases include success scenarios, database access, and service coordination

These are triggered after unit tests in the CI pipeline.

4.5 Non-function and architecture attributes testing

Performance and load testing are conducted using **K6**. Simulations include making requests to several random endpoints every second.