# TQS: Quality Assurance manual

*António Alberto 114622, Diogo Fernandes 114137, Henrique Oliveira 113585, Raquel Vinagre 113736*
V2025-06-08

## Contents

## 1 Project management

### 1.1 Assigned roles

Raquel Vinagre – Team Manager
António Alberto – Product Owner
Diogo Fernandes - DevOps
Henrique Oliveira – QA Engineer

### 1.2 Backlog grooming and progress monitoring

Team members regularly update JIRA with new changes as the project is developed. Tasks are distributed among members according to their role, making sure everyone contributes the same amount. Every week, a new sprint starts and members focus on the tasks planned for the week, updating the board. As for epic development, user stories with more points are prioritized. Once these are completed/ almost completed and tested, members focus on the rest of the stories. X-ray is setup in JIRA, allowing the creation of commits and branches directly from the platform, which helps maintain both code development and project management within the same guidelines.

## 2 Code quality management

### 2.1 Team policy for the use of generative AI

We stand for the use of AI-assistants during production to accelerate development and reduce boilerplate. However, we do not take AI-generated code quality for granted and carefully review and make necessary alterations before making use of it.

**Do's:**

a) Generation of boilerplate code
b) Reviewing of PRs
c) Drafting of documentation and comments
d) Inquire about possible implementations or refactoring
e) Writing Unit/Integration tests (the most basic ones)
f) Help with debugging

**Don'ts:**

g) Blindly trust AI code
h) Accept AI-generated changes without reviewing
i) Accept without reviewing AI-generated tests for edge cases and complex, project-specific cases

### 2.2 Guidelines for contributors

**Coding style**

We adopted a clean, pragmatic, and layered architecture, with separation of concerns between layers (controllers, services, repositories, entities) on the backend and component-based, reactive state management on the frontend. Both codebases prioritize readability and maintainability.

**Code reviewing**

In terms of code reviewing, we made sure to appoint as reviewer the person closest to the issue the PR addressed. When necessary, the reviewer could appoint a second reviewer for a different opinion. AI was scarcely used as a code reviewer.

## 2.3 Code quality metrics and dashboards

**Backend (Spring Boot)**:

j) Tool Used: **SonarQube**

k) Integration: CI/CD via GitHub Actions

l) Features:

m) - Analyzes Java code for bugs, code smells, vulnerabilities, and complexity
- Collects test coverage metrics using JaCoCo, triggered by mvn verify

We defined the following quality gates for the backend through SonarQube:

n) Code Coverage: >= 90% (via JaCoCo)

**o)** No new critical bugs or vulnerabilities

p) No duplicated code blocks in new code

q) Maintainability Rating: A

**Frontend (React):**

r) Tools Used: **ESLint** and **Prettier**

s) Purpose:
- ESLint catches common JavaScript/TypeScript issues and enforces best practices
- Prettier enforces consistent formatting across the codebase

t) These tools are configured to run locally and can be integrated into CI if needed

All this was done in hopes of encouraging writing of tested, clean and maintainable code, while also making it easier to catch bugs early on and avoid future technical debt.

# 3 Continuous delivery pipeline (CI/CD)

## 3.1 Development workflow

**Coding workflow**

The stories are assigned based primarily on user preference and proficiency. Developers can pick user stories based on their want to develop them and bring it up with the Team Manager. Then the Team Manager can weigh the story's focus against the developer's skills (e.g. if it is a more backed-heavy story, it probably should not be assigned to someone who primarily works with frontend). After work is distributed, a branch is created via Jira for that user story, where future child-branches can be created to specifically address parts of the user story (e.g. create a child branch for the service logic relevant for the User Story ; create another child branch for unit testing, and a different one for integration testing, ...)

**Definition of done**

Full implementation of the user story, each step has been reviewed and carefully tested. All frontend and backend tests pass, covering each step of the user story.

### 3.2 CI/CD pipeline and tools

**CI Pipeline:**

Our CI pipeline runs automatically on every pull request and every push to develop/main.

For the frontend, we use **ESLint** and **Prettier** to enforce consistent code style and formatting. These checks are required to pass before a PR can be merged.

On the backend, we integrate **SonarCloud** with Maven to perform static analysis and track code quality. The analysis covers test coverage, code smells, bugs, and other metrics defined by the SonarCloud quality gate. Like the frontend tools, these checks are enforced; a PR cannot be merged unless all checks succeed.

In short, every code change is validated for style, correctness, and quality before it enters the repository.

**CD Pipeline:**

Deployment is handled via a **GitHub Action** that runs on a self-hosted runner deployed on the provided VM. The action is triggered by pushes to the main/develop branches. It checks out the repository and runs the full application using Docker Compose. All services are containerized and orchestrated together in this setup, which ensures reproducibility and consistent behavior across environments.
As we already had the application fully containerized, we chose to just use docker compose instead of building an image to, for example, DockerHub. This approach allows us to spin up the complete application stack with a single command and ensures a smooth, automated deployment pipeline.

### 3.3 System observability

For observability we used **Micrometer,** integrated into the Spring Boot backend to expose application-level metrics. These metrics are scraped by **Prometheus**, including:

u) HTTP request rates, errors, and durations

v) JVM memory and thread usage

w) Database query timings

Aside from that, logs are collected from the backend using **Promtail** and aggregated by **Loki**. These Logs are searchable in **Grafana**, enabling traceability and debugging of:

x) User actions

y) API failures

z) System warnings and exceptions

Finally, we use **k6** for performance/load testing, simulating real user interactions. k6 results are stored in **InfluxDB** and also visualized in Grafana.

For visualization purposes, we created 2 dashboards: one for system monitoring metrics and logs (JVM (Micrometer) + Logs) and the other for monitoring k6 results (k6 Load Testing Results).

# 4 Software testing

## 4.1 Overall testing strategy

aa) We defined a **TDD approach at the Unit level** wherever feasible, to define expected behavior before implementation and catch regressions early, using **Junit 5** and **Mockito**.

bb) For our Integration Tests we used **Spring Boot integration tests** for full controller–service–repository flows and **REST-assured** for verifying HTTP endpoints from our external Geoapify API.

cc) For E2E testing we used **Cypress** to simulate user interaction with Jolteon, following the steps in our **User Stories**.

dd) In terms of CI Integration, we use **GitHub Actions**, executed on every push and pull request. The **backend pipeline** runs mvn test with coverage reporting, while the **frontend pipeline** runs npm test and Selenium E2E tests. Additionally, **Linting checks** are also included to ensure code quality.

**ee)** To ensure Static Analysis and Code Quality, we use **SonarQube** (for code smells and bugs detection, and technical debt analysis) with **JaCoCo** for code coverage.

ff) Regarding Load Testing, we chose **K6**, performing load and spike tests.

## 4.2 Functional testing and ATDD

In our project, **functional testing** follows a **black-box approach,** focusing on validating that the system behaves according to **user-facing requirements**, regardless of internal implementation.

We align with ATDD practice of defining acceptance criteria collaboratively, based on our user stories. We also tried to follow the "three amigos" practice (collaboration between customer, developer and tester) but since we lack a "customer", we instead work on the acceptance criteria with the one who wrote/will write implement the functionality (developer), the one who will test it (tester) and the Product Owner, since it is the closest we can get to a real customer, and has the deepest understanding of our project requirements and user stories.

Functional tests are **mandatory** for all user stories with non-trivial logic, and are achieved with Cypress tools for frontend testing.

## 4.3 Developer facing tests (unit, integration)

We followed a structured approach for developer-facing tests to ensure code correctness and maintainability.

**Unit Tests**:
Written using JUnit 5 and Mockito. We were expected to write unit tests for all business logic, validations, and bug fixes. Key focus areas included Services and Controllers. Tests must be short, fast and isolated.

**Integration Tests**:
Used to validate multi-layer behavior of Service + Repository with real DB interactions. These use Spring Boot's testing annotations and often run against an in-memory DB.

**API Tests**:
We use MockMvc for controller testing to verify endpoints, status codes, and request/response formatting. Service tests use mocks to isolate logic from repositories or external systems.

Tests are triggered during development and reviewed via PRs.

## 4.4 Non-function and architecture attributes testing

Our project's non-function tests are done with **k6**, integrated with **InfluxDB** and **Grafana** for visualization. We defined load tests to simulate realistic user behavior (e.g., concurrent bike rentals and trip completions). And spike tests to test the availability of Jolteon when under stress.

Regarding our execution policy, these tests are triggered manually using Docker Compose (docker compose run --rm k6). We meant not to integrate these tests in the CI pipeline to avoid unnecessary resource consumption during builds. Lastly, tests explicitly account for valid rejection scenarios (e.g., "user already has an active rental", or "station full"). These are logged and excluded from threshold failure conditions to reflect true system performance.