deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Carolina Prata [114246], Diogo Domingues [114192], João Varela [113780]
v2025-06-09

Contents

1 Project management

1.1 Assigned roles

After some discussion between the members we decided the following distribution was the best for us:

- João Varela: DevOps Master
- Carolina Prata: Product Owner and Team Coordinator
- Diogo Domingues: QA Engineer

Due to the team having only three members one of us has to have two roles, besides these roles all of us also perform the Developer role.

1.2 Backlog grooming and progress monitoring

The project team utilizes Jira Cloud with the Scrum framework to manage and organize all development activities. Work is structured hierarchically into Epics, User Stories, and Tasks, reflecting both functional and non-functional requirements as defined in the specification document.

To ensure continuous alignment with project objectives, weekly backlog refinement sessions are conducted under the coordination of the Team Leader. These sessions serve to review and clarify existing user stories, add or modify tasks in response to evolving requirements, adjust story point estimates, re-prioritize work items, and validate acceptance criteria. This iterative grooming process ensures that the backlog remains accurate, clear, and actionable.

Each sprint begins with a sprint planning session, during which the team selects a subset of backlog items based on current priorities and estimated capacity. Progress throughout the sprint is systematically monitored using story point tracking, burndown charts, and custom Jira dashboards, which provide real-time visibility into issue status, team velocity, and sprint completion forecasts.

Furthermore, the team adopts a proactive approach to requirement-level test coverage. All user stories and tasks are integrated with Xray, a test management tool embedded in Jira. This integration ensures full traceability from requirements to test cases and results, enabling early identification of coverage gaps. Xray's reporting features support continuous quality assurance through comprehensive coverage analysis and traceability reports.

2 Code quality management

2.1 Team policy for the use of generative AI

Generative AI is very useful and easy to use here are the do's and dont's.

- **Do's:**
    a. AI may be used to help write code or tests and troubleshooting;
    b. AI may also be used to help understand pieces of code that might be unclear;
    c. AI might also be used to suggest improvements or generate documentation drafts.
- **Dont's:**

a. Do NOT push AI code without testing it;
b. Beware of AI comments, while some are useful others are just pure useless, so filter these code comments;
c. Beware of revealing secrets like API tokens or DB credentials to generative AI;
d. Do not copy large chunks of code without understanding their purpose;
e. AI can be a good tool to help writing tests, but always verify that they are doing what they are supposed to.

2.2 Guidelines for contributors

Coding style

Coding style of this project will be very similar to AOS project for Java Development and Airbnb React Style Guide for most of the React development along with these accessibility rules. Some rules that are worth emphasising are:

- Be consistent in both Java and React development one of the most simple yet effective rules;
- As for Java development, follow the Java language,library and style rules as well as the Javatest styles outlined in the AOS project.
- For React development follow mainly the style guide present in Airbnb React Style Guide with special highlight for the rules regarding naming, quotes, alignment, spacing and props styles as well as the accessibility rules present here.

Code reviewing

Code reviewing is essential to ensure that the code delivered by the team is high quality, secure, and maintainable. An effective code review helps catch bugs, ensures consistency, and allows the team to better understand and collaborate on the written code.

Code reviews should be conducted after the developer has completed their changes (whether bug fixes or new features/user stories), tested them locally, and submitted a Pull Request (PR) with the proposed changes. The PR should include a clear description outlining the goals and impact of the changes, the type of modifications made, how the changes were tested, and screenshots if applicable.

Code Review Roles and Process:

- The developer who wrote the code submits the PR for review.
- For the dev and release branches, one team member is enough to review the code.
- For the main branch, two team members must approve the PR to ensure quality and security before merging.

The review should be thorough. Reviewers should check:

- Whether the code works correctly and solves the intended problem or implements the desired feature.
- If the code is written in a clear, maintainable way and follows the team's coding standards.
- That no bugs, security vulnerabilities, or performance issues are introduced.
- That the code adheres to project-specific guidelines such as accessibility, naming conventions, and best practices.

Feedback should be constructive,i.e, reviewers should offer actionable suggestions for improvement and highlight any issues needing attention. Reviews should always be respectful and aim to support the author in improving the code.

Before submitting a PR, the developer should test the code locally and ensure it passes all tests. The PR must also pass all quality gates defined in SonarQube and not introduce any new vulnerabilities (checked via CodeQL analysis). Once all checks are passed, the author should request peer reviews.

2.3 Code quality metrics and dashboards

For static code analysis, we use CodeQL for security and vulnerability scanning, and SonarQube Cloud for evaluating test coverage on new code, maintainability, reliability, and additional security practices.

CodeQL must not detect any critical issues in the submitted code. For SonarQube, we use the default quality gate, which enforces the following conditions:

- New code test coverage is greater than or equal to 80.0%
- Duplication in the new code is less than or equal to 3.0%
- No new issues are introduced (Security Rating A, and Reliability C)
- All new Security Hotspots are reviewed (100%)

These thresholds were chosen as a balanced approach that is neither too strict nor too lenient.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

To start coding, follow these steps:

- Assign a JIRA task to yourself and create a new branch in the (correct) repository.
- Begin implementing the solution based on pre-existing tests (TDD approach).
- Ensure all existing tests pass and write new ones if necessary.
- Once development is complete, create a PR and request peer review.
- Address any feedback from reviewers as needed.

Definition of done

A user story is considered complete when it is ready for release, meaning the user can complete the feature via the frontend and it works as expected.

3.2 CI/CD pipeline and tools

Our CI process ensures that:

- Builds are successful upon each PR submission.
- Unit and integration tests run and pass automatically.
- SonarQube verifies the build quality on the backend (test coverage, quality gates).
- CodeQL checks for security vulnerabilities.

On the frontend, a custom build script is used alongside CodeQL and SonarQube analysis.

For continuous delivery, we use GHCR to build Docker images tagged with release versions (e.g., vX.X). These images are automatically deployed to the VM using WatchTower. Since main only accepts tested and reviewed code, it is always in a deployable state.

3.3 System observability

To ensure proactive system monitoring, we use Prometheus for metrics collection and Grafana for data visualization. Dashboards include key stats such as CPU usage and memory consumption. Alarms are set up to detect and notify of backend or database failures. Additional alerts and dashboards can be configured as needed.

3.4 Artifacts repository

We maintain two deployment repositories:

- Deployment (deprecated) - Previously used for deployment using references to frontend/backend repositories.
- Deployment with GHCR (current) - Uses GHCR Docker images built during GitHub releases.

4 Software testing

4.1 Overall testing strategy

Our testing strategy is based on Test-Driven Development (TDD) using JUnit 5 with necessary Spring Boot dependencies and annotations. REST-Assured is used for HTTP-level testing. All tests must pass for a PR to be accepted.

4.2 Functional testing and ATDD

Basic functional tests are usually in place due to our TDD practice. However, developers are responsible for writing more complex or scenario-specific functional tests as needed. These tests simulate the system from a user's perspective (e.g., bookings, reservations, route calculations) and operate in a closed-box manner.

4.3 Developer facing tests (unit, integration)

Unit tests are open-box and validate the internal logic of components like services and controllers. While most unit tests are created via TDD, developers can write additional tests when refactoring or fixing bugs.

Integration tests usually focus on interactions with external APIs. These are typically closed-box from the developer's perspective, as the internal workings of external services are not known. Integration tests validate correct data flow and behavior between components.

API testing is included in ITs.

4.4 Exploratory testing

Exploratory testing was not actively performed as part of the development and testing workflow in this project. The team primarily relied on automated and scripted testing approaches, such as Test-Driven Development (TDD), unit tests, integration tests, and functional testing aligned with specific user stories.

Although exploratory testing was not formally adopted, it can serve as a valuable complement to automated testing. It allows testers or developers to interact freely with the application to uncover unexpected behaviors, usability issues, and edge-case bugs that may not be captured by predefined test cases. In particular, exploratory testing often leads to informal usability evaluation such as identifying confusing interfaces, unclear labels, inefficient workflows, or inconsistent interaction without requiring full usability test sessions.

In future iterations, incorporating exploratory testing during QA phases or before major releases could help improve overall quality and user experience by catching issues that automated tests may miss.

4.5 Non-function and architecture attributes testing

The k6 test simulates multiple users (Virtual Users or VUs) making reservation requests to the backend API over a period of time, assessing the system's performance under different traffic loads. The primary goals are to:

- Ensure the system handles multiple concurrent reservations efficiently.
- Monitor response times and ensure they meet the required thresholds.
- Verify that no conflicts occur when multiple users attempt to reserve the same charger at the same time.

Test Configuration and Flow

The test is designed with several stages to simulate different usage patterns and load conditions, including ramp-up periods, steady loads, spikes, and ramp-downs:

- Ramp-Up:
    - 5 minutes with a gradual increase in load, from 0 to 50 Virtual Users (VUs). This simulates the system starting to handle traffic.
    - 5 minutes to increase the load to 100 VUs. This represents normal usage during peak times.
    - 5 minutes to further increase the load to 150 VUs, simulating an increased number of users, such as during promotions or high-demand periods.
- Steady Load:

    - For 5 minutes, the load is kept steady at 100 VUs to simulate a consistent user base.

- Spike Test:

    - In 5 minutes, the number of users spikes to 150 VUs, simulating a sudden surge in traffic. This helps verify if the system can handle traffic spikes without performance degradation.

- Ramp-Down:

    - Finally, the load is gradually reduced back to 0 VUs over 5 minutes, simulating a decrease in traffic as the system reaches a quieter state after peak demand.

Thresholds and Metrics

To ensure the system's performance meets expectations, the following thresholds are defined for key metrics:

- http_req_duration: 95% of the requests should complete in under 100ms. This ensures that the system remains fast even under increasing load.

- http_req_failed: No more than 1% of requests should fail. This measures the reliability of the reservation system, ensuring that most reservation requests are successfully processed.

- success_rate: The success rate should be 100%. This indicates that all requests should complete without any errors (e.g., conflicts).

- response_time: The average response time for reservation creation should be under 20ms, ensuring that users do not experience delays during the reservation process.

Test Execution and Monitoring

- Virtual Users (VUs): The test starts with 50 VUs and gradually increases up to 150 VUs, testing how well the system scales as more users interact with it simultaneously.

- Response Time Tracking: The response times for each reservation request are tracked to ensure they remain within acceptable limits.

    - The Trend metric is used to track response times.

    - A Rate metric is used to track the success rate of the requests.

The test is designed to provide a comprehensive understanding of how the reservation system performs under normal and stress conditions, focusing on scalability and the ability to handle a high volume of concurrent requests without failures.

Key Test Results and Observations

- Response Time: The system must process 95% of requests under 100ms. This ensures that users are not waiting long periods to complete their reservations, which is crucial for a seamless user experience.

- Failure Rate: The failure rate is kept below 1%, ensuring that the system remains reliable even under heavy load.

- Concurrency Handling: The test checks how well the system handles concurrent requests, especially when multiple users attempt to reserve the same charger at the same time. The system's ability to manage reservation conflicts efficiently is crucial.

- Spikes and Steady Load: The system is able to handle sudden traffic spikes (e.g., 150 VUs for 5 minutes) without significant degradation in performance or failures, demonstrating its robustness under variable load conditions.