TQS: Quality Assurance manual

David Amorim [112610], Francisca Silva [112841], Guilherme Amaral [113207], Henrique Freitas [114990]

v2025-06-09

Contents

1 Project management

1.1 Assigned roles

Roles	Assignee
Team Leader	David Amorim
Product Owner	Guilherme Amaral
QA Engineer	Francisca Silva
DevOps master	Henrique Freitas
Developer	All the team members

1.2 Backlog grooming and progress monitoring

We are using the backlog of JIRA for management of the project. There we created the user stories and grouped them into epics. Each user story was composed of a set of tasks that were assigned to different team members. We only considered a user story completed if all the tasks were completed successfully, all the acceptance criteria are covered, and all the tasks were tested.
In the end of a sprint, we plan the next one, to assure that we continue to work with coordination. This way, we have meetings every week to review and prioritize the work to be done.
On a regular basis, progress is tracked by using story points. This way we start to implement the user stories with greater story points.
We also integrated in Jira the XRay to monitor the level of coverage in our tests.

2 Code quality management

2.1 Team policy for the use of generative AI

The team position about the use of AI-assistants is that we don't use the AI to generate new code, but we can use it to help fix bugs or errors. This way we can improve our code capabilities and be more involved with the project.
We don't use copilot for code reviews because we believe that this way a more human friendly review can be better for learning purposes.

2.2 Guidelines for contributors

Coding style

We decided that we would follow the coding style of Google project .
For instance, we would use:

a) UpperCamelCase for class names;
b) lowerCamelCase for method names;
c) Block indentation: +4 spaces[Adapted];
d) One statement per line;

Code reviewing

For a more effective code review, the reviews will occur after all automation checks and before merging. Once the code is ready, we will mark someone to review the code.
We consider a feature done when the quality gate is passed and the tests are done and passed.
We will not integrate this process with AI tools. If more than one person as contributed to the pull request, neither the owner of the PR nor the person that contributed can close the PR.

2.3 Code quality metrics and dashboards

To assure the quality of our code we used the SonarQube cloud analysis.
We applied the default metrics of the SonarQube:

e) 0 new issues;

f) 0 Security Hotspots;

g) More that 80% coverage on the new code;

h) Less than 3% deduplication lines on new code;

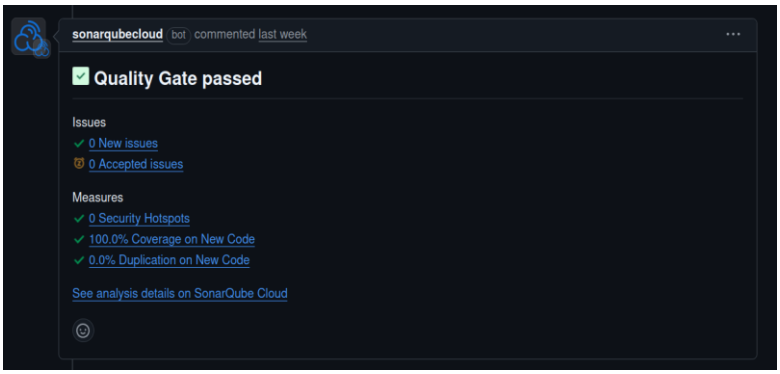


Fig.1 - Example of the Quality Gate passed on a pull request

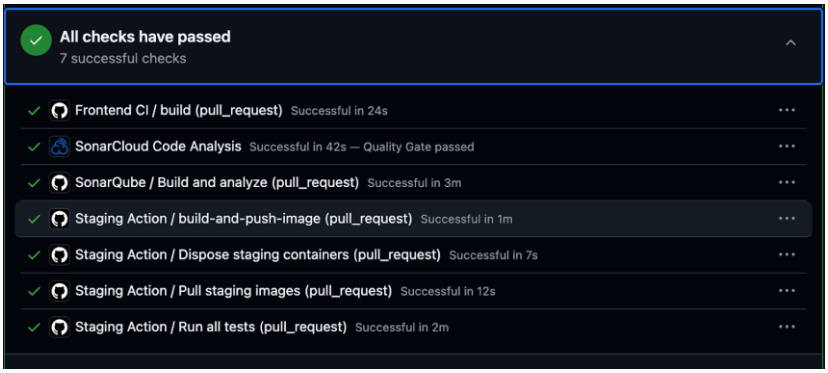


Fig. 2 – Example of the checks done to merge the pull request

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

A user story is divided into several subtasks every single ticket has only one responsible by it. Development begins with the assignment of a **user story** taken from a project management system (e.g., GitHub Projects, Jira). Then a branch is created via the Jira ticket with the task associated to the element of the group.

The work is done and tested locally. Once ready:

i) The branch is pushed and a **Pull Request is opened to the "dev" branch**.

j) After review and approval, it is merged into it, which triggers the staging process.

Definition of done

Our team's definition of "Done" means that a user story is fully implemented according to the acceptance criteria, thoroughly tested locally, and passes all automated checks, including SonarQube quality gates. The code must be reviewed, always by at least 1 person, and approved through a pull request, covered by relevant unit and integration tests, and successfully deployed to the staging environment. Once validated, it's merged into main and deployed to production.

3.2 CI/CD pipeline and tools

Continuous Integration

We use **GitHub Actions** to automate continuous integration:

k) Every push triggers automated workflows.

l) Frontend changes trigger specific actions.

m) **SonarQube** checks code quality on all pushes.

n) The **Staging pipeline** runs automated tests before code reaches main.

Core tools:

o) GitHub Actions

p) SonarQube Cloud

q) Docker

r) GitHub Packages (for image registry)

Continuous Delivery

Once code is merged into main:

s) Docker images for frontend and backend are **automatically built**.

t) Images are **pushed to GitHub Packages**.

u) The provided virtual machine then through the hosted github action-runner **pulls the images**.

v) The project is **automatically deployed to production** using those images.

3.3 System observability

For system observability, we rely on automated testing and real-time monitoring to ensure the health and reliability of each deployment. All changes go through unit, integration, and Cucumber tests in the staging environment, with results tracked via XRAY. During deployment, we use K6 to run load tests and stress the system under realistic conditions. For real-time metrics and alerting, we integrate Prometheus and Spring Actuator with a Grafana interface, which allows us to visualize system performance, detect anomalies, and respond proactively to potential issues in both staging and production environments.

3.4 Artifacts repository [Optional]

For artifact management, we use **GitHub Packages** to store and version our Docker images for both staging and production, ensuring traceability, easy rollbacks, and consistent deployments across environments. Making the clear distinction of both images via docker version tags.

4 Software testing

4.1 Overall testing strategy

We adopted a BDD strategy for testing our application.
Firstly, we would write scenarios that would describe how the system should work so that all of the team members knew exactly how things should be implemented.
After this we would implement the backend of each user story and then we would make unitary tests and integration tests (we used **Rest-Assured** and MockMvc) to see if everything was working properly and to make the integration with the frontend easier.
Then we would make the frontend and integrate the backend, then we would do the **Cucumber** tests to allow us to validate all the features implemented. In the end, we would do the K6 performance tests.

One of the most important steps of our CI process is to check if all tests created have passed.

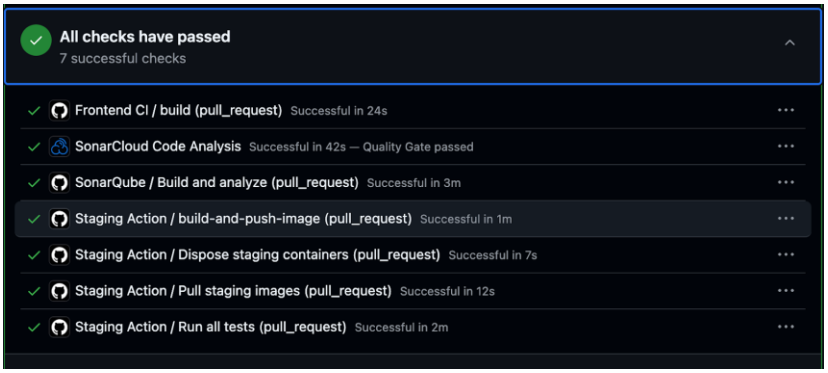


Fig. 3 – Example of the checks done to merge the pull request

4.2 Functional testing and ATDD

We adopt a **closed box** approach for writing functional tests and we decided that we would develop them when we have a new user story or when we have something that needs to be validated.

```
@MSEV-19
Feature: Register in the application
  Scenario: Valid sign up
    When I open the app
    And click on login
    And click on signup
    And fill the signup form
    And submit the signup form
    Then I should be redirected to the home page
    And I should see a logout button

  Scenario: Invalid sign up
    When I open the app
    And click on login
    And click on signup
    And fill the signup form with invalid password
    And submit the signup form
    Then I should see an alert saying "Passwords mismatch"
```

```
@MSEV-18
Feature: See charger details
  Scenario: See details of a charger
    Given that I have made a station search
    When Click on a station
    Then I should see a list of its chargers
    And each charger should display its type
    And each charger should display its pricing
```

Fig. 4 – Example of Cucumber tests

4.3 Developer facing tests (unit, integration)

We implemented unitary tests to make sure that all the modules of our application were working properly individually and used an **open box** approach. For all new methods created that were part of implemented features in our application, we wrote a unitary test. We created these tests for controllers, services and repositories to ensure the correctness of them isolated.
The most relevant unit tests that we considered were the ones involved with the reservations because they had more edge cases.

We also implemented integration tests to verify that the components work together as well. For these tests we used Rest Assured for API testing and to validate the services and MockMvc to test the controllers. Depending on what was tested, we used **an open and closed box** approach.

4.4 Exploratory testing

We don't have a strategy for non-scripted tests but we've done some to test edge cases and to improve our application, having in mind our business logic.

4.5 Non-function and architecture attributes testing

We've done some performance tests using K6 like normal usage, ramp and spike tests. We measure metrics like success and failure of the HTTP requests, the duration of them and other checks.

We tested the methods:

w) Authentication;
x) Get stations;
y) Get chargers of a specific station;

We considered these limits to better understand if the system was working as expected:

```
const thresholds = {
    http_req_duration: ['p(95)<1100'],
    http_req_failed: ['rate<0.01'],
    checks: ['rate>0.98'],
};
```

Fig. 5- Thresholds used on the performance tests

These were the results that we obtained and as we can see the average HTTP request duration is 630 ms and we had 46875 requests done successfully.

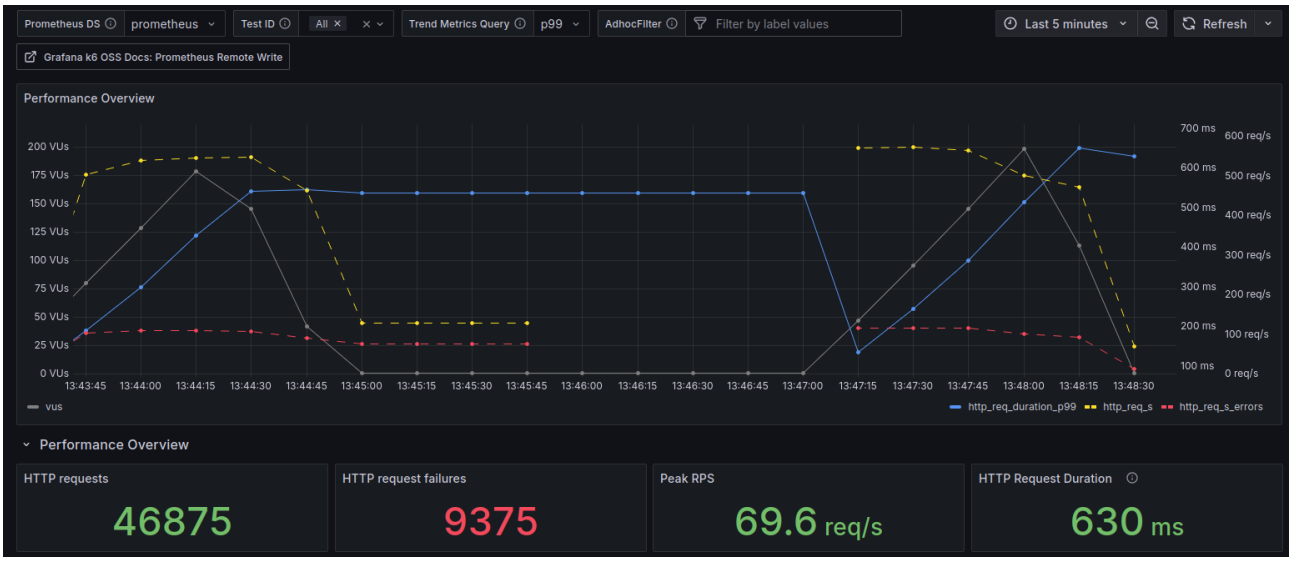
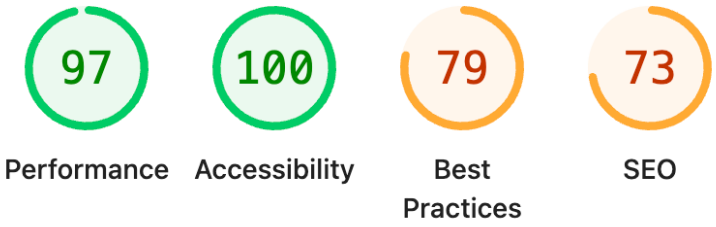


Fig. 6 – Results of the performance tests

We used the lighthouse to perform more tests and here are the results:



We got good results at performance and accessibility metrics, however, we lost some points at best practices and SEO, essentially because we don't use https.