45426 Teste e Qualidade de Software

TQS: Manual de Quality Assurance

André Ribeiro [112974], Violeta Ramos [113170], Diogo Guedes [114256]

v09/06/2025

1 Project management **2**
1.1 Assigned roles 2
1.2 Backlog grooming and progress monitoring 2

2 Code quality management **4**
2.1 Team policy for the use of generative AI 4
2.2 Guidelines for contributors (coding style) 4
2.3 Code quality metrics and dashboards 6
2.4 Xray 7

3 Continuous delivery pipeline (CI/CD) **9**
3.1 Development workflow 9
3.2 CI/CD pipeline and tools 10
3.2.1 Continuous Integration (CI) 10
3.2.2 Continuous Deployment (CD) 13
3.3 System observability 14

4 Software testing **15**
4.1 Overall strategy for testing 15
4.2 Functional testing/acceptance 15
4.3 Unit tests 16
4.4 System and integration testing 17
4.5 Performance testing 18

1 Project management

1.1 Assigned roles

The team consists of a team coordinator, a product owner, a QA engineer, a DevOps master, and developers. The assigned person and the responsibilities of each role are as follows:

- Team Coordinator (All members of the team): Since one of the members dropped out, we opted to take on the role ourselves
- Product owner (Diogo Guedes): Represents the interests of stakeholders and possesses a deep understanding of the product and the application domain, the team will turn to the Product Owner to clarify any questions about expected product features. Should be involved in accepting incremental solutions.
- QA Engineer (QA Engineer): Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure the quality of the deployment. Monitors that team follows agreed QA practices.
- DevOps master (Violeta Ramos): Responsible for the (development and production) infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparing the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.
- Developer (All members of the team): Development tasks which can be tracked by monitoring the pull requests/commits in the team repository.

1.2 Backlog grooming and progress monitoring

For this project, the user story is the unit for tracking and acceptance. For that, we define together the user stories missing on every weekly meeting, add them to the backlog, decide the story points necessary to complete each one and choose in what iteration they will be Concluído. Then, during that iteration, any person is free to assign themselves to an user story present on the ”A fazer” column of the backlog.

Git branches are created from sub-tasks of stories. After completing a sub-task, the developer must create a Pull Request to the main branche. After finishing all subtasks, the developer must mark the story as “Concluido”.

On our Jira board, we have the columns ”A fazer”, ”Em progresso”, ”Em revisão” and ”Concluído”. For moving the user stories between them, we defined a set of Jira automation rules:

- ”Em progresso” →”Em revisão”: When a Pull Request (on the story branch) is created
- ”Em revisão” →”Concluído”: When Pull Request is merged

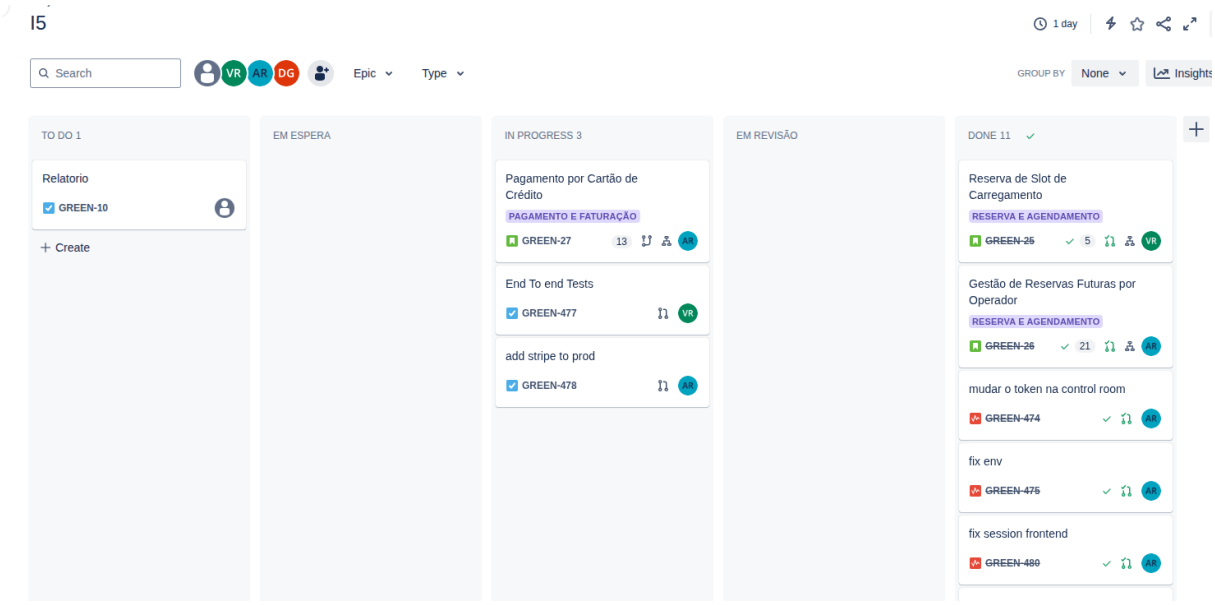


Figure 1: Backlog During The Iteration 5

Figure 2: Burnup Report At The End Of Iteration 4

2 Code quality management

2.1 Team policy for the use of generative AI

The team acknowledges generative AI as a productivity tool but establishes clear usage boundaries:

- **Permitted:**
  - Boilerplate/repetitive code generation (e.g., getters/setters)
  - Documentation drafting
  - Test case scaffolding
  - Syntax/refactoring suggestions
- **Prohibited:**
  - Core business logic implementation
  - Security/auth-related code
  - Architectural decisions
  - Test oracle determination (expected outcomes)
- **Mandatory practices:**
  - All AI-generated code requires human review and validation
  - Developers maintain full ownership and understanding of all committed code

2.2 Guidelines for contributors (coding style)

The code style for this project strictly adheres to the Google Java Style Guide as the foundation, supplemented by the consistency principle: new contributions should match the style of existing surrounding code to maintain readability and ease of maintenance.

Key enforced conventions include:

- **2-space indentation** for all blocks
- **Column limit** of 100 characters
- **K&R brace style** with mandatory braces (even for single-line statements)

- **One variable per declaration**
- **CamelCase naming** (PascalCase for types, camelCase for methods/variables)
- **Vertical whitespace rules** (single blank line between members)
- **Overload grouping** (consecutive placement without intervening code)
- **Annotation placement** above language keywords
- **Exception handling** (never catch generic `Exception`, never ignore exceptions)
- **Test method naming** (`test`*Feature_State* pattern)

All contributions must pass automated style verification through our GitHub Actions workflow, which enforces style compliance using the official `google-java-format` tool under the hood. The workflow configuration is:

```yaml
name: Lint Java code
on:
  push:
  pull_request:
jobs:
  formatting:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: axel-op/googlejavaformat-action@v4
        with:
          args: "--replace"
```

All contributions must pass automated style verification using the official `google-java-format` tool. Additionally, SonarQube static analysis is integrated into the review process to address code quality issues and eliminate bad smells.

As for TypeScript, we follow the ESLint recommendations. For that, each one of us have the linter extension on our IDE, which provides warnings when some bad practice is detected. The rule is to always fix the issue, with the exception of fixes that would break the code.

2.3 Code quality metrics and dashboards

For the static code analysis, we defined a workflow with Github Actions, that runs every time a pull request is created. This workflow runs all the tests and sends the new code implemented to be analyzed with SonarQube. Then, the developer consults the analysis, which can pass or not the quality gate. If it doesn't pass, the developer must refactor the code until it complies. If it does, the developer must still check if there are any issues and fix the most critical. There are some exceptions to the previous rule, namely in the case of impossibility to resolve the issues.

The defined quality gate is the default provided by SonarQube. We decided to use the default as base because we believe it is already strict enough to ensure the coding best practices while being realistic. The conditions of the quality gate used can be seen on Figure 3.

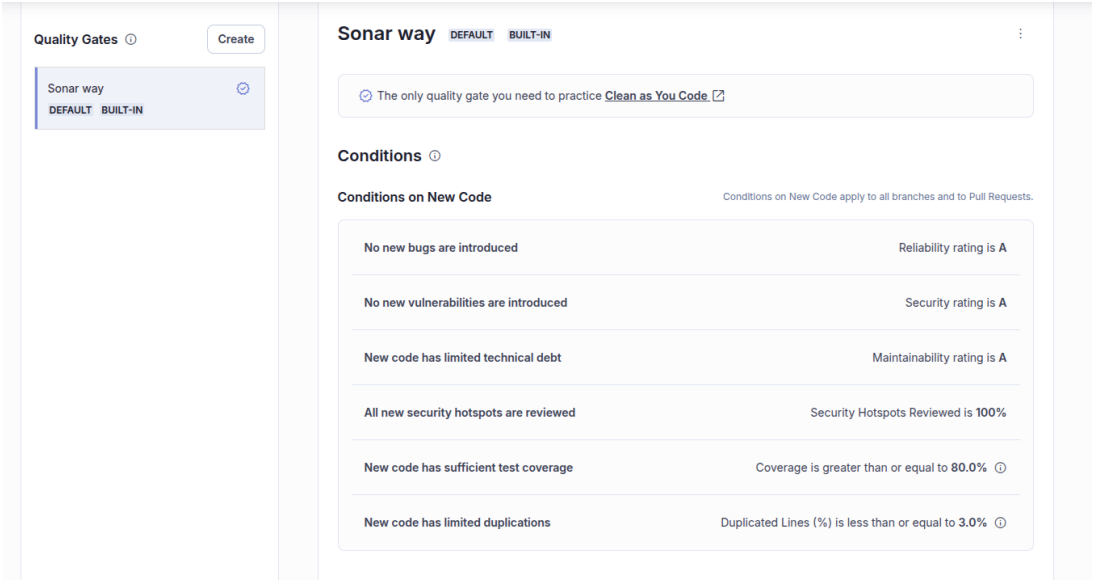


Figure 3: Sonar Quality Gate

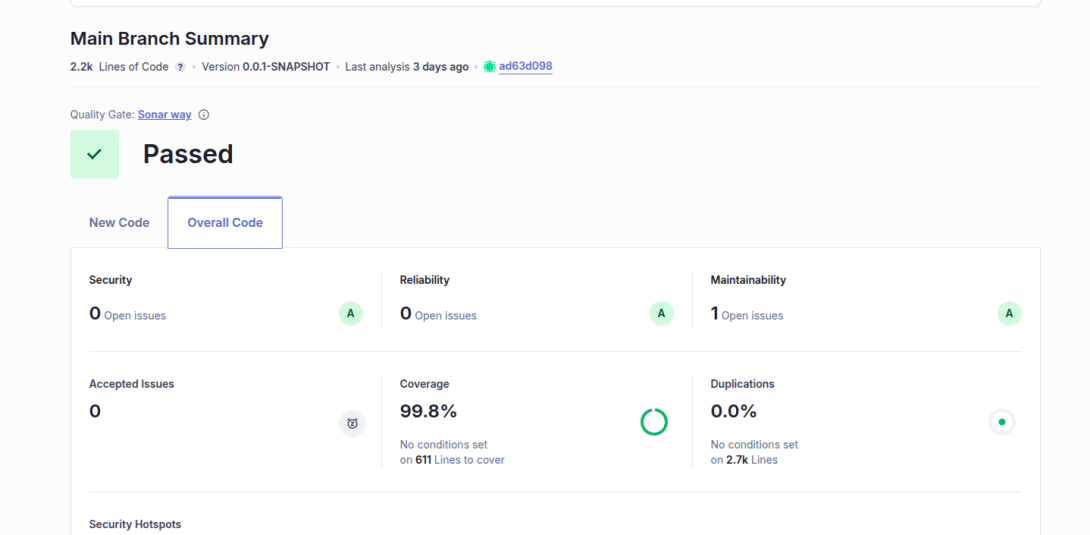


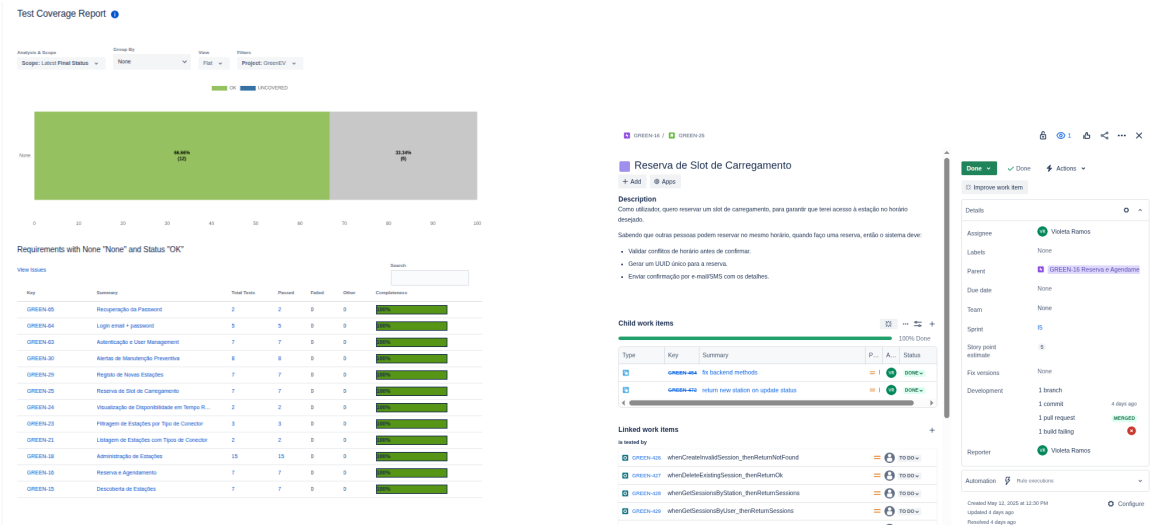
Figure 4: Sonar Dashboard on the Backend Repository

2.4 Xray

We used Xray to connect tests to their respective user stories and see their result in Jira, only for integration tests. This tool is very useful for checking whether a user story has passed its testing requirements, before merging it into the `main` branch(es). It allows for a feature-based, non-code-related vision of software testing, which isn't possible with testing tools like JUnit or Selenium.

Each time a new user story is created or new tests are implemented, we use the Xray user interface to connect themselves. This method was chosen over the one where user stories are connected through an annotation on each test, due to a more simplified and less cluttered organization.

Results are uploaded to Xray automatically from the JUnit reports, already linked to the respective user-storie, using CI/CD. Check section 3.2 for more information on that.



(a) Testing dashboard

(b) Testing board related to a user story

Figure 5: Xray dashboard and example of a user story

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

An organization was created, with each micro-service having its own repository, to improve organization, when compared to a giant monorepo. Besides those, a repository named `control-room` aggregates all these repos as Git submodules and contains the Docker Compose files (both development and production) to run the project.

The adopted workflow is the GitHub one, where the `main` branch contains all the development, instead of a `dev` branch, which is typical in a Git flow.

As mentioned in section 1.2, branches were created for user stories sub-tasks. Therefore those branches follow the naming convention `GREEN-[number]-[issue_name]`, so that the GitHub for Jira extension works and we can check branch / pull request status on Jira.

All repositories have branch protection rules in the `main` branch, so that the following is assured:

- A pull request is required before merging
- At least one person must review the pull request
- When new commits are pushed to a branch where a PR is open, all existing code reviews are dismissed
- SonarCloud's Quality Gate status must be *Passed*
- All review conversations must be resolved

Regarding the code review, although some descriptions and approval messages were written in a more informal way (with emojis, AI-generated poems, or even slangs), a thorough review was still made. This approach ensures that the code review process is both rigorous and engaging, allowing a collaborative environment without compromising on quality. The key rule was that nobody who committed in the branch could review its corresponding pull request, for a less biased code evaluation.

Also, all GitHub Actions workflows must successfully run (check section 3.2), and there was no merge until all those conditions were assured. This leads to a cleaner, less buggy code.

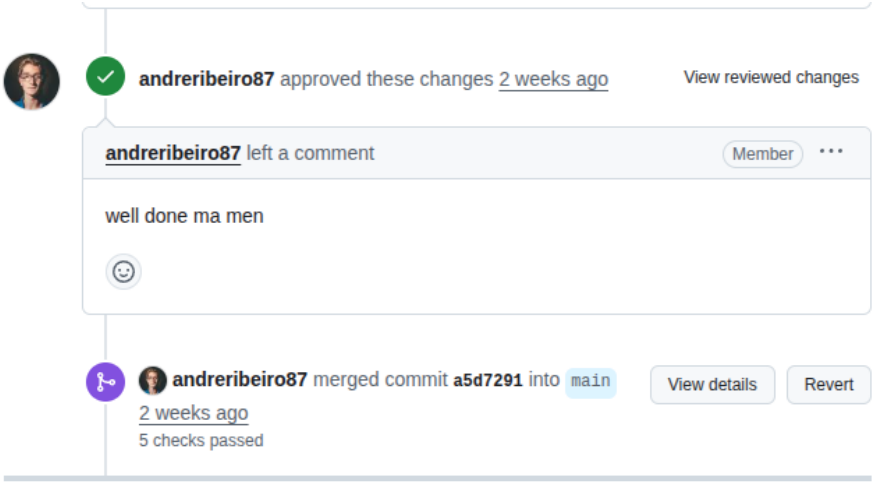


Figure 6: Pull Request Approved (After making the changes requested on Figure 7)

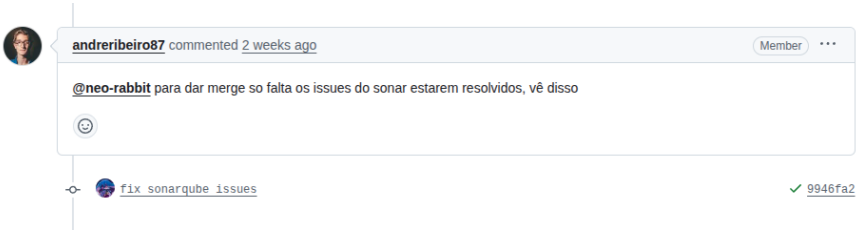


Figure 7: Review of a Pull Request in the Project (Asking for changes)

A user story is considered done when all implementation and tests are done, and when all story-related tests successfully run.

3.2 CI/CD pipeline and tools

Several CI/CD pipelines were built using GitHub Actions, each one adapted for the repository context. Besides that, SonarCloud was used for quality assurance in code.

These pipelines ensure better code quality, less time in manual analysis, and less human bias regarding issues and potential bugs.

3.2.1 Continuous Integration (CI)

Since a GitHub organization was used instead of a monorepo (see section 3.1), it was easier to adapt CI workflows to their repository context.

On the backend repo, there are several workflows running each time a commit is pushed to the `main` branch, such as:

- Unit and integration **backend tests**, using JUnit
- **Frontend tests**, using Selenium and Cucumber
- Upload of results to **SonarCloud**, which defines a Quality Gate (check section 2.3)
- Upload of results to **Xray**, for associating tests to user stories (check section 2.4)

On the frontend repos, the workflows are a bit different:

- **SonarCloud analysis**, which defines a Quality Gate
- **Performance-wise & quality analysis**, using Lighthouse (check section 4.5)

```yaml
name: Tests & Sonar analysis

on:
  push:
    branches:
      - main
  pull_request:
    types: [ opened, synchronize, reopened ]
  workflow_dispatch:

jobs:
  build:
    name: Build, test and analyze
    runs-on: ubuntu-latest
    permissions: read-all
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Set up JDK
        uses: actions/setup-java@v4
```

```yaml
        with:
          java-version: 21
          distribution: "zulu"

      - name: Cache SonarCloud packages
        uses: actions/cache@v4
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar

      - name: Cache Maven packages
        uses: actions/cache@v4
        with:
          path: ~/.m2
          key: "${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}"
          restore-keys: ${{ runner.os }}-m2

      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: |
          mvn -B verify failsafe:integration-test sonar:sonar \
          -Dsonar.projectKey=GreenEV-TQS24-25_backend

      - name: Publish results to Xray
        uses: mikepenz/xray-action@v3
        with:
          username: ${{ secrets.XRAY_CLIENT_ID }}
          password: ${{ secrets.XRAY_CLIENT_SECRET }}
          testFormat: "junit"
          testPaths: "**/TEST-*.xml"
          testExecKey: "GREEN-73"
```

```
58          projectKey: "GREEN"
```

Actions workflow for running backend tests and upload them to SonarCloud and Xray

Besides the existing CI on the `main` branch, there are workflows running on Pull Requests as well. In most cases, the running workflows are the same as the ones on the `main` branch, which have both triggers. This allows checking if the new code will bring any trouble to already existing code quality.

After tests are run, SonarCloud and Lighthouse (the latter only in frontend repositories) comment on the Pull Request with the result: SonarCloud will output its Quality Gate, whereas Lighthouse will provide metrics regarding performance and quality for the associated pages.

3.2.2 Continuous Deployment (CD)

Each time a push is made to the `main` branch of the backend or frontend repositories, a workflow runs with the following steps:

- **Build and push a Docker image to `ghcr.io`**: this step takes each repository's production Dockerfile, and builds an updated version of the respective image, based on the latest updates in the `main` branch. The output package is then pushed to the GitHub Container Registry, so that it can be used on the deployment itself.
- **Commit the submodule changes to the `control-room` repository**: One of the biggest drawbacks of using Git submodules is that each change in the default branch of a submodule requires a commit to update the submodule in the parent repository. We're using GitHub Actions to do it automatically, saving time and effort, as well as helping to trigger the deployment process, mentioned below.

When a push is made to the `main` branch of the `control-room`, a workflow that runs on the deployment machine (i.e., the deti virtual machine) pulls the latest Docker images from `ghcr.io` and then recreates the containers with the updated images. This ensures that the deployment is automatic - requiring no human intervention - and that the deployed version is always packed with the latest features, improvements and bug fixes.

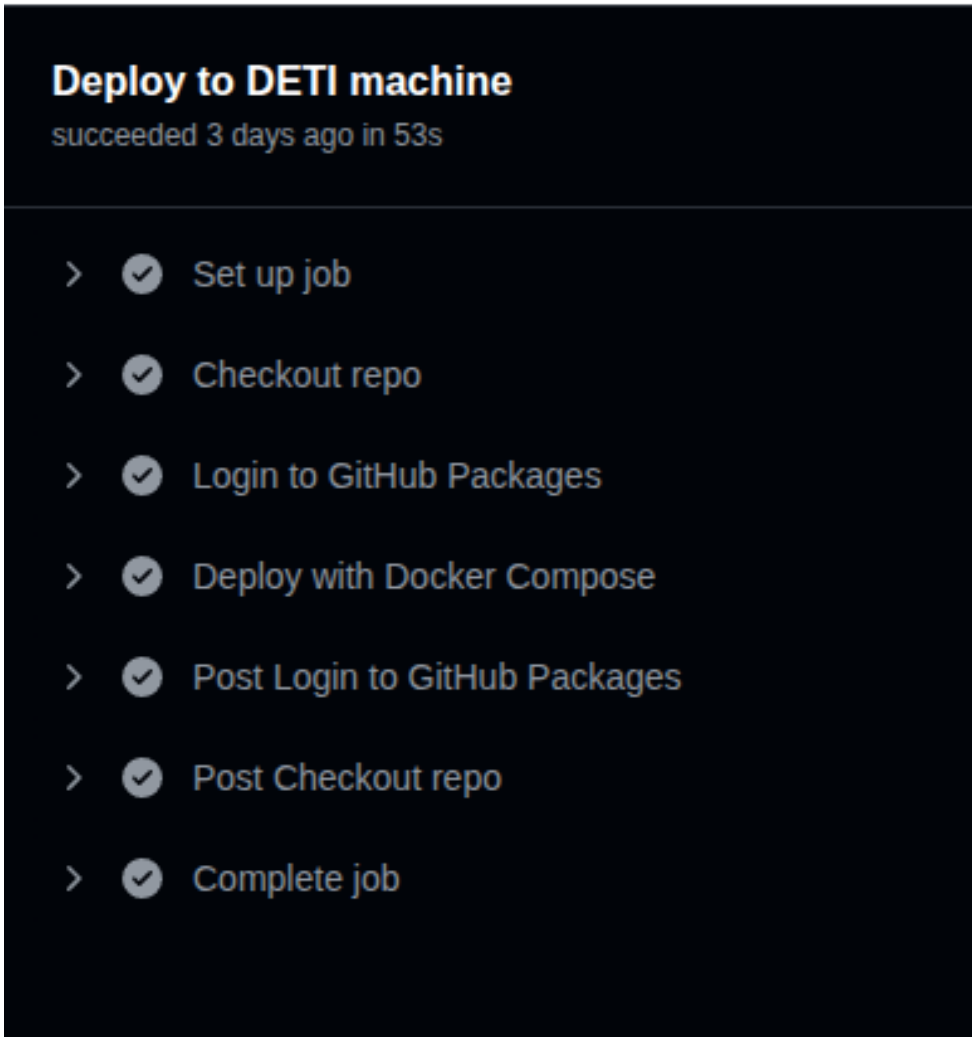


Figure 8: Deployment workflow run in Actions

3.3 System observability

Docker health checks are implemented for both the database and the Spring Boot backend, the latter using Spring Actuator endpoints. This allows for checking whether essential containers are working correctly, them having an unhealthy state when something goes wrong. Frontend containers don't have health checks, since there is no easy way to check if a React-based server is working properly.

Additionally, some logging was implemented, persisted in file.

4 Software testing

4.1 Overall strategy for testing

The overall strategy adopted for implementing the tests tried to follow a Test-Driven Development (TDD) approach, at least at the beginning, where the API definition was preceded by the elaboration of necessary unit and integration tests. Initially, tests and how they would be written were planned. Then, application development began, with any failing tests prompting necessary changes to the application code until the test passed smoothly. This approach allowed for maintaining clean code and ensuring that system features were working correctly from the start of development.

Additionally, a Behavior-Driven Development (BDD) strategy with Selenium WebDriver was adopted, using the Cucumber tool, to perform functional tests. This allowed for controlling and verifying expected system behaviors from the users' perspective.

For integration tests, REST-Assured was used, which is a widely used library for testing RESTful services, making it a suitable choice for testing the interaction between different system components, such as APIs and web services.

Moreover, Lighthouse, being an open-source automated tool, was used to perform performance tests. Lighthouse helps in improving performance, quality, and correctness related to web applications. With this, performance, accessibility, best practices and SEO were evaluated, ensuring quality.

By using different tools for different types of tests, we were able to effectively address various aspects of our system, aiming to maintain clean, quality, and efficient code throughout the development cycle.

4.2 Functional testing/acceptance

To conduct the functional tests, as mentioned in section 4.1, we adopted the BDD strategy using Selenium WebDriver in conjunction with Cucumber. We implemented functional tests for the main stories, carefully crafting scenarios to cover different usage situations and possible user interactions with the app.

The policy followed in these tests was the "black box", as they were written without directly accessing the system's internal logic and the focus was on the system's input and output, without

concern for internal implementation. All these tests were conducted through interaction with the user interface.

Furthermore, part of the approach is also based on the user perspective, as the tests are conducted in the same way a user would interact with the system, navigating through the interface and filling in fields.

4.3 Unit tests

Unit testing is critical to our software development process, ensuring individual components function as expected. In this types of tests, we assess smaller functional units of code. By this, when a test fails, we can quickly isolate the area where of the code that has the bug or the error. Our policy emphasizes high coverage metrics, such as statement and branch coverage, to enhance code quality and reliability. We write granular, isolated tests using frameworks JUnit and Mockito, integrating them into our CI/CD pipeline for automatic execution with every build.

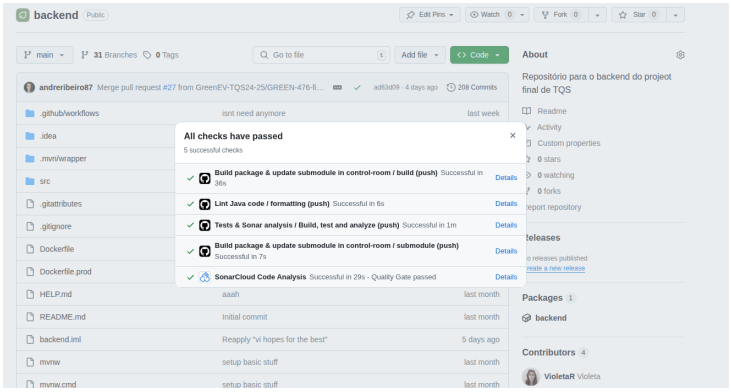


Figure 9: CI/CD pipeline - automatic execution

Tests are self-documented, peer-reviewed, and must be included in pull requests, with user stories considered complete only when all related tests are Concluído. We track and monitor test coverage and quality metrics using tools like SonarQube and Jira. The image bellow connects to the CI/CD topic presented before.

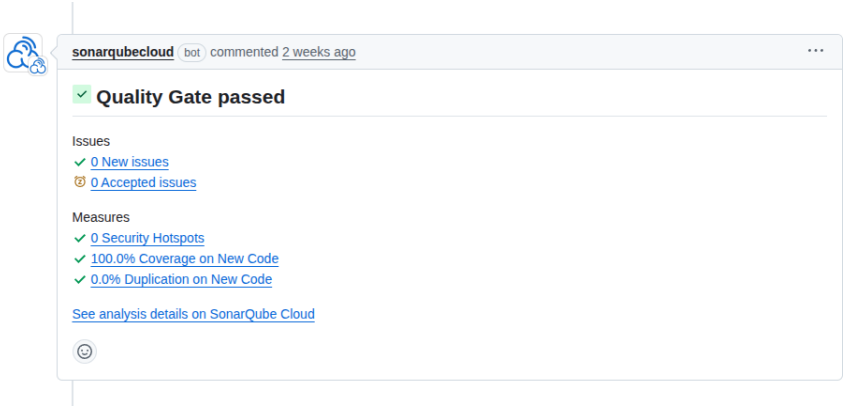


Figure 10: Sonarcould after Unit testing PR

4.4 System and integration testing

Integration testing verifies that different modules or services in the application work together as expected. The implementation strategy was to use SpringBootTest with Test Containers and RestAssured for coverage and an effective approach. It applies SpringBootTest by loading the application context during tests, therefore allowing testing in an environment similar to production. Test Containers creates and manages temporary Docker containers during a test, providing isolated and controlled environments for testing interactions with databases and external services. In addition, RestAssured was used to send HTTP requests to the application's API in the course of tests, ensuring the validation of the integration between all the different parts of the system. These combined tools enabled efficient and reliable execution of integration tests.

These tests are also automated in the CI/CD pipeline and include end-to-end scenarios to validate real-world usage.

For the backend tests, we did a integration test for testing the backend application by controlling the environment - PostgreSQL in a docker container).

```
1 @TestConfiguration(proxyBeanMethods = false)
2 public class TestBackendApplication {
3     public static void main(String[] args) {
4         SpringApplication.from(BackendApplication::main)
5         .with(TestBackendApplication.class).run(args);
6     }
7     @Bean
```

```
8       @ServiceConnection
9       PostgreSQLContainer<?> postgresContainer() {
10          return new PostgreSQLContainer<>(DockerImageName.parse("postgres:17"));
11      }
12  }
13
```

Container test

4.5 Performance testing

For backend API performance testing, we used k6 - an open-source load testing tool that simulates realistic traffic patterns and measures system behavior under stress. Our k6 test replicates EV platform usage through four core workflows: public station lookups (40% of requests), vehicle management (30%), session bookings (20%), and payments (10%), with automated validation of response time and error rate thresholds.

For performance testing, we used Lighthouse, which is an open-source, automated tool for improving the quality of web pages. It has audits for performance, accessibility, progressive web apps, SEO, etc. The figure below illustrates our Lighthouse performance testing automation setup, showcasing the integration of Lighthouse analysis into our development pipeline.

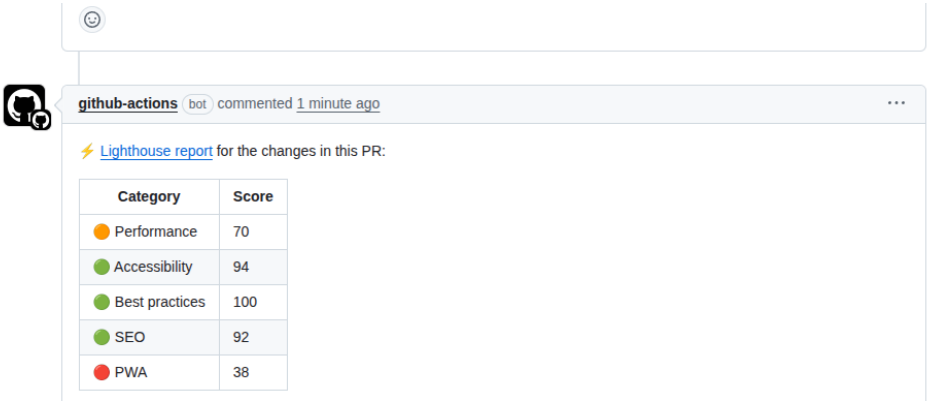


Figure 11: Lighthouse performance testing automation