TQS: Quality Assurance manual

Afonso Ferreira [113480], Ricardo Antunes [115243], Tomás Brás [112665]
v2025-04-25

1 Project management

1.1 Assigned roles

Role	Assignee
Team Leader	Afonso Ferreira
Product Owner	Tomás Brás
QA Engineer	Ricardo Antunes
DevOps Master	Afonso Ferreira

1.2 Backlog grooming and progress monitoring

We will have weekly meetings, following the Iterations plan, in order to officially organize the backlog with tasks that are relevant to the next iteration. New features should be planned in this period (ex: implementing a user story), while tasks and issues are to be spontaneously created once we find them (ex: a bug is found in the website that needs fixing).

We will try to complete user stories by priority so that the most important features are implemented in time for the MVP.

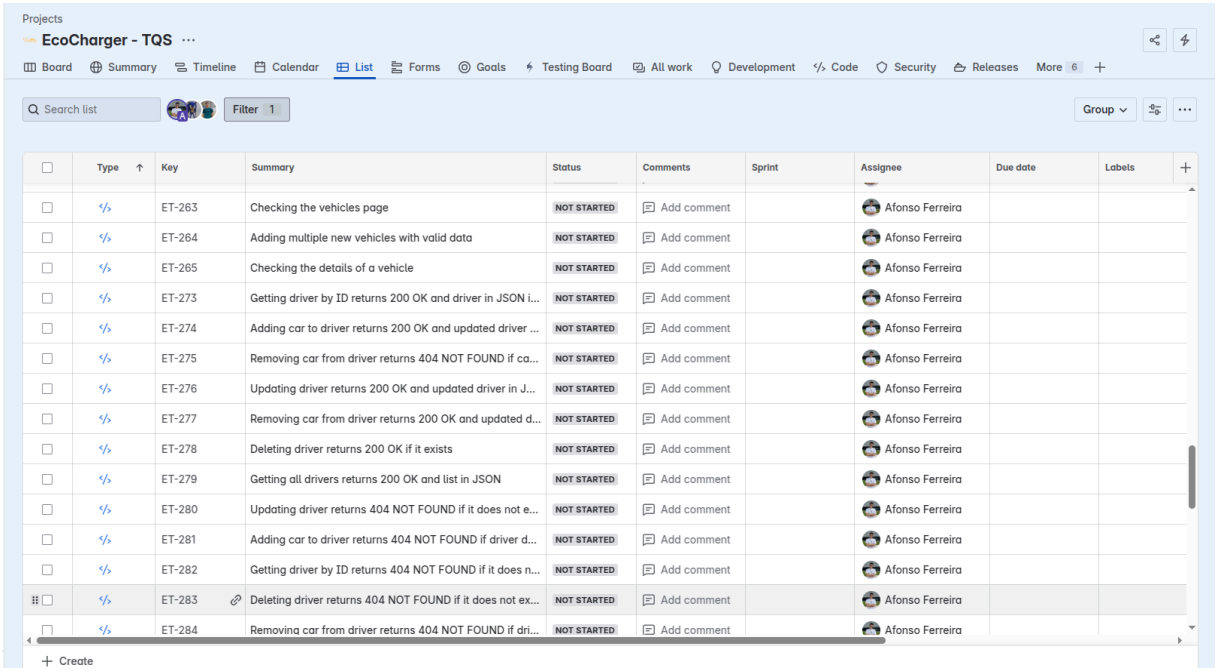
We will use story points in order to estimate the effort required in order to fulfill the task. The following scale will be implemented to issue story points:

- 0-2 hours - 1 story point
- 2-4 hours - 2 story points
- 4-6 hours - 3 story points
- 6-8 hours - 4 story points
- 8+ hours - 5 story points

Thanks to X-ray, our backlog contains a log of all tests run. It provides a full log of tests run when pull requests to the develop or master branches are made, thanks to a workflow run that sends a generated report to Xray.

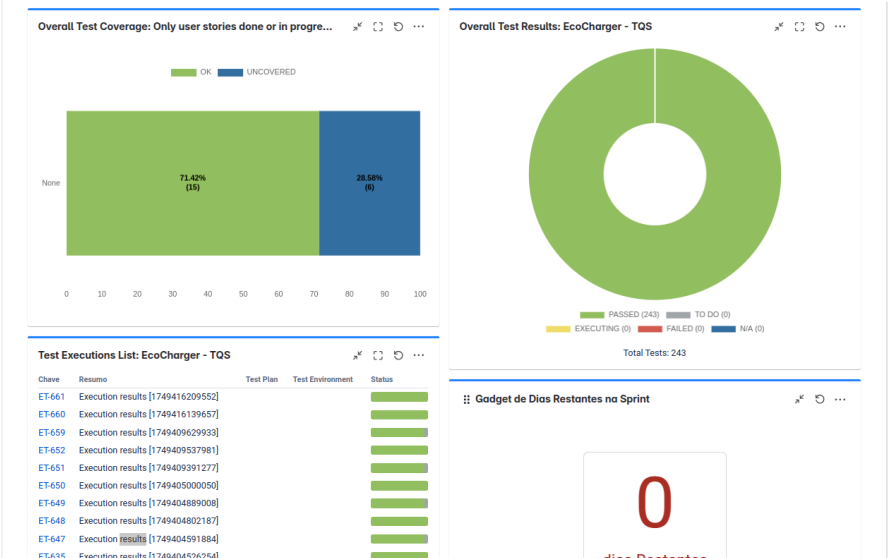
All of these tests are linked to the respective user stories automatically (using @Requirement) except for the functional tests which are linked manually.

Example of these tests:



To have an overall track of the project, we configured a custom dashboard combining test coverage, execution results, and sprint timeline indicators. The Test Coverage Report visualizes which user stories (specifically those marked as Done or In Progress) are currently covered by tests, helping us identify any gaps. The Overall Test Results widget provides a real-time breakdown of the entire test suite's outcomes, showing the number of tests passed, failed, in execution, or not yet run. We've also included a **Test Execution List**, which offers quick access to recent test runs, along with their status bars for immediate insight. Finally, the **Sprint Countdown Gadget** informs the team how many days are left in the current sprint, ensuring the developers are aware of the deadlines. This setup gives the team a centralized, real-time overview of project health and delivery progress.

Note: The 4 uncovered user stories are user stories related with the ability for third party users to use our public API (using Swagger), therefore they weren't tested since tests of other user stories already covered the issues the endpoints could have.



2 Code quality management

2.1 Team policy for the use of generative AI

The use of generative AI is allowed for this project. However, there are some restrictions in place:

- You must not use generative AI to describe your work in pull requests or in any other areas. Not understanding your own code is a recipe for disaster
- If you used generative AI to generate code, you must review it and signal the AI-generated code with a comment, so that others can pay special attention to it when reviewing the code
- You must delete all unnecessary comments placed by the generative AI, for example on CSS files

Fulfilling these guidelines transparently will allow for less rejected pull requests and more and better work being done.

2.2 Guidelines for contributors

Coding style
Java code (Backend) -> Google Styling Guide

This Coding Style will be enforced by Github Actions, which automatically will change the code to match the guidelines Google showcases. Some notable guidelines that are not covered by the automatic formatter:

- For a source file containing classes, the file name consists of the case-sensitive name of the top-level class (of which there is exactly one), plus the `.java` extension.
- **Wildcard imports**, static or otherwise, **are not used**.
- Class names are written in UpperCamelCase. Method names are written in lowerCamelCase. Constant names use `UPPER_SNAKE_CASE`: all uppercase letters, with each word separated from the next by a single underscore.
- At the *minimum*, Javadoc is present for every *visible* class, member, or record component, with a few exceptions noted below. A top-level class is visible if it is `public`; a member is visible if it is `public` or `protected` and its containing class is visible; and a record component is visible if its containing record is visible.

React JSX Code (Frontend) -> AirBnB Styling Guide

This Coding Style will be enforced by Github Actions, which will fail and flag errors if the code is not in accordance with the guidelines AirBnB suggests. Some guidelines that are not covered by the checker:

- Use PascalCase for filenames. E.g., ReservationCard.jsx
- Use the filename as the component name.
- Do not use displayName for naming components. Instead, name the component by reference.

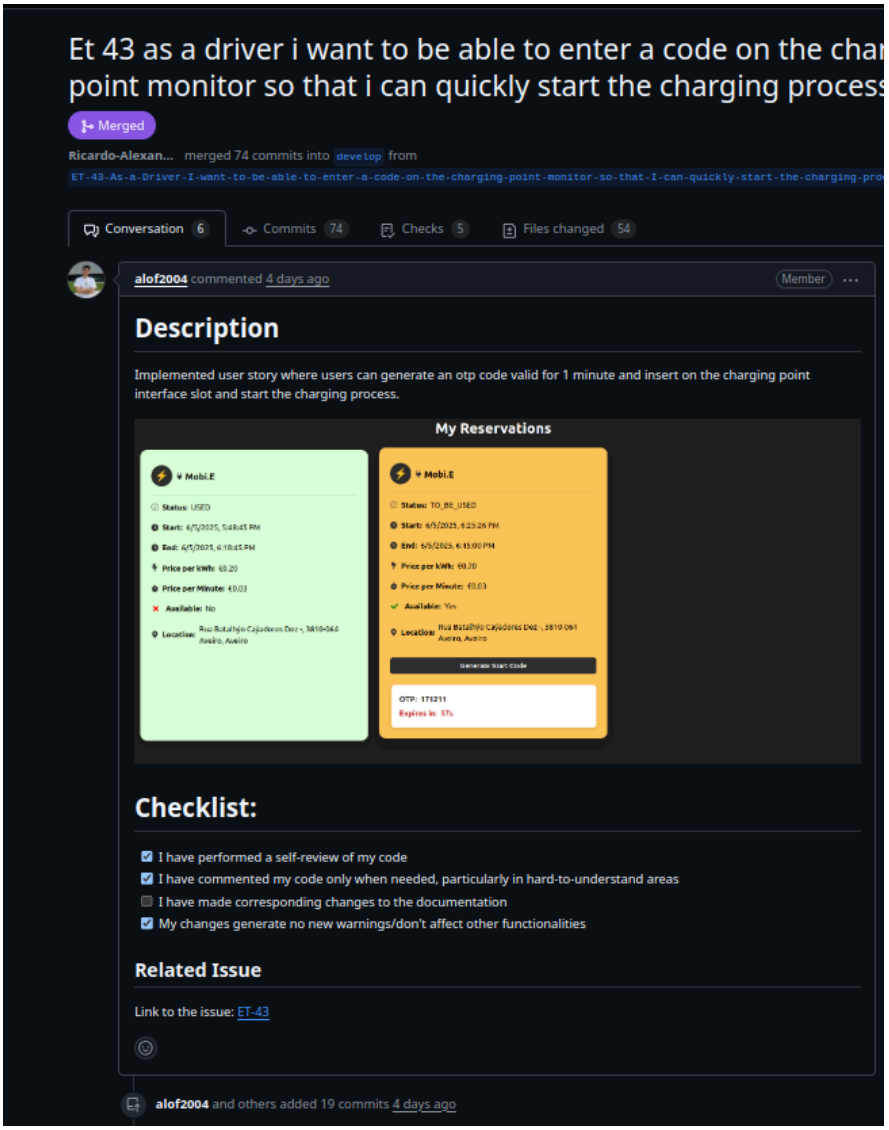
- Always use camelCase for prop names, or PascalCase if the prop value is a React component.
- Always define explicit defaultProps for all non-required props.
- Do not use underscore prefix for internal methods of a React component.

Code reviewing

Code will be reviewed upon Pull Requests, using the Pull Request template we have established that requires describing the changes made, why they were made and proof of testing. This ensures that whoever is reviewing the Pull Request is aware of all of the context behind it and the related user story. We will also have Copilot review every pull request as a means of allowing us to catch bad code practices that otherwise may not be caught (especially when PR's contain a lot of code).

We enforce conversation resolution and if someone's code is changed, they must approve this change. This ensures that the pusher is aware of feedback, and that everyone that created code can comment on changes made to it, making sure that code mishaps are not missed.

In order for a PR to be accepted, then the frontend and backend code styling must be followed, as well as minimum coverage on tests that all must pass. Below is an example of a PR that was merged after all conversations were resolved

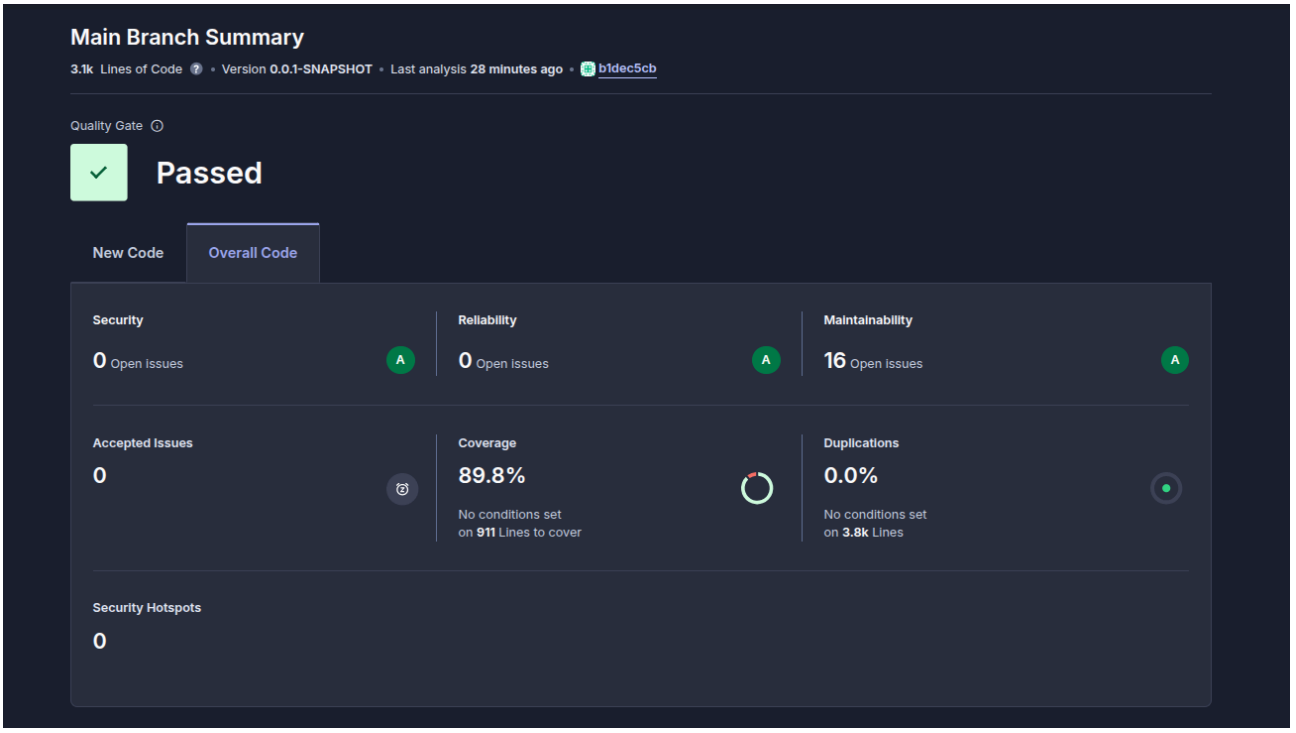


2.3 Code quality metrics and dashboards

We will use SonarCloud to help monitorize code smells and test coverage. We have enforced the following metrics:

- Code Coverage: >= 80% - we enforce this as a means to ensure that all main features are covered in tests; we do not want useless tests to be created just to get coverage
- Code Duplication: <= 3% - sometimes it is useful to copy a tiny bit of code so we do offer some leeway on this metric, but the norm should be to follow DRY principles and whenever possible place duplicated code on a function to reuse that code.

Additionally, technical debt should not increase by more than 1 hour with each new PR, and no PR can follow through with Blocker or High level Code Smells. This ensures code quality has at least *an acceptable* level of standart, without forcing developers to make the code “perfect” since our available time is not high.



3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

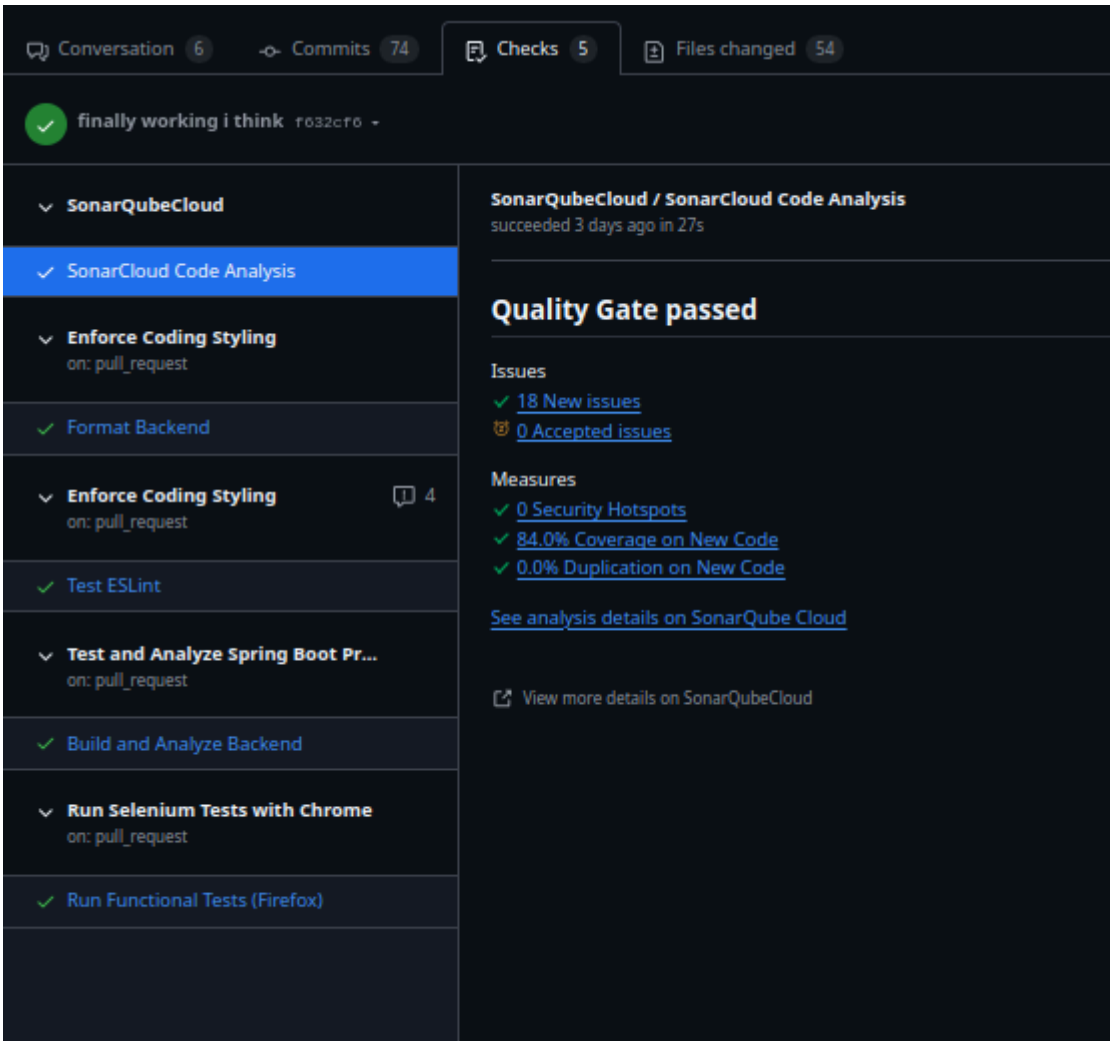
Coding workflow

The workflow adopted for our project was the gitflow workflow where users create a specific branch that originated from the ‘develop’ branch to implement a specific user story. After they complete the user story a pull request should be made into the develop branch which will be auto reviewed by Copilot and will need an approving review from someone other than the last pusher.

This reviewer should carefully review the code and request any necessary changes to the code. The **PR creator is responsible for merging the PR** after all comments and required changes have been addressed. This ensures that they are aware of all feedback and conversations before the code is merged. In the end of each sprint where code has been developed we will do a release to the master branch with fully working code.

Definition of done

A user story is considered done when the branch that corresponds to the respective user story has been completed is merged into **'develop'** branch. For the corresponding PR to be accepted by peers we should guarantee that the code being pushed has at least 80% of coverage on new code. All functional tests covering key aspects of the user story and all aspects of the acceptance criteria. Besides the functional tests, all automated tests (unit, integration, functional) should also be implemented and connected to the respective user story on JIRA. All of the tests must pass, as well as the frontend and backend coding style metrics must be followed in order for the merge to be possible. Below is an example of a User Story where all checks passed.



3.2 CI/CD pipeline and tools

For the CI pipeline, we divided it into four different workflows. This way, we separate different responsibilities within the development lifecycle, allowing each workflow to focus on a specific task. This modular approach improves maintainability, speeds up feedback, and isolates failures, making debugging easier.

- **frontend-styling.yml** handles linting for the frontend using ESLint, ensuring code quality and consistency in JavaScript/TypeScript files.

```yaml
name: Enforce Coding Styling

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

jobs:
  test-eslint:
    name: Test ESLint
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18'

      - name: Install dependencies
        working-directory: ./frontend
        run: npm install

      - name: Run ESLint
        working-directory: ./frontend
        run: npm run lint
```

- **styling.yml** is responsible for automatically formatting the backend Java code using Google Java Format.

```yaml
name: Enforce Coding Styling

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

jobs:
  format-backend:
    name: Format Backend
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Run Google Java Format
        uses: axel-op/googlejavaformat-action@v4
        with:
          args: "--skip-sorting-imports --replace"
          skip-commit: true

      - name: Show git diff
        run: git --no-pager diff

      - name: Commit and push changes if needed
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: |
          git config user.name "GitHub Actions"
          git config user.email "actions@github.com"

          branch_name="${GITHUB_HEAD_REF:-${GITHUB_REF#refs/heads/}}"

          git add .
          if ! git diff --cached --quiet; then
            git commit -m "Apply Google Java Format"
            git push origin HEAD:$branch_name
          else
            echo "No formatting changes to commit."
          fi

```

- **functional-tests.yml** runs Selenium-based end-to-end tests in a containerized environment with Firefox, simulating real user interactions.

```yaml
name: Run Selenium Tests with Chrome

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

jobs:
  selenium-firefox-tests:
    name: Run Functional Tests (Firefox)
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: 17
          distribution: 'zulu'

      - name: Write .env file
        run: |
          cat <<EOF > .env
          SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/ecocharger
          SPRING_DATASOURCE_USERNAME=ecocharger
          SPRING_DATASOURCE_PASSWORD=ecocharger
          EOF

      - name: Write test application.properties
        run: |
          mkdir -p backend/src/test/resources
```

- **build.yml** focuses on unit testing, code coverage with Jacoco, static analysis using SonarQube, and reporting results to Jira Xray.

```yaml
name: Test and Analyze Spring Boot Project (Unit Only)

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


jobs:
  build-and-analyze:
    name: Build and Analyze Backend
    runs-on: ubuntu-latest
    services:
      db:
        image: postgres:15
        env:
          POSTGRES_DB: ecocharger
          POSTGRES_USER: ecocharger
          POSTGRES_PASSWORD: ecocharger
        ports:
          - 5432:5432
        options: >-
          --health-cmd "pg_isready -U ecocharger"
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: 17
          distribution: 'zulu'

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

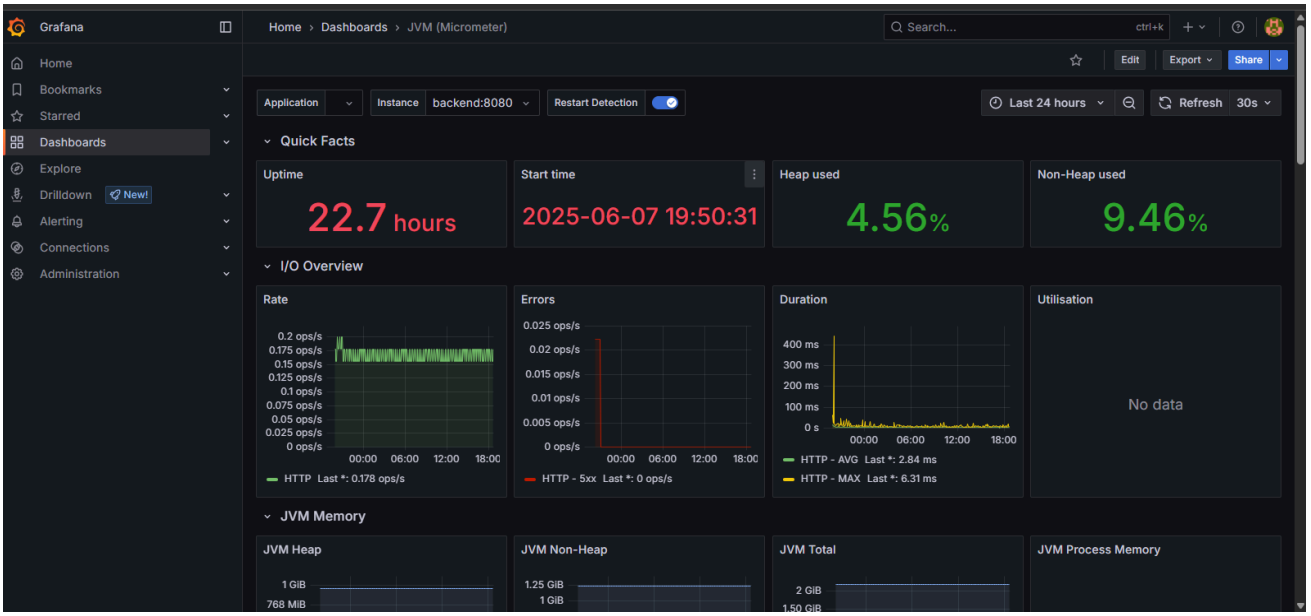
For the **CD pipeline**, we set up an automated workflow that deploys the application to the IEETA virtual machine whenever changes are pushed to the master branch. The process includes checking out the latest code, setting up environment and configuration files, and then rebuilding and restarting the Docker containers. Before each deployment, it stops any running containers and clears the Docker cache to ensure a clean setup. Secrets like Stripe keys and JWT tokens are securely injected from GitHub. This setup ensures deployments are fast, consistent, and require no manual steps.

If we had more time, we would've implemented a **staging environment** to test deployments before pushing to production. This would allow us to catch potential issues in a controlled setting, reducing the risk of downtime or bugs affecting end users. A staging workflow would mirror the production setup but on different ports.

3.3 System observability

In order to get some Observability into our system, we use Grafana + Prometheus which allows us to get metrics about uptime and resource uses, as well as http requests on our system. At a later stage, as future work, we can use a tool such as Loki to capture logs, and stack traces, completing the trifecta needed to guarantee observability of the system.

Here is a sample screenshot:



4 Software testing

4.1 Overall testing strategy

As a team, considering the timeframe that we have to complete the project, and because requirements on features are not robust when implementing them, we opted not to follow TDD. As such, we first write the features, then write the tests. For the backend, there are 3 levels of tests - Controller tests, Service Tests and Integration Tests

Controller tests mock the service and test how the controller reacts to payload being directed to each endpoint, each test expecting a different result. Service tests on the other hand mock the

repositories and perform the same tests as the controllers, only on a different level. Integration tests combine both aspects while also testing the repository and models.

We will also implement BDD and write tests using Cucumber + Selenium to navigate the solution's frontend, verifying that the user stories that we implement are fulfilled.

4.2 Functional testing and ATDD

When we created each user story, we developed Acceptance Criteria which must be fulfilled upon implementing the story. The actual code implementation of the test is done after implementing the user story, as this way, we don't restrict ourselves when writing the code, particularly in the frontend. We can add additional acceptance criteria upon developing the user story, allowing us to develop code more fluidly in a short amount of time.

These tests are performed using a combination of Cucumber and Selenium, where we navigate the website following a scenario with a given background, attempting to fulfill the criteria defined in the cucumber feature. Each US is detailed in a different .feature, and steps specific to that user story are also translated to Selenium in a separate .java file.

Here is an example of a functional test:

```
Feature: Check and Manage Vehicles
    As a Driver,
    I want to register my car in the application
    So that I can use it at charging stations and receive accurate insights about energy usage, CO₂ savings, and cost efficiency.

    Background:
        Given I am on the login page
        When I enter "ricardo.antunes2002@gmail.com" into the "email" field
        And I enter "banana" into the "password" field
        And I click the "login-button"
        And I navigate to the vehicles page

    Scenario: Checking the vehicles page
        Then I should see a table of vehicles
        And the table should have 2 rows
        And the table should have the following columns:
            | ID | Name | Make | Actions |
        And the table should contain the following data:
            | ID | Name                | Make    |
            | 3  | Hyundai Kona Electric | Hyundai |
            | 4  | Renault Zoe           | Renault |
```

4.3 Unit tests

Aiming to test each component independently unit tests were written with Junit while mocking the underlying components the first component depends on for its functioning with Mockito. RestAssured with mockMvc was also used to validate the RestAPI and to promote easier code readability and maintainability These tests were written following a top-down approach: firstly we'll develop the tests with the controller mocking service tests, then the service with mock repository tests and ending with the repository tests.

Developers are expected to write unit tests for **every piece of new business logic** they implement. These tests should cover all architectural layers, but should especially focus on the **controller and service layers**, which typically contain the most critical logic and integration points.

These run as previously mentioned in our Build and Analysis CI pipeline which will run the different unit tests, generating a JaCoCO report, running a SonarCloud analysis and sending the results to the Xray Cloud.

4.4 System and integration testing

In our integration tests we followed the openbox principles, where we would perform the tests, knowing the code and how it works, using the tests results / failures in order to pinpoint where in the code the issue was. Since the goal of Integration tests is to see how the system behaves as a whole, mimicking its behavior in production, we used **@SpringBootTests** in order to load the application context.

We will use **REST Assured** to make requests to the controllers in a more human-readable way, and we will be mocking services that call external dependencies.

We created these tests using a combination of **@Testcontainers** + **Flyway,** mimicking the production DB while using a mocked / temporary one, already pre-populated with data so that it may assist us in our tests. These integration tests are integrated into the pipeline much like the other tests, being automatically assigned to their respective user stories using Xray and the **@Requirement** annotation

Here is an example of an integration test using an external dependency (Stripe)

```java
@Test
@DisplayName("Finalize Stripe payment with mock session_id")
void testFinalizeTopUpStripe() throws Exception {
  String json =
      """
      {
          "amount": 5.0,
          "simulateSuccess": false
      }
      """;

  var sessionId =
      RestAssuredMockMvc.given()
          .contentType("application/json")
          .body(json)
          .when()
          .post("/api/v1/driver/" + savedDriver.getId() + "/balance")
          .then()
          .statusCode(200)
          .extract()
          .jsonPath()
          .getString("sessionId");

  // Simulate success callback
  RestAssuredMockMvc.given()
      .when()
      .get("/api/v1/driver/checkout-success?session_id=" + sessionId)
      .then()
      .statusCode(200)
      .body("status", equalTo("success"));
}
```

4.5 Non-function and architecture attributes testing

Our performance tests should target the main features of our product that will be used the most frequently by users, and that contain complex logic that can take a while for the backend to realize. Load tests are essential to figuring out if a feature is efficient or not, especially if it relies on external dependencies.

In our project, we must have:

- Breakpoint Tests - to measure the limits of our system
- Stress Tests - to place our system under pressure on a short amount of time
- Soak Tests - to test the durability of our system

We will use K6 in order to run these tests, and once we implement core functionalities such as station booking and car charging. Below is an example of a test ran while we were developing the system.

```
  THRESHOLDS

    checks
    ✓ 'rate>0.95' rate=99.63%

    http_req_duration
    ✓ 'p(80)<2000' p(80)=undefined

    http_req_failed
    ✓ 'rate<0.05' rate=0.00%


  TOTAL RESULTS

    checks_total.....................: 41192  340.853872/s
    checks_succeeded.................: 99.63% 41042 out of 41192
    checks_failed....................: 0.36%  150 out of 41192

    ✓ login succeeded
    ✓ token received
    ✗ login time < 2000ms
      ↳  98% — ✓ 10148 / ✗ 150
    ✓ reservation created

    HTTP
    http_req_duration.....................................................: avg=726.85ms min=5.5ms med=574.31ms max=4.29s p(90)=1.64s p(95)=1.94s
      { expected_response:true }..........................................: avg=726.85ms min=5.5ms med=574.31ms max=4.29s p(90)=1.64s p(95)=1.94s
    http_req_failed.......................................................: 0.00%  0 out of 20596
    http_reqs.............................................................: 20596  170.426936/s

    EXECUTION
    iteration_duration....................................................: avg=2.46s    min=1.02s med=2.23s    max=7.03s p(90)=4.14s p(95)=4.57s
    iterations............................................................: 10298  85.213468/s
    vus...................................................................: 9      min=2          max=500
    vus_max...............................................................: 500    min=500        max=500

    NETWORK
    data_received.........................................................: 16 MB  131 kB/s
    data_sent.............................................................: 7.0 MB 58 kB/s
```