universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Hugo Ribeiro [113402], João Neto [113482], Rodrigo Abreu [113626], Eduardo Lopes [103070]
v2025-04-25

Contents

1 Project management

1.1 Assigned roles

Role	**Member**
Team Coordinator	Hugo Ribeiro
Product Owner	João Neto
QA Engineer	Rodrigo Abreu
DevOps Master	Eduardo Lopes
Developer	Everyone

1.2 Backlog grooming and progress monitoring

For the development of our project our group adopted the Agile methodology and decided to use Jira as the project managing tool. Our Team Coordinator is responsible for assigning tasks to each team member and defining the respective story points, priority and corresponding deadlines.
Our Product Owner defined the core user stories and respective acceptance criteria. We aim to implement the respective user stories based on their priority, so the tasks assigned are accordingly to this.

Story points have the following meaning:
- 1 story point: 0-2h of estimate work
- 2 story points: 2-4h of estimate work
- 3 story points: 4-6h of estimate work
- 4 story points: 6-8h of estimate work
- 5 story points: 8+h of estimate work

2 Code quality management

2.1 Team policy for the use of generative AI

Our team encourages the use of AI assistants, such as GitHub Copilot and ChatGPT, as valuable tools to improve productivity and speed up development. However, it's important to understand that AI is a tool, not a source of truth. It can produce convincing but incorrect, insecure, or inefficient code. The responsibility for ensuring correctness, security, and maintainability still lies entirely with the developer submitting the code. AI can assist with writing code, but it cannot take responsibility for its quality. That responsibility belongs to you.

To maintain the integrity of our codebase, no code, regardless of whether it was written manually or with AI assistance, should be pushed to the dev or main branches without review and approval by at least two developers. This ensures shared accountability, consistent standards, and a safeguard against introducing bugs or poor practices. If you're using AI to assist with your work, you must be able to fully understand and explain any code you include. If you don't understand what the AI produced, it shouldn't be in the codebase.

For newcomers, here are some practical guidelines to follow when working with AI tools:

Do:

- Use AI to explore solutions, patterns, or unfamiliar syntax before implementing your own.
- Let AI help you write boilerplate code or test scaffolding, then refine it to match our standards.
- Review all AI-generated code as if it were written by a junior developer—check it thoroughly.
- Use AI to help you understand code or clarify documentation, then validate what you learn.
- Lean on AI to improve readability, suggest better naming, or enhance comments and docs.

Don't:

- Copy and paste AI-generated code into production or test files without understanding how it works.
- Trust AI output blindly, treat it as a starting point, not a final solution.
- Skip writing tests or documentation because "AI wrote it."
- Use AI to make critical architectural or security decisions without consulting the team.
- Bypass the review pr
- Assume AI-generated code needs less scrutiny.

2.2 Guidelines for contributors

To ensure code consistency, readability and maintainability since the beginning, the following code guidelines were defined. For the Maven Spring Boot Backend (in Java) the group adopted the Google Java Style Guide. For the frontend in React JS we adopted the Airbnb React/JSX Style Guide guidelines.
A github-actions workflow was set up using Google Java Format, to automatically format the code to comply with the Google Java Style automatically and ensure the guidelines are met.

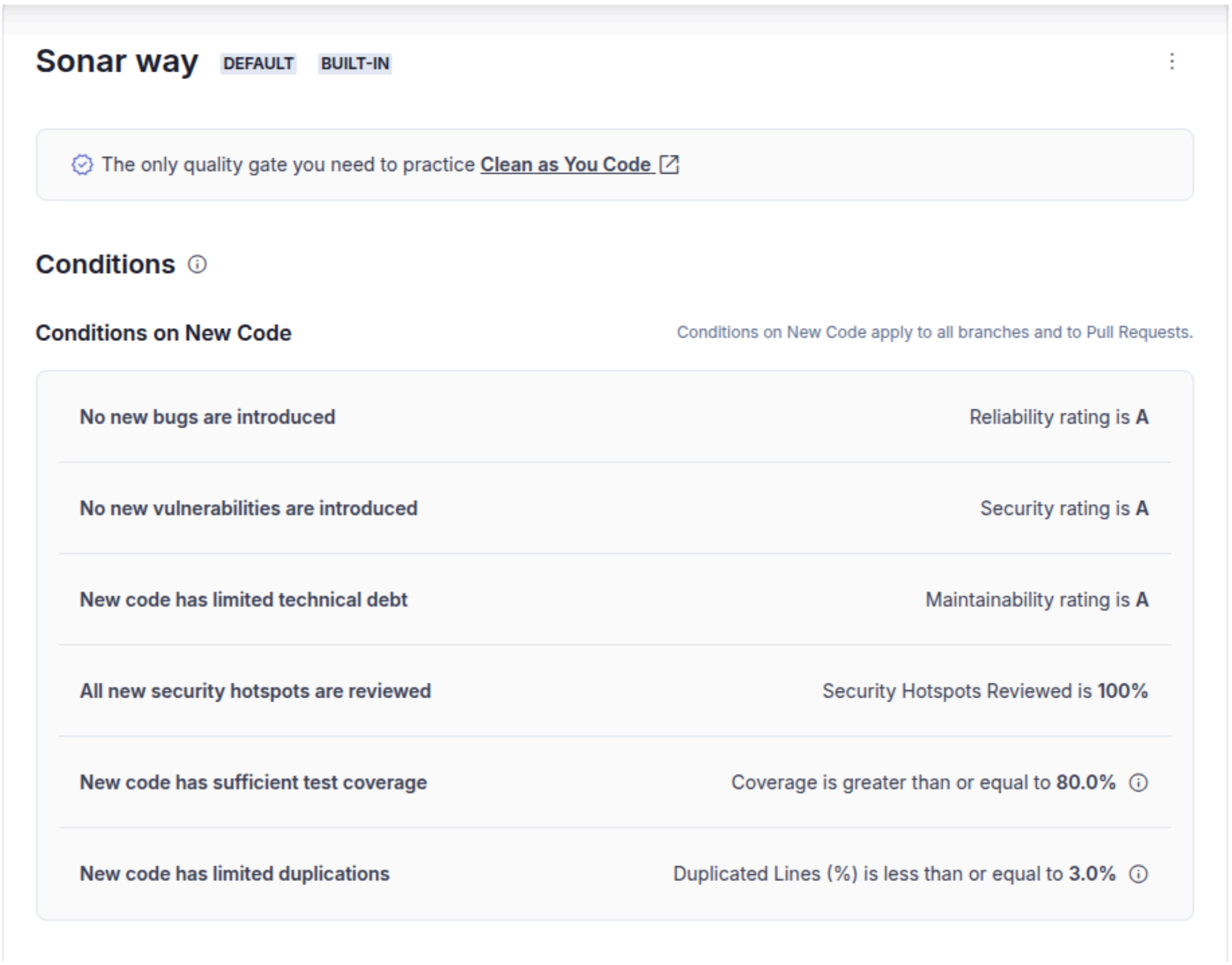
Code reviewing

Every pull request to main or dev branches should be reviewed by at least 2 other members, one of them must always be the product owner. No AI tools were implemented for code reviewing.

2.3 Code quality metrics and dashboards

For static code analysis SonarQube was integrated from the beginning to detect bad smells and other issues on the code. We also implemented a github-actions workflow that runs every time a commit is done to the main and dev branches and in every pull request.

We used the default quality gate named "Sonar Way" from the beginning and ensured the project met the objectives on each iteration.



Sonar way DEFAULT BUILT-IN

The only quality gate you need to practice Clean as You Code

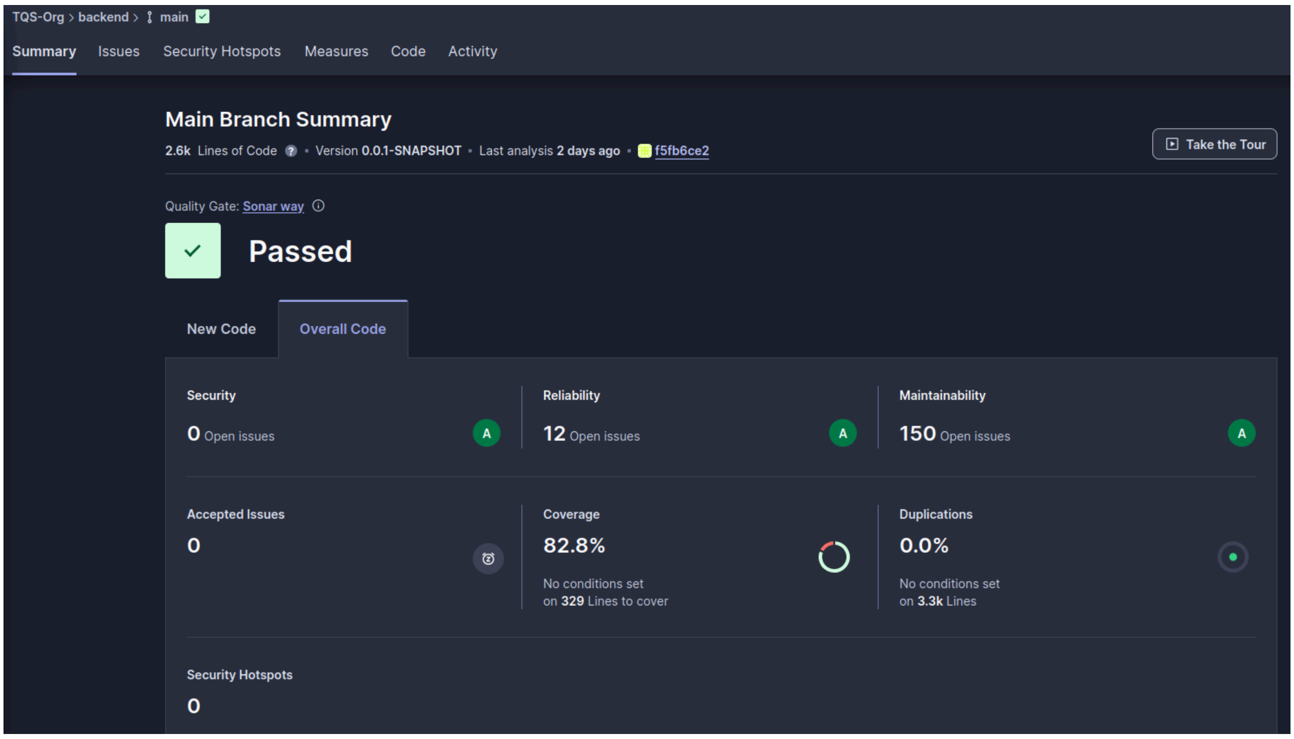
Conditions

Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

No new bugs are introduced	Reliability rating is **A**
No new vulnerabilities are introduced	Security rating is **A**
New code has limited technical debt	Maintainability rating is **A**
All new security hotspots are reviewed	Security Hotspots Reviewed is **100%**
New code has sufficient test coverage	Coverage is greater than or equal to **80.0%**
New code has limited duplications	Duplicated Lines (%) is less than or equal to **3.0%**

Final Quality Gate results:



Main Branch Summary

2.6k Lines of Code • Version 0.0.1-SNAPSHOT • Last analysis 2 days ago • f5fb6ce2

Quality Gate: Sonar way

Passed

New Code | Overall Code

Security: 0 Open issues (A)	Reliability: 12 Open issues (A)	Maintainability: 150 Open issues (A)
Accepted Issues: 0	Coverage: 82.8% No conditions set on 329 Lines to cover	Duplications: 0.0% No conditions set on 3.3k Lines
Security Hotspots: 0		

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

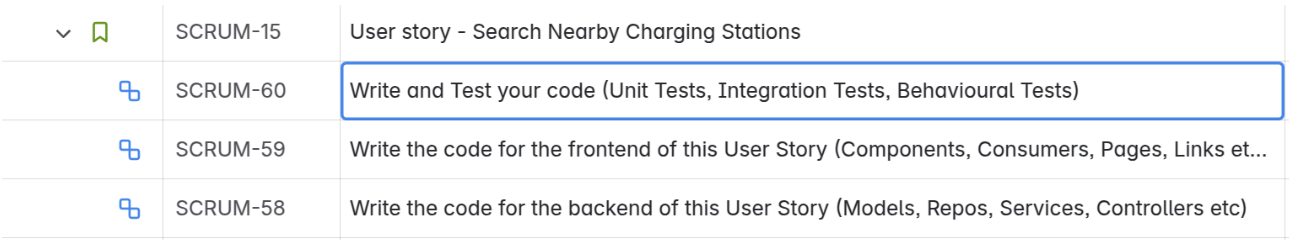
Coding workflow

We adopted the gitflow as our project workflow. Flow steps:

- At the start of each iteration the tasks are divided by each team member based on each role and priority.
- To effectively complete a task, a member must verify if the task definition of done is satisfied. Then creates a pull request that must be reviewed for at least 2 members to be validated. Each member must carefully analyse the content before accepting/rejecting the pull request.
- All pull requests must target the ‘dev’ branch. This branch, each iteration release, is merged into the main branch.
- When a pull request is accepted the respective issue is marked as done, and it's closed.

How to write code in our Project:

- You will be assigned an User Story for development.
- Each User Story will have 3 subtasks, you will see something like this:
- Create a new branch with base ‘dev’ named after the User Story you are implementing ex:’SCRUM-15 Search Nearby Charging Stations’
- Develop the necessary code for the backend of this User Story.
- Develop the necessary code for the frontend of this User Story.
- Merge the testing branch into the User Story branch.
- Finally, request a review by making a PR into ‘dev’



	SCRUM-15	User story - Search Nearby Charging Stations
	SCRUM-60	Write and Test your code (Unit Tests, Integration Tests, Behavioural Tests)
	SCRUM-59	Write the code for the frontend of this User Story (Components, Consumers, Pages, Links et...
	SCRUM-58	Write the code for the backend of this User Story (Models, Repos, Services, Controllers etc)

Definition of done

We consider an User Story to be completed when the branch regarding it is merged into dev.

3.2 CI/CD pipeline and tools

For our project, we implemented a robust and efficient Continuous Integration (CI) and Continuous Delivery (CD) pipeline leveraging GitHub Actions. This automation-first approach ensures rigorous validation, comprehensive testing, and seamless deployment of code changes, significantly enhancing code quality, shortening feedback loops, and streamlining release cycles.

Our CI process is tightly integrated with GitHub Actions and is automatically triggered on push events to the main branch as well as on pull requests (PRs). Additionally, developers can initiate workflows manually through the GitHub Actions interface using the workflow_dispatch event, allowing for flexible testing on demand.
The CI system is composed of several modular workflows, each designed to handle distinct stages of the development lifecycle:

- **Backend Build and Quality Assurance**: A core workflow named CI with Maven and SonarCloud compiles the Java backend using Maven, executes both unit and integration tests, performs static code analysis via SonarCloud, and gathers code coverage metrics using JaCoCo. These metrics are uploaded to SonarCloud and Jira Xray, ensuring traceability and enabling robust quality monitoring. The workflow also utilizes dependency caching to reduce build times and improve overall efficiency.
- **Code Formatting Enforcement**: To maintain consistent coding standards across the codebase, we established separate formatting workflows for the frontend and backend. The frontend workflow enforces JavaScript styling rules via ESLint with the Airbnb configuration, while the backend workflow applies Google-Java-Format to Java code. If any formatting discrepancies are detected, the workflows automatically commit and push the corrected files, promoting clean, standardized code.
- **Functional Testing with Selenium**: A dedicated workflow titled Run Selenium Tests with Chrome orchestrates end-to-end functional testing. This workflow launches all necessary services using Docker Compose, verifies the frontend's readiness, and executes behavior-driven development (BDD) tests using the Cucumber testing framework. ChromeDriver is used to simulate real user interactions, and all test results are reported to Jira Xray to ensure comprehensive documentation and traceability.
- **Dynamic Environment Configuration**: Each workflow dynamically generates required configuration files such as .env and application.properties, using context-specific variables. This guarantees that all test executions and builds occur in stable, reproducible environments, reducing configuration drift and environmental inconsistencies.

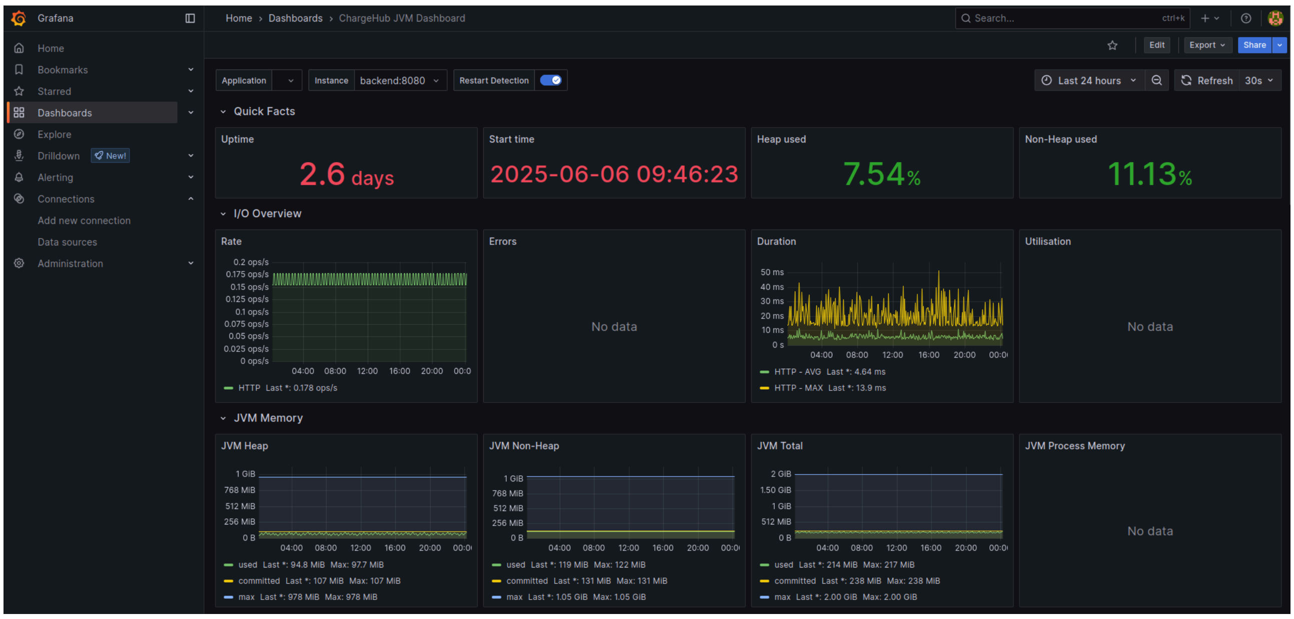
Our Continuous Delivery (CD) pipeline extends the automation principles established in our Continuous Integration (CI) process and is likewise powered by GitHub Actions. Deployments are strictly gated, triggered only after the successful completion of both the core CI workflow and the Selenium-based functional tests. This ensures that only thoroughly tested and validated code is promoted to the production environment.
The deployment workflow operates on a self-hosted GitHub Actions runner, which provides greater control over the deployment infrastructure and enhances security and flexibility. Once initiated, the workflow checks out the latest code from the main branch, generates the necessary service-specific environment configuration files required for production, and deploys the full application stack using Docker Compose. This deployment leverages pre-built containers for all essential components, including the backend, frontend, database, and observability tools such as Grafana.
By automating both the environment provisioning and deployment process while enforcing strict validation gates, our CD pipeline significantly reduces the risk of human error, improves the reliability of deployments, and accelerates the delivery lifecycle. This end-to-end automation empowers the team to iterate rapidly, maintain high standards of code quality, and consistently deliver value to end users with confidence.

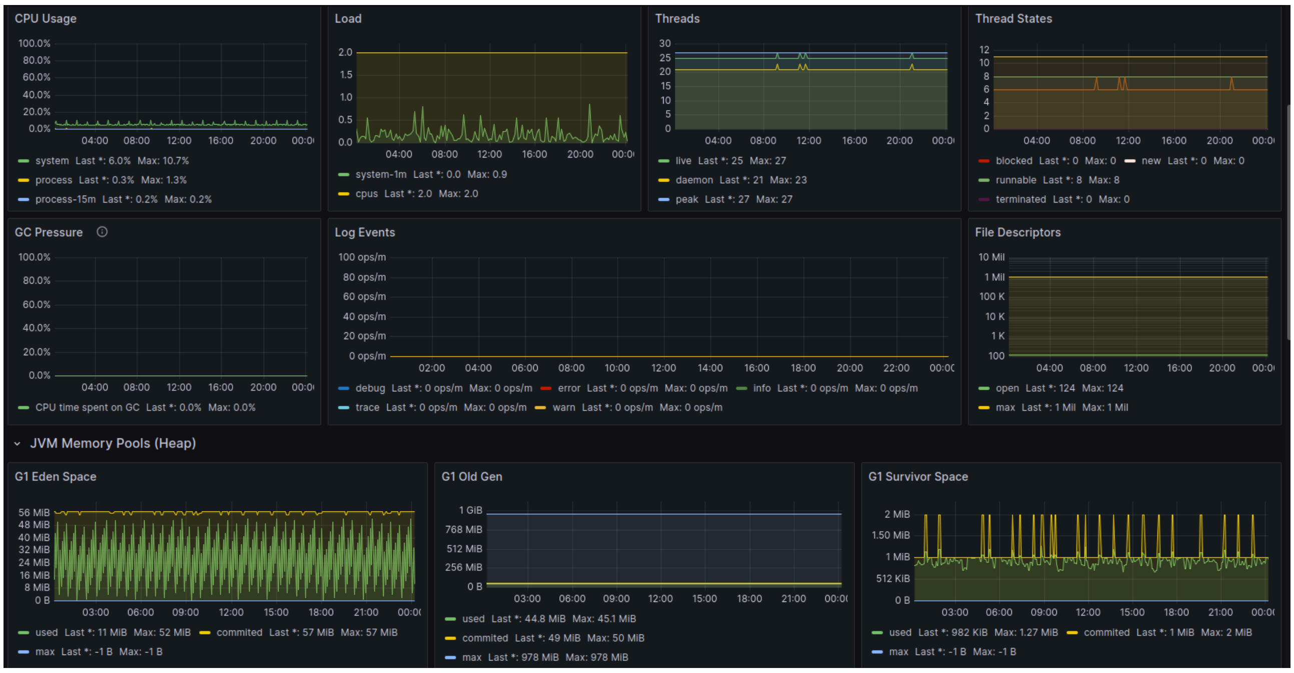
3.3 System observability

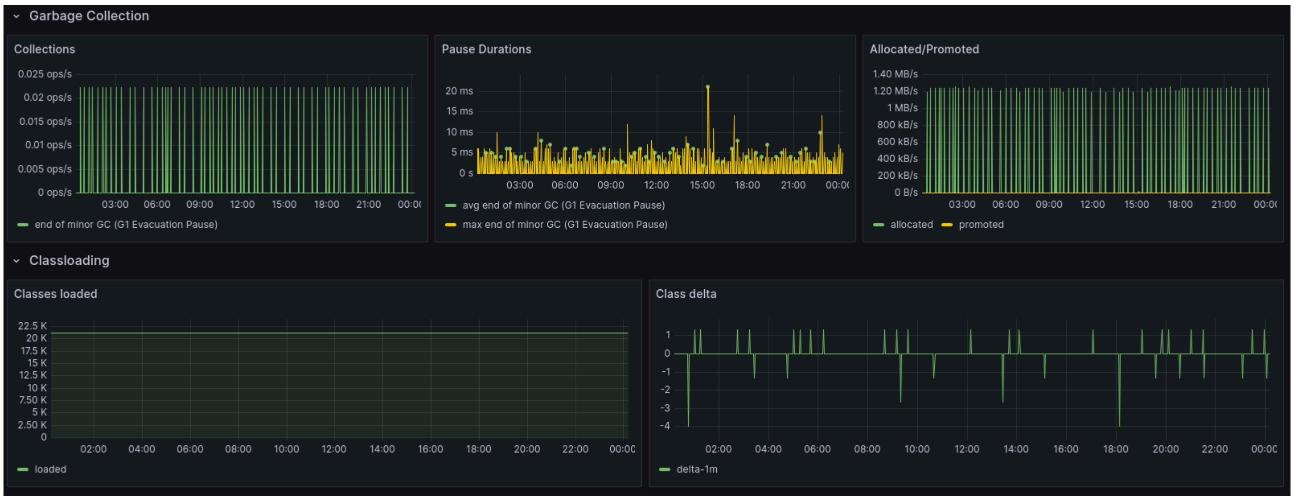
To ensure proactive monitoring and comprehensive observability of the system’s operational conditions, we integrated Prometheus and Grafana into our deployment architecture. The backend application exposes detailed runtime metrics through Spring Boot Actuator’s Prometheus endpoint. These metrics include data on HTTP request rates and latencies, JVM memory usage, garbage collection activity, thread counts, and application-specific custom metrics.

Prometheus is configured to regularly scrape these exposed metrics and store them in a time-series database. This setup enables continuous collection of performance and health data, forming the basis for real-time monitoring and historical analysis. Grafana is then used to visualize these metrics through customizable dashboards that provide an at-a-glance view of system health, resource consumption, and application behavior.

This observability stack ensures that the system remains transparent and measurable at all times. By continuously collecting, visualizing, and evaluating operational data, it supports faster troubleshooting, data-driven optimization, and a high degree of confidence in the system’s stability and performance in production.


Grafana
Home › Dashboards › ChargeHub JVM Dashboard
Home
Bookmarks
Starred
Dashboards
Explore
Drilldown New!
Alerting
Connections
Add new connection
Data sources
Administration
Search... ctrl+k
Edit
Export
Share
Application
Instance backend:8080
Restart Detection
Last 24 hours
Refresh 30s
Quick Facts
Uptime 2.6 days
Start time 2025-06-06 09:46:23
Heap used 7.54%
Non-Heap used 11.13%
I/O Overview
Rate
HTTP Last *: 0.178 ops/s
Errors No data
Duration
HTTP - AVG Last *: 4.64 ms
HTTP - MAX Last *: 13.9 ms
Utilisation No data
JVM Memory
JVM Heap
used Last *: 94.8 MiB Max: 97.7 MiB
committed Last *: 107 MiB Max: 107 MiB
max Last *: 978 MiB Max: 978 MiB
JVM Non-Heap
used Last *: 119 MiB Max: 122 MiB
committed Last *: 131 MiB Max: 131 MiB
max Last *: 1.05 GiB Max: 1.05 GiB
JVM Total
used Last *: 214 MiB Max: 217 MiB
committed Last *: 238 MiB Max: 238 MiB
max Last *: 2.00 GiB Max: 2.00 GiB
JVM Process Memory No data


CPU Usage
system Last *: 6.0% Max: 10.7%
process Last *: 0.3% Max: 1.3%
process-15m Last *: 0.2% Max: 0.2%
Load
system-1m Last *: 0.0 Max: 0.9
cpus Last *: 2.0 Max: 2.0
Threads
live Last *: 25 Max: 27
daemon Last *: 21 Max: 23
peak Last *: 27 Max: 27
Thread States
blocked Last *: 0 Max: 0
new Last *: 0 Max: 0
runnable Last *: 8 Max: 8
terminated Last *: 0 Max: 0
GC Pressure
CPU time spent on GC Last *: 0.0% Max: 0.0%
Log Events
debug Last *: 0 ops/m Max: 0 ops/m
error Last *: 0 ops/m Max: 0 ops/m
info Last *: 0 ops/m Max: 0 ops/m
trace Last *: 0 ops/m Max: 0 ops/m
warn Last *: 0 ops/m Max: 0 ops/m
File Descriptors
open Last *: 124 Max: 124
max Last *: 1 Mil Max: 1 Mil
JVM Memory Pools (Heap)
G1 Eden Space
used Last *: 11 MiB Max: 52 MiB
commited Last *: 57 MiB Max: 57 MiB
max Last *: -1 B Max: -1 B
G1 Old Gen
used Last *: 44.8 MiB Max: 45.1 MiB
commited Last *: 49 MiB Max: 50 MiB
max Last *: 978 MiB Max: 978 MiB
G1 Survivor Space
used Last *: 982 KiB Max: 1.27 MiB
commited Last *: 1 MiB Max: 2 MiB
max Last *: -1 B Max: -1 B


Garbage Collection
Collections
end of minor GC (G1 Evacuation Pause)
Pause Durations
avg end of minor GC (G1 Evacuation Pause)
max end of minor GC (G1 Evacuation Pause)
Allocated/Promoted
allocated
promoted
Classloading
Classes loaded
loaded
Class delta
delta-1m

4 Software testing

4.1 Overall testing strategy

The QA Engineer in conformity with the rest of the members defined the following testing strategy. When it is reasonable, developers should implement Test Driven Development (TDD) with a top-down approach, starting from the controller, then the service, then the repository, and ending with full context integration tests. This should be implemented in a flexible way, depending on the available time and the complexity of tasks, always prioritizing the quality of the final product.

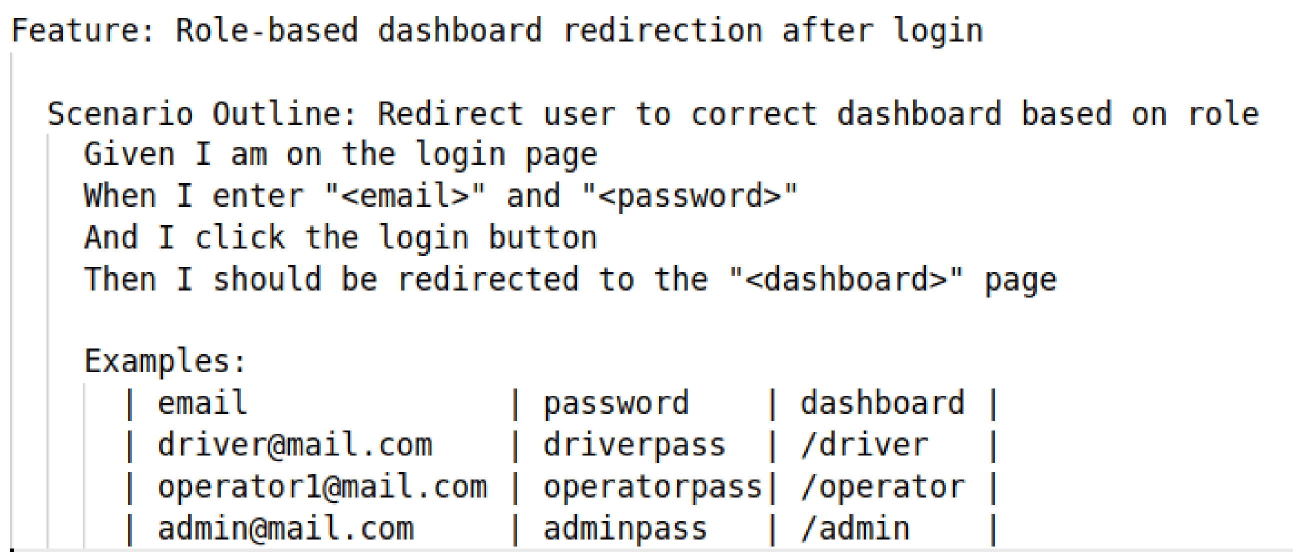
To perform functional tests the QA Engineer defined that the group would adopt a Behaviour Driven Development (BDD) approach using Cucumber and the Selenium Web Driver, to simulate the user's perspective interacting with the web app. For some features the selenium web driver is not applicable. In such cases, experimental testing should be applied to ensure feature quality.

4.2 Functional testing and ATDD

Our team's functional testing strategy is centered on validating system behavior through user-centric test scenarios, ensuring alignment with client requirements. To achieve this, we employ a Behaviour-Driven Development (BDD) approach using Cucumber, defining feature scenarios in a structured and human-readable format to improve collaboration between developers, testers, and stakeholders.

For automation, we integrate Selenium WebDriver with Cucumber, enabling dynamic simulation of user interactions within the system on the Google Chrome Browser. This combination provides a powerful framework for executing functional tests consistently while maintaining flexibility and scalability. The Selenium-Cucumber integration ensures efficient execution of user flows, strengthens test automation reliability, and enhances our overall quality assurance pipeline.

Example:



```
Feature: Role-based dashboard redirection after login

  Scenario Outline: Redirect user to correct dashboard based on role
    Given I am on the login page
    When I enter "<email>" and "<password>"
    And I click the login button
    Then I should be redirected to the "<dashboard>" page

    Examples:
      | email               | password    | dashboard |
      | driver@mail.com     | driverpass  | /driver   |
      | operator1@mail.com  | operatorpass| /operator |
      | admin@mail.com      | adminpass   | /admin    |
public class LoginSteps {
  private WebDriver driver;
  private WebDriverWait wait;

  @Before
  public void setup() {
    WebDriverSingleton.initialize();
    driver = WebDriverSingleton.getDriver();
    wait = WebDriverSingleton.getWait();
  }

  @Given("I am on the login page")
  public void iAmOnTheLoginPage() {
    driver.get("http://localhost:3000/");
    System.out.println("Current URL: " + driver.getCurrentUrl());
    assertTrue(driver.getCurrentUrl().equals("http://localhost:3000/"));
  }

  @When("I enter {string} and {string}")
  public void iEnterEmailAndPassword(String email, String password) {
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector(".login-input")));
    driver.findElements(By.cssSelector(".login-input")).get(0).sendKeys(email); // Email field
    driver.findElements(By.cssSelector(".login-input")).get(1).sendKeys(password); // Password field
  }

  @And("I click the login button")
  public void iClickLoginButton() {
    WebElement loginButton =
        wait.until(ExpectedConditions.elementToBeClickable(By.cssSelector(".login-button")));
    loginButton.click();
  }

  @Then("I should be redirected to the {string} page")
  public void iShouldBeRedirectedToDashboard(String expectedPath) {
    wait.until(webDriver -> webDriver.getCurrentUrl().contains(expectedPath));
    String currentUrl = driver.getCurrentUrl();
    assertTrue(
        currentUrl.contains(expectedPath),
        "Expected to be on: " + expectedPath + " but was: " + currentUrl);
  }
}
```

4.3 Developer facing tests (unit, integration)

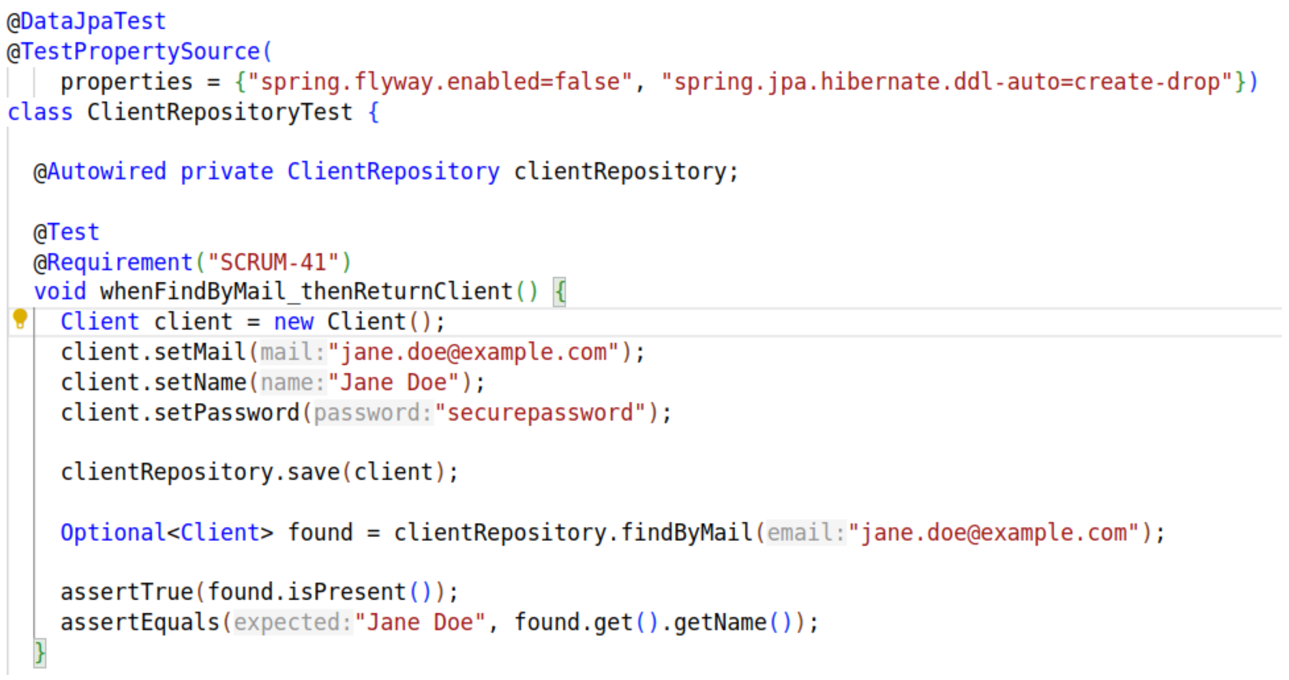
Each developer will write its own tests so we naturally adopt a developer perspective approach for unit tests and integration tests. For integration tests, we decided to adopt the close box approach, since it is the last tests implemented in each feature and we only want to ensure the feature implementation is working correctly on a full flow.

4.3.1 Unitary tests

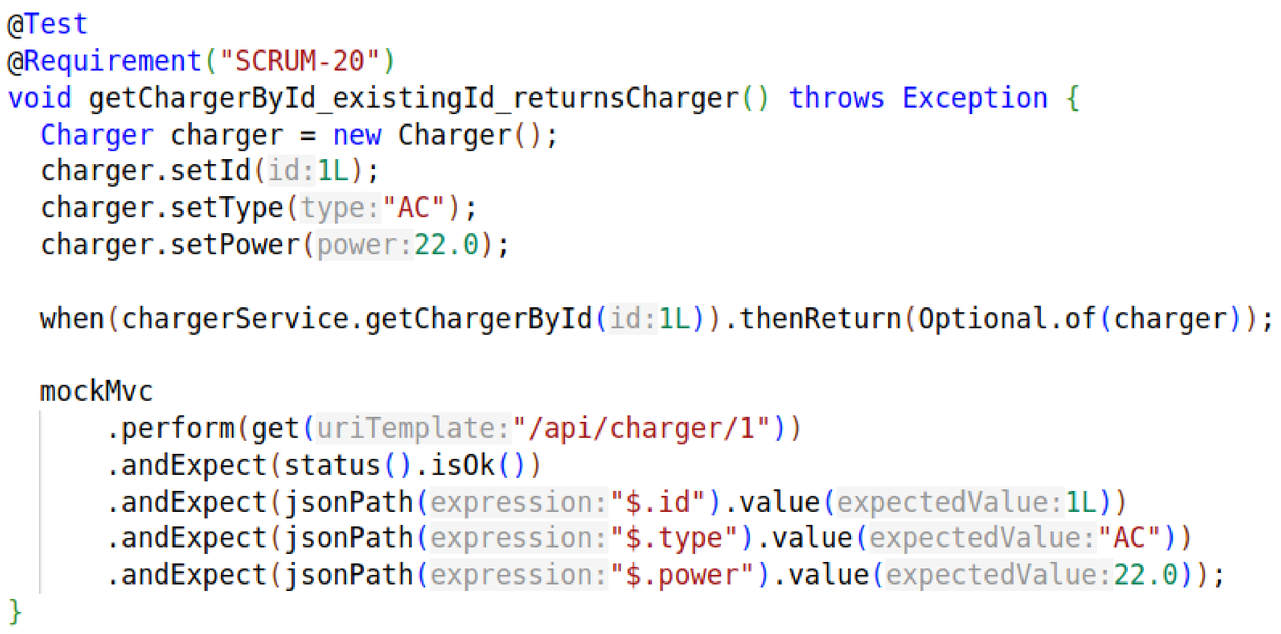
As explained before we adopted a TDD top-down approach for our backend testing. Since we start by writing tests and develop the endpoint mapping on the controller for each new feature to be implemented. The services behaviour for this endpoints test should be mocked using MockMVC.
Unitary tests should also be done for the new business logic that will be implemented in the service layer for the new feature. These tests should mock the dependencies on the respective repositories, which will be implemented on the next step.
After ensuring the business logic is correctly implemented and passing the tests. The data access layer should be implemented, starting by writing the corresponding tests focusing on complex

queries, to ensure the implementation is done correctly. For the tests we load only JPA-related instrumentation.

Example of a unitary tests on repository, service and controller, respectively:



```java
@DataJpaTest
@TestPropertySource(
    properties = {"spring.flyway.enabled=false", "spring.jpa.hibernate.ddl-auto=create-drop"})
class ClientRepositoryTest {

  @Autowired private ClientRepository clientRepository;

  @Test
  @Requirement("SCRUM-41")
  void whenFindByMail_thenReturnClient() {
    Client client = new Client();
    client.setMail(mail:"jane.doe@example.com");
    client.setName(name:"Jane Doe");
    client.setPassword(password:"securepassword");

    clientRepository.save(client);

    Optional<Client> found = clientRepository.findByMail(email:"jane.doe@example.com");

    assertTrue(found.isPresent());
    assertEquals(expected:"Jane Doe", found.get().getName());
  }
```



```java
@Test
@Requirement("SCRUM-20")
void getChargerById_existingId_returnsCharger() throws Exception {
  Charger charger = new Charger();
  charger.setId(id:1L);
  charger.setType(type:"AC");
  charger.setPower(power:22.0);

  when(chargerService.getChargerById(id:1L)).thenReturn(Optional.of(charger));

  mockMvc
      .perform(get(uriTemplate:"/api/charger/1"))
      .andExpect(status().isOk())
      .andExpect(jsonPath(expression:"$.id").value(expectedValue:1L))
      .andExpect(jsonPath(expression:"$.type").value(expectedValue:"AC"))
      .andExpect(jsonPath(expression:"$.power").value(expectedValue:22.0));
}
```

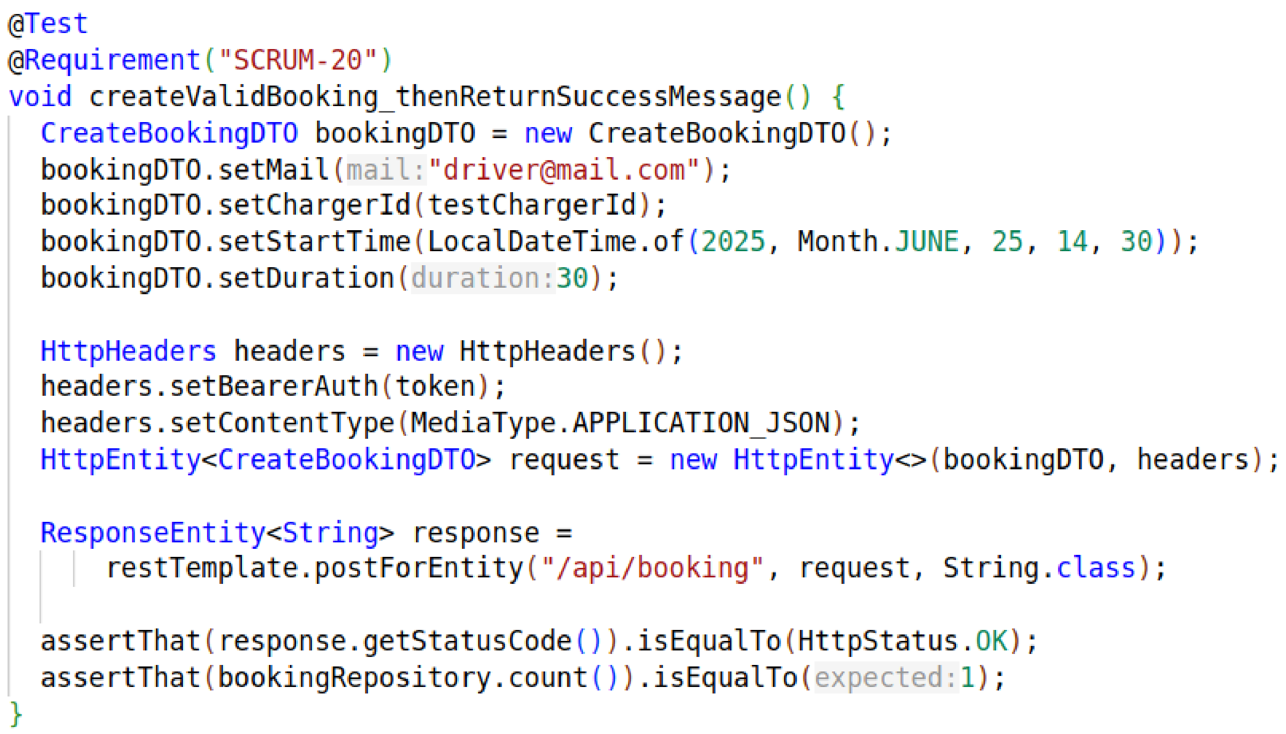
```java
@ExtendWith(MockitoExtension.class)
@MockitoSettings(strictness = Strictness.LENIENT)
public class ChargerServiceTest {

  @Mock private ChargerRepository chargerRepository;
  @Mock private BookingRepository bookingRepository;
  @Mock private ChargingSessionRepository chargingSessionRepository;

  @Mock private StationRepository stationRepository;

  @InjectMocks private ChargerService chargerService;

  @Test
  @Requirement("SCRUM-20")
  void getChargerById_existingId_returnsCharger() {
    Station station = new Station();

    Charger charger = new Charger();
    charger.setStation(station);
    charger.setId(id:1L);
    charger.setType(type:"DC");

    when(chargerRepository.findById(id:1L)).thenReturn(Optional.of(charger));

    Optional<Charger> result = chargerService.getChargerById(id:1L);

    assertThat(result.isPresent()).isTrue();
    assertThat(result.get().getType()).isEqualTo(expected:"DC");
  }
```

4.3.2 Integration tests

After completing unitary tests for each layer, full scope integration tests should be done for the feature to ensure the full functionality flow is working. The integration tests should use Rest Template to create the Rest client and use an in-memory database.

After this is completed and all tests are passed, if the feature implies, frontend code may be implemented respecting the BDD approach.

Example of an integration test:



```java
@Test
@Requirement("SCRUM-20")
void createValidBooking_thenReturnSuccessMessage() {
    CreateBookingDTO bookingDTO = new CreateBookingDTO();
    bookingDTO.setMail(mail:"driver@mail.com");
    bookingDTO.setChargerId(testChargerId);
    bookingDTO.setStartTime(LocalDateTime.of(2025, Month.JUNE, 25, 14, 30));
    bookingDTO.setDuration(duration:30);

    HttpHeaders headers = new HttpHeaders();
    headers.setBearerAuth(token);
    headers.setContentType(MediaType.APPLICATION_JSON);
    HttpEntity<CreateBookingDTO> request = new HttpEntity<>(bookingDTO, headers);

    ResponseEntity<String> response =
            restTemplate.postForEntity("/api/booking", request, String.class);

    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(bookingRepository.count()).isEqualTo(expected:1);
}
```

4.4 Exploratory testing

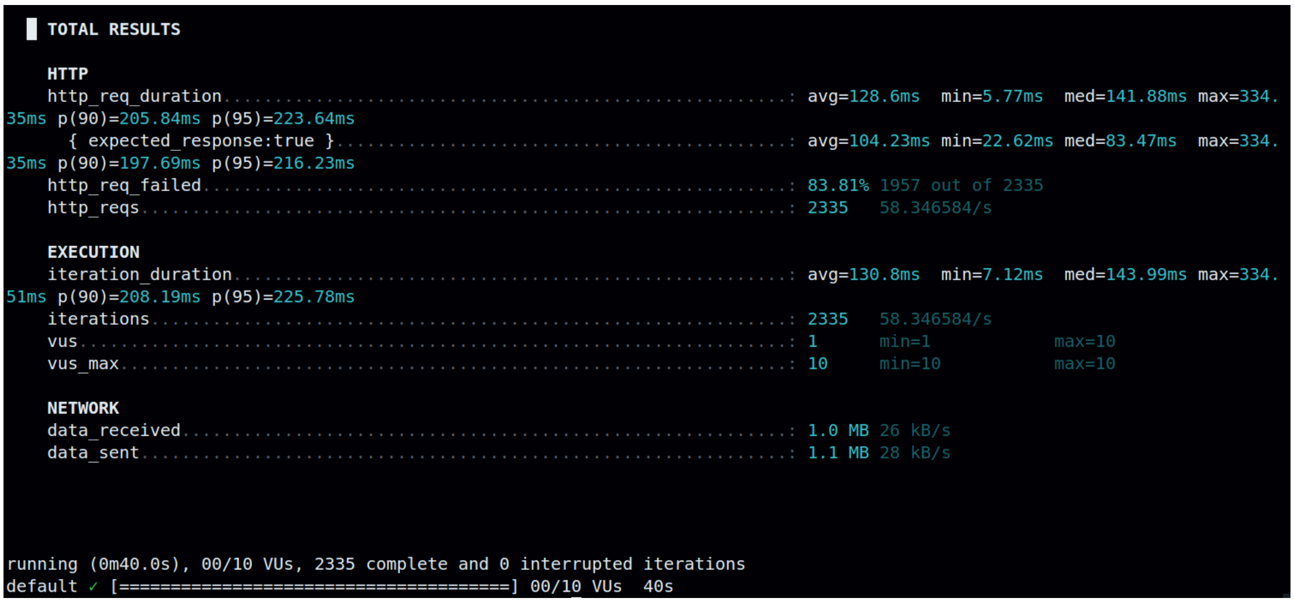
Each developer performed exploratory testing by manually interacting with the web interface, validating functionality across key user flows. Beyond UI-based checks, they also tested API endpoints both within the interface and directly using Postman, ensuring comprehensive coverage and identifying edge cases beyond scripted test scenarios.

Exploratory testing proved particularly valuable for features where Selenium WebDriver testing was less practical. For instance, in the Charging Management epic, testing the user story for unlocking a charger with a token was better suited to exploratory testing. Since the charger unlock occurs several minutes after the corresponding booking, making the Selenium driver wait for this duration was not an optimal approach.

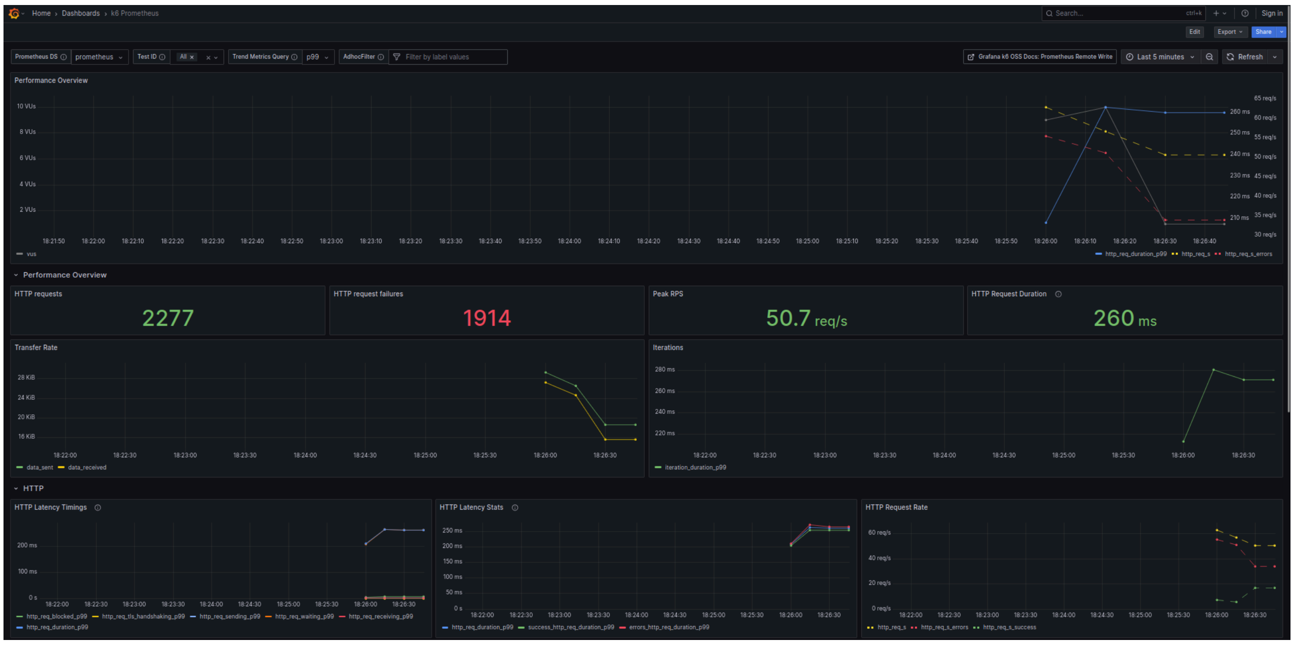
While these tests were not formally documented, each pull request for a new feature included a video demonstrating its correct execution, providing clear validation of functionality.

4.5 Non-functional and architecture attributes testing

Our project includes policies for assessing performance and architectural resilience through targeted testing. Load testing is performed using **k6** on key API endpoints to evaluate backend performance under stress.
During testing, we observed a drop in HTTP success rates due to multiple virtual users (VUs) concurrently attempting to book the same slot. This contention led to failed booking attempts returning HTTP 400 responses, which impacted the overall success metrics.

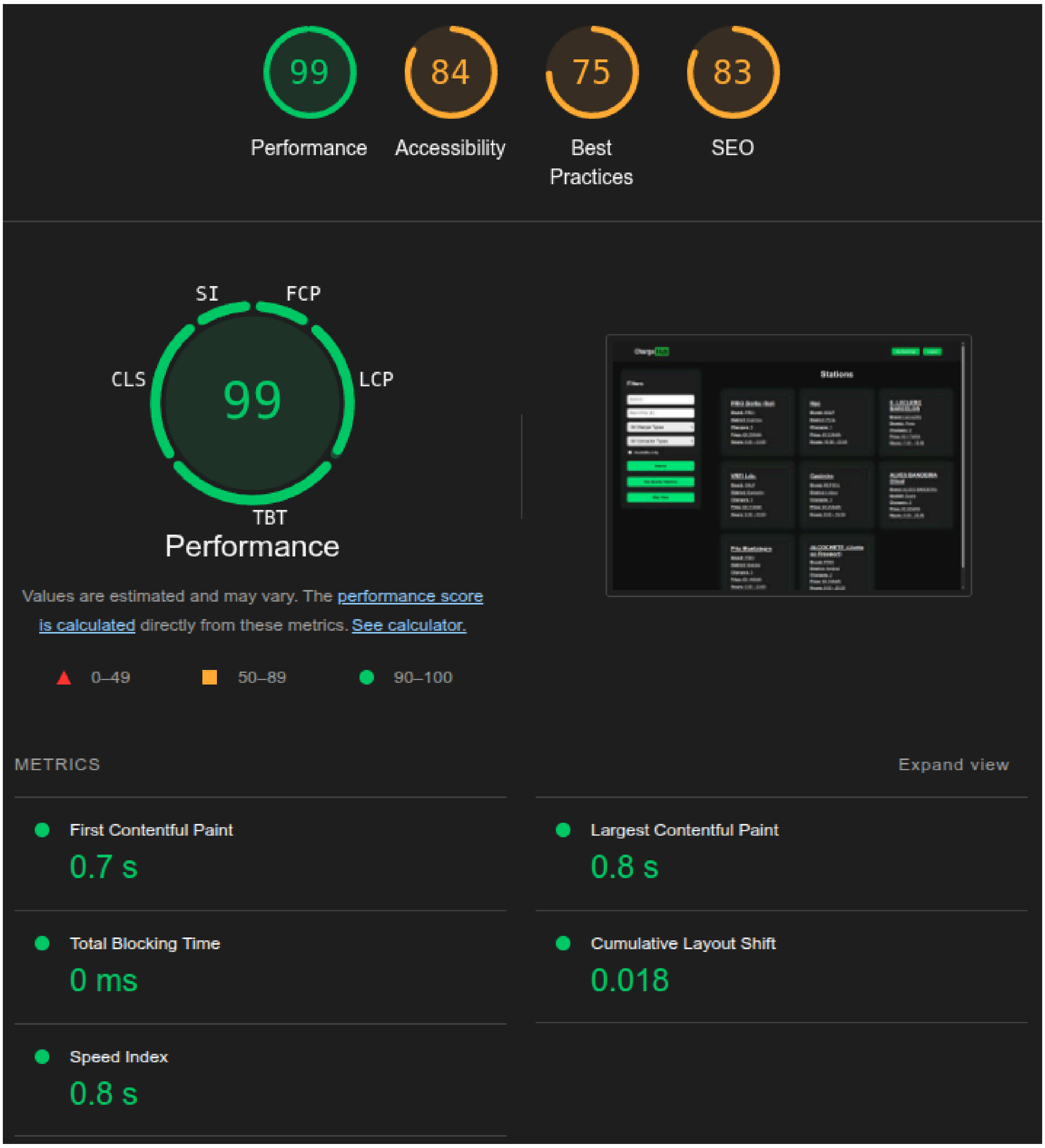


```
█ TOTAL RESULTS

    HTTP
    http_req_duration......................................................: avg=128.6ms  min=5.77ms  med=141.88ms max=334.
35ms p(90)=205.84ms p(95)=223.64ms
      { expected_response:true }...........................................: avg=104.23ms min=22.62ms med=83.47ms  max=334.
35ms p(90)=197.69ms p(95)=216.23ms
    http_req_failed........................................................: 83.81% 1957 out of 2335
    http_reqs..............................................................: 2335   58.346584/s

    EXECUTION
    iteration_duration.....................................................: avg=130.8ms  min=7.12ms  med=143.99ms max=334.
51ms p(90)=208.19ms p(95)=225.78ms
    iterations.............................................................: 2335   58.346584/s
    vus....................................................................: 1      min=1            max=10
    vus_max................................................................: 10     min=10           max=10

    NETWORK
    data_received..........................................................: 1.0 MB 26 kB/s
    data_sent..............................................................: 1.1 MB 28 kB/s



running (0m40.0s), 00/10 VUs, 2335 complete and 0 interrupted iterations
default ✓ [====================================] 00/10 VUs  40s
```



HTTP requests 2277
HTTP request failures 1914
Peak RPS 50.7 req/s
HTTP Request Duration 260 ms

For frontend performance, we integrate **Lighthouse** to measure metrics such as loading speed, accessibility, and best practices compliance. These tools help ensure the system meets performance expectations and remains scalable and user-friendly.



99 Performance

84 Accessibility

75 Best Practices

83 SEO

SI FCP CLS 99 LCP TBT

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● First Contentful Paint
0.7 s

● Largest Contentful Paint
0.8 s

● Total Blocking Time
0 ms

● Cumulative Layout Shift
0.018

● Speed Index
0.8 s