

TQS: Quality Assurance manual

Bárbara Galiza [105937], Diana Miranda [107457], Diogo Falcão [108712], Rúben Garrido [107927]

v2024-05-27

1	Project management	1
1.1	Team and roles	1
1.2	Agile backlog management and work assignment	2
2	Code quality management	3
2.1	Guidelines for contributors (coding style)	3
2.2	Code quality metrics and dashboards	3
2.3	Xray	5
3	Continuous delivery pipeline (CI/CD)	6
3.1	Development workflow	6
3.2	CI/CD pipeline and tools	7
3.2.1	Continuous Integration (CI)	8
3.2.2	Continuous Deployment (CD)	10
3.3	System observability	10
4	Software testing	11
4.1	Overall strategy for testing	11
4.2	Functional testing/acceptance	12
4.3	Unit tests	12
4.4	System and integration testing	16
4.5	Helpers	17
4.6	Performance testing	17

1 Project management

1.1 Team and 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 (Diogo Falcão): Ensure a fair distribution of tasks and that members adhere to the plan. Promote optimal collaboration within the team and proactively address any arising issues. Ensure timely delivery of the requested project outcomes.
- Product owner (Bárbara Galiza): 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 (Diana Miranda): 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 (Rúben Garrido): 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 Agile backlog management and work assignment

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 done. Then, during that iteration, any person is free to assign themselves to an user story present on the "To do" column of the backlog.

Git branches are created from stories and sub-tasks of these stories. After completing a sub-task, the developer must create a Pull Request to the story branch. Given this project uses different repositories for each frontend component and one for the backend (integrated via git submodules), the story branch must be created in all repositories related to it, which is normally one of the frontend repositories and the backend one. After finishing all subtasks, the developer must create a Pull Request from the story branch to the main branch.

On our Jira board, we have the columns "To do", "In progress", "In review" and "Done". For moving the user stories between them, we defined a set of Jira automation rules:

- "To do" → "In progress": When story branch is created
- "In progress" → "In review": When a Pull Request (on the story branch) is created
- "In review" → "Done": When Pull Request is merged

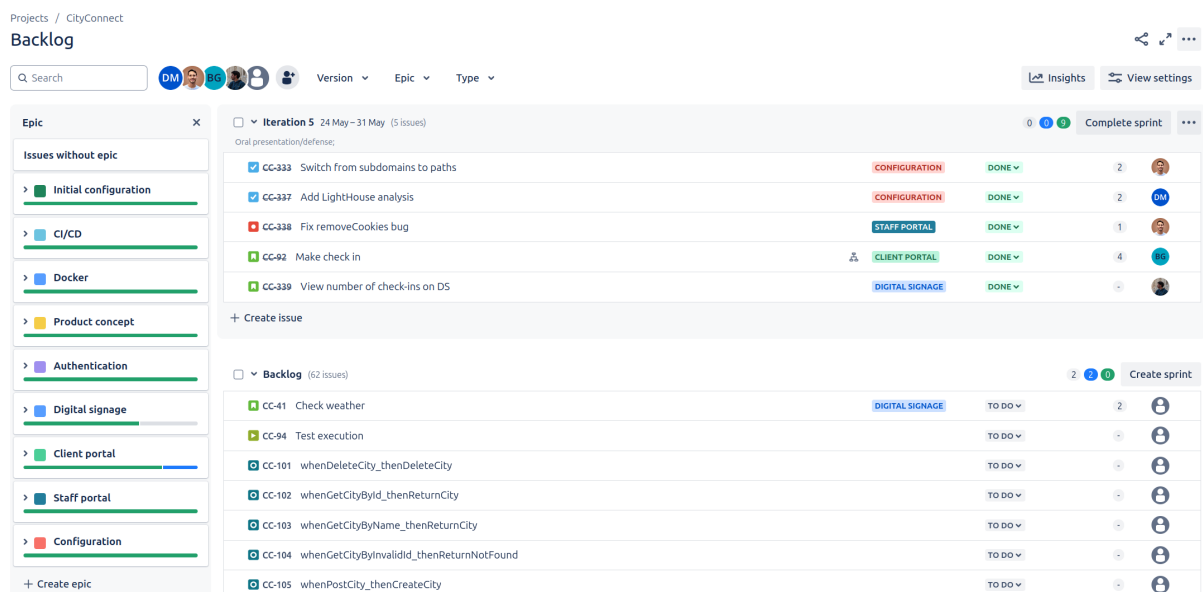


Figure 1: Backlog During The Iteration 5



Figure 2: Burnup Report At The End Of Iteration 4

2 Code quality management

2.1 Guidelines for contributors (coding style)

The code style adopted on this project is based on the conventions used for each programming language and on the consistence rule, meaning new code introduced should follow the same aspect of surrounding code, in order to make it more readable and maintainable.

Regarding the Java rules, we follow the AOSP Java code style for contributors guidelines, focusing primarily on the Java style rules such as "Write short methods", "Define fields in standard places", "Use standard brace style", and other rules such as "Don't ignore exceptions" and "Follow test method naming conventions". Additionally, we use the SonarQube suggestions to fix the bad smells that arise.

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. Besides that, a few ESLint plugins were installed, for both Tailwind CSS and Tanstack Query, so that the rules are extended to those packages.

2.2 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, with the difference of the duplication percentage

allowed, which we defined as 10%. 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, and we changed the duplication because we found it was failing the gate in duplication that couldn't be fixed too many times. The conditions of the quality gate used can be seen on Figure 3.

Conditions ?
Add Condition

Conditions on New Code

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

Metric	Operator	Value		
Coverage	is less than	80.0%		
Duplicated Lines (%)	is greater than	10.0%		
Maintainability Rating	is worse than	A		
Reliability Rating	is worse than	A		
Security Hotspots Reviewed	is less than	100%		
Security Rating	is worse than	A		

Figure 3: Sonar Quality Gate

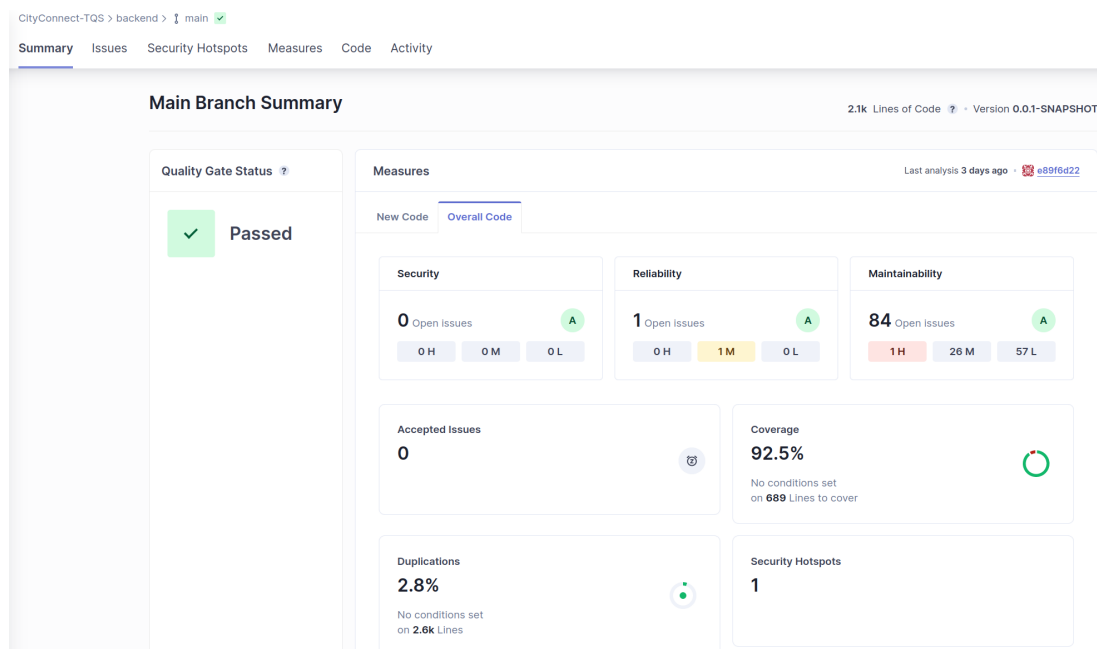


Figure 4: Sonar Dashboard on the Backend Repository

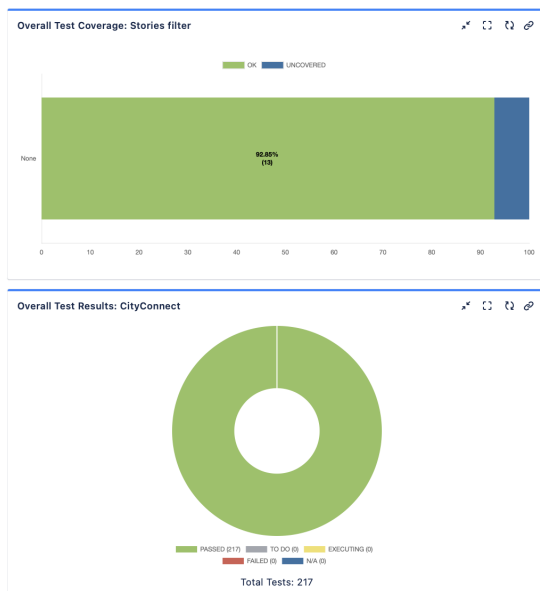
2.3 Xray

We used Xray to connect tests to their respective user stories and see their result in Jira. 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.

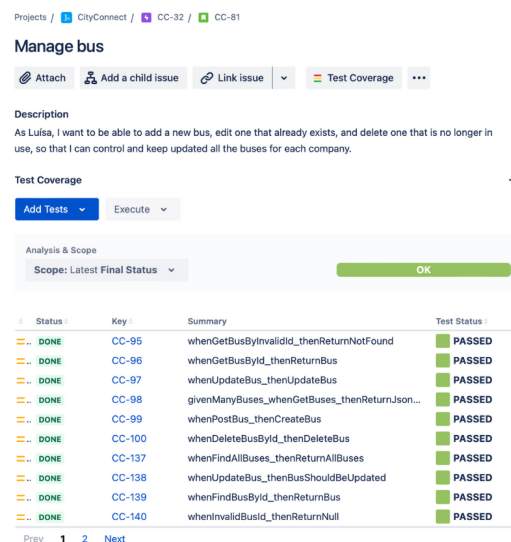
Each time a new user story is created or new tests are implemented, we use the Xray user interface to connect themselves. This method was chosen over the one where user stories are connected through an annotation on each test, due to a more simplified and less cluttered organization.

A testing dashboard was also created, with a summary of passing/failing test results, as well as user story coverage. The only uncovered user story is the one that was pushed to future work (Check weather), so there are no tests available yet for that.

Results are uploaded to Xray automatically from the JUnit reports, 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 and their respective sub-tasks. Therefore those branches follow the naming convention **CC-[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.

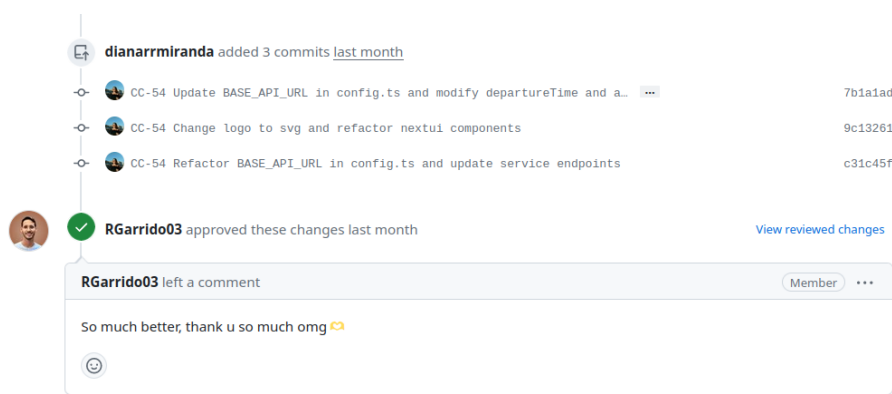


Figure 6: Pull Request Approved (After making the changes requested on Figure 7)

Merged Cc 54 frontend #2
RGarrido03 merged 15 commits into CC-38-Create-trip from CC-54-Frontend last month

CC-54 add input validations d2c387e 3 participants

RGarrido03 requested changes last month [View reviewed changes](#)

[Lock conversation](#)

RGarrido03 left a comment • edited by dianarmiranda

I added some comments and requested changes about using the built-in NextUI error messages, as well as color scheme and SVGs.

```
src/components/modalCreateTrip.tsx Outdated
137 +      }}
138 +      onBlur={handleBlur}
139 +    />
140 +    {state.meta.errors && <span className="text-red-500">{state.meta.errors}</span>}
```

RGarrido03 last month

NextUI components support the `isInvalid` and `errorMessage` props. Please use them instead of plain `span` below the component.

Reply...

Resolve conversation

```
src/components/modalCreateTrip.tsx Outdated
167 +      >
168 +      {(item) => <AutocompleteItem key={item.id} textValue={item.name}>{item.name}</AutocompleteItem>
169 +    }
170 +    {state.meta.errors && <span className="text-red-500">{state.meta.errors}</span>}
```

RGarrido03 last month

NextUI components support the `isInvalid` and `errorMessage` props. Please use them instead of plain `span` below the component.

Reply...

Resolve conversation

```
src/components/modalCreateTrip.tsx Outdated
198 +      onBlur={handleBlur}
199 +      calendarProps={{className: "bg-transparent"}}
200 +    />
201 +    {state.meta.errors && <span className="text-red-500">{state.meta.errors}</span>}
```

RGarrido03 last month

NextUI components support the `isInvalid` and `errorMessage` props. Please use them instead of plain `span` below the component.

Reply...

Resolve conversation

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. The story is not merged into the `main` branch until these requirements are met, as mentioned above.

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 2.2)
- Upload of results to **Xray**, for associating tests to user stories (check section 2.3)
- GitHub's **CodeQL** analysis

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

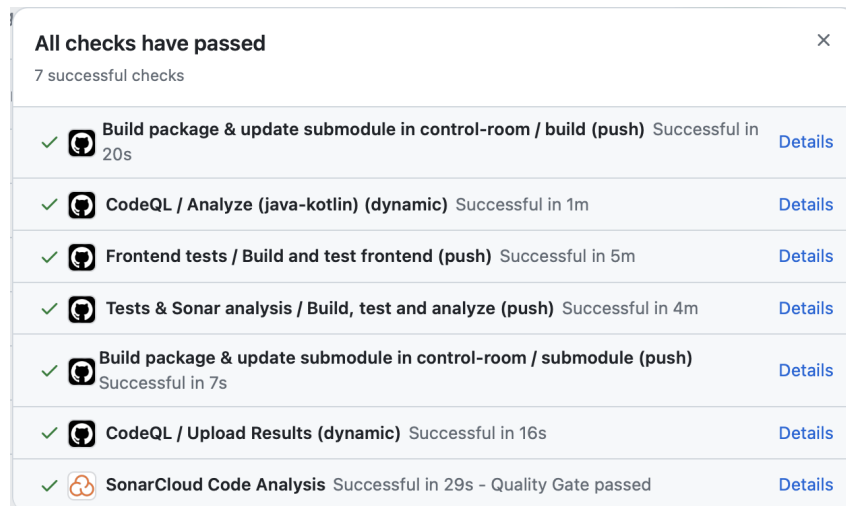
- **SonarCloud analysis**, which defines a Quality Gate
- **Performance-wise & quality analysis**, using Lighthouse (check section 4.6)
- GitHub's **CodeQL** analysis

Workflow file for this run
 .github/workflows/test.yaml at e89f6d2

```

1 name: Tests & Sonar analysis
2
3 on:
4   push:
5     branches:
6       - main
7   pull_request:
8     types: [opened, synchronize, reopened]
9   workflow_dispatch:
10
11 jobs:
12   build:
13     name: Build, test and analyze
14     runs-on: ubuntu-latest
15     permissions: read-all
16     steps:
17       - name: Checkout code
18         uses: actions/checkout@v4
19         with:
20           fetch-depth: 0
21
22       - name: Set up JDK
23         uses: actions/setup-java@v4
24         with:
25           java-version: 21
26           distribution: "zulu"
27
28       - name: Cache SonarCloud packages
29         uses: actions/cache@v4
30         with:
31           path: ~/.sonar/cache
32           key: ${{ runner.os }}-sonar
33           restore-keys: ${{ runner.os }}-sonar
34
35       - name: Cache Maven packages
36         uses: actions/cache@v4
37         with:
38           path: ~/.m2
39           key: "${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}"
40           restore-keys: ${{ runner.os }}-m2
41
42       - name: Build and analyze
43         env:
44           GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
45           SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
46           CURRENCY_API_KEY: ${{ secrets.CURRENCY_API_KEY }}
47         run: |
48           SPRING_APPLICATION_JSON='${"currencyapi.apikey": "${CURRENCY_API_KEY}"}' \
49           mvn -B -Dtest=pt.ua.deti.tqs.backend.functionat.* verify failsafe:integration-test sonar:sonar \
50           -Dsonar.projectKey=CityConnect-TQS_backend
51
52       - name: Publish results to Xray
53         uses: mikepenz/xray-action@v3
54         with:
55           username: ${{ secrets.XRAY_CLIENT_ID }}
56           password: ${{ secrets.XRAY_CLIENT_SECRET }}
57           testFormat: "JUnit"
58           testPaths: "*/surefire-reports/TEST-*.xml"
59           testExecKey: "CC-94"
60           projectKey: "CC"
```

Figure 8: Actions workflow for running backend tests and upload them to SonarCloud and Xray

Figure 9: Workflow run status in the `main` branch

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.

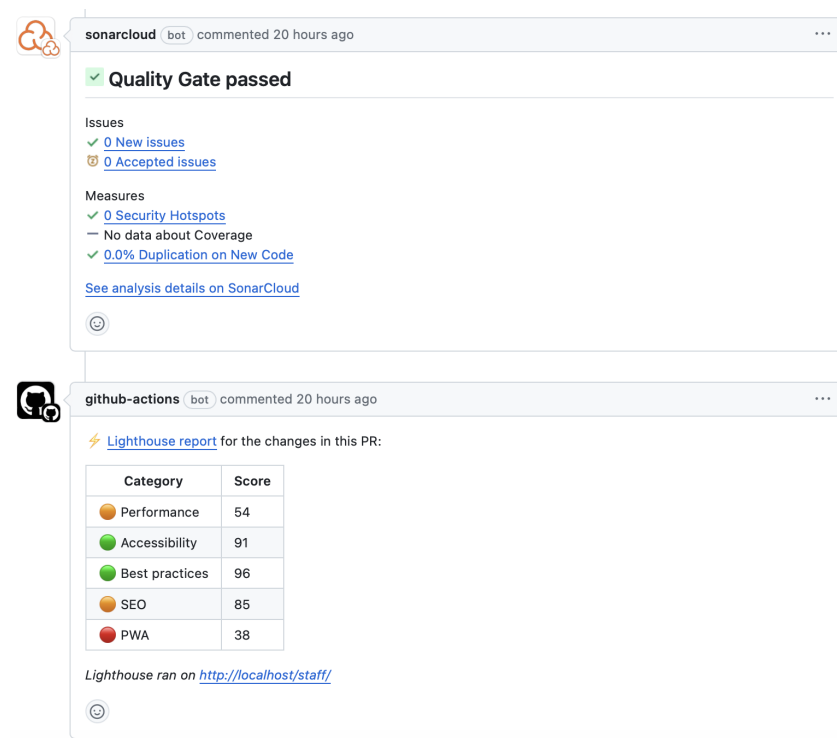


Figure 10: SonarCloud and Lighthouse results as PR comments

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.

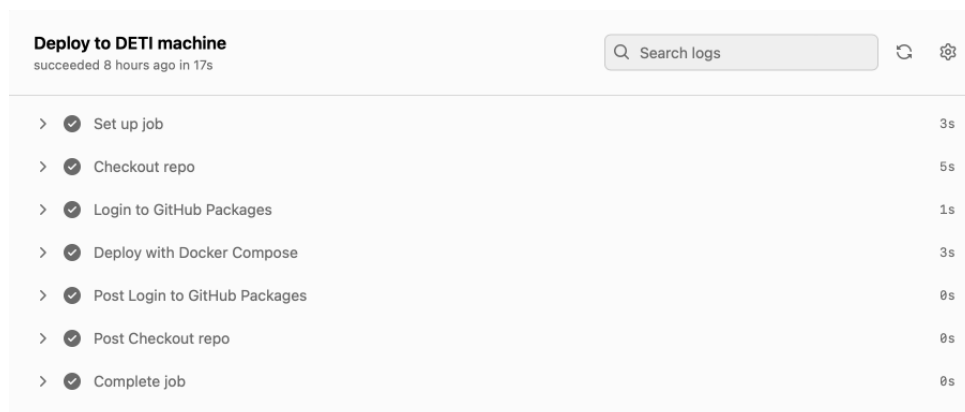


Figure 11: Deployment workflow run in Actions

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 regarding the update of trip status sent to the digital signage, given it's a core feature that requires a connection between two components. However, this logging wasn't persisted on files.

4 Software testing

4.1 Overall strategy for testing

The overall strategy adopted for implementing the tests followed a Test-Driven Development (TDD) approach, 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, SEO, and PWA concerning the staff portal, client portal, and digital signage were evaluated, ensuring quality in all web apps.

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.

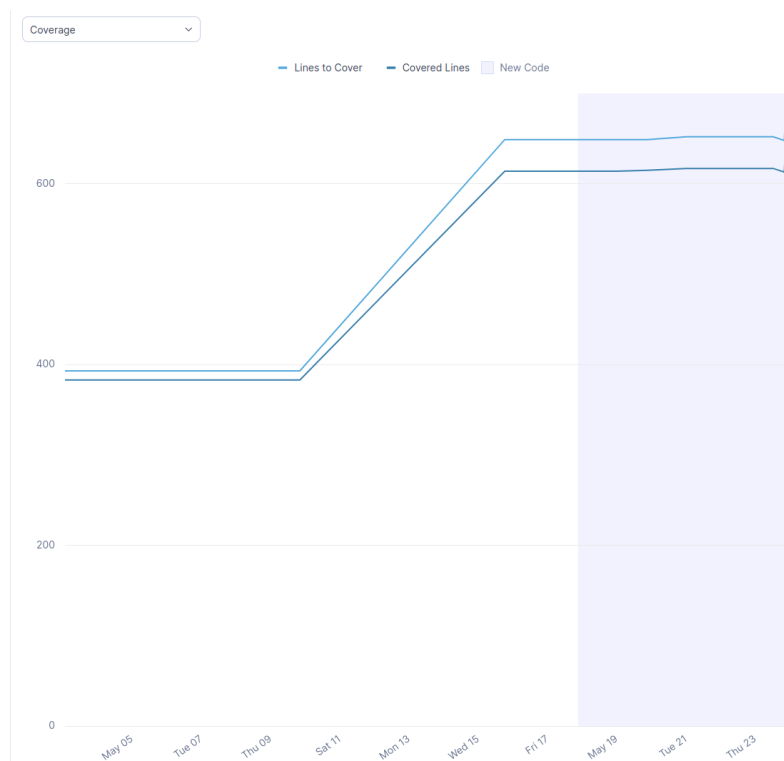


Figure 12: Test coverage throughout development

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.

```
@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("pt/ua/deti/tqs/backend/functional/client")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "pt.ua.deti.tqs.backend.functional.client")
public class ClientCucumberTest {

    private static WebDriver driver;

    public static WebDriver getDriver() {
        if (driver == null) {
            FirefoxOptions options = new FirefoxOptions();
            options.addArguments("-headless");
            driver = new FirefoxDriver(options);
        }
        return driver;
    }
}
```

Figure 13: Client Class for Running Frontend Tests on Client Portal

```
Feature: Check existing trips

Scenario: Check existing trips
    When I navigate to "http://localhost/"
    And I fill in departure with 3
    And I fill in arrival with 1
    And I search for trips
    Then I should see a list of trips

Scenario: Check existing trips on a specific date
    When I navigate to "http://localhost/"
    And I fill in departure with 3
    And I fill in arrival with 1
    And I fill in date with "05/24/2024"
    And I search for trips
    Then I should see a list of trips on date "05/24/2024" or after this date

Scenario: Check existing trips but there are none
    When I navigate to "http://localhost/"
    And I fill in departure with 3
    And I fill in arrival with 1
    And I fill in date with "05/24/2025"
    And I search for trips
    Then I should see a message saying there "No trips available for this date"
```

Figure 14: Feature for Functional Tests of the "Check Trips Available" Story (Example of One of the Tests Implemented)

```
public class CheckTripsSteps {
    private final WebDriver driver = ClientCucumberTest.getDriver();

    @When("I fill in departure with {int}")
    public void iFillInDepartureWith(int number) throws InterruptedException {
        Thread.sleep(1000);
        driver.findElement(By.id("origin")).click();
        driver.findElement(By.id("departure" + number)).click();
    }

    @And("I fill in arrival with {int}")
    public void iFillInArrivalWith(int number) {
        driver.findElement(By.id("destination")).click();
        driver.findElement(By.id("arrival" + number)).click();
    }

    @And("I fill in date with {string}")
    public void iFillInDateWith(String date) {
        String[] parts = date.split("/");
        driver.findElement(By.cssSelector("#departureTime>div>div:nth-child(1)")).sendKeys(parts[0]);
        driver.findElement(By.cssSelector("#departureTime>div>div:nth-child(3)")).sendKeys(parts[1]);
        driver.findElement(By.cssSelector("#departureTime>div>div:nth-child(5)")).sendKeys(parts[2]);
    }

    @And("I search for trips")
    public void iSearchForTrips() throws InterruptedException {
        driver.findElement(By.id("searchBtn")).click();
    }
}
```

Figure 15: Partial Steps for Implementing the "Check Existing Trips" Feature

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.

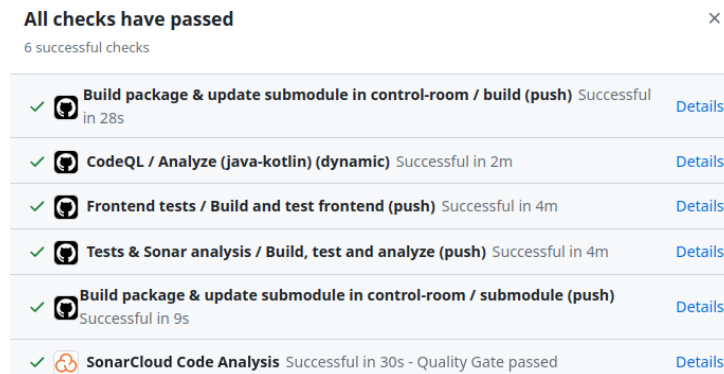


Figure 16: CI/CD pipeline - automatic execution

Tests are documented, peer-reviewed, and must be included in pull requests, with user stories considered complete only when all related tests are done. 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.

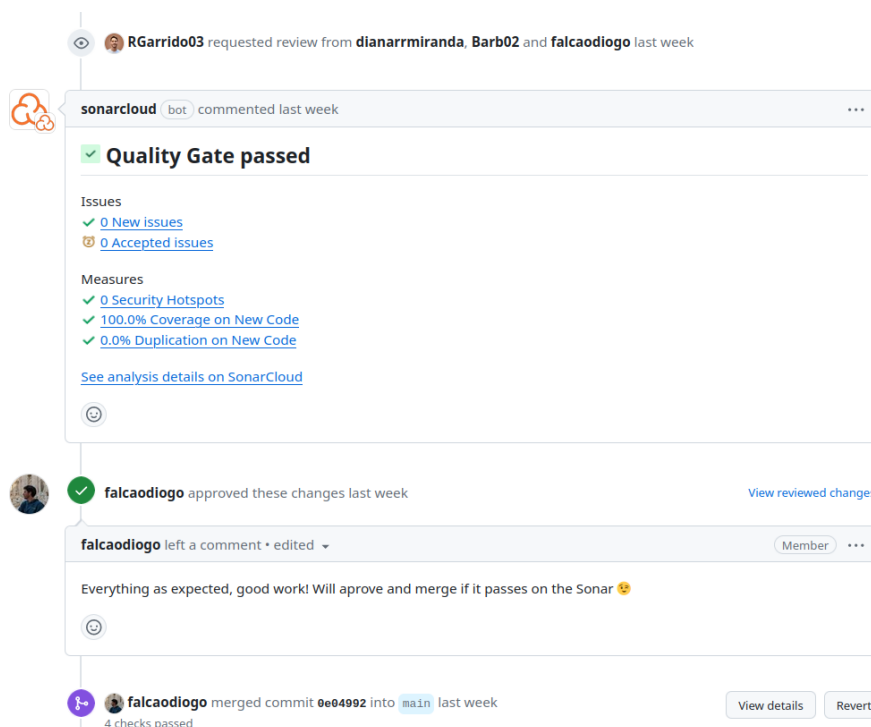


Figure 17: Sonarcloud after Unit testing PR

- Controllers Testing

First, we have the controllers tests that were done for all entities (BusControllerTest, CityControllerTest, ReservationControllerTest, StatsControllerTest, TripControllerTest and UserControllerTest). The tests were made

with Mockito to mock each service tested and the token creation for the defined permissions of each operation according to the roles of the actor. The files have an average of 7 tests and below, we see an example of a unit test for the user service.

```

225  @Test
226  void whenUpdateUser_thenUpdateUser() throws Exception {
227      User user = new User();
228      user.setId(1L);
229      user.setName("John Doe");
230      user.setEmail("johndoe@ua.pt");
231      user.setPassword("password");
232      user.setRoles(List.of(UserRole.USER, UserRole.STAFF));
233
234      LoginResponse loginResponse = new LoginResponse(user.getId(), user.getName(), user.getEmail(), user.getRoles(), "token", 123456789L);
235
236      when(service.updateUser(Mockito.any(Long.class), Mockito.any(User.class))).thenReturn(loginResponse);
237
238      mockMvc.perform(put("/api/backoffice/user/1")
239                  .contentType(MediaType.APPLICATION_JSON)
240                  .content(new ObjectMapper().writeValueAsString(user)))
241              .andExpect(status().isOk())
242              .andExpect(jsonPath("$.id", is(1)))
243              .andExpect(jsonPath("$.name", is("John Doe")))
244              .andExpect(jsonPath("$.email", is("johndoe@ua.pt")))
245              .andExpect(jsonPath("$.roles", hasSize(2)))
246              .andExpect(jsonPath("$.roles[0]", is(UserRole.USER.toString())))
247              .andExpect(jsonPath("$.roles[1]", is(UserRole.STAFF.toString())));
248
249      verify(service, times(1)).updateUser(Mockito.any(Long.class), Mockito.any(User.class));
250  }
251
252  @Test
253  void whenDeleteUser_thenDeleteUser() {
254      RestAssuredMockMvc.given().mockMvc(mockMvc)
255                          .when().delete("/api/public/user/1")
256                          .then().statusCode(200);
257
258      verify(service, times(1)).deleteUser(1L);
259  }
260  }

```

Figure 18: Update user controller test

- Services Testing

Once again, we did services integration testing for all the services there are in the project (BusServiceTest, CityServiceTest, CurrencyServiceTest, ReservationServiceTest, StatsServiceTest, TripServiceTest, TripStatusSchedulerServiceTest and UserServiceTest). To test the repositories, we do a set up function using the BeforeEach annotation. In the next example, we can see the both the set up and the first tests of the bus service.

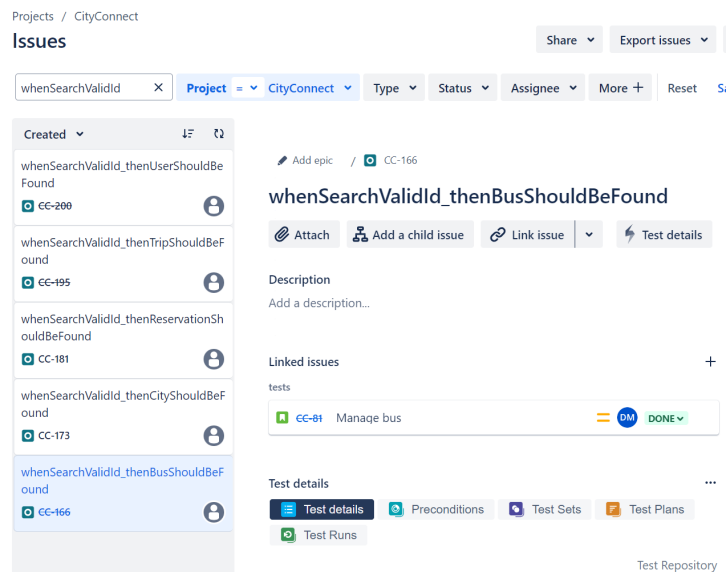
```

18 @ExtendWith(MockitoExtension.class)
19 class BusServiceTest {
20     @Mock(lenient = true)
21     private BusRepository busRepository;
22
23     @InjectMocks
24     private BusService busService;
25
26     @BeforeEach
27     public void setUp() {
28         Bus bus1 = new Bus();
29         bus1.setId(1L);
30         bus1.setCapacity(50);
31         Bus bus2 = new Bus();
32         bus2.setId(2L);
33         bus2.setCapacity(100);
34         Bus bus3 = new Bus();
35         bus3.setId(3L);
36         bus3.setCapacity(50);
37
38         List<Bus> allBuses = List.of(bus1, bus2, bus3);
39
40         Mockito.when(busRepository.findById(12345L)).thenReturn(Optional.empty());
41         Mockito.when(busRepository.findById(bus1.getId())).thenReturn(Optional.of(bus1));
42         Mockito.when(busRepository.findAll()).thenReturn(allBuses);
43     }
44
45     @Test
46     void whenSearchValidId_thenBusShouldBeFound() {
47         Bus found = busService.getBus(1L);
48
49         assertThat(found).isNotNull();
50         assertThat(found.getCapacity()).isEqualTo(50);
51     }
52
53     @Test
54     void whenSearchInvalidId_thenBusShouldNotBeFound() {
55         Bus fromDb = busService.getBus(12345L);
56         assertThat(fromDb).isNull();
57     }

```

Figure 19: Bus service test

We recall that using X-ray, all the tests performed in our project are assign and connected to issues in Jira.



The screenshot shows the Jira interface for the 'CityConnect' project. On the left, a list of issues is displayed, including 'whenSearchValidId_thenUserShouldBeFound' (CC-200), 'whenSearchValidId_thenTripShouldBeFound' (CC-195), 'whenSearchValidId_thenReservationShouldBeFound' (CC-181), 'whenSearchValidId_thenCityShouldBeFound' (CC-173), and 'whenSearchValidId_thenBusShouldBeFound' (CC-166). The right panel shows the details for issue 'whenSearchValidId_thenBusShouldBeFound'. It includes a description field, a 'Linked issues' section showing a link to 'tests' (CC-01), and a 'Test details' section with tabs for 'Test details', 'Preconditions', 'Test Sets', 'Test Plans', and 'Test Runs'. The 'Test details' tab is active, showing a test 'CC-01 Manage bus' with a 'DONE' status.

Figure 20: Jira bus service test

- Repository Testing

For the repositories tests (DataJpaTests), just like all of the other tests types, tested all entities and did tests covering all the available CRUD operations (BusRepositoryTest, CityRepositoryTest, ReservationRepositoryTest, TripRepositoryTest and UserRepositoryTest). In the next image, we see one of the many tests performed to the trips repository.

```

139  @Test
140  void whenFindTripsByArrival_thenReturnCorrectTrips() {
141      City city = Utils.generateCity(entityManager);
142      Bus bus = Utils.generateBus(entityManager);
143
144      Trip trip1 = new Trip();
145      trip1.setDeparture(city);
146      trip1.setArrival(city);
147      trip1.setBus(bus);
148      trip1.setDepartureTime(LocalDateTime.now());
149      trip1.setArrivalTime(LocalDateTime.now().plusHours(3));
150      trip1.setPrice(50);
151      Trip trip2 = new Trip();
152      trip2.setDeparture(city);
153      trip2.setArrival(city);
154      trip2.setBus(bus);
155      trip2.setDepartureTime(LocalDateTime.now());
156      trip2.setArrivalTime(LocalDateTime.now().plusHours(2));
157      trip2.setPrice(100);
158      Trip trip3 = new Trip();
159      trip3.setDeparture(city);
160      trip3.setArrival(city);
161      trip3.setBus(bus);
162      trip3.setDepartureTime(LocalDateTime.now());
163      trip3.setArrivalTime(LocalDateTime.now().plusHours(1));
164      trip3.setPrice(50);
165      Trip trip4 = new Trip();
166      trip4.setDeparture(city);
167      trip4.setArrival(city);
168      trip4.setBus(bus);
169      trip4.setDepartureTime(LocalDateTime.now());
170      trip4.setArrivalTime(LocalDateTime.now().plusHours(1));
171      trip4.setPrice(50);
172      trip4.setStatus(TripStatus.ARRIVED);
173
174      entityManager.persist(trip1);
175      entityManager.persist(trip2);
176      entityManager.persist(trip3);
177      entityManager.persist(trip4);
178
179      Iterable<Trip> trips = tripRepository.findByArrivalAndStatusNotOrderByArrivalTimeAsc(city, TripStatus.ARRIVED, Limit.of(2));
180      assertThat(trips).hasSize(2).contains(trip3, trip2);
181  }

```

Figure 21: Trip repository test

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).

```

10 @TestConfiguration(proxyBeanMethods = false)
11 public class TestBackendApplication {
12
13     @Bean
14     @ServiceConnection
15     PostgreSQLContainer<> postgresContainer() {
16         return new PostgreSQLContainer<>(DockerImageName.parse("postgres:latest"));
17     }
18
19     public static void main(String[] args) {
20         SpringApplication.from(BackendApplication::main).with(TestBackendApplication.class).run(args);
21     }
22 }
23

```

Figure 22: Container test

4.5 Helpers

We conducted a category of helper tests for the external currency API (as seen bellow).

```

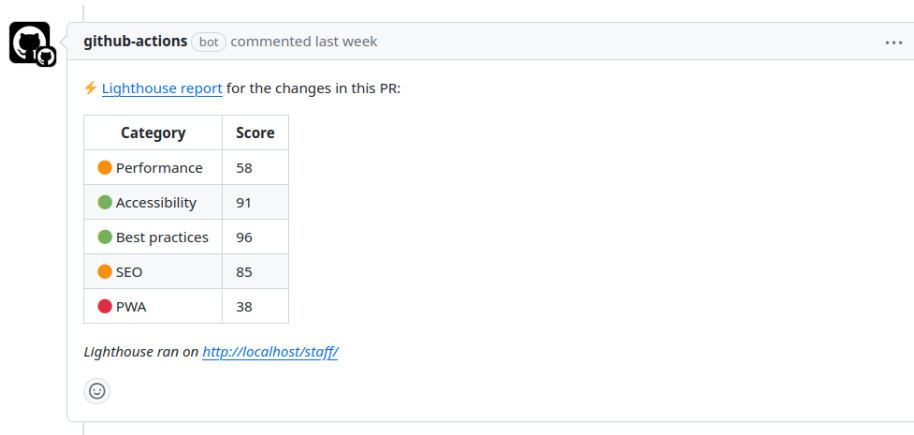
3 import org.junit.jupiter.api.Assertions;
4 import org.junit.jupiter.api.Test;
5
6 import java.util.Map;
7
8 class CurrencyApiResponseTest {
9     @Test
10     void testConstructor() {
11         Map<Currency, Double> data = Map.of(Currency.USD, 1.08, Currency.EUR, 1.0, Currency.GBP, 0.92);
12         CurrencyApiResponse actualCurrencyApiResponse = new CurrencyApiResponse(data);
13         Assertions.assertEquals(data, actualCurrencyApiResponse.getData());
14     }
15
16     @Test
17     void testNoArgsConstructor() {
18         CurrencyApiResponse actualCurrencyApiResponse = new CurrencyApiResponse();
19         Assertions.assertNull(actualCurrencyApiResponse.getData());
20     }
21 }
22
23

```

Figure 23: Currency API test

4.6 Performance testing

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.



github-actions bot commented last week

⚡ [Lighthouse report](#) for the changes in this PR:

Category	Score
Performance	58
Accessibility	91
Best practices	96
SEO	85
PWA	38

Lighthouse ran on <http://localhost/staff/>

😊

Figure 24: Lighthouse performance testing automation