HW1: Mid-term assignment report

Pedro Daniel Fonseca Ramos [876543], v2024-04-08

1 Introduction

1.1 Overview of the work

This project involves the creation of a full-stack application that is able to manipulate and manage a set of bus trips, allowing users to search for routes between cities, look for trips and provide the required information for obtaining a ticket.

The created application is named RoadRoam and is composed of a frontend and backend contained inside a set of dockers manipulated by a docker-compose instance.

The main focus of this assignment is to simulate a real-world scenario where we as developers create a well tested, maintainable and clean project, relying on libraries such as JUnit, Selenium and Cucumber to write easy to read tests with a large coverage.

Overall, the testing tools used for testing were:

- Junit 5 for unit and repository tests
- Mockito and AssertJ for service and repository tests
- Selenium with Cucumber for the frontend UI, action flow and communication testing
- MockMvc, hamcrest and Mockito for the integration tests

Other tools such as SonarCloud and github actions were used to implement automatic testing, code coverage checks and overall static code quality checks.

The Springdoc library was used to generate the documentation for the endpoints, which is available at http://localhost:8080/docs.

1.2 Current limitations

Due to the extent of this assignment's requirements (full stack), some issues have appeared mostly while producing the frontend such as the inability to fully utilize all the features provided by the backend, having the style of the pages fit only select monitors and browsers and sometimes sending incorrect requests to the backend

.

Selenium frontend tests were also skipped for the github workflow and SonarCloud scans, since they made these tools crash after trying to load the pages.

Some Selenium tests fail if the frontend takes a while to load, this was remedied with waits for a web element to load, but they can rarely still fail, making a rerun necessary.

As for the backend, some code remains uncovered by tests and the currency API implementation makes it very hard to mock, so it was omitted from this project.

All of the required features were implemented and some additional ones as well.

Some extra features for the user such as editing and canceling trips are not supported by the current implementation of either the frontend or the backend.

Administrative features such as adding trips, routes, cities, etc, are available in the backend but not supported by the frontend.

2 Product specification

2.1 Functional scope and supported interactions

The current application was designed for accommodating the needs of a user who intends to search for a route between two cities, list the bus trips available between them, book the desired trip with his information and then see the list of acquired tickets.

Usage Scenario:

- Searching for a Bus Route between two cities and booking tickets for one of the trips.

Actors/Roles:

- User: Initiates the process by searching for bus routes, selecting a trip, and booking tickets.

Issues:

- Navigate the interface for a smooth booking process;
- Assure that the backend is optimized for the task flow created by the frontend.

Goals/Context:

- The user intends to find a suitable bus route between two cities;
- The user wants to browse through available bus trips and select one that fits their schedule and preferences;
- After selecting a trip, the user wishes to securely book tickets with their personal information;
- The user wants to see all the acquired tickets.

Scenario/Steps:

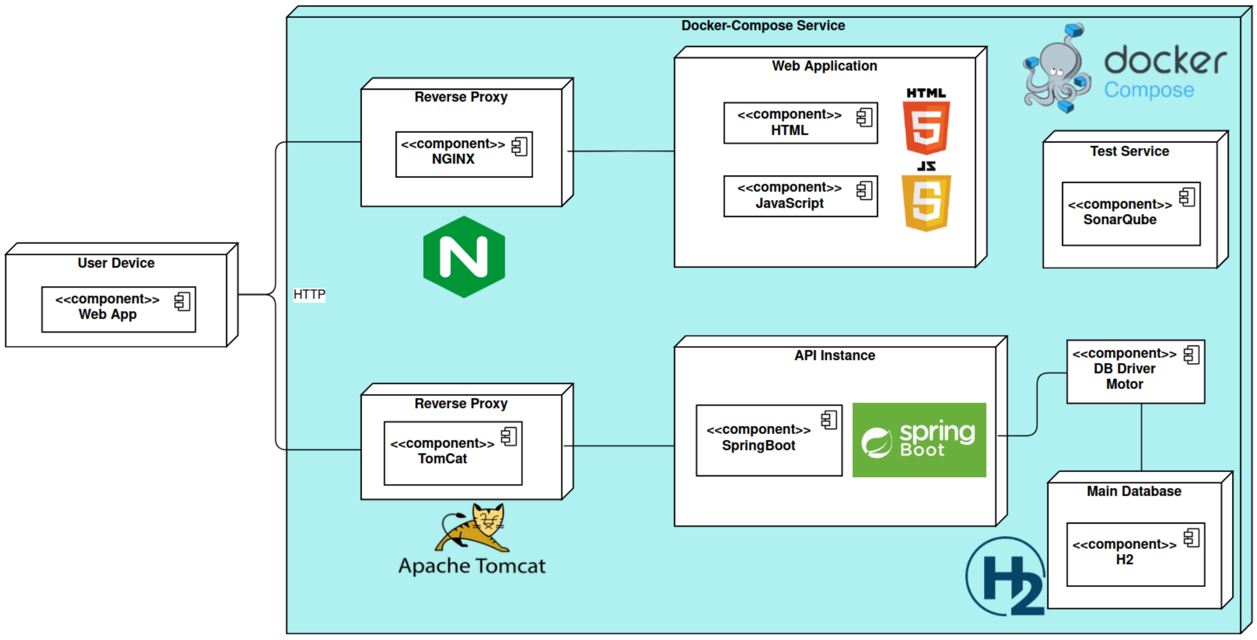
1. User opens the application and selects the departure and destination cities.
2. The system retrieves the available bus trips and displays them to the user.
3. User selects a desired trip from the list of available options.
4. The system prompts the user to enter their personal information for ticket booking (name, contact details, etc.) as well as the currency, intended seat number and number of travelers.
5. User provides the necessary information.
6. The system calculates the total price and presents it to the user.
7. User confirms the price and submits the request for the ticket.
8. The system validates the given information and generates a ticket with the user's information and trip details.
9. The user can view the acquired ticket in the application for reference and boarding.

2.2 System architecture

The system architecture comprises a simple frontend <-> backend relationship where the frontend is a set of web pages served by an NGINX reverse proxy and the backend is a spring boot application running inside a maven environment with an in-memory H2 database.

All the processes were contained utilizing a docker-compose system.

The docker-compose also integrates a SonarQube service that allows for local scans and tests of the full application.



Frontend:

In this configuration, the frontend consists of a set of web pages and javascript files served by an NGINX reverse proxy.

The server is used to handle incoming web requests and route them to the appropriate files.

Backend:

The backend is responsible for processing requests from the client, handling business logic, and interacting with the database.

In this architecture, the backend is implemented using Spring Boot running inside a Maven environment acting as a build automation tool.

The backend utilizes an in-memory H2 database for testing, which should be switched for a normal database in a production environment.

Tomcat is utilized as the servlet container to serve the backend application.

Docker-compose:

Docker-compose is used for defining and running the dockerized applications, allowing an admin to easily manage the entire application stack with a single command.

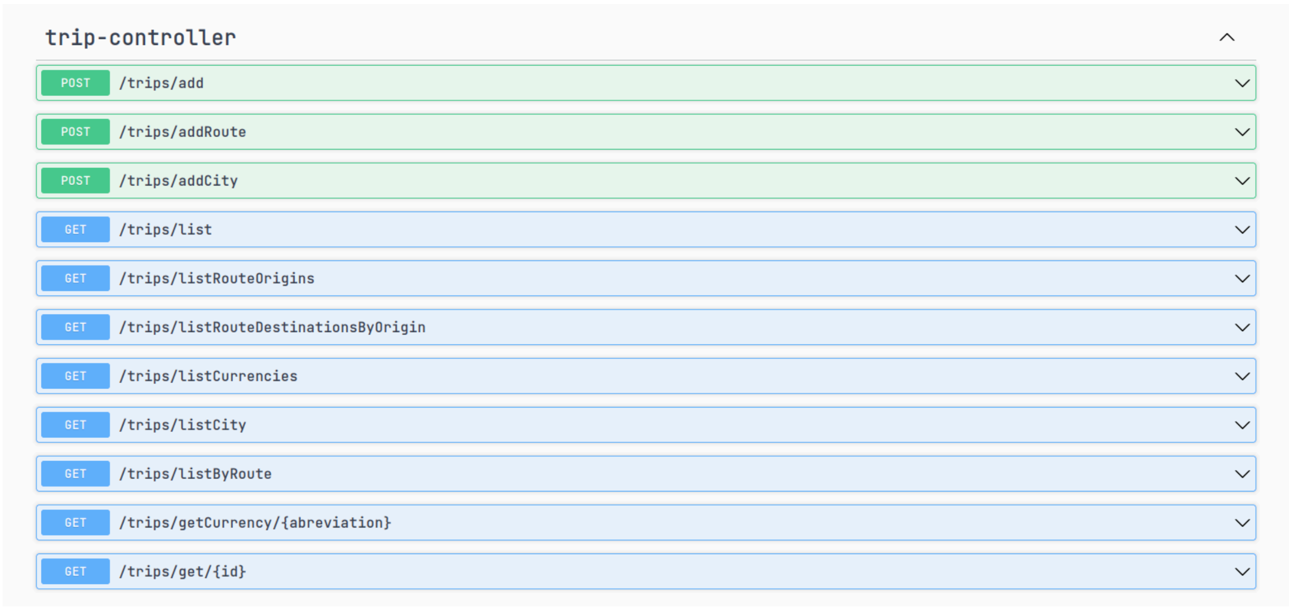
SonarQube:

SonarQube is used to statically analyze code of the backend, find bugs, vulnerabilities, and code smells.
It was set up to perform an automated code review at every docker rebuild and provides actionable insights to improve code quality.

2.3 API for developers

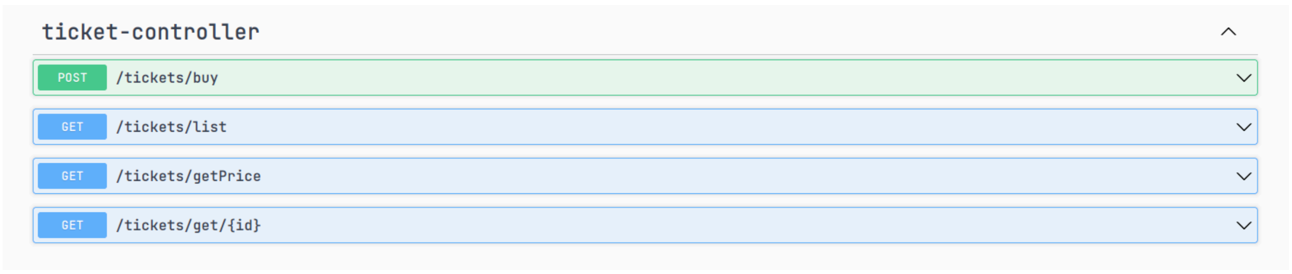
Although the final frontend is limited in functionality, the API itself supports many more features that would be useful in a real-life utilization of our API, such as administrative endpoints, more search functionality and cache usage statistics.

For the surrounding problem domain, a developer can:
- Add Cities, Routes and Trips;
- List Cities, Routes and Trips;
- List all Cities that are origins of at least one Route;
- List all Route destination Cities with a specified origin City ;
- List all currencies
- Search a currency by the abreviation;
- Search for a specific trip information.


trip-controller

POST /trips/add
POST /trips/addRoute
POST /trips/addCity
GET /trips/list
GET /trips/listRouteOrigins
GET /trips/listRouteDestinationsByOrigin
GET /trips/listCurrencies
GET /trips/listCity
GET /trips/listByRoute
GET /trips/getCurrency/{abreviation}
GET /trips/get/{id}

For the primary problem domain, a developer can:
- Purchase a ticket;
- List all purchased tickets;
- Get the estimated price of a ticket based on the input values;
- Get the information of a specific ticket.



ticket-controller

POST /tickets/buy

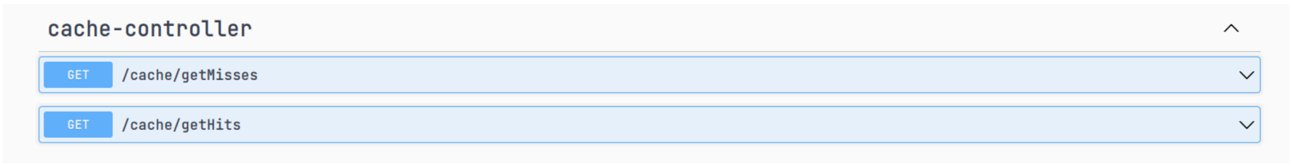
GET /tickets/list

GET /tickets/getPrice

GET /tickets/get/{id}

For cache statistics, a developer can:

- Get the number of cache hits;
- Get the number of cache misses.

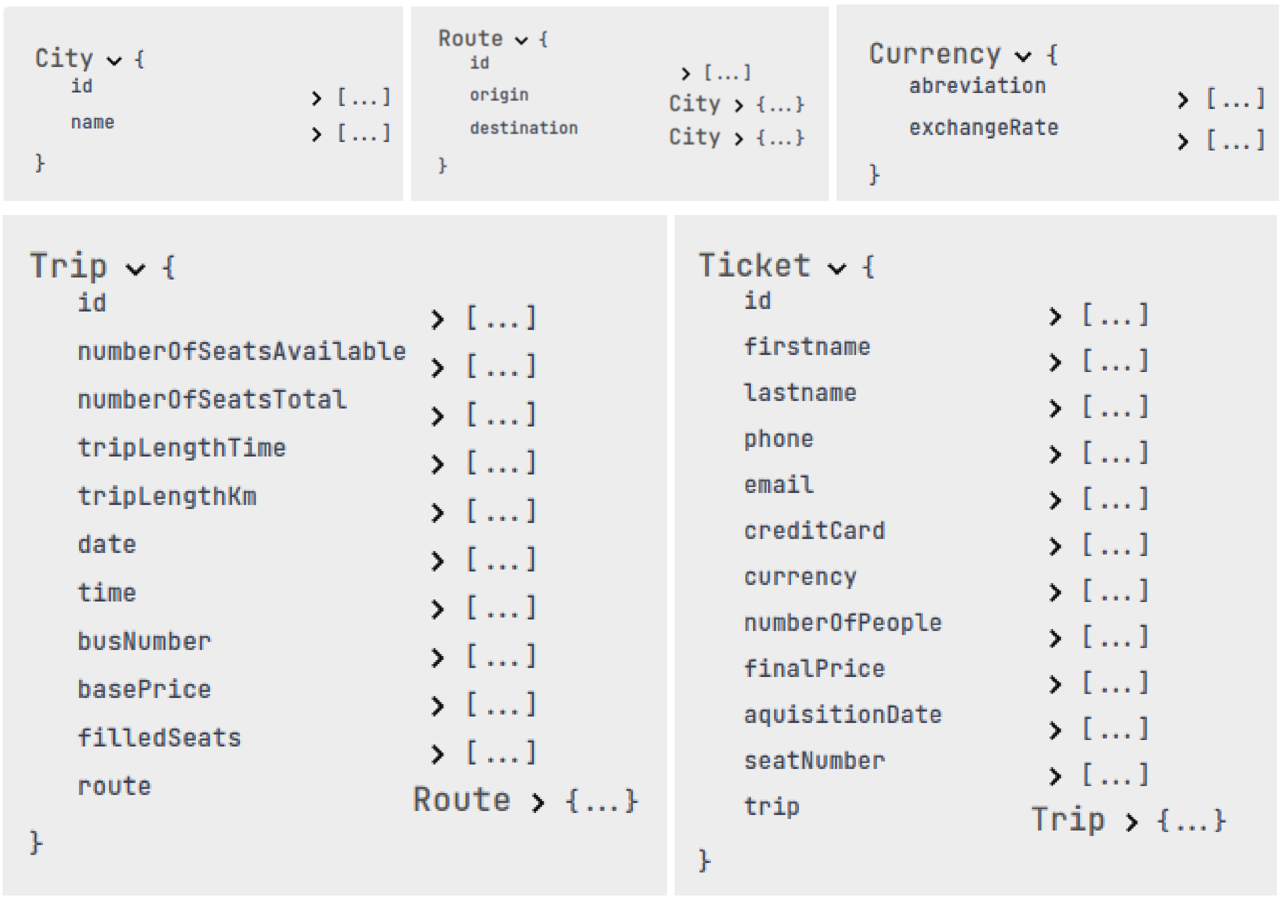


cache-controller

GET /cache/getMisses

GET /cache/getHits

And finally, all of the schemas for the main models are also documented:



```
City ⌄ {
   id                         > [...]
   name                       > [...]
}
```

```
Route ⌄ {
   id                         > [...]
   origin                     City > {...}
   destination                City > {...}
}
```

```
Currency ⌄ {
   abreviation                > [...]
   exchangeRate               > [...]
}
```

```
Trip ⌄ {
   id                         > [...]
   numberOfSeatsAvailable     > [...]
   numberOfSeatsTotal         > [...]
   tripLengthTime             > [...]
   tripLengthKm               > [...]
   date                       > [...]
   time                       > [...]
   busNumber                  > [...]
   basePrice                  > [...]
   filledSeats                > [...]
   route                      Route > {...}
}
```

```
Ticket ⌄ {
   id                         > [...]
   firstname                  > [...]
   lastname                   > [...]
   phone                      > [...]
   email                      > [...]
   creditCard                 > [...]
   currency                   > [...]
   numberOfPeople             > [...]
   finalPrice                 > [...]
   aquisitionDate             > [...]
   seatNumber                 > [...]
   trip                       Trip > {...}
}
```

3 Quality assurance

3.1 Overall strategy for testing

For the initial development of the project, a small portion of the main code (models and simple repositories) was first developed to establish the grounds for the basic functions of the project.
After this, some simple unit tests were developed to ensure that the functionalities of the internal classes created were working as expected, namely for the validators and some of the functions of the models.
Then some of the more simple functionalities of the controllers and services were added with the relevant tests being created after.
For the more complex functions of the controllers, the tests were created alongside the code implementation.
Finally the tests for the cache and integration were created.

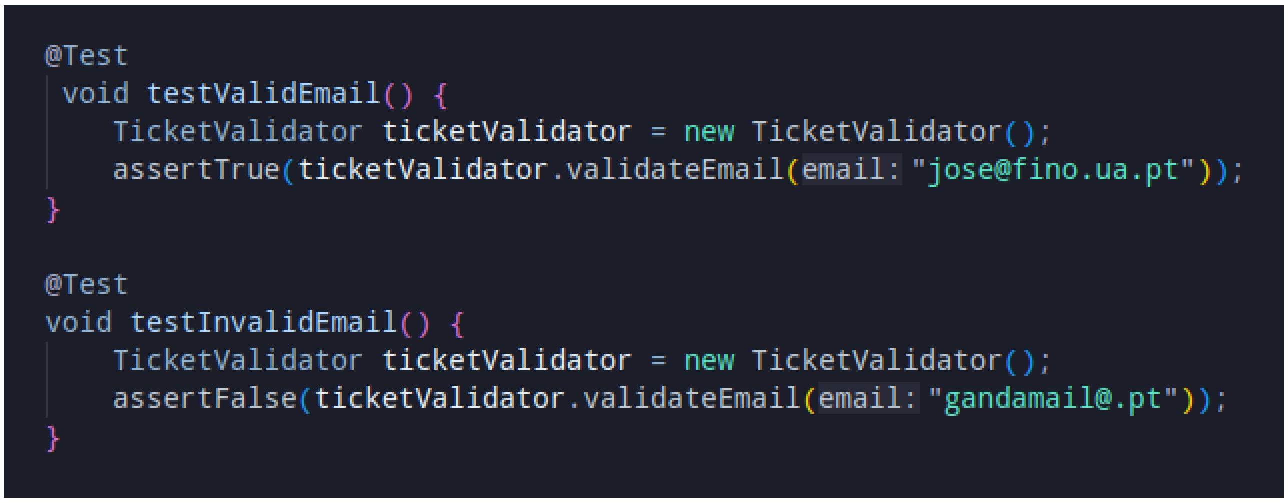
As for the frontend, it was first fully built and then a mixture of Selenium and Cucumber was used to ensure that the user workflow was working correctly and the values displayed in the website were the ones expected.
The Selenium and Cucumber tests also allowed for testing the internal connection between the dockerized images, ensuring that if a connection could not be established, the test would fail.

In conclusion, most of the tests were written right after or alongside the actual code implementation, and all of the complicated and failure-prone functions were tested.

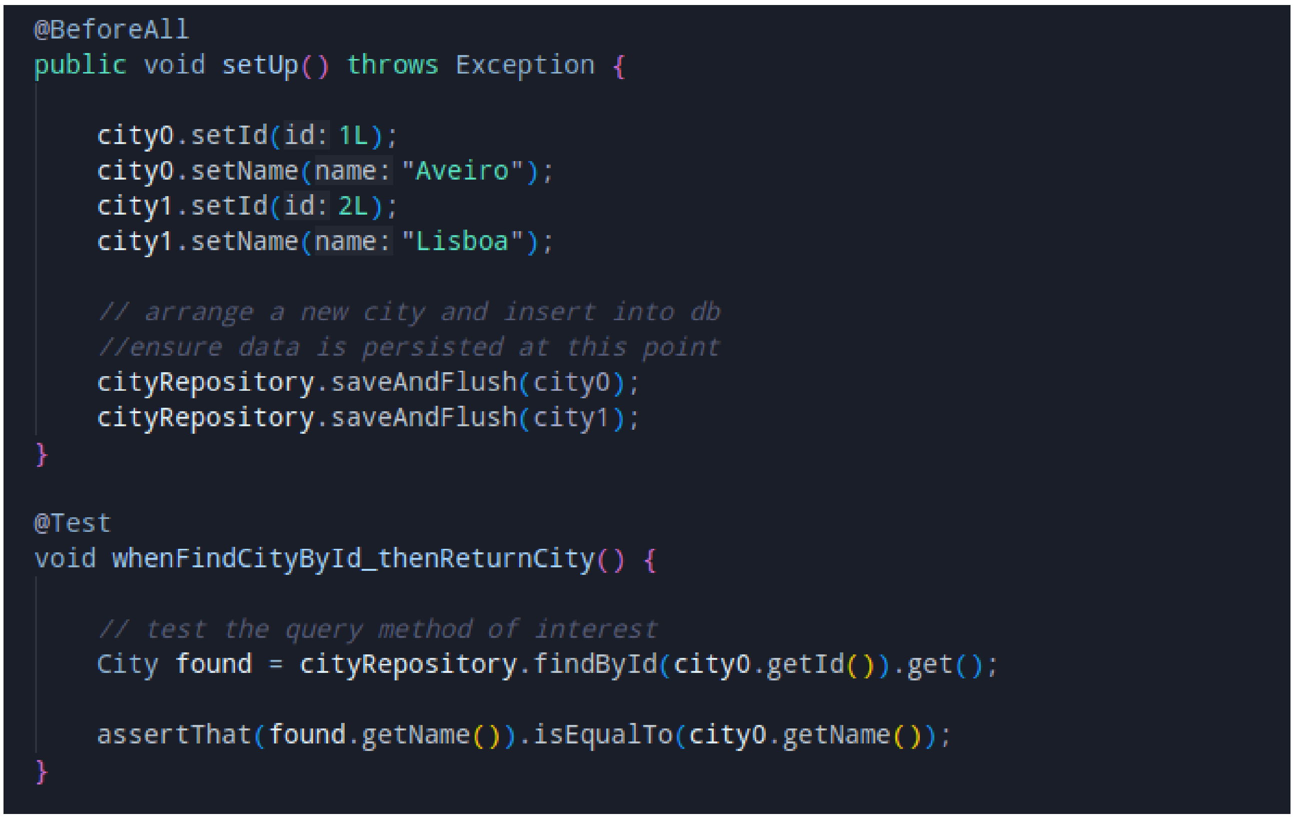
Almost all of the functions and parameter values were tested in order to cover as much of the relevant code branches as possible, leading to a final test count of 79 tests for the whole application.

3.2 Unit and integration testing

The unit testing was used mainly for the testing of “static” functions that do not depend on any other functionalities, such as Validators and repositories.

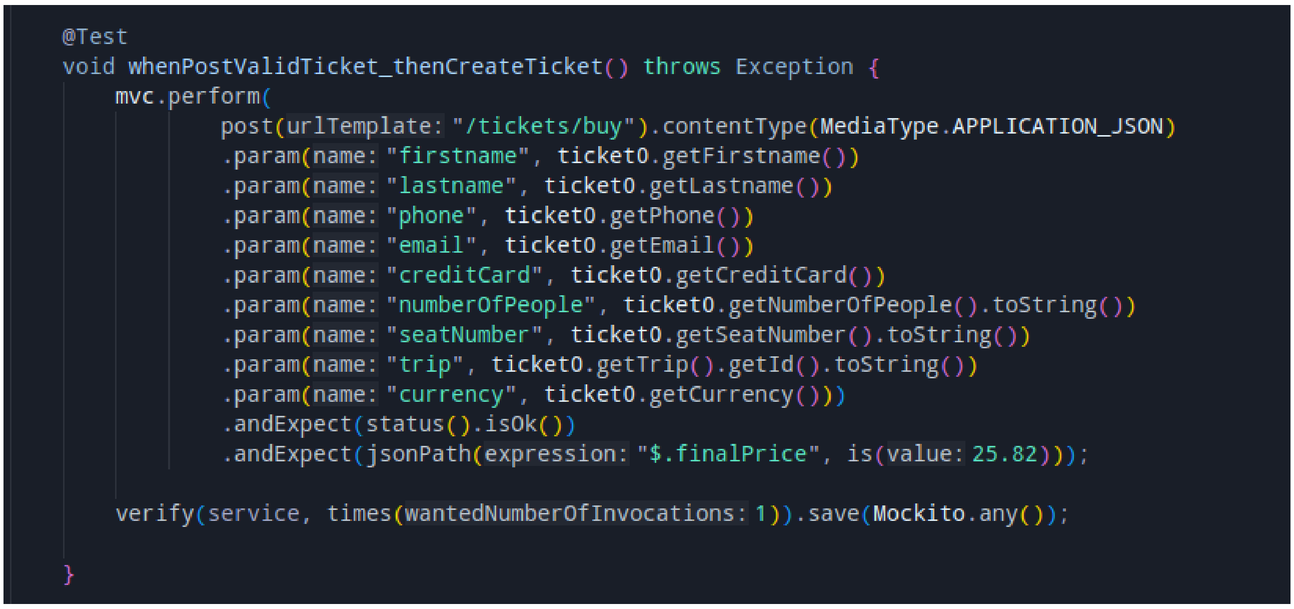


```java
@Test
 void testValidEmail() {
    TicketValidator ticketValidator = new TicketValidator();
    assertTrue(ticketValidator.validateEmail(email: "jose@fino.ua.pt"));
}

@Test
void testInvalidEmail() {
    TicketValidator ticketValidator = new TicketValidator();
    assertFalse(ticketValidator.validateEmail(email: "gandamail@.pt"));
}
```

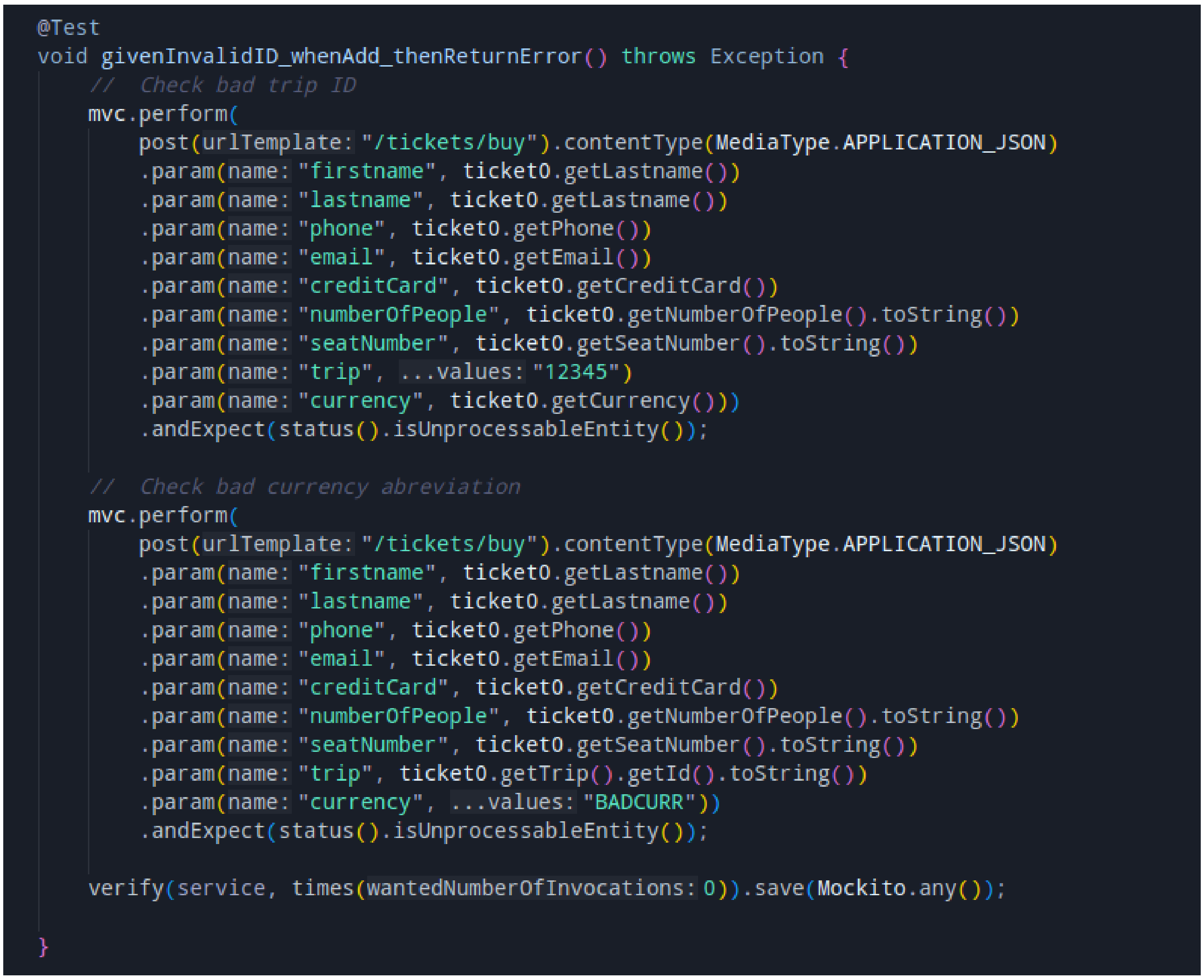


```java
@BeforeAll
public void setUp() throws Exception {

    city0.setId(id: 1L);
    city0.setName(name: "Aveiro");
    city1.setId(id: 2L);
    city1.setName(name: "Lisboa");

    // arrange a new city and insert into db
    //ensure data is persisted at this point
    cityRepository.saveAndFlush(city0);
    cityRepository.saveAndFlush(city1);
}

@Test
void whenFindCityById_thenReturnCity() {

    // test the query method of interest
    City found = cityRepository.findById(city0.getId()).get();

    assertThat(found.getName()).isEqualTo(city0.getName());
}
```

Integration testing was used mostly for testing the responses of controllers, so that all the integrations from all the built classes from top to bottom (controller <-> service <-> repository <-> model) were guaranteed to be working correctly.



```java
@Test
void whenPostValidTicket_thenCreateTicket() throws Exception {
    mvc.perform(
            post(urlTemplate: "/tickets/buy").contentType(MediaType.APPLICATION_JSON)
            .param(name: "firstname", ticket0.getFirstname())
            .param(name: "lastname", ticket0.getLastname())
            .param(name: "phone", ticket0.getPhone())
            .param(name: "email", ticket0.getEmail())
            .param(name: "creditCard", ticket0.getCreditCard())
            .param(name: "numberOfPeople", ticket0.getNumberOfPeople().toString())
            .param(name: "seatNumber", ticket0.getSeatNumber().toString())
            .param(name: "trip", ticket0.getTrip().getId().toString())
            .param(name: "currency", ticket0.getCurrency()))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression: "$.finalPrice", is(value: 25.82)));

    verify(service, times(wantedNumberOfInvocations: 1)).save(Mockito.any());

}
```

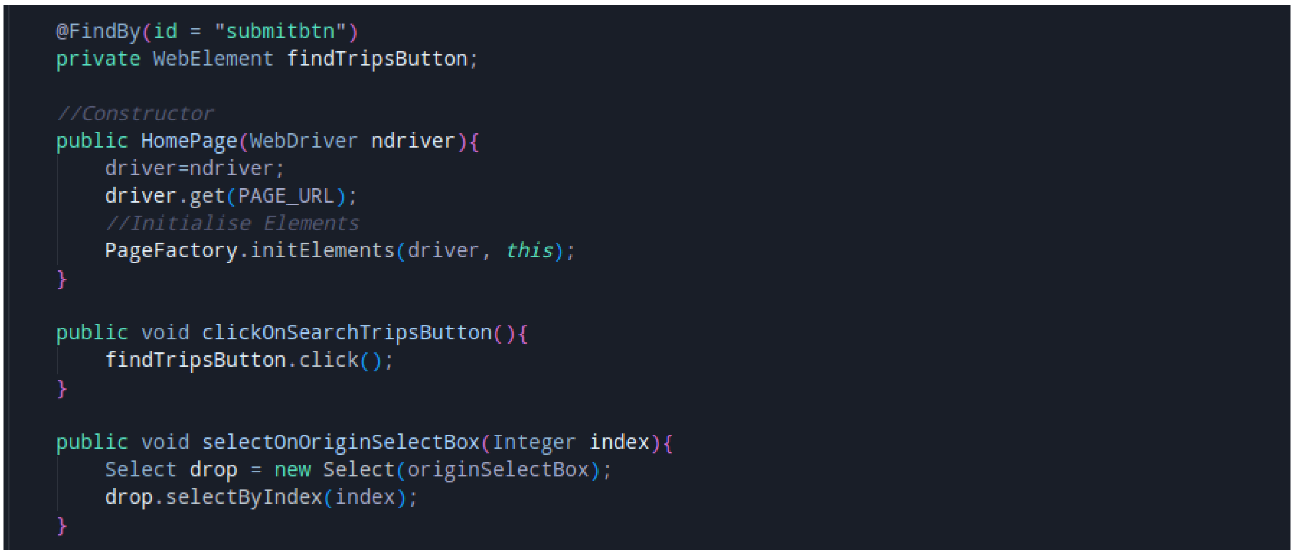

```java
@Test
void givenInvalidID_whenAdd_thenReturnError() throws Exception {
    //  Check bad trip ID
    mvc.perform(
        post(urlTemplate: "/tickets/buy").contentType(MediaType.APPLICATION_JSON)
        .param(name: "firstname", ticket0.getLastname())
        .param(name: "lastname", ticket0.getLastname())
        .param(name: "phone", ticket0.getPhone())
        .param(name: "email", ticket0.getEmail())
        .param(name: "creditCard", ticket0.getCreditCard())
        .param(name: "numberOfPeople", ticket0.getNumberOfPeople().toString())
        .param(name: "seatNumber", ticket0.getSeatNumber().toString())
        .param(name: "trip", ...values: "12345")
        .param(name: "currency", ticket0.getCurrency()))
        .andExpect(status().isUnprocessableEntity());

    //  Check bad currency abreviation
    mvc.perform(
        post(urlTemplate: "/tickets/buy").contentType(MediaType.APPLICATION_JSON)
        .param(name: "firstname", ticket0.getLastname())
        .param(name: "lastname", ticket0.getLastname())
        .param(name: "phone", ticket0.getPhone())
        .param(name: "email", ticket0.getEmail())
        .param(name: "creditCard", ticket0.getCreditCard())
        .param(name: "numberOfPeople", ticket0.getNumberOfPeople().toString())
        .param(name: "seatNumber", ticket0.getSeatNumber().toString())
        .param(name: "trip", ticket0.getTrip().getId().toString())
        .param(name: "currency", ...values: "BADCURR"))
        .andExpect(status().isUnprocessableEntity());

    verify(service, times(wantedNumberOfInvocations: 0)).save(Mockito.any());

}
```

3.3 Functional testing

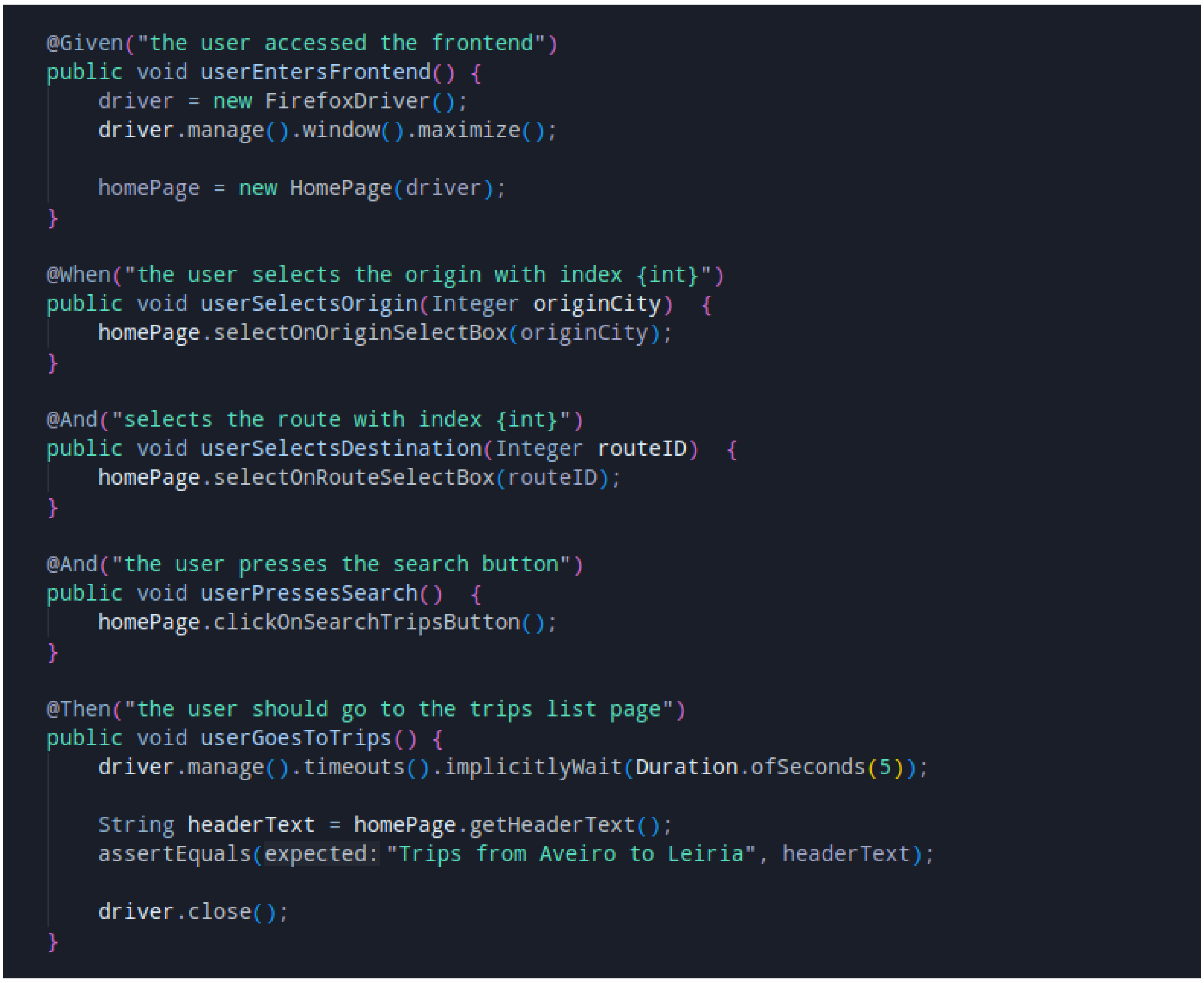
Functional testing of the user side of the application was done with a mixture of Selenium and Cucumber.

Cucumber was used to write reader-friendly tests that also represent a workflow of the user, while Selenium was used to access, manipulate and obtain the data and UI elements present in the webpage.

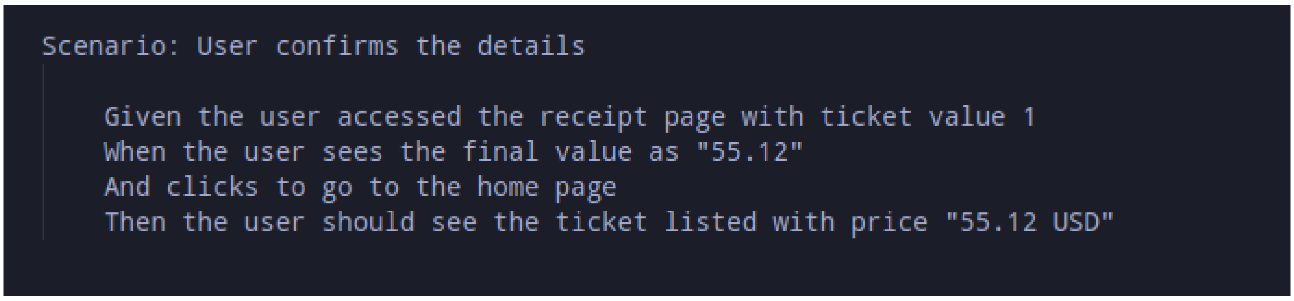
These tests were implemented after the application was created.



```
    @FindBy(id = "submitbtn")
    private WebElement findTripsButton;

    //Constructor
    public HomePage(WebDriver ndriver){
        driver=ndriver;
        driver.get(PAGE_URL);
        //Initialise Elements
        PageFactory.initElements(driver, this);
    }

    public void clickOnSearchTripsButton(){
        findTripsButton.click();
    }

    public void selectOnOriginSelectBox(Integer index){
        Select drop = new Select(originSelectBox);
        drop.selectByIndex(index);
    }
```



```
    @Given("the user accessed the frontend")
    public void userEntersFrontend() {
        driver = new FirefoxDriver();
        driver.manage().window().maximize();

        homePage = new HomePage(driver);
    }

    @When("the user selects the origin with index {int}")
    public void userSelectsOrigin(Integer originCity)  {
        homePage.selectOnOriginSelectBox(originCity);
    }

    @And("selects the route with index {int}")
    public void userSelectsDestination(Integer routeID)  {
        homePage.selectOnRouteSelectBox(routeID);
    }

    @And("the user presses the search button")
    public void userPressesSearch()  {
        homePage.clickOnSearchTripsButton();
    }

    @Then("the user should go to the trips list page")
    public void userGoesToTrips() {
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(5));

        String headerText = homePage.getHeaderText();
        assertEquals(expected: "Trips from Aveiro to Leiria", headerText);

        driver.close();
    }
```



```
Scenario: User confirms the details

    Given the user accessed the receipt page with ticket value 1
    When the user sees the final value as "55.12"
    And clicks to go to the home page
    Then the user should see the ticket listed with price "55.12 USD"
```

3.4 Code quality analysis

The quality analysis of the static code is performed in two ways:

- Locally with SonarQube and Jacoco
  - The docker-compose instance contains a SonarQube service that automatically analyzes and compares the code changes and the full application code against previous runs and a set of coding guidelines.
  - This process saves all the processed data inside the local filesystem.

- Remotely with SonarCloud and Jacoco
  - The github workflow contains a service that automatically analyzes the uploaded code and submits the results to the SonarCloud website, which can be accessed here: https://sonarcloud.io/summary/overall?id=tqs-hw_road-roam

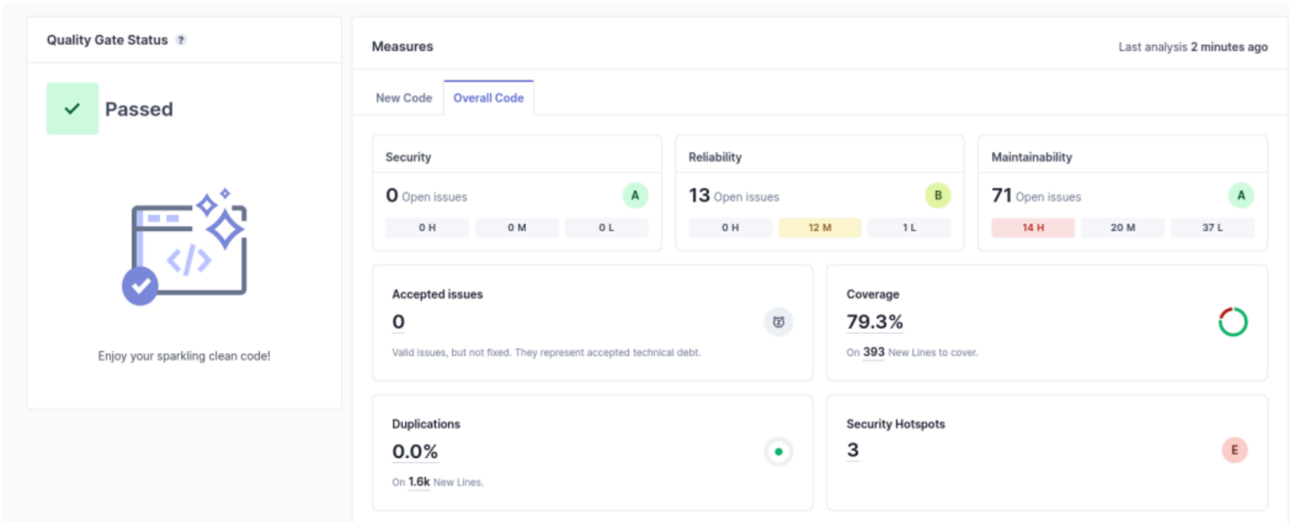
Both methods produce mostly the same results, since the most defining difference between the two services is that one is processed locally while the other is cloud based.

Both analyses were executed multiple times throughout the final stages of development, and the results helped a lot in the creation of more tests for cases and execution branches that were missed.

The initial results showed a large set of open issues and a total coverage of 79.3%, but this percentage was misleading.

Most of the edge-cases, such as weird parameter inputs, were not tested but since they were implemented along side a lot of other functions, they would not affect the coverage by much.

This means that despiste obtaining a good-looking coverage score, some of the most critical functions had potentially dangerous execution branches that were left uncovered, and could lead to serious security and service problems in a real-world application.


Quality Gate Status

Passed

Enjoy your sparkling clean code!

Measures — Last analysis 2 minutes ago

New Code | Overall Code

Security: 0 Open issues — A — 0 H, 0 M, 0 L

Reliability: 13 Open issues — B — 0 H, 12 M, 1 L

Maintainability: 71 Open issues — A — 14 H, 20 M, 37 L

Accepted issues: 0 — Valid issues, but not fixed. They represent accepted technical debt.

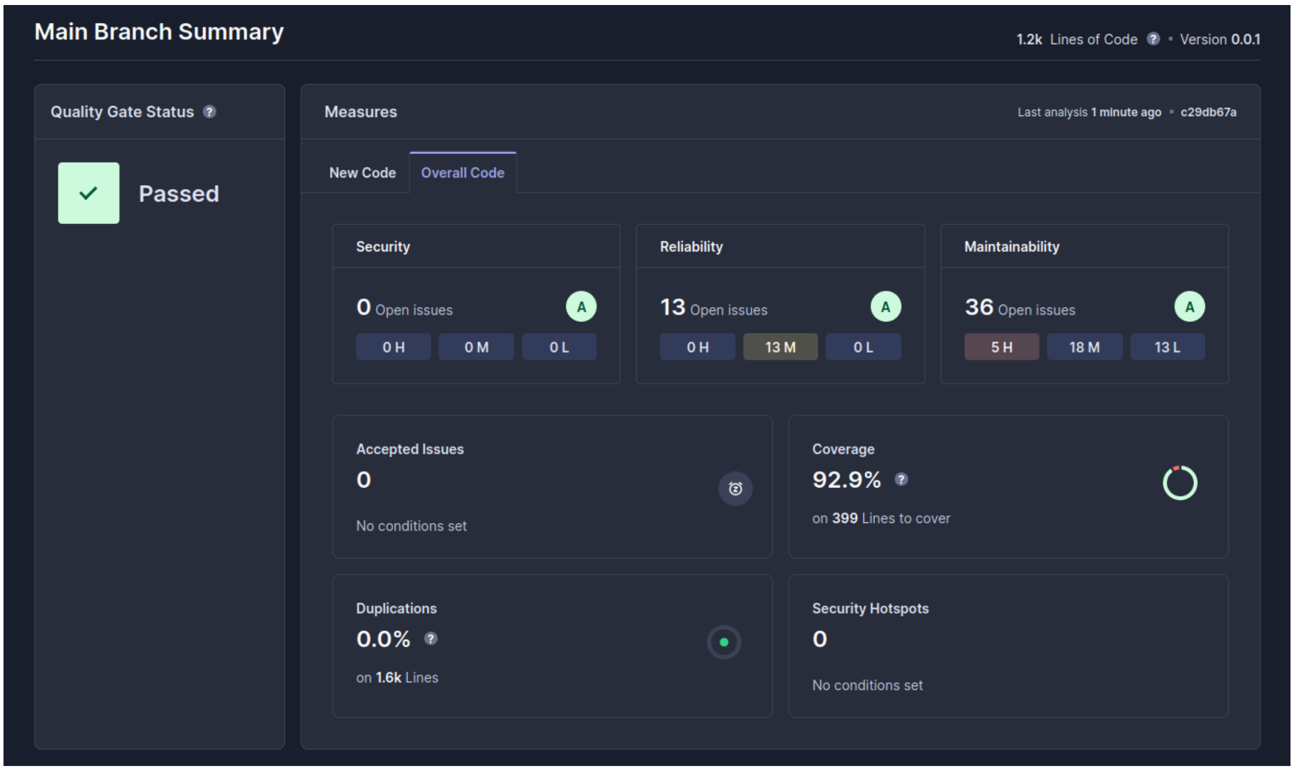
Coverage: 79.3% — On 393 New Lines to cover.

Duplications: 0.0% — On 1.6k New Lines.

Security Hotspots: 3 — E

After some iterations, the new code coverage results presented a lot less open issues, as well as a noticeably bigger coverage percentage, but most importantly, all of the weird edge-cases were now fully covered by the tests implemented.

Some issues such as having the package name contain a number or having non-static and non-final class variables (code smell) without getters and setters were not as severe as other issues such as having classes without any tests.


Main Branch Summary

1.2k Lines of Code · Version 0.0.1

Quality Gate Status: Passed

Measures — Last analysis 1 minute ago · c29db67a

New Code | Overall Code

Security: 0 Open issues (A) — 0 H, 0 M, 0 L

Reliability: 13 Open issues (A) — 0 H, 13 M, 0 L

Maintainability: 36 Open issues (A) — 5 H, 18 M, 13 L

Accepted Issues: 0 — No conditions set

Coverage: 92.9% on 399 Lines to cover

Duplications: 0.0% on 1.6k Lines

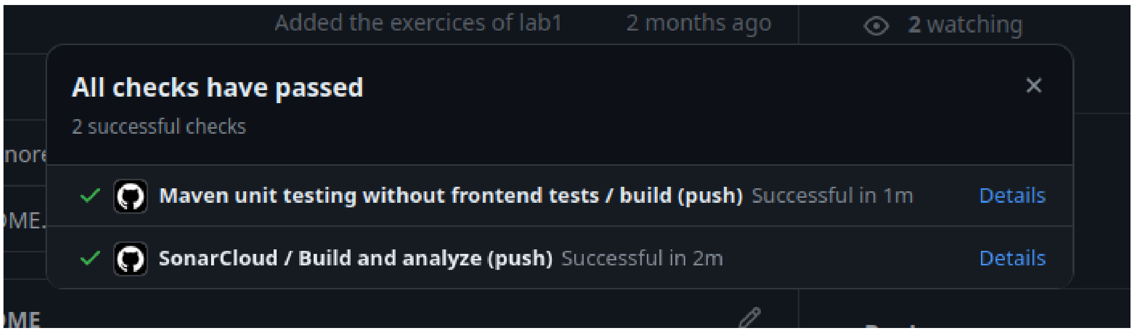
Security Hotspots: 0 — No conditions set

3.5 Continuous integration pipeline

For the purposes of automating unit tests and the code analysis tool, a github CI pipeline was created using GitHubs Actions, allowing these tests to be executed as soon as the new code is pushed/merged into the main branch of the repository.

The implemented workflows were:

- **Unit Tests**: The CI pipeline includes steps to run automated unit tests for the backend components. This ensures that any code changes do not introduce regressions and maintain the expected functionality of the application.
- **Code Analysis with SonarCloud**: Another step in the pipeline involves triggering code analysis using SonarCloud. This step scans the codebase for potential bugs, security vulnerabilities, and code smells, providing valuable insights to improve code quality.



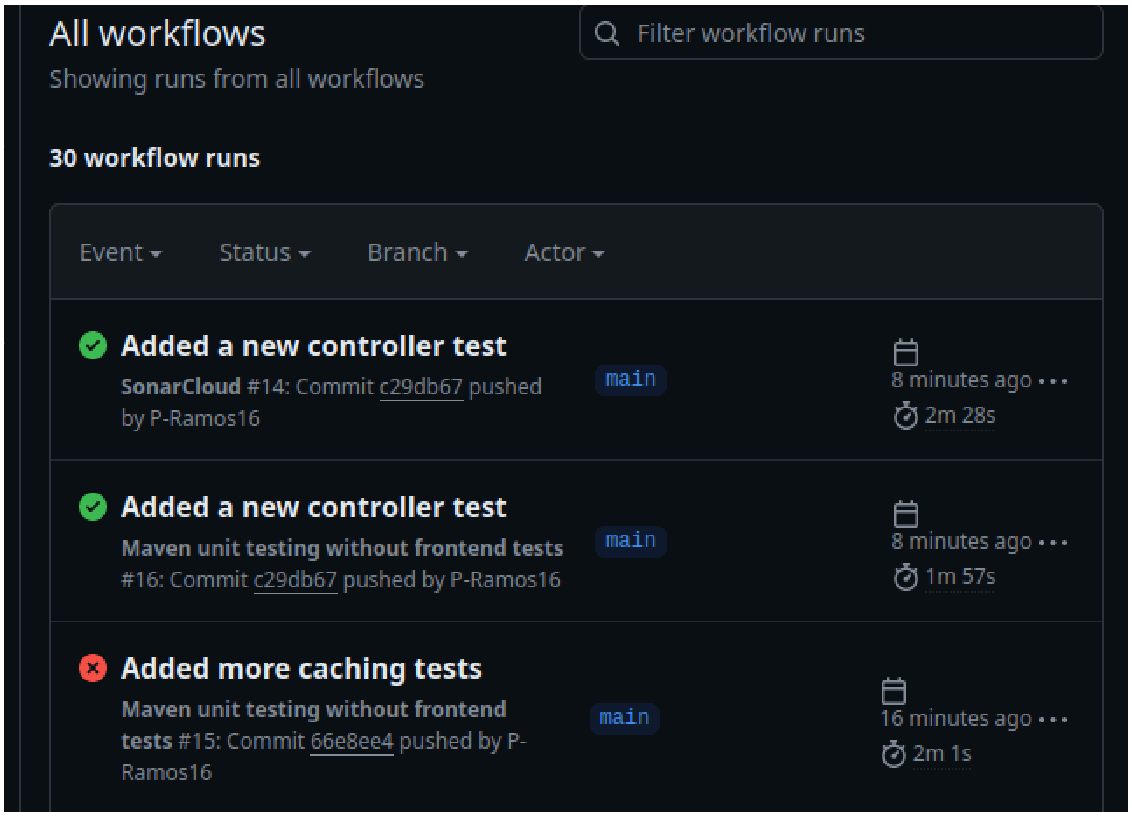
Added the exercices of lab1 2 months ago 2 watching

All checks have passed

2 successful checks

✓ Maven unit testing without frontend tests / build (push) Successful in 1m Details

✓ SonarCloud / Build and analyze (push) Successful in 2m Details

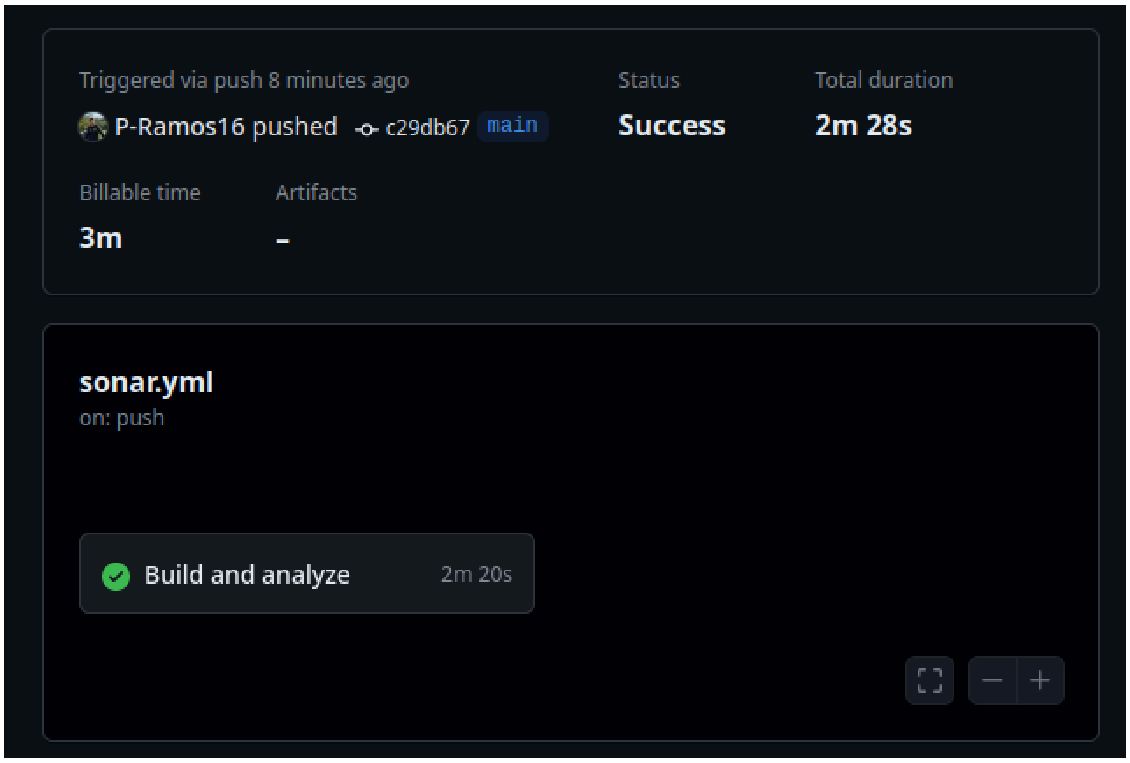

All workflows
Showing runs from all workflows

30 workflow runs

✓ **Added a new controller test**
SonarCloud #14: Commit c29db67 pushed by P-Ramos16 — main — 8 minutes ago — 2m 28s

✓ **Added a new controller test**
Maven unit testing without frontend tests #16: Commit c29db67 pushed by P-Ramos16 — main — 8 minutes ago — 1m 57s

✗ **Added more caching tests**
Maven unit testing without frontend tests #15: Commit 66e8ee4 pushed by P-Ramos16 — main — 16 minutes ago — 2m 1s


Triggered via push 8 minutes ago
P-Ramos16 pushed c29db67 main

Status: Success
Total duration: 2m 28s
Billable time: 3m
Artifacts: –

sonar.yml
on: push

✓ Build and analyze 2m 20s

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/P-Ramos16/TQS_107348
Video demo	< short video demonstration of your solution; consider including in the Git repository>

QA dashboard (online)	https://sonarcloud.io/summary/overall?id=tqs-hw_road-roam
CI/CD pipeline	https://github.com/P-Ramos16/TQS_107348/tree/main/.github/workflows

Reference materials

General Springboot examples:
- https://www.baeldung.com/spring-boot

Currency API:
- https://www.exchangerate-api.com/

SpringDoc:
- https://www.baeldung.com/spring-rest-openapi-documentation

Selenium WebDriver:
- https://www.selenium.dev/documentation/webdriver/

Cucumber Browser Automation:
- https://cucumber.io/docs/guides/browser-automation/?lang=java

SonarQube:
- https://www.baeldung.com/sonar-qube

CI Github maven test workflow:
- https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven

CI Github SonarCloud:
- https://github.com/SonarSource/sonarcloud-github-action