

# HW1: Mid-term assignment report

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



<b>1 Introduction</b>	<b>2</b>
1.1 Overview of the work	2
1.2 Current limitations	2
<b>2 Product specification</b>	<b>3</b>
2.1 Functional scope and supported interactions	3
2.2 System architecture	5
2.2.a Frontend	5
2.2.b Backend	5
2.2.c Docker-compose	6
2.2.d SonarQube:	6
2.3 API for developers	6
2.3.a Problem Domain - Trips, Routes, Cities and Currencies	6
2.3.b Problem Domain - Tickets	7
2.3.c Problem Domain - Cache Statistics	7
2.3.d Problem Domain - Schemas	7
<b>3 Quality assurance</b>	<b>8</b>
3.1 Overall strategy for testing	8
3.2 Unit and integration testing	9
3.2.a Unit Testing	9
3.2.b Integration Testing	10
3.3 Functional testing	11
3.4 Code quality analysis	13
3.4.a Locally	13
3.5 Continuous integration pipeline	16
<b>4 References &amp; resources</b>	<b>18</b>

# 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 **SonarQube**, **SonarCloud** and **Github Actions** were used to implement automatic testing, code coverage checks and overall static code quality evaluations.

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. A feature that allows **searching for trips by the date** and time of departure would also be a good improvement to the project.

**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 the tickets with his information.

#### Issues:

- Navigate the interface for a smooth booking process;
- Assure that the backend is optimized for the task flow and provides all the features required 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 for the selected route 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, updating the seats in the bus.

9. The user can view the acquired ticket in the application for reference and boarding.

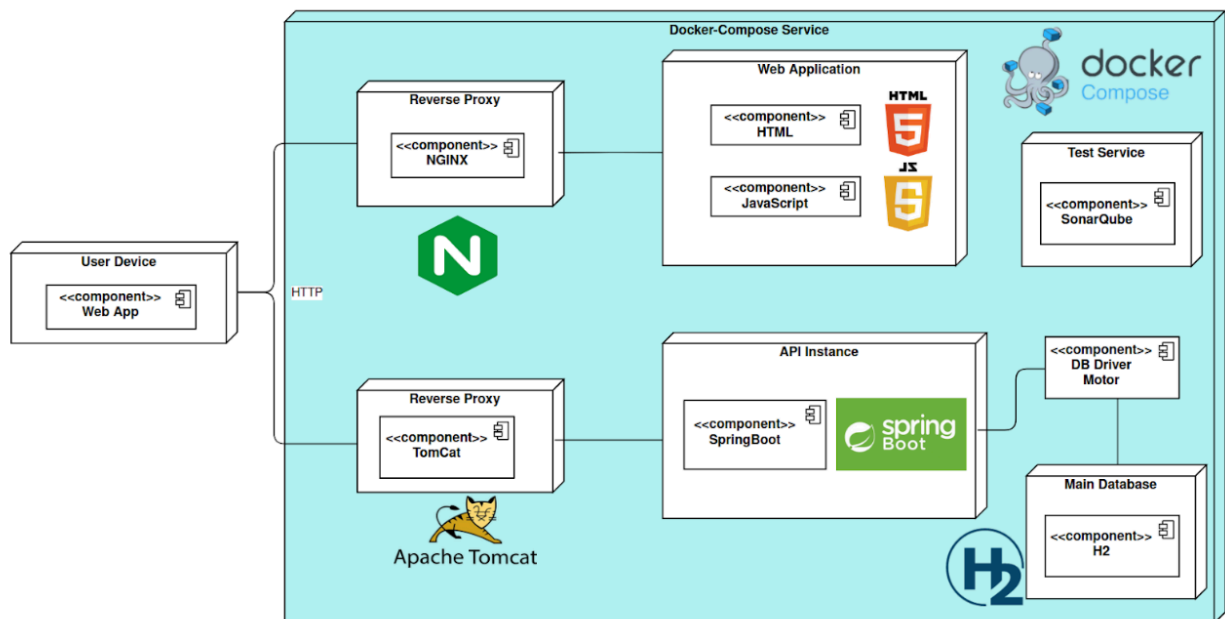
## 2.2 System architecture

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.

A **H2** in-memory database was added to allow the persistence of system data, but the code for an easy switch to an MySQL database is also included.

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

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



### 2.2.a 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.

### 2.2.b 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 that is set up as a **build automation tool**.

The backend utilizes an in-memory **H2** database for testing, which could be switched for a normal database (ex: MySQL) in a production environment.

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

## 2.2.c 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.

## 2.2.d 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.

### 2.3.a Problem Domain - Trips, Routes, Cities and Currencies

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 abbreviation;
- **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/{abbreviation}	▼
GET	/trips/get/{id}	▼



### 2.3.b Problem Domain - Tickets

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}	▼

### 2.3.c Problem Domain - Cache Statistics

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	▼

### 2.3.d Problem Domain - Schemas

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

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

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

Currency > {
abbreviation > [...]
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**, simple **repositories** and **validators**) was first developed to establish the **grounds** for the basic functions of the project.

After this, some **simple unit tests** were developed with **JUnit 5** 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**. This early testing also helped guarantee that all of the libraries and code used in these tests was functioning without any issues.

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.

These controller tests were implemented using the **MockMvc** library to allow the simulation of real-world requests and the service tests were made using the **Mockito** library to mock the functionality of the repositories.

Finally the tests for the **cache** and **system 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 that the values displayed in the website were the ones **expected**. This means that the frontend tests took advantage of the BDD principles for creating easy to read and 'use case focused' tests.

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 combinations** were tested in order to **cover as much of the relevant code branches as possible**, leading to a final test count of **89 tests** for the whole application and a final coverage of over **93%**.

The tests themselves were written with **different libraries and testing ideologies**, to better utilize the strengths of each component for all the categories of tests and also allowing a more **diverse testing experience**.

## 3.2 Unit and integration testing

### 3.2.a Unit 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**.

```
@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"));
}
```

```
@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());
}
```

### 3.2.b Integration Testing

**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**.

```
@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());
}
```

```

@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 abbreviation
    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**, utilizing the **BDD** principles.

This means that the **Cucumber** library was used to write **reader-friendly tests** that also textually 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:

#### 3.4.a Locally

- 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**;
- These actions are executed every time the backend **docker image is rebuilt** locally.

#### 3.4.b Remotely

- 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](https://sonarcloud.io/summary/overall?id=tqs-hw_road-roam);
- These actions are executed every time the **main branch** of the repository is updated via a **push** or a **merge**.

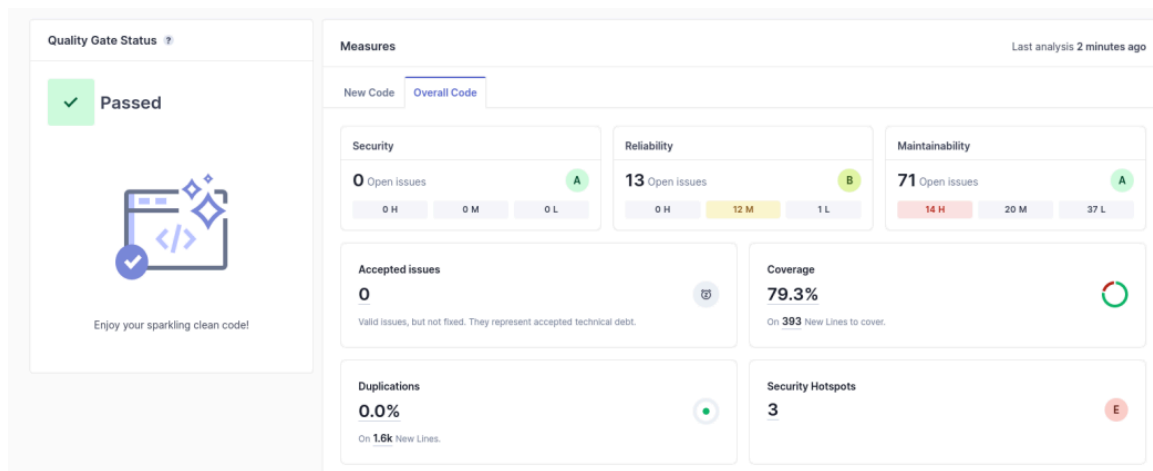
Both methods produce **mostly the same results**, since the most defining difference between the two services is that one is processed locally while the other is cloud based.

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** (over 70) and a total coverage of **79.3%**, but this percentage was misleading.

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

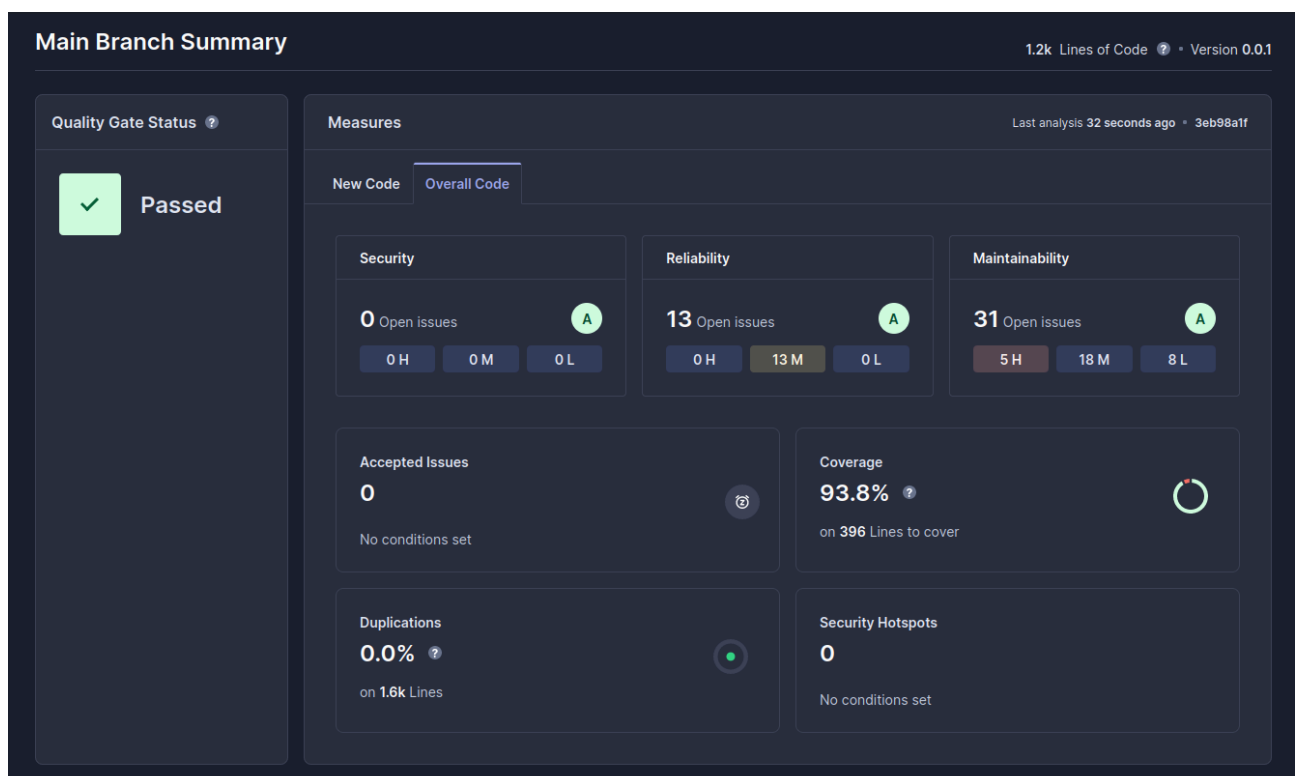
This means that despite obtaining a **good-looking coverage score**, some of the most **critical functions** had potentially dangerous execution branches that were **left uncovered**, and could lead to serious security and service problems in a real-world application.



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 other small **code smells** were considered not as severe as some of the other issues found, such as having whole complex functions **without any tests**.

This is why some issues were **left open**, as they do not really impact the quality of the code and other more critical issues were instead chosen to be fixed.




















This analysis allowed a better understanding of the **total coverage of all the classes** and provided a very easy way to check the **quality of all old and new code** committed to the main branch.

With **SonarCloud** we can easily navigate through all the application classes and decide if a specific condition that is missing tests is **worth testing** or not (most are).

This means that **after the first execution** of the analysis, we easily identified the **tests that needed to be written** and the **critical issues** that needed to be fixed.

Coverage 93.8% [See history](#) New code: since previous version

	Coverage	Uncovered Lines	Uncovered Conditions
 src/main/java/hw/roadroam/services/CurrencyService.java	89.5%	5	3
 src/main/java/hw/roadroam/services/TripService.java	90.0%	1	0
 src/main/java/hw/roadroam/controllers/TripController.java	90.4%	5	8
 src/main/java/hw/roadroam/controllers/TicketController.java	92.7%	1	7
 src/main/java/hw/roadroam/SimpleCORSFilter.java	93.3%	1	–
 src/main/java/hw/roadroam/services/TicketValidator.java	95.0%	0	1
 src/main/java/hw/roadroam/controllers/CacheController.java	100%	0	–
 src/main/java/hw/roadroam/models/City.java	100%	0	–
 src/main/java/hw/roadroam/services/CityService.java	100%	0	0
 src/main/java/hw/roadroam/models/Currency.java	100%	0	–
 src/main/java/hw/roadroam/models/Route.java	100%	0	–
 src/main/java/hw/roadroam/services/RouteService.java	100%	0	0
 src/main/java/hw/roadroam/models/Ticket.java	100%	0	–
 src/main/java/hw/roadroam/services/TicketService.java	100%	0	0
 src/main/java/hw/roadroam/models/Trip.java	100%	0	0

Most of the **issues left** in the code pertained to the usage of '@Autowired' instead of a constructor injector, and most of the untested execution branches are **simple functions** that should not miss and **spring-generated code** that should not break.

```

36
37 p.ramo...    @Autowired
38              private TicketService ticketService;
39              @Autowired
40              private TripService tripService;
41              @Autowired
               Remove this field injection and use constructor injection instead.
42              private CurrencyService currencyService;
43 p.ramo...
44 p.ramo...    private static TicketValidator tValidator = new TicketValidator();
45 p.ramo...    private static final Logger logger = LoggerFactory.getLogger(TicketController.class);
46 p.ramo...
47 p.ramo...    @PostMapping("/buy")
48              public ResponseEntity<Ticket> buyTicket(@RequestParam String firstname, @RequestParam String lastname,

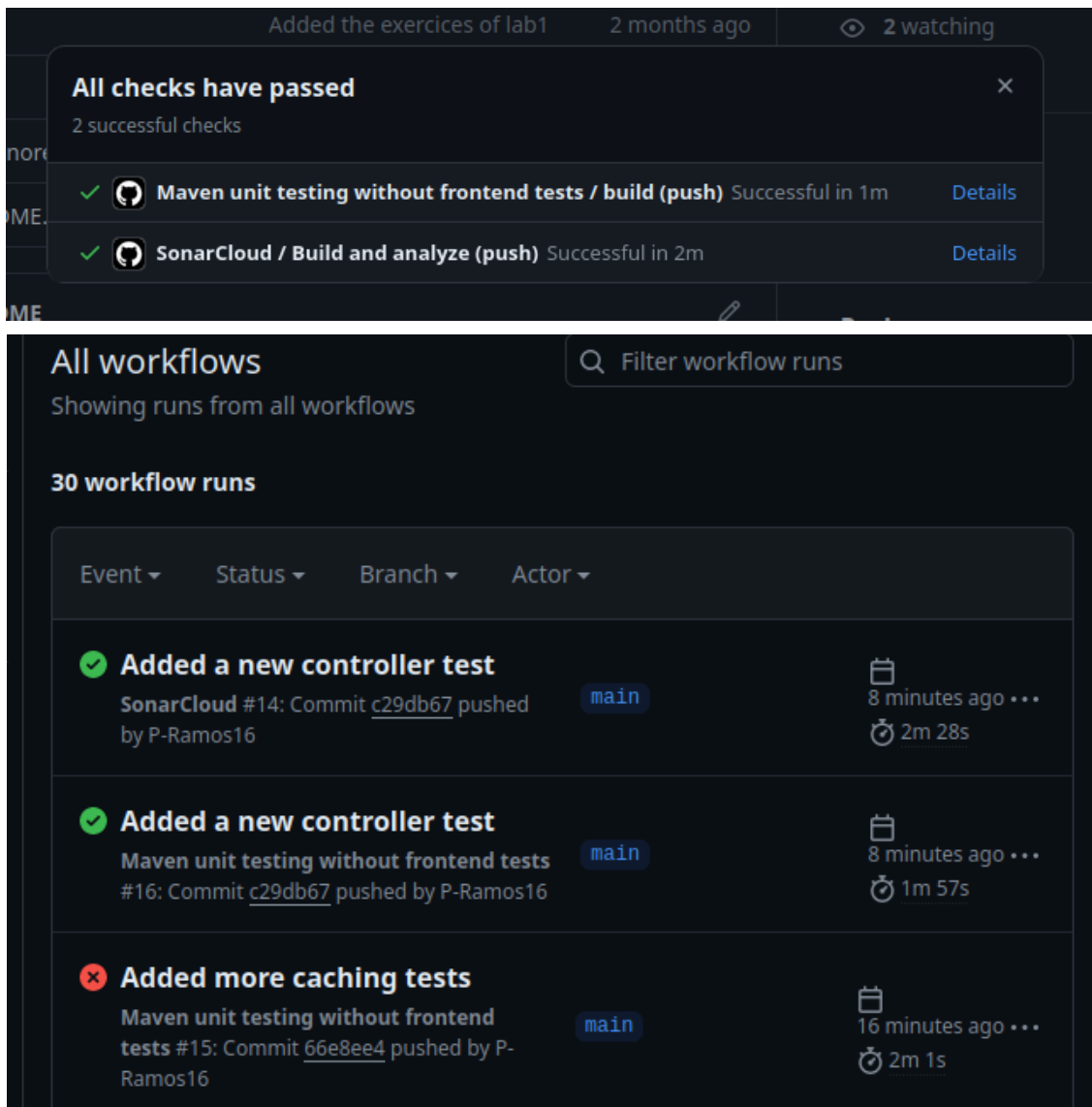
```

### 3.5 Continuous integration pipeline

For the purposes of **automating unit tests** and the **code analysis** tool, a **github CI pipeline** was created using **GitHub 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.





The screenshot displays the GitHub Actions interface. At the top, a notification states "All checks have passed" with "2 successful checks". Below this, two workflow runs are listed: "Maven unit testing without frontend tests / build (push)" and "SonarCloud / Build and analyze (push)", both successful. The main section, "All workflows", shows "30 workflow runs". A table lists the following runs:

Event	Status	Branch	Actor	Run Name	Commit	Pushed by	Time
✓	Success	main	P-Ramos16	Added a new controller test	c29db67	P-Ramos16	2m 28s
✓	Success	main	P-Ramos16	Added a new controller test	c29db67	P-Ramos16	1m 57s
✗	Failure	main	P-Ramos16	Added more caching tests	66e8ee4	P-Ramos16	2m 1s

Triggered via push 8 minutes ago

Status

Total duration

 P-Ramos16 pushed  c29db67 main

**Success**

**2m 28s**

Billable time


Artifacts

**3m**




-

**sonar.yml**

on: push

 **Build and analyze**

2m 20s

## 4 References & resources

### Project resources

Resource:	URL/location:
Git repository	<a href="https://github.com/P-Ramos16/TQS_107348">https://github.com/P-Ramos16/TQS_107348</a>
Video demo	<a href="https://github.com/P-Ramos16/TQS_107348/blob/main/HW1/reports/HW_Video.mp4">https://github.com/P-Ramos16/TQS_107348/blob/main/HW1/reports/HW_Video.mp4</a>
QA dashboard (online)	<a href="https://sonarcloud.io/summary/overall?id=tgs-hw_road-roam">https://sonarcloud.io/summary/overall?id=tgs-hw_road-roam</a>
CI/CD pipeline	<a href="https://github.com/P-Ramos16/TQS_107348/tree/main/.github/workflows">https://github.com/P-Ramos16/TQS_107348/tree/main/.github/workflows</a>

### 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>