deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Ana Alexandra Antunes [876543], v2022-04-07

1 Introduction

1.1 Overview of the work

For the midterm assignment given by our TQS teacher, we were challenged to create an application that facilitates the operations of a bus company. The application is fully functional and ready to be deployed, although with some potential shortcomings. With this application, users can select from available trips between six cities in Europe and book tickets for those buses. Additionally, users can review all the tickets they have purchased by providing the token they received upon booking.

1.2 Current limitations

This application as some limitations that might be able to make it difficult to use. Does not have authentication neither a token based one, so you really have to keep the token to see your tickets again. You cannot re-book a trip or change anything about it, once is done you have no interface to change it, although you can make it through the API. There is also neither driver nor administrator interface, and even if you are a passenger you cannot choose the seat you are sitting on, that is not supported as well

From the requested in the assignment, the work has:

- A REST API that can be invoked invoked by external clients
- A minimalist web app designed in HTML/CSS/JS and Bootstrap CSS that allows the user to interacts
- The cache is also working and being called every 5 minutes (TTL)
- A logger to see the API being called

2 Product specification

2.1 Functional scope and supported interactions

As I said before in the introduction, the functional scope is all about the functionality of a bus company , and this app is made in order to help it run the company. In this app you’re able to see all trips, you’re able to make buy a ticket and you are also able to check the tickets you already bought. The main actor of this application is the passenger, who is the main focus here in our application. Maybe a driver would want to check the tickets so it can also be a actor.

2.2 System architecture

In the system architecture, Spring Boot is utilized for building the application, while Thymeleaf, a templating engine, is employed for creating the frontend, leveraging Spring Boot's seamless support. Within the Spring Boot application, there exist Controller, Service, and Model units crucial for shaping the architecture.

The Controller plays a pivotal role in managing incoming HTTP requests initiated by user interactions with the application. It processes these requests and delivers responses to the requester. Handling the processing aspect are the services within the Spring Boot application. These services house the business logic and act as an intermediary between the Controller, UI, and data access layer, including the models.

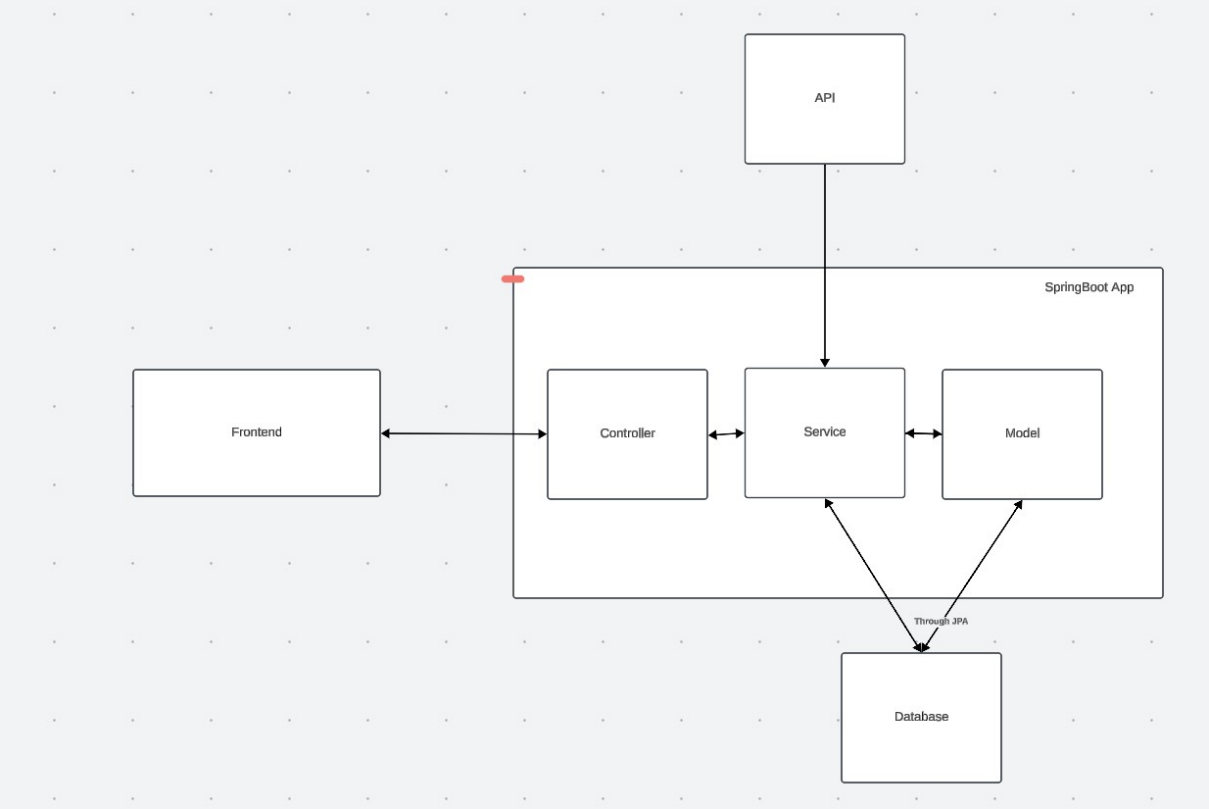
The Service unit assumes significant responsibility, overseeing transformations, validations, data manipulation, and overall organization of the application. It ensures coherence and integrity within the system by orchestrating various operations.

Next in line are the Model units, responsible for enforcing the structure and attributes designated for incoming data. This encapsulation of data ensures adherence to predefined schemas before storing it in the H2 database, managed

by Spring Boot. The JPA Entity facilitates interaction with the database, acting as a gateway for accessing and manipulating data.

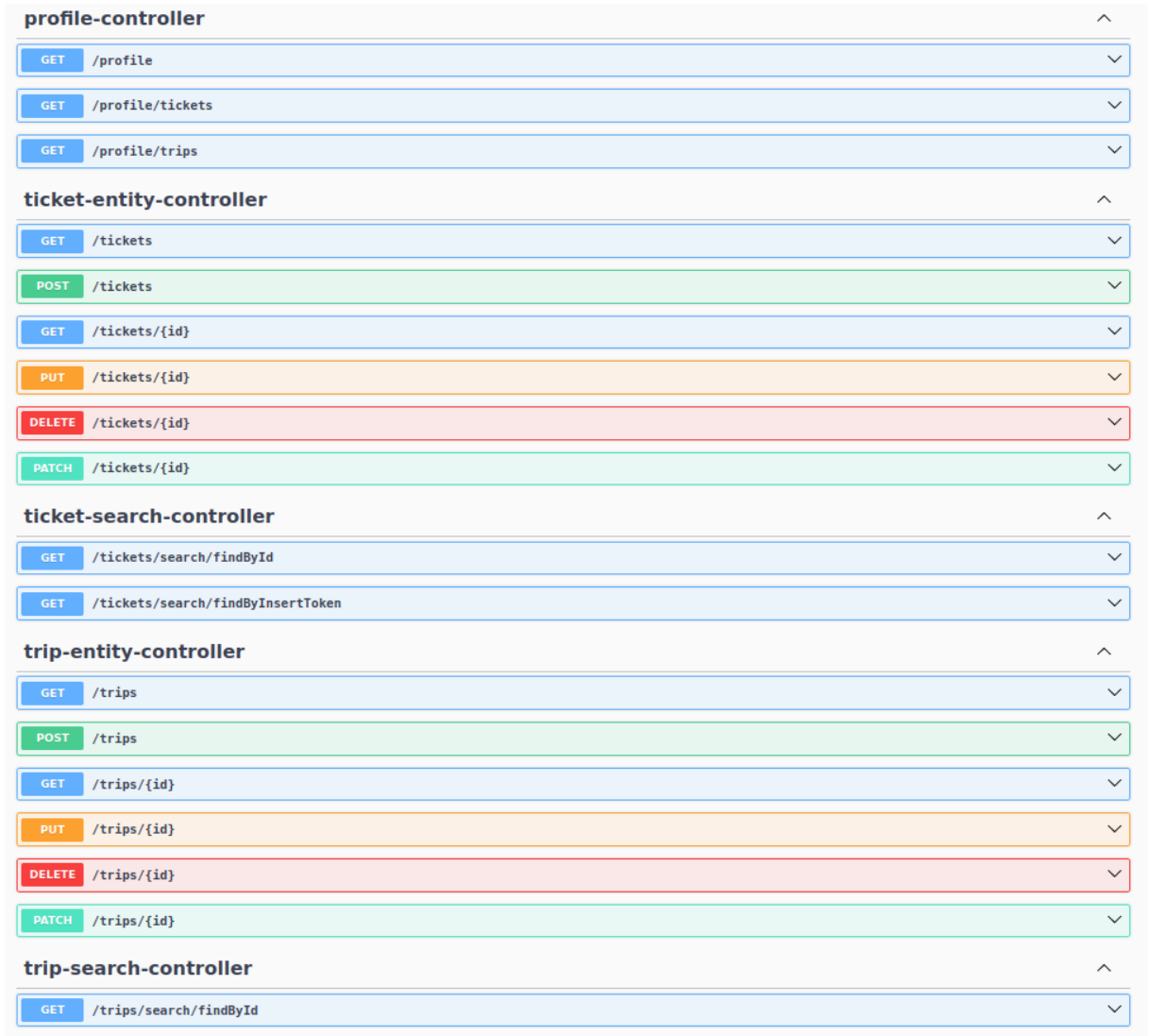
For handling exchange rates, the system leverages a free API named Frankfurter, allowing unrestricted requests without the need for tokens or authentication. These requests are made by a cache service at regular intervals of five minutes to ensure the currency rates remain up-to-date. The cache mechanism employed operates within the Spring Boot main web service, optimizing performance and reliability.

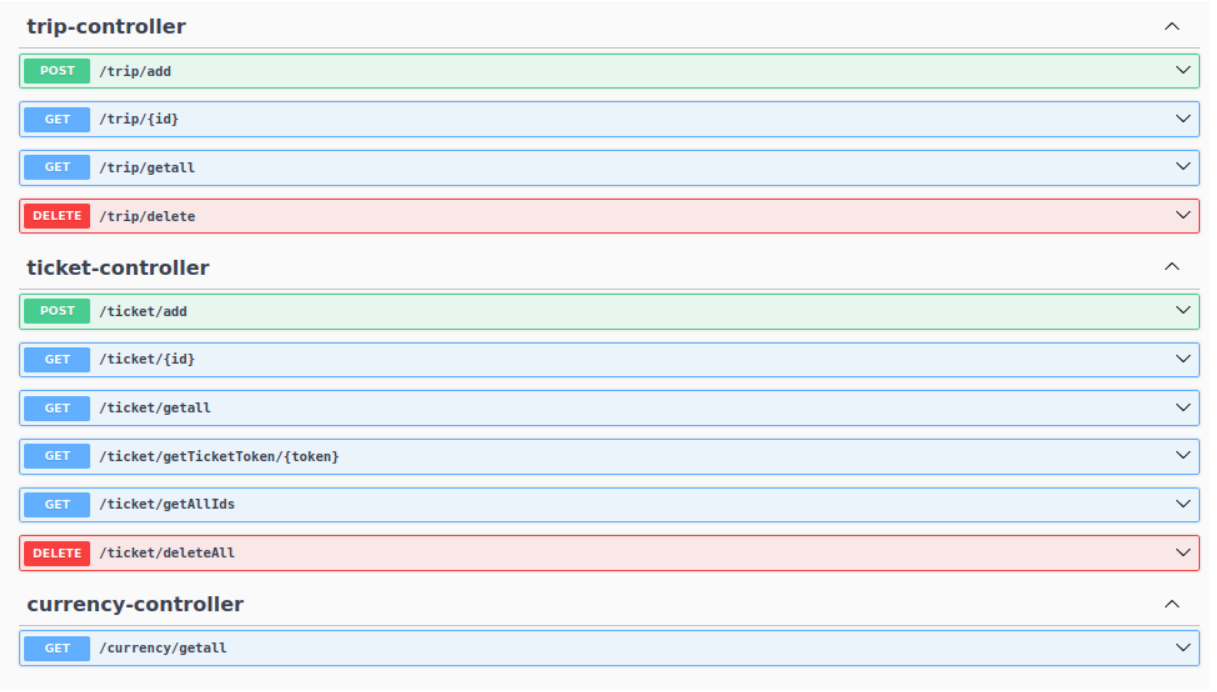
In the end is all supported by a docker, and is configured to localhost:8080.

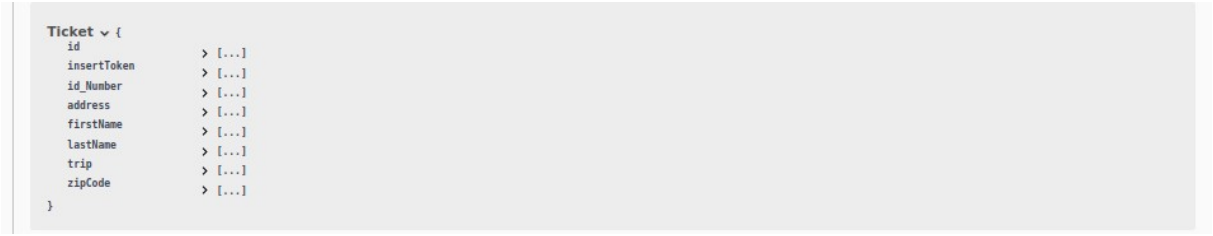


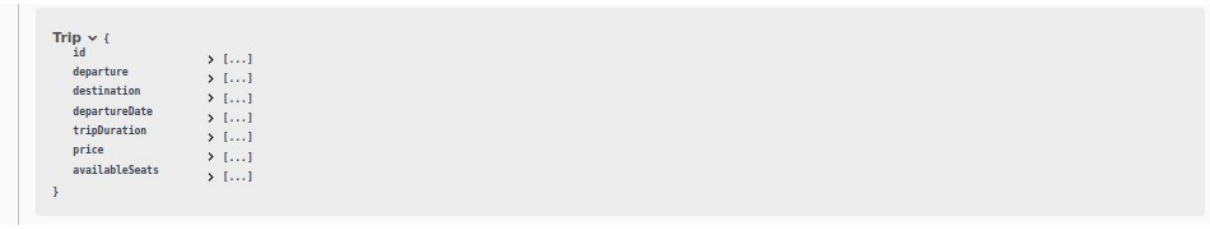
2.3 API for developers

Using Swagger, I was able to get all the schemas and endpoints of my API, here are the most important:


profile-controller
GET /profile
GET /profile/tickets
GET /profile/trips
ticket-entity-controller
GET /tickets
POST /tickets
GET /tickets/{id}
PUT /tickets/{id}
DELETE /tickets/{id}
PATCH /tickets/{id}
ticket-search-controller
GET /tickets/search/findById
GET /tickets/search/findByInsertToken
trip-entity-controller
GET /trips
POST /trips
GET /trips/{id}
PUT /trips/{id}
DELETE /trips/{id}
PATCH /trips/{id}
trip-search-controller
GET /trips/search/findById


trip-controller
POST /trip/add
GET /trip/{id}
GET /trip/getall
DELETE /trip/delete
ticket-controller
POST /ticket/add
GET /ticket/{id}
GET /ticket/getall
GET /ticket/getTicketToken/{token}
GET /ticket/getAllIds
DELETE /ticket/deleteAll
currency-controller
GET /currency/getall





3 Quality assurance

3.1 Overall strategy for testing

The development strategy I employed involved Test-Driven Development (TDD) for both the service and controller components of the API. These unit tests were meticulously designed to guide the construction of the application, ensuring that each piece of functionality was thoroughly tested and met the specified requirements.

In addition to TDD, I utilized Selenium to facilitate testing of the frontend developed for this project. To achieve this, I first executed Maven to run the application on localhost:8080. While I could have utilized the Spring Boot AppContext, I opted for running a separate Spring Boot instance for each Selenium test to avoid potential conflicts.

Cucumber played a vital role in testing the cache functions from a user's perspective. I was able to create clear and concise test scenarios that accurately reflected user interactions and expectations.

In total, the project comprises 40 tests, each crafted and validated to ensure correctness and reliability throughout the development process.

3.2 Unit and integration testing

Talking about this unit testing on this application. Unit testing is a very important tool to test the isolation and each component of the system by itself. Mine are fast executing and all running correctly. Starting with repository, this tests uses assertions to make sure the repository is keeping everything in the database.

To give the example we will use the **TicketRepositoryTest** example.

Here we need put @DataJpaTest to test the JPA Entity

```
@DataJpaTest
public class TicketRepositoryTest {

    @Autowired
    private TicketRepository ticketRepository;
```

These is the part where I setup the the ticket in the repository:

```
@BeforeEach
public void setUp() {
    Ticket ticket = new Ticket();
    ticket.setAddress(address:"Avenida 25 de Abril, Alijó");
    ticket.setFirstName(firstName:"Gonçalo");
    ticket.setLastName(lastName:"Ferreira");
    ticket.setId_Number(id_Number:123456789L);
    ticket.setZipCode(zipCode:"5070-011");
    ticket.setTrip(trip:1L);
    ticket.setInsertToken(insertToken:"T-123456789");
    Ticket savedTicket = ticketRepository.save(ticket);
    ticketId = savedTicket.getId();
}
```

Then have tests to make sure every interaction with the database is completely ok. This one is suppose to find to find a ticket by an ID and make sure it is the right one:

```
@Test
public void testFindById() {
    Ticket ticket = ticketRepository.findById(ticketId).get();
    assertEquals(expected:"Avenida 25 de Abril, Alijó", ticket.getAddress());
    assertEquals(expected:"Gonçalo", ticket.getFirstName());
    assertEquals(expected:"Ferreira", ticket.getLastName());
    assertEquals(expected:123456789L, ticket.getId_Number());
    assertEquals(expected:"5070-011", ticket.getZipCode());
    assertEquals(expected:"T-123456789", ticket.getInsertToken());
}
```

This one makes sure that all the tickets are deleted from the database:

```
@Test
public void testDeleteAll() {
    assertEquals(expected:1, ticketRepository.findAll().size());
    ticketRepository.deleteAll();
    assertThat(ticketRepository.findAll().isEmpty());
}
```

This tries to delete a ticket by its id:

```
@Test
public void testDeleteById() {
    assertEquals(expected:1, ticketRepository.findAll().size());
    ticketRepository.delete(ticketRepository.findById(ticketId).get());
    assertThat(ticketRepository.findAll().isEmpty());
}
```

This finds the ticket by the unified token to get it:

```
@Test
public void testFindByInsertToken() {
    Ticket ticket = ticketRepository.findByInsertToken(insertToken:"T-123456789").get();
    assertEquals(expected:"Avenida 25 de Abril, Alijó", ticket.getAddress());
    assertEquals(expected:"Gonçalo", ticket.getFirstName());
    assertEquals(expected:"Ferreira", ticket.getLastName());
    assertEquals(expected:123456789L, ticket.getId_Number());
    assertEquals(expected:"5070-011", ticket.getZipCode());
    assertEquals(ticketId, ticket.getId());
}
```

This one finds a ticket , changes its first name and then sees if the name has changed:

```
@Test
public void testFindChangeAndSave(){
    Ticket t = ticketRepository.findById(ticketId).get();
    t.setFirstName(firstName:"João");
    ticketRepository.save(t);

    Ticket t_aftersave = ticketRepository.findById(ticketId).get();

    assertEquals(t_aftersave.getFirstName(), actual:"João");
}
```

In the Controllers class, I am going to explain the test with the **TicketControllerTest**

With the @WebMvcTest defined to the TicketController to help it run, we are going to use MockMvc to mock the HTTP requests to the endpoints needed in the controllers. We will also use the @MockBean to mock the services responses, to help us configurate all the responses as we want.

Here is the setup made before each test:

```
@WebMvcTest(TicketController.class)
class TicketControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private TicketService ticketService;

    private Ticket t;
    private Ticket t2;

    @BeforeEach
    void setUp() {
        t = new Ticket();
        t.setAddress(address:"Avenida 25 de Abril, Alijó");
        t.setFirstName(firstName:"Gonçalo");
        t.setId(id:1L);
        t.setLastName(lastName:"Ferreira");
        t.setId_Number(id_Number:123456789L);
        t.setZipCode(zipCode:"5070-011");
        t.setInsertToken(insertToken:"T-123456789");

        t2 = new Ticket();
        t2.setAddress(address:"Rua da Laranja Doce, Castedo");
        t2.setFirstName(firstName:"João");
        t.setId(id:2L);
        t2.setLastName(lastName:"Milafres");
        t2.setId_Number(id_Number:123456789L);
        t2.setZipCode(zipCode:"5070-022");
        t2.setInsertToken(insertToken:"T-234567890");

        when(ticketService.getAllTickets()).thenReturn(List.of(t, t2));
        when(ticketService.getTicketByToken(token:"T-123456789")).thenReturn(t);
        when(ticketService.getTicketByID(id:1L)).thenReturn(t);
        when(ticketService.getAllIds()).thenReturn(List.of(e1:1L,e2:2L));
    }
```

This is a test where I mock a request to get all the tickets to the endpoint, see if the status of the request is 200 and then check if the JSON received is similar tothe mocked in the service :

```
@Test
void getAllTicketsTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get(urlTemplate:"/ticket/getall").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression:"$[0].address").value(expectedValue:"Avenida 25 de Abril, Alijó"))
            .andExpect(jsonPath(expression:"$[0].firstName").value(expectedValue:"Gonçalo"))
            .andExpect(jsonPath(expression:"$[0].lastName").value(expectedValue:"Ferreira"))
            .andExpect(jsonPath(expression:"$[0].id_Number").value(expectedValue:123456789L))
            .andExpect(jsonPath(expression:"$[0].zipCode").value(expectedValue:"5070-011"))
            .andExpect(jsonPath(expression:"$[0].insertToken").value(expectedValue:"T-123456789"))
            .andExpect(jsonPath(expression:"$[1].address").value(expectedValue:"Rua da Laranja Doce, Castedo"))
            .andExpect(jsonPath(expression:"$[1].firstName").value(expectedValue:"João"))
            .andExpect(jsonPath(expression:"$[1].lastName").value(expectedValue:"Milafres"))
            .andExpect(jsonPath(expression:"$[1].id_Number").value(expectedValue:123456789L))
            .andExpect(jsonPath(expression:"$[1].zipCode").value(expectedValue:"5070-022"))
            .andExpect(jsonPath(expression:"$[1].insertToken").value(expectedValue:"T-234567890"));
}
```

This is another test that in the same way the last did, requests the endpoint, checks if is 200 (ok!) and then makes sure the answer is correct:

```
@Test
void getTicketByTokenTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get(urlTemplate:"/ticket/getTicketToken/T-123456789").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression:"$.address").value(expectedValue:"Avenida 25 de Abril, Alijó"))
            .andExpect(jsonPath(expression:"$.firstName").value(expectedValue:"Gonçalo"))
            .andExpect(jsonPath(expression:"$.lastName").value(expectedValue:"Ferreira"))
            .andExpect(jsonPath(expression:"$.id_Number").value(expectedValue:123456789L))
            .andExpect(jsonPath(expression:"$.zipCode").value(expectedValue:"5070-011"))
            .andExpect(jsonPath(expression:"$.insertToken").value(expectedValue:"T-123456789"));
}
```

This are the rest of the tests:

```
@Test
void getTicketByIDTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get(urlTemplate:"/ticket/1").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression:"$.address").value(expectedValue:"Avenida 25 de Abril, Alijó"))
            .andExpect(jsonPath(expression:"$.firstName").value(expectedValue:"Gonçalo"))
            .andExpect(jsonPath(expression:"$.lastName").value(expectedValue:"Ferreira"))
            .andExpect(jsonPath(expression:"$.id_Number").value(expectedValue:123456789L))
            .andExpect(jsonPath(expression:"$.zipCode").value(expectedValue:"5070-011"))
            .andExpect(jsonPath(expression:"$.insertToken").value(expectedValue:"T-123456789"));
}

@Test
void getAllIdsTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get(urlTemplate:"/ticket/getAllIds").contentType(MediaType.APPLICATION
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression:"$[0]").value(expectedValue:1L))
            .andExpect(jsonPath(expression:"$[1]").value(expectedValue:2L));
}

@Test
void saveTicketTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate:"/ticket/add")
            .contentType(MediaType.APPLICATION_JSON)
            .content(content:"{\"address\":\"Rua da Laranja Doce, Castedo\",\"firstName\":\"João\",\"lastName\":\
            .andExpect(status().isOk());
}

@Test
void deleteAllTicketsTest() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.delete(urlTemplate:"/ticket/deleteAll")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk());
}
```

For the Services, I used the @Mock and the @InjectMocks. These are some annotations from Mockito and Junit that inject some mock objects in the classes under test. For this example I'll show the **TripServiceTest**

Here I make the the service be injected by the mocked repository:

```
public class TripServiceTest {

    @InjectMocks
    private TripService tripService;

    @Mock
    private TripRepository tripRepository;

    private Trip t;
    private Trip t2;
    private List<Trip> trips = new ArrayList<>();
```

And then I make a before each where I set some trips and then with the addTripsTest I verify is the service is actually calling the repository:

```
@BeforeEach
void setUp() {
    MockitoAnnotations.openMocks(this); // DOCUMENTATION WAY TO CORRECT NULL POINTER EXCEPTION

    t = new Trip();
    t.setDeparture(departure:"Porto");
    t.setDestination(destination:"London");
    t.setDepartureDate(new Date());
    t.setTripDuration(tripDuration:3);
    t.setPrice(price:140);
    t.setAvailableSeats(availableSeats:10);

    t2 = new Trip();
    t2.setDeparture(departure:"Madrid");
    t2.setDestination(destination:"Barcelona");
    t2.setDepartureDate(new Date());
    t2.setTripDuration(tripDuration:1);
    t2.setPrice(price:40);
    t2.setAvailableSeats(availableSeats:4);
}

@Test
public void addTripsTest() {
    tripService.saveTrip(t);
    tripService.saveTrip(t2);

    verify(tripRepository, times(wantedNumberOfInvocations:1)).save(t);
    verify(tripRepository, times(wantedNumberOfInvocations:1)).save(t2);
}
```

These are other types of tests that use asserts to help testing those trips and the mocked responses from the repository:

```
@Test
public void getTripsWhenEmpty() {
    when(tripRepository.findAll()).thenReturn(trips);

    List<Trip> t_fromservice = tripService.getAllTrips();

    assertThat(trips).isEmpty();
    assertThat(t_fromservice).isEmpty();

    verify(tripRepository, times(wantedNumberOfInvocations:1)).findAll();
}

@Test
public void getTripByIdTest() {
    tripRepository.save(t);
    when(tripRepository.findById(id:1L)).thenReturn(java.util.Optional.of(t));

    Trip trip = tripService.getTripById(id:1L);

    assertThat(trip).isEqualTo(t);
    verify(tripRepository, times(wantedNumberOfInvocations:1)).findById(id:1L);
}
```

These are the rest of the tests

```java
@Test
public void addTripsTest() {
    tripService.saveTrip(t);
    tripService.saveTrip(t2);

    verify(tripRepository, times(wantedNumberOfInvocations:1)).save(t);
    verify(tripRepository, times(wantedNumberOfInvocations:1)).save(t2);
}

@Test
public void deleteTripsTest() {
    tripService.deleteTrip(t);
    tripService.deleteTrip(t2);

    verify(tripRepository, times(wantedNumberOfInvocations:1)).delete(t);
    verify(tripRepository, times(wantedNumberOfInvocations:1)).delete(t2);
}

@Test
public void deleteAllTripsTest() {
    tripService.deleteAllTrips();

    verify(tripRepository, times(wantedNumberOfInvocations:1)).deleteAll();
}
```

```java
@Test
public void getAllTripsTest() {
    trips.add(t);
    trips.add(t2);

    when(tripRepository.findAll()).thenReturn(trips);

    List<Trip> t_fromservice = tripService.getAllTrips();

    assertThat(trips).hasSize(expected:2)
                     .containsExactlyInAnyOrder(t, t2);
    assertThat(t_fromservice).hasSize(expected:2)
                     .containsExactlyInAnyOrder(t, t2);

    verify(tripRepository, times(wantedNumberOfInvocations:1)).findAll();
}
```

Also my Integration tests are not running, I tried failsafe and also I have in controllers a test called ITTest that was supposed to be running but I missed on doing that.

There is also a APITest to make sure is running:

```java
class APITest {

    Logger logger = mock(classToMock:Logger.class);

    @Test
    void testCall() throws IOException {
        APICaller apiCaller = new APICaller();
        apiCaller.call();

        Map<String, Object> currencies = apiCaller.getCurrencies();

        assertNotNull(currencies);
        assertFalse(currencies.isEmpty());

    }

}
```

3.3 Functional testing

In the functional testing part, I have used Cucumber + Selenium to test all the things user can interact with, beign the frontend developed or the cache that is receiving the API.

I used the Selenium Interface Extension to make some tests that then exported to java code. Some of the tests needed to be adjusted once some of the given methods were deprecated.

This is an example of a test to fill the form and buy a trip, called **FillingTheFormTest**.

Most of this test was exported, but some methods like the assert to check the alert message needed to be changed, but they are running correctly.

```java
@ExtendWith(SeleniumJupiter.class)
public class FillingTheFormTest {

    private WebDriver driver;
    private Map<String, Object> vars;

    @BeforeEach
    public void setUp( ) {
        this.driver = new FirefoxDriver();
        vars = new HashMap<>();
    }

  @Test
  public void fillingTheForm() {
    driver.get(url:"http://localhost:8080/buying.html?tripId=1");
    driver.findElement(By.id(id:"firstName")).click();
    driver.findElement(By.id(id:"firstName")).sendKeys(...keysToSend:"Gonçalo");
    driver.findElement(By.id(id:"lastName")).click();
    driver.findElement(By.id(id:"lastName")).sendKeys(...keysToSend:"Ferreira");
    driver.findElement(By.id(id:"idNumber")).click();
    driver.findElement(By.id(id:"idNumber")).click();
    driver.findElement(By.id(id:"idNumber")).sendKeys(...keysToSend:"123456789");
    driver.findElement(By.id(id:"address")).click();
    driver.findElement(By.id(id:"address")).sendKeys(...keysToSend:"Alijó");
    driver.findElement(By.id(id:"zipCode")).click();
    driver.findElement(By.id(id:"zipCode")).sendKeys(...keysToSend:"2020-20");
    driver.findElement(By.cssSelector(cssSelector:".btn")).click();
    Alert alert = driver.switchTo().alert();
    String alertMessage = alert.getText();
    assert(alertMessage.contains(s:"Ticket added successfully"));
    alert.accept();
    driver.close();
  }
}
```

Using Cucumber I tested the CacheService interaction, here is the Steps Java File:

```java
public class CacheServiceSteps {

    @Mock
    private CacheService cacheService;

    private String key;
    private Map<String, Object> value;

    @Before
    public void setup() {
        MockitoAnnotations.openMocks(this);
    }

    @Given("^a key \"([^\"]*)\"$")
    public void a_key(String key) {
        this.key = key;
    }

    @Given("^a value with key \"([^\"]*)\" and value \"([^\"]*)\"$")
    public void a_value_with_key_and_value(String key, String value) {
        this.value = new HashMap<>();
        this.value.put(key, value);
    }

    @When("^I put the value into the cache with key \"([^\"]*)\"$")
    public void i_put_the_value_into_the_cache_with_key(String key) {
        cacheService.putIntoCache(key, value);
    }

    @When("^I get all values from the cache$")
    public void i_get_all_values_from_the_cache() {
        cacheService.getFromCache(key:"currencies");
    }


    @Then("^the value should be retrievable from the cache with key \"([^\"]*)\"$")
    public void the_value_should_be_retrievable_from_the_cache_with_key(String key) {
        when(cacheService.getFromCache(key)).thenReturn(value);
        Map<String, Object> retrievedValue = cacheService.getFromCache(key);
        assertEquals(value, retrievedValue);
    }
```

And then I use this cache.feature file to test putting into a value the cache, clearing the cache and then retrieving the currencies from the cache:

```
@cache_sample
Feature: Cache Service With Cucumber

  Scenario: Put value into cache
    Given a key "testKey"
    And a value with key "currency" and value "USD"
    When I put the value into the cache with key "testKey"
    Then the value should be retrievable from the cache with key "testKey"

  Scenario: Clear cache
    Given a key "testKey"
    And a value with key "currency" and value "USD"
    When I put the value into the cache with key "testKey"
    And I clear the cache
    Then the value should not be retrievable from the cache with key "testKey"

  Scenario: Retrieve currencies from cache
    Given a key "currencies"
    And a value with key "USD" and value "1.0"
    And a value with key "EUR" and value "0.85"
    And a value with key "GBP" and value "0.75"
    When I get all values from the cache
    Then the value should be retrievable from the cache with key "currencies"
    And the value should not be retrievable from the cache with key "unknown_key"
```

3.4 Code quality analysis

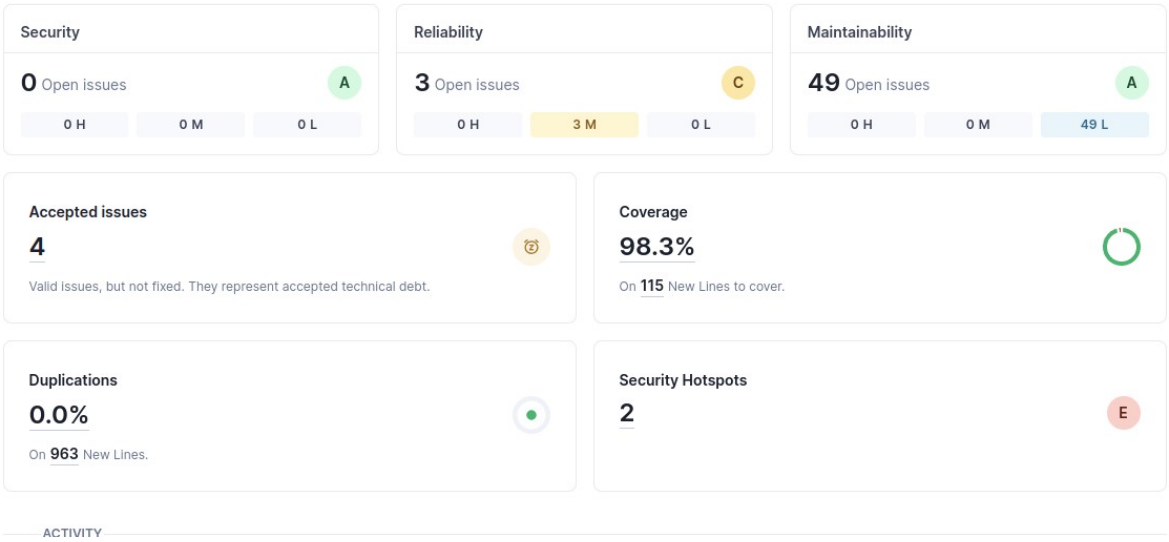
Like we used in lab6, I used docker to install the sonarqube and then created a project and created a project token to run my mvn with the sonar.

This is the command I used:

mvn verify sonar:sonar -Dsonar.host.url=http://localhost:9000 -Dsonar.projectKey=hw1 -Dsonar.login=sqp_0ee652bef2f252c076b9e3f1fb5e0ac4933d6e1b

And after checking the localhost:9000 (default port), I got a 119 issues and 76% coverage

After changing a lot of method names names, packages, instancing a lot of repeated values, and overall making a whole lot more tests I ended up with this stats:



I used 2 Generated values in the cache service and in the DataLoader, cause they are generated with the SpringBootApp Starting.

4 References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/goncalomf20/TQS_107853
Video demo	Video is in the github
QA dashboard (online)	[**optional**; if you have a quality dashboard available online (e.g.: sonarcloud), place the URL here]
CI/CD pipeline	[**optional**; if you have th CI pipeline definition in a server, place the URL here]
Deployment ready to use	[**optional**; if you have the solution deployed and running in a server, place the URL here]

Reference materials

Swagger UI to get a the API Endpoint

Youtube videos (A lot of them)

AI to solve some problems (copilot and gemini)

Mockito Documentation