

HW1: Mid-term assignment report

Tiago Caridade Gomes [108307], v2024-04-10

HW1: Mid-term assignment report.....	1
1 Introduction.....	1
1.1 Overview of the work.....	1
1.2 Current limitations.....	1
2 Product specification.....	2
2.1 Functional scope and supported interactions.....	2
2.2 System architecture.....	2
2.3 API for developers.....	2
3 Quality assurance.....	3
3.1 Overall strategy for testing.....	3
3.2 Unit and integration testing.....	3
3.3 Functional testing.....	3
3.4 Code quality analysis.....	4
3.5 Continuous integration pipeline [optional].....	4
4 References & resources.....	4

1 Introduction

1.1 Overview of the work

Unfortunately, I was unable to implement a docker-compose to run the project more efficiently. At the same time, due to some change in the program, which I was unable to identify, it is not possible to make the reservation completely correctly through my frontend, that is, it does not save the information entered by the user, something that we can check in the video, we can also check if we POST through the Swagger UI.

1.2 Current limitations

Unfortunately, I was unable to implement a docker-compose to run the project more efficiently. At the same time, due to some change in the program, which I was unable to identify, it is not possible to make the reservation completely correctly through my frontend, something that we can see in the video.

2 Product specification

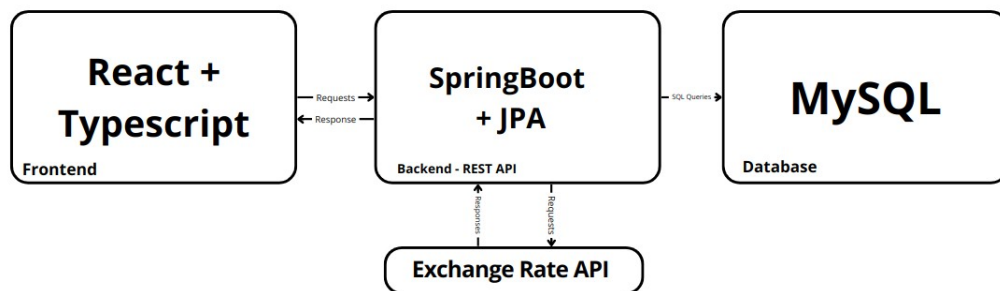
2.1 Functional scope and supported interactions

Actor: Passenger

The actors who will use the system will be passengers who are looking to book bus tickets between two cities. This way the frontend is something simple where you enter the two cities and desired date. They are then presented with all the routes that satisfy the request. After choosing the route they want, they enter their details and make the reservation, where they are informed of the reservation ID that they can use to check it.

2.2 System architecture

The diagram illustrates a modern web application with a frontend built using React and TypeScript, a backend powered by Spring Boot and JPA that functions as a REST API, and a MySQL database. Additionally, the backend fetches data from an external Exchange Rate API. This setup allows for a responsive, scalable, and maintainable application, with clear separation of concerns between user interface, business logic, and data storage.



2.3 API for developers

REST API documentation was generated using Swagger for Maven. It can be accessed at <http://localhost:8080/swagger-ui/index.html#/>.

ticket-controller		^
POST	/api/tickets/createreservation	▼
GET	/api/tickets/reservation/{id}	▼
route-controller		^
GET	/api/routes/routes	▼
GET	/api/routes/route/{id}	▼
currency-exchange-controller		^
GET	/api/exchange-rate/all	▼

3 Quality assurance

3.1 Overall strategy for testing







In my project, I specifically chose not to utilize Test-Driven Development (TDD) for testing the backend. Instead, my strategy focused on employing Behavior-Driven Development (BDD) with Cucumber for the frontend. This approach was adopted to ensure a direct alignment with user expectations and requirements, enabling the development of frontend functionalities that were both relevant and closely tied to user stories. By leveraging Cucumber for BDD on the frontend, I was able to create tests that served as clear, executable specifications based on real user scenarios, thus ensuring that the frontend development was driven by actual user needs. This decision to differentiate the testing approaches for backend and frontend allowed me to optimize the development process, ensuring clarity, relevance, and user-centricity in the functionalities delivered.

3.2 Unit and integration testing

In order to implement the tests, I chose to rely on the Jacoco report to guide me. I started by implementing unit tests to test entities and behaviors. Then Service level tests, with dependency isolation using mocks and finally Integration tests on my own API with Spring Boot MockMvc.

demo Sessions

demo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
 <code>tqs.example.init</code>	<div><div></div></div>	100%	<div><div></div></div>	92%	1 22	0 88	0 15	0 4
 <code>tqs.example.entity</code>	<div><div></div></div>	100%	n/a	n/a	0 74	0 107	0 74	0 4
 <code>tqs.example.service.impl</code>	<div><div></div></div>	100%	<div><div></div></div>	100%	0 10	0 39	0 8	0 2
 <code>tqs.example.dto</code>	<div><div></div></div>	100%	n/a	n/a	0 26	0 38	0 26	0 1
 <code>tqs.example.controller</code>	<div><div></div></div>	100%	<div><div></div></div>	100%	0 10	0 28	0 8	0 2
 <code>tqs.example</code>	<div><div></div></div>	100%	n/a	n/a	0 2	0 3	0 2	0 1
Total	0 of 1 151	100%	1 of 22	95%	1 144	0 303	0 133	0 14

Created with [JaCoCo 0.8.8.202204050719](#)

3.3 Functional testing

For testing the web client, I employed Selenium for a structured and maintainable approach. Tests were designed to simulate user input of locations into the application, verifying the correctness of the returned data. Additionally, I incorporated Cucumber to articulate test scenarios in natural language, bridging the gap between technical implementations and business requirements. This combination of tools and methodologies enabled a thorough and user-centric testing process.

```
Feature: Bus Ticket Booking Service

Scenario: Book a bus ticket from Braga to Porto
  Given I am on the bus service homepage
  When I enter "Braga" as the origin
  And I enter "Porto" as the destination
  And I select the travel date as "01-04-2024"
  And I change the currency
  Then I click the "Find Routes" button

  When I select the route with the index "1"
  And I fill in the passenger details
  And I fill in the payment details
  And I click the "Purchase Ticket" button
  And I should see a confirmation message
  And I check the reservation
  Then I go back the homepage
```

```
@Given("I am on the bus service homepage")
public void i_am_on_the_bus_service_homepage() {
    driver = new ChromeDriver();
    js = (JavascriptExecutor) driver;
    driver.get(url:"http://localhost:5173/");
}

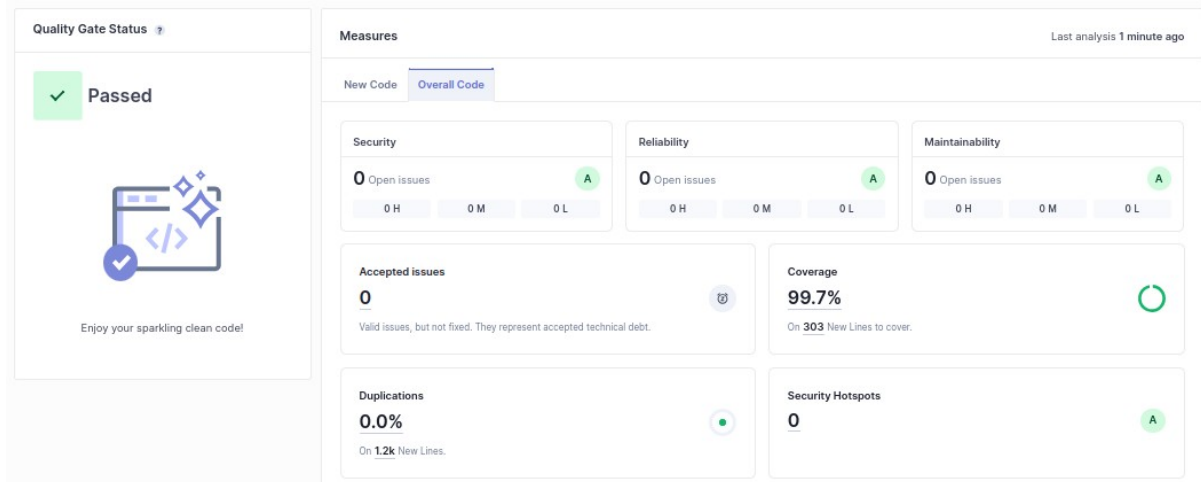
@When("I enter {string} as the origin")
public void i_enter_as_the_origin(String origin) {
    driver.findElement(By.id(id:"origin")).sendKeys(origin);
}

@And("I enter {string} as the destination")
public void i_enter_as_the_destination(String destination) {
    driver.findElement(By.id(id:"destination")).sendKeys(destination);
}

@And("I select the travel date as {string}")
public void i_enter_as_the_date(String date) {
    driver.findElement(By.name(name:"date")).sendKeys(date);
}
```

3.4 Code quality analysis

For code quality analysis, SonarQube was my tool of choice, utilized at critical junctures during the application's development lifecycle. By closely monitoring and addressing the code smells identified by SonarQube, I systematically enhanced the quality of my code. This practice allowed me to refine and optimize the application's codebase, ensuring adherence to best coding practices and improving overall software health.



3.5 Continuous integration pipeline [optional]

I did not implement continuous integration pipeline.

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/caridade1706/TQS_108307
Video demo	https://youtu.be/tLQrL-oaY-k
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

- <https://www.exchangerate-api.com/>