HW1: Mid-term assignment report

Daniel Augusto Serra Madureira [107603], v2024-04-09

1 Introduction

1.1 Overview of the work

This report presents the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy.
BusIn is a platform where is possible to buy bus tickets. This product allows users to select origin and destine cities, as well as the departure day, and see which trips are available for them to buy.
It also shows previously bought tickets, so users can easily access them.
We also offer ways for people to see their tickets in wither euro or American dollars.

1.2 Current limitations

Currently we are limited to a group of cities to which is possible to travel. We lack authentication, as well as proper payment validation, so in the meanwhile tickets are stored locally and with only name and email reservation.

2 Product specification

2.1 Functional scope and supported interactions

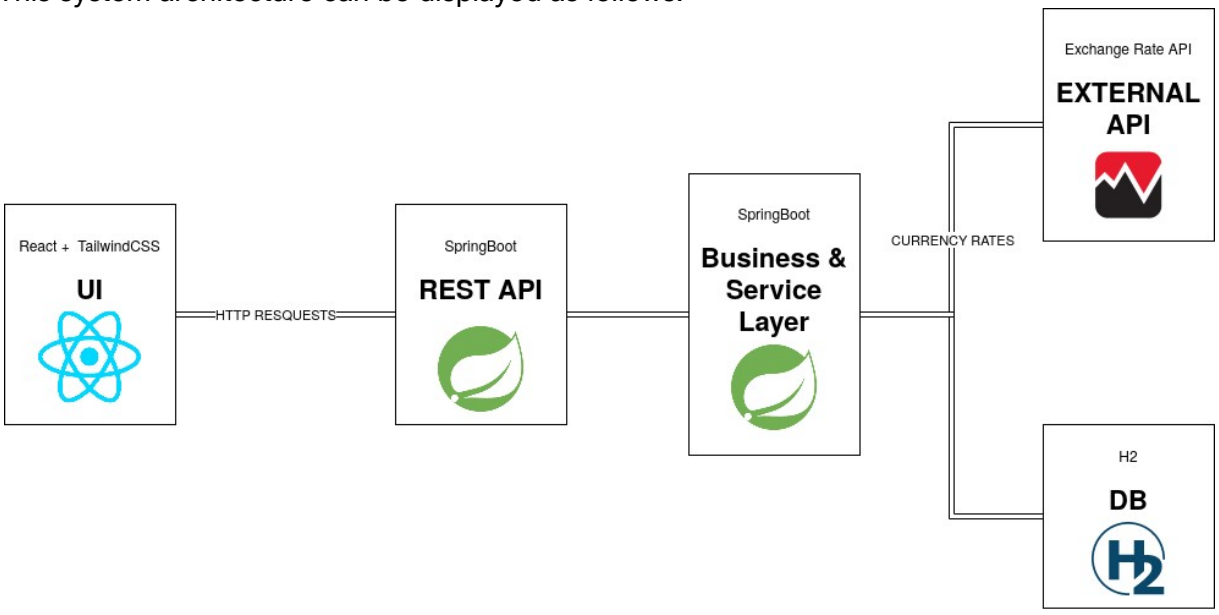
This application main usage scenario is quite simple. A user (**main actor**) chooses between a list of trips, filtered or not, and buys the ticket for the desired trip. Later, we can view that ticket on the platform.

The platform itself should support these interactions through an interface:

- In the HomePage, it is possible to filter the trips by origin city, destine city and departure date.
- In the TripsPage, it is possible to see the trip's main information like departure time and bus, as well as choose a trip to later purchase a ticket.
- In the TripDetailsPage, users can select an available seat and write their information in order to buy the ticket. This page has validation, so invalid format emails won't pass to the db.
- After purchase the ticket, users can see it under the MyTicketsPage, where the ticket's information is displayed for the user to see. It shows the ticket unique identifier, as well as the departing time.

2.2 System architecture

This system architecture can be displayed as follows:



As described by the diagram, this system consists in five major layers that interact with each other:

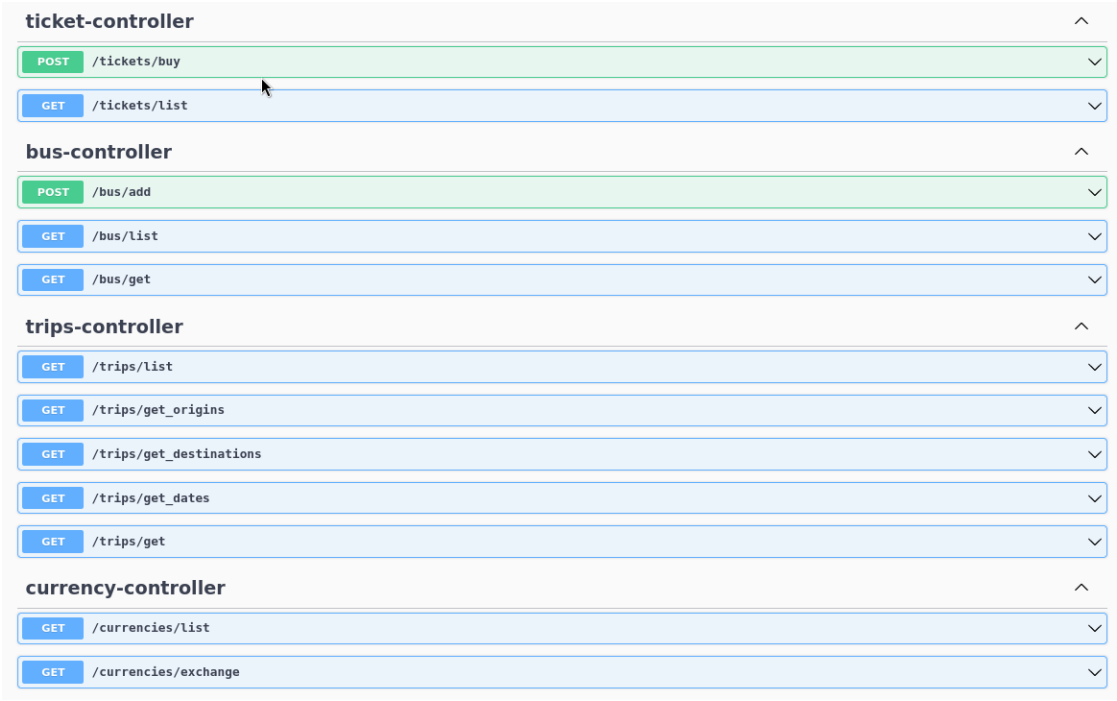
- **User Interface** – it shows the system information to the user, as well as receives his input and passes it to the backend through HTTP Requests. We used React.js with TailwindCSS to handle both the connection logic with the REST API (axios) and the styling (tailwind + daisyui);

- **REST API** – Makes the connection between the interface and the business layer. We choose to use SpringBoot for both its reliability and its simplicity to make a functional API effortlessly.
- **Business and Service Layer –** The heart of the system. This layer is responsible for properly handle the data in the system, as well as transform it and use it to make the system work.
- **External API** – We used this api to handle the currency exchange. We opted for an external API in order to reduce our work and to achieve better and more real conversion rates.
- **DataBase** – We used H2 as our database, mainly because it has a great integration with SpringBoot and yet is simple and powerful.

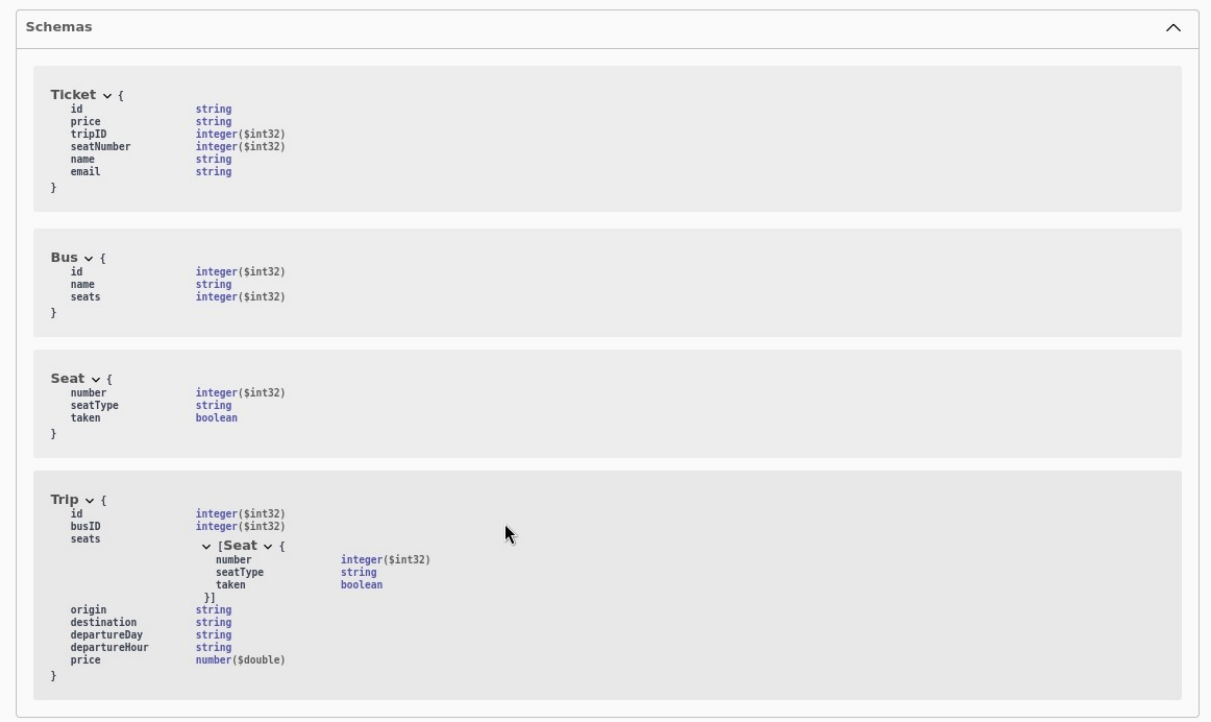
2.3 API for developers

Our system uses a REST API exposed on *localhost:8080* through docker.
Our endpoints are documented via Swagger on *localhost:8080/docs* , so developers can easily find their way through our application. We currently have endpoint for:

1. Buying/ Listing tickets
2. Adding/Listing buses
3. Listing trips, origins and destines
4. Get/ Exchange currencies in our system, with implemented hourly cache.



We also have our schemas documented there:



3 Quality assurance

3.1 Overall strategy for testing

In our test development strategy, we opted for a combination of Cucumber and Selenium to drive our testing efforts, although we didn't implement all the envisioned use cases initially. We believed this combination would offer a robust framework for automating our tests, providing clarity through Cucumber's behavior-driven development (BDD) approach and efficiency through Selenium's web application testing capabilities.
Despite not covering every use case with this approach, we did implemented a very strong base of tests, from unit to integration test Unit 5 and Mockito for unit and integration testing. By employing these tools, we ensured thorough coverage across different layers of our application, verifying both individual components and their interactions within the system. While we recognized the importance of extending our test coverage further, establishing a robust foundation with JUnit 5 and Mockito laid the groundwork for future expansions.

3.2 Unit and integration testing

Our unit testing is more present to test our caching mechanism as well as email validation. These components were deemed essential for ensuring data integrity and input validation. However, due to SpringBoot’s nature, where components seemly interact with each other, we opted not to extensively employ unit tests across all modules. Instead, we strategically allocated testing resources, focusing on essential areas while relying on integration testing to validate component interactions. For scenarios like checking if a bus is full, we addressed them within the appropriate layers, such as the controller layer, where integration testing sufficed to ensure overall system functionality and coherence.

```java
class ValidationTest {

    private static TicketController controller;

    tabnine: test | explain | document | ask
    @SuppressWarnings("static-access")
    @Test
    @DisplayName("Test email validation")
    void testEmailValidation() {
        assertTrue(controller.validateEmail(email:"daniel.madureira@ua.pt"));
    }

    tabnine: test | explain | document | ask
    @SuppressWarnings("static-access")
    @Test
    @DisplayName("Test invalid email validation")
    void testInvalidEmailValidation() {
        assertFalse(controller.validateEmail(email:"daniel.madureiraua.pt"));
        assertFalse(controller.validateEmail(email:"daniel.madureira@uapt"));
    }
```

```java
class CacheTests {

    private static CurrencyService currencyService = new CurrencyService(ttl:2000);

    tabnine: test | fix | explain | document | ask
    @Test
    @DisplayName("Test if cache is invalid after ttl")
    void testIfCacheIsInvalidAfterTTL() throws Exception {
        currencyService.exchange(from:"EUR", to:"USD");
        assertTrue(currencyService.isCacheValid());

        Thread.sleep(2000);
        assertFalse(currencyService.isCacheValid());
    }

    tabnine: test | fix | explain | document | ask
    @Test      You, 19 hours ago • unit tests done and integration with sonarcloud
    @DisplayName("Test if cache is valid during ttl")
    void testIfCacheIsValidDuringTTL() throws Exception {
        currencyService.exchange(from:"EUR", to:"USD");
        assertTrue(currencyService.isCacheValid());

        Thread.sleep(1000);
        assertTrue(currencyService.isCacheValid());
    }
```

Integration tests were a crucial part of our backend testing strategy. We made extensive use of TestRestTemplate to verify communication and functionality across various layers of our application.
Some examples include:

```java
@Test
@DisplayName("Test when exchange currency then return exchange rate")
void whenExchangeCurrency_thenReturnExchangeRate() {
```

```java
@Test
@DisplayName("Test when get a bus then return a bus")
void whenGetBus_thenReturnBus() {
```

```java
@Test
@DisplayName("Test when buying a ticket for a seat that does not exist then give error")
void whenPostTicketForInvalidSeat_thenGiveError() {
```

3.3 Functional testing

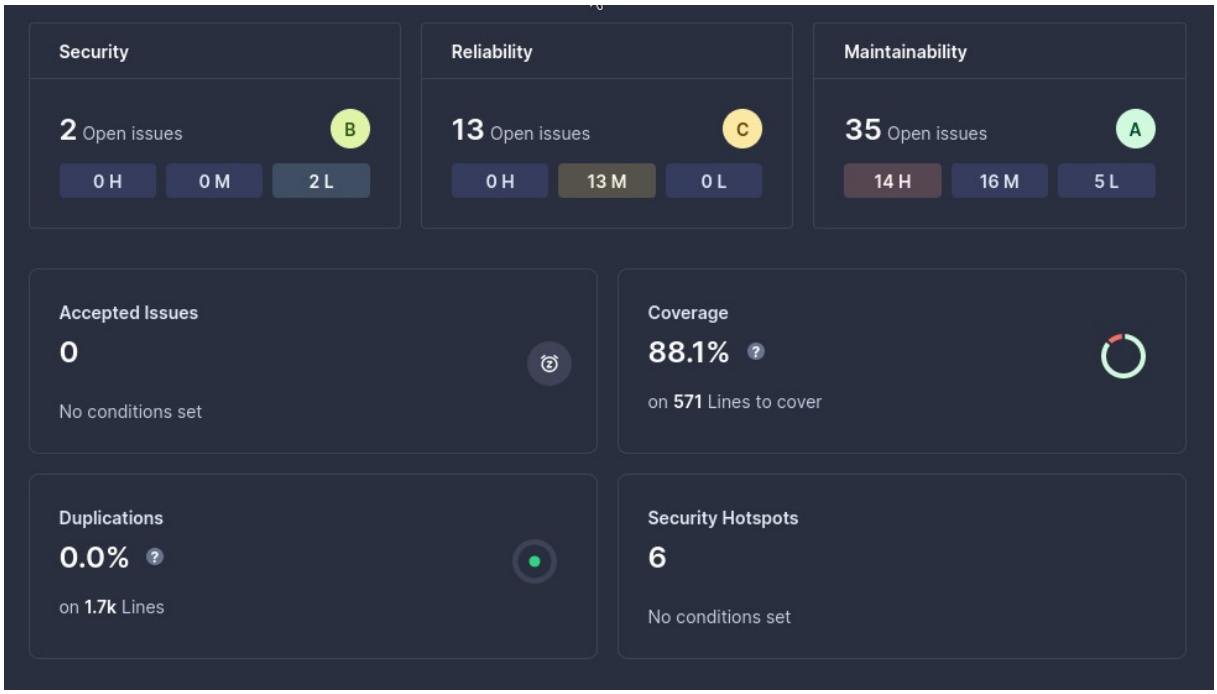
Despite not being the focus of our strategy, we implemented a test for our main use case with Selenium and Cucumber.
Our test was first written in Gherkin and later implemented in Java employing the Page Object Model in order to represent the website pages in Java.

We can see bellow our functional test in Gherkin:

```
Feature: Buying tickets
    Scenario: User buys a ticket from Aveiro to Mirandela
        Given User is on the homepage
        When User selects Aveiro as origin and Mirandela as destination
        Then User should see available trips
        When User selects a trip
        And User selects a seat
        And User enters name and email
        And User buys the ticket
        Then User should see the ticket details
```

3.4 Code quality analysis

Code quality analysis was done with SonarCloud and Jacoco.
The results are shown below:



Despite having open issues, they are mostly related to the SonarCloud analysis. For example, the high level maintainability issues are:

This analysis is incorrect, because there are more than one assertion implemented in this function.
Because of that, I considered a decent code analysis.. Most of our code seems to be in line with the observations we made.

During development, we encountered and successfully resolved issues such as the autowiring of constructors. By addressing these challenges, we ensured smooth integration and functionality within our application. This proactive approach to problem-solving not only improved the efficiency of our development process but also enhanced the overall stability and reliability of our codebase.

The overall code coverage reached 88.1% with a total of 76 tests, which is pretty good considering the 1000+ lines of code that makes this application.

4 References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/Dan1m4D/TQS-107603
Video demo	In the folder under the name demo.webm
QA dashboard (online)	[**optional**; if you have a quality dashboard available online (e.g.: sonarcloud), place the URL here]
Deployment ready to use	The solution runs locally through a docker running ‘docker compose up’ on the directory root

Reference materials

https://www.baeldung.com/spring-rest-openapi-documentation
https://www.exchangerate-api.com/docs/overview
https://docs.sonarsource.com/sonarcloud/getting-started/github/
https://docs.sonarsource.com/sonarcloud/enriching/test-coverage/overview/