HW1: Mid-term assignment report

Rodrigo Martins Graça [107634]

April 9, 2024

Contents

1 Introduction

1.1 Overview of work

This report presents the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy. My application, BusWay, is designed to streamline the process of purchasing bus tickets from various locations to others, offered by different companies, with prices varying based on different times.

1.2 Current limitations

Currently the Reservations do not have seats, and with that its possible to make unlimited reservations in the same Bus. For future features, maybe there should also be an Edit and Cancel Reservation Feature that would allow the users to make the wanted changes in their reserve.

2 Product specification

2.1 Functional scope and supported interactions

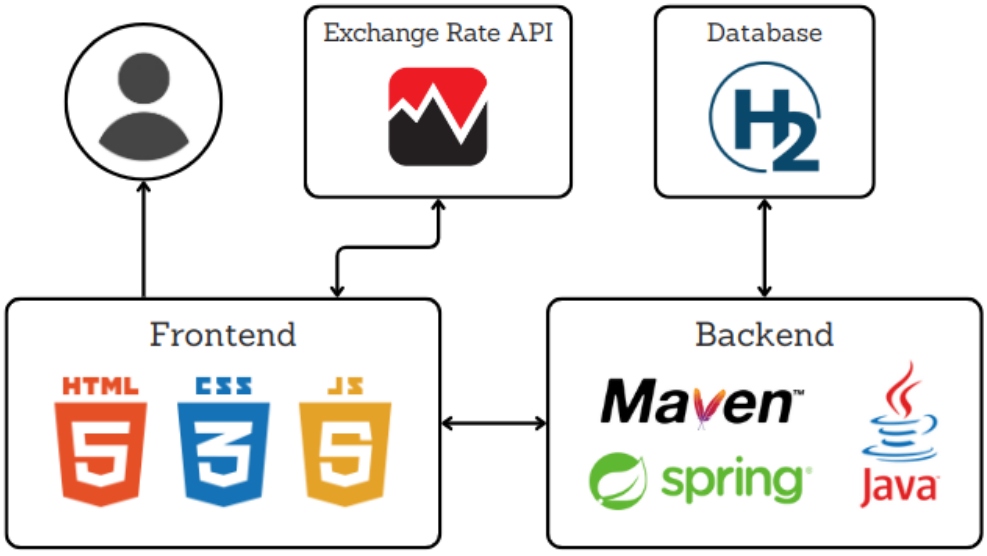
The web application is very intuitive and simple. Has a nice and catchy design for the users eye that will make them confident in using it. About the Use Scenario, each user, can firstly choose their pick-up location and destination, which will lead to the tickets available. After choosing the ticket based on Company, Date and Price, they will be redirected to the purchase page, which they need to fill with their info for completing the purchase. Finally, a table with all the info about the Bus ticket and Reservation will show up.

2.2 System architecture

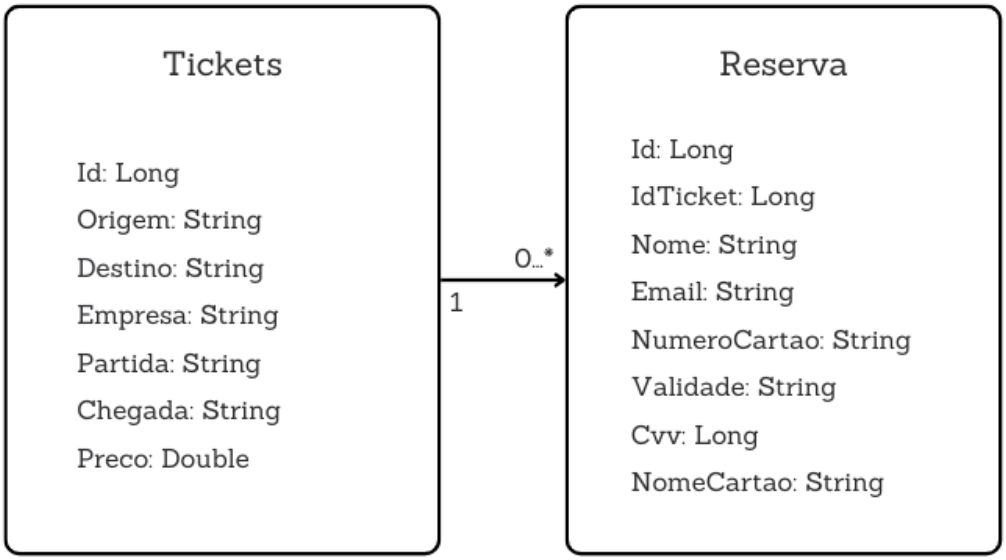
The HTML-based frontend was styled using CSS to tailor its appearance, while JavaScript was employed to integrate the application with the API.

On the backend, Spring Boot was utilized alongside the H2 database.

Real-time currency exchange rates are directly integrated into the frontend, which are fetched from an external API called Exchange Rate.

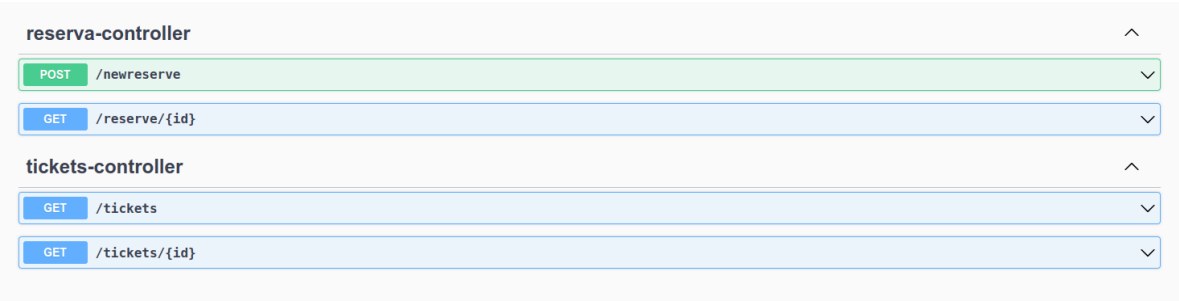


The database has the entities Tickets and Reserva, and they have a relationship of One to Many respectively.



2.3 API for developers

The REST API documentation was obtained with the Swagger dependency from Maven. After the dependency is added to the pom.xml, it can acessed from this url:
http://localhost:8080/swagger-ui.html



This API has 3 methods GET and one POST, which the functions of them are:

- POST a New Reserve
- GET a Reserve by Id
- GET all the tickets
- GET a Ticket by Id

3 Quality assurance

3.1 Overall strategy for testing

The overall strategy used in my test development was a mix with Test-Driven Development and a Test Automation Framework. In the beginning I built the skeleton of the classes I thought would be necessary as I was making sure everything would still compile without errors.

After, I wrote the tests for those classes which made me able to start doing tests during the implementations of the classes.

Although, it was a good approach, I still had to change some classes and add others, which forced me to write more tests, during the implementation. Even with that, some of the tests were useful for early detection of errors in implementation of classes.

3.2 Unit and integration testing

For quality assurance, a comprehensive approach was adopted which included both unit testing and integration testing.

Unit Testing

Unit tests were developed for various components of the application including services, controllers, repositories, and entities. These tests were aimed at ensuring that each unit of the application behaves as expected in isolation. Mockito was used to mock dependencies and isolate the units being tested.

Below is a example of a unit test for the service:

```java
@ExtendWith(MockitoExtension.class)
public class ReservaServiceTest {
    @Mock
    private ReservaRepository reservaRepository;

    @InjectMocks
    private ReservaService reservaService;

    @BeforeEach
    public void setUp() {
        reset(reservaRepository);
    }

    @Test
    @DisplayName("Get reserva by id")
    public void testReservaServiceGetById() {
        Reserva reserva = new Reserva(idticket:1L, nome:"Joao", email:"joao@gmail.com", numeroCartao:"1234 1234 1234 1234", validade:"12/23", c…123L, "Joao Silva");
        Reserva reserva2 = new Reserva(idticket:2L, nome:"Maria", email:"maria@gmail.com", numeroCartao:"1234 1234 1234 1234", validade:"12/23", c…123L, "Maria Silva");
        reserva.setId(id:1L);
        reserva2.setId(id:2L);

        Mockito.when(reservaRepository.findById(id:1L)).thenReturn(java.util.Optional.of(reserva));

        Reserva reservaFound = reservaService.getReservabyId(id:1L);

        assertThat(reservaFound).isEqualTo(reserva);
        Mockito.verify(reservaRepository, Mockito.times(wantedNumberOfInvocations:1)).findById(id:1L);
    }

    @Test
    @DisplayName("Post reserva")
    public void testReservaServicePost() {
        Reserva reserva = new Reserva(idticket:1L, nome:"Joao", email:"joao@gmail.com", numeroCartao:"1234 1234 1234 1234", validade:"12/23", c…123L, "Joao Silva");

        Mockito.when(reservaRepository.save(reserva)).thenReturn(reserva);

        Reserva reservaSaved = reservaService.reserveTicket(reserva);

        assertThat(reservaSaved).isEqualTo(reserva);
        Mockito.verify(reservaRepository, Mockito.times(wantedNumberOfInvocations:1)).save(reserva);
    }
}
```

Integration Testing

Integration tests were performed to verify the interactions between different components of the system. These tests were conducted without mocking, making actual calls to the respective components. REST Assured library was used to make requests to the controller and assert the responses.

This approach ensured that the application behaves correctly as a whole. An example integration test is provided below in the TicketsIntegrationTest class.

This comprehensive testing approach helped in early detection of errors and ensured the reliability and robustness of the application.

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@TestPropertySource(locations = "../../application.properties")
class TicketsIntegrationTest {

    @LocalServerPort
    int randomServerPort;

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private TicketsRepository ticketsRepository;

    @BeforeEach
    public void setUp() {
        ticketsRepository.deleteAll();
    }

    @AfterEach
    public void tearDown() {
        ticketsRepository.deleteAll();
    }

    @Test
    @DisplayName("Get all tickets")
    void testGetAllTickets() {
        Tickets ticket1 = new Tickets(origem:"Porto", destino:"Viseu", empresa:"BuzzTuga", partida:"2021-10-10 10:00", chegada:"2021-10-10 12:00", prec…10.0);
        Tickets ticket2 = new Tickets(origem:"Porto", destino:"Viseu", empresa:"Autocarros de Portugal", partida:"2021-10-10 14:00", chegada:"2021-10-10 16:00", prec…10.0);
        ticketsRepository.save(ticket1);
        ticketsRepository.save(ticket2);

        ResponseEntity<Tickets[]> responseEntity = restTemplate.getForEntity(url:"/tickets", responseType:Tickets[].class);
        Tickets[] tickets = responseEntity.getBody();

        assertThat(tickets)
            .isNotNull()
            .hasSize(expected:2);
        assertThat(tickets).extracting(Tickets::getOrigem).containsOnly(...values:"Porto", "Porto");
        assertThat(tickets).extracting(Tickets::getDestino).containsOnly(...values:"Viseu", "Viseu");
        assertThat(tickets).extracting(Tickets::getEmpresa).containsOnly(...values:"BuzzTuga", "Autocarros de Portugal");
        assertThat(tickets).extracting(Tickets::getPartida).containsOnly(...values:"2021-10-10 10:00", "2021-10-10 14:00");
        assertThat(tickets).extracting(Tickets::getChegada).containsOnly(...values:"2021-10-10 12:00", "2021-10-10 16:00");
        assertThat(tickets).extracting(Tickets::getPreco).containsOnly(...values:10.0, 10.0);
    }
```

3.3 Functional testing

The web client functional tests have been implemented using Selenium and the Page Object pattern. These tests involve interacting with user interface elements and verifying the system's behavior.

The tests have been designed to simulate user behavior when making a bus ticket reservation through the web application. They follow a specific sequence of steps to select origin and destination, fill in reservation details, and confirm the transaction.

Here's an excerpt of one of the implemented tests:

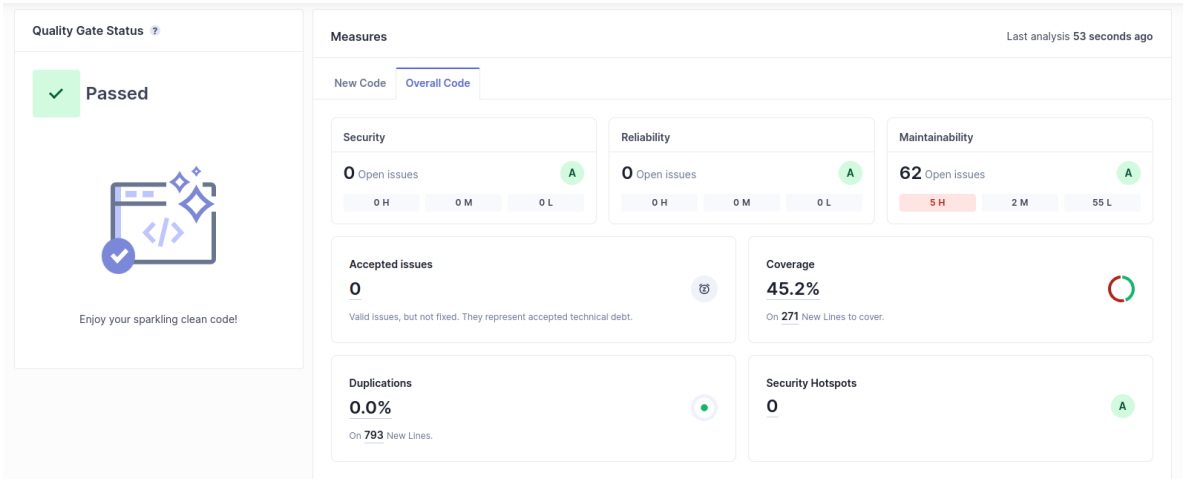
```java
@ExtendWith(SeleniumJupiter.class)
public class SeleniumTest {
  private ChromeDriver driver;

  @Test
  public void backend() throws InterruptedException {

    WebDriverManager.chromedriver().setup();
    ChromeOptions options = new ChromeOptions();

    options.addArguments("--remote-allow-origins=*");

    // 1 | open | http://
    driver = new ChromeDriver(options);
    driver.get(url:"http://localhost:8000/html/index.html");
    //driver.get("http://127.0.0.1:5500/frontend/html/index.html");
    driver.manage().window().setSize(new Dimension(width:1920, height:1048));

    // 2 | wait until page ready
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(1));
    wait.until(ExpectedConditions.visibilityOfElementLocated(By.id(id:"pickup")));


    // 3 | select | id=pickup | label=Faro
    {
      WebElement dropdown = driver.findElement(By.id(id:"pickup"));
      dropdown.findElement(By.xpath(xpathExpression:"//option[. = 'Faro']")).click();
    }
    // 4 | mouseDownAt | id=pickup | 0,-0.75
    {
      WebElement element = driver.findElement(By.id(id:"pickup"));
      Actions builder = new Actions(driver);
      builder.moveToElement(element).clickAndHold().perform();
    }
    // 5 | mouseMoveAt | id=pickup | 0,-0.75
    {
      WebElement element = driver.findElement(By.id(id:"pickup"));
      Actions builder = new Actions(driver);
      builder.moveToElement(element).perform();
    }
    // 6 | mouseUpAt | id=pickup | 0,-0.75
    {
      WebElement element = driver.findElement(By.id(id:"pickup"));
      Actions builder = new Actions(driver);
      builder.moveToElement(element).release().perform();
    }
```

3.4 Code quality analysis

For Code Quality Analysis it was used SonarQube. It was used in multiple different times in the development of the application. The first test was done not so in the beginning of implementation but still with the Integration Tests not done, and some problems in the other tests.

Although it did not had Security or Reliability Issues, there was a lot of coverage missing as well as some Maintainability and before doing the integration tests, I prioritized fixing this bugs and smells. Some of them were some Asserts missing and public classes that I needed to remove the 'public'.

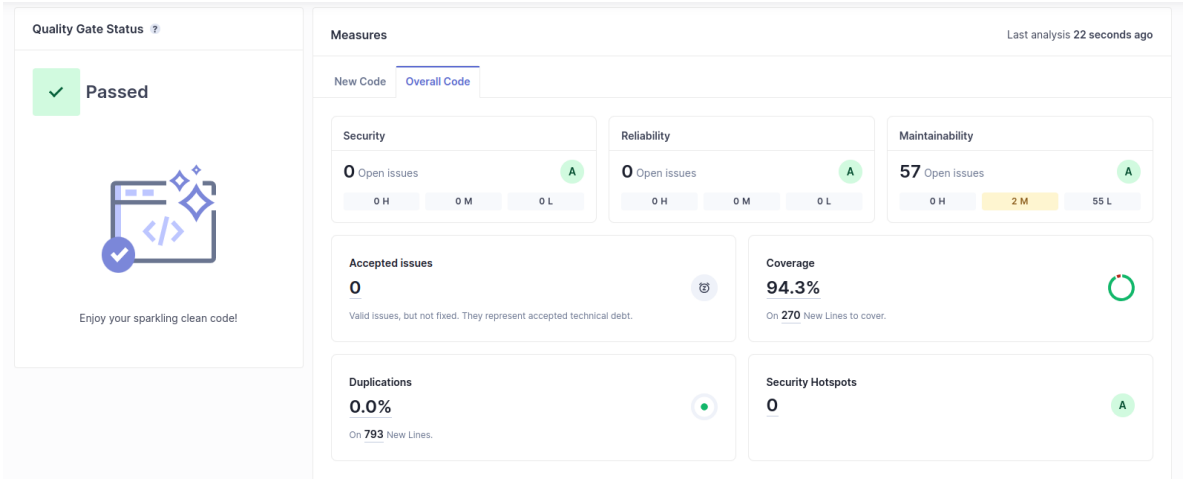
In the picture below you can see the results from the first test that included High Impact, Medium Impact Maintainability Issues and Low Coverage.



After fixing all the bugs, and smells I finally implemented the Integration Tests which came with some Medium Impact Maintainability Issues. I had to fix those too, and then I worried about the Coverage.

The biggest problem about it was that the DataInitializer did not had any test, and since it was a bigger file it brought the coverage to a low percentage. The moment I made a Test for that file, the coverage rised from that low percentage to around 92%, which is a big upgrade. I covered some other lines of the code as well but didn't made that much difference only rising to 94%.

And below you can see the final SonarQube Test I did before the submission.



4 References resources

4.1 Project resources

Resource	URL/location
Git repository	https://github.com/rodrigograc4/TQS_107634
Video demo	https://github.com/rodrigograc4/TQS_107634/tree/main/HW1/report

4.2 Reference materials

Exchange-Rate API