HW1: Mid-term assignment report

Joaquim Vertentes Rosa [109089], v2024-04-09

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 **2**
3.1 Overall strategy for testing 2
3.2 Unit and integration testing 2
3.3 Functional testing 3
3.4 Code quality analysis 3
3.5 Continuous integration pipeline [optional] 3

4 References & resources **3**

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.
The application developed is a simple full-stack web application for online bus ticket booking. Its primary purpose is to provide a services API for searching bus connections between two cities and booking reservations for passengers. The backend services and the user interface were built using the Spring Boot framework with Maven.

1.2 Current limitations

- Automatic documentation of the API endpoints was not possible because Swagger documents @RestController's, and @Controller's were used in this project. An attempt was made to use the OpenAPI dependency to generate documentation, but no results were obtained.

- A more realistic logic for the booking service was not implemented.
- The project deployment was not performed.

2 Product specification

2.1 Functional scope and supported interactions

This application is intended for anyone who needs to book bus tickets online to travel between different cities in mainland Portugal.

Usage Scenarios:

Booking a trip: The user accesses the platform, enters the origin city, destination city, and the date of their trip, and clicks the search button. They are redirected to a page with the search results, where they select the preferred trip. At checkout, the user fills out the form with their details and clicks the "Pay Now" button to make the reservation payment. After payment, a reservation code is provided to the user.

Checking reservation details: The user clicks on the NavBar button to check the reservation details. A text input box will appear for the user to enter the reservation code provided earlier. Upon entering the reservation code and clicking the check button, the user should see the details of their reservation.

2.2 System architecture

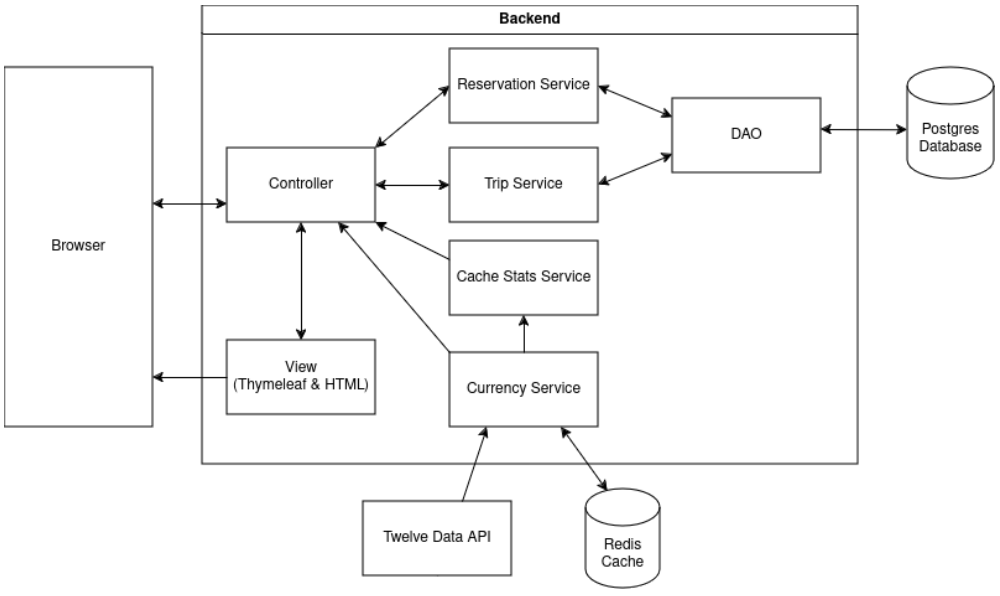

Fig 1 - System architecture

The backend was built using Spring Boot, while the presentation layer utilized Spring Boot along with Thymeleaf templates. The Twelve Data API was integrated to fetch currencies and exchange rates. Postgres served as the database to store trip and reservation data, while Redis was used for caching currency data and rates.

2.3 API for developers

2.3.1 Problem Domain

- **GET "/" -** Returns the HTML of the main page, with the list of cities dynamically inserted into the selects.
- **GET "/results?origin={origin}&destination={destination}&date={date}&currency={currency}" -** Returns the HTML of the results page, dynamically inserting the trips that match the user's search. When a new currency is selected, this endpoint is called again with the updated currency, converting the base price to that currency. By default, the currency is Euro.
- **GET "/checkout?origin={origin}&destination={destination}&date={date}&tripId={id}&currency={currency}&finalPrice={finalPrice}" -** Returns the HTML of the checkout page with forms for the user to fill out with their personal information. The converted price is inserted into the page. The first three parameters are used to return to the previous page, and the tripId, currency, and finalPrice are used in creating the reservation.
- **POST "/create-reservation" -** Sends JSON-formatted user form data plus the three attributes mentioned above in the message body and returns the generated ID for the created reservation.
- **GET "/success?reservationId={reservationId}" -** Returns the HTML of the successful purchase page with the code generated during reservation creation inserted into the page.
- **GET "/reservation" -** Returns the HTML of the reservation details verification page.
- **POST "/reservation" -** Receives the reservation ID entered by the user and returns the details of that reservation.

2.3.2 Cache Usage Statistics

- **GET "/cacheStats" -** Returns the HTML of the page for viewing cache usage statistics that store conversion rates. It returns the number of cache hits, the number of misses, the number of puts, and the total number of requests.

3 Quality assurance

3.1 Overall strategy for testing

Initially, the approach was to create HTML templates, so most controllers were implemented using static data to visualize the interface. After completing the UI, the controller was adapted to receive data from the services, and implementation proceeded as tests were conducted. Unit tests were conducted first, followed by integration tests, and finally, functional testing with Cucumber and Selenium.

3.2 Unit and integration testing

Unit tests were conducted to test the services, namely the currency service, the trip service, and the reservation service. Mockito was used to simulate the behavior of repositories and other components with which these services interacted. Additionally, unit tests were performed on the controllers to ensure they were fulfilling their purpose, mocking the business logic layer used by them. Integration tests were also carried out on the controllers, using MockMvc, to verify if the backend as a whole was functioning correctly.

```
@Mock(lenient = true)
private TripRepository tripRepository;

@InjectMocks
private TripServiceImpl tripService;

@BeforeEach
void setUp() {
    Trip trip1 = new Trip(id:1L, origin:"Porto", destination:"Lisboa", LocalDate.of(2024, 05, 05), Time.valueOf("08:00:00"),
            Time.valueOf("12:00:00"), duration:"4h", priceEuro:20.0, busId:"BUS001", availableSeats:50);

    Trip trip2 = new Trip(id:2L, origin:"Lisboa", destination:"Porto", LocalDate.of(2024, 05, 05), Time.valueOf("08:00:00"),
            Time.valueOf("12:00:00"), duration:"4h", priceEuro:20.0, busId:"BUS002", availableSeats:50);

    List<Trip> trips = Arrays.asList(trip1, trip2);

    when(tripRepository.findByOriginAndDestinationAndDate(origin:"Porto", destination:"Lisboa", LocalDate.of(2024, 05, 05)))
            .thenReturn(Arrays.asList(trip1));
    when(tripRepository.findByOriginAndDestinationAndDate(origin:"Lisboa", destination:"Porto", LocalDate.of(2024, 05, 05)))
            .thenReturn(Arrays.asList(trip2));
    when(tripRepository.findById(id:1L)).thenReturn(java.util.Optional.of(trip1));
    when(tripRepository.findById(id:2L)).thenReturn(java.util.Optional.of(trip2));
    when(tripRepository.findAll()).thenReturn(trips);
}

@Test
void testGetAllCities() {
    Set<String> cities = tripService.getAllCities();

    assertThat(cities, hasSize(size:2));

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

Fig. 2 - TripService Unit Test Example

```
@BeforeEach
void setUp() throws Exception {
        SimpleDateFormat format = new SimpleDateFormat("HH:mm:ss");
        java.util.Date d1 = format.parse("08:00:00");
        java.sql.Time departureTime = new java.sql.Time(d1.getTime());

        java.util.Date d2 = format.parse("12:00:00");
        java.sql.Time arrivalTime = new java.sql.Time(d2.getTime());

        trip1 = new Trip(id:1L, origin:"Lisboa", destination:"Porto", LocalDate.parse("2024-05-05"), departureTime,
                        arrivalTime, duration:"4h", priceEuro:10.0, busId:"BUS007", availableSeats:50);

        trip1 = tripRepository.save(trip1);
}

@Test
void testIndex() throws Exception {
        mvc.perform(get(urlTemplate:"/")).andExpect(status().isOk()).andExpect(view().name(expectedViewName:"index"))
                        .andExpect(model().attribute(name:"cities",
                                        new HashSet<>(Arrays.asList("Lisboa", "Porto"))));
}
```

Fig. 3 - PageController Integration Test Example

3.3 Functional testing

For user-facing test cases, Selenium was used in conjunction with Cucumber and the Page Object pattern to implement scenarios for bus ticket booking and verification of booking details. Here's a code snippet created for this purpose:

```
    private WebDriver driver;
    private IndexPage indexPage;
    private ResultsPage resultsPage;
    private CheckoutPage checkoutPage;
    private SuccessPage successPage;
    private ReservationPage reservationPage;
    private String reservationId;

    @Given("the user is on the main page")
    public void theUserIsOnTheMainPage() {
        driver = WebDriverManager.firefoxdriver().create();
        driver.get(url:"http://localhost:8080/");
        indexPage = new IndexPage(driver);
    }

    @When("the user selects {string} as the departure city")
    public void theUserSelectsAsTheDepartureCity(String origin) {
        indexPage.selectOrigin(origin);
        assertEquals(origin, indexPage.getOrigin());
    }

    @And("the user selects {string} as the arrival city")
    public void theUserSelectsAsTheArrivalCity(String destination) {
        indexPage.selectDestination(destination);
        assertEquals(destination, indexPage.getDestination());
    }

    @And("the user selects {string} as the departure date")
    public void theUserSelectsAsTheDepartureDate(String date) {
        indexPage.inputDate(date);
        assertEquals(date, indexPage.getDate());
    }
```

Fig. 4 - CucumberSteps.java

3.4 Code quality analysis

The tool used for static code analysis was Sonarqube. This tool identified potential issues, vulnerabilities, and areas of improvement, which I addressed to achieve cleaner and safer code. The final version of the code achieved positive results in the quality gate, with an impressive coverage percentage of 93.4% and only four medium-level issues identified.

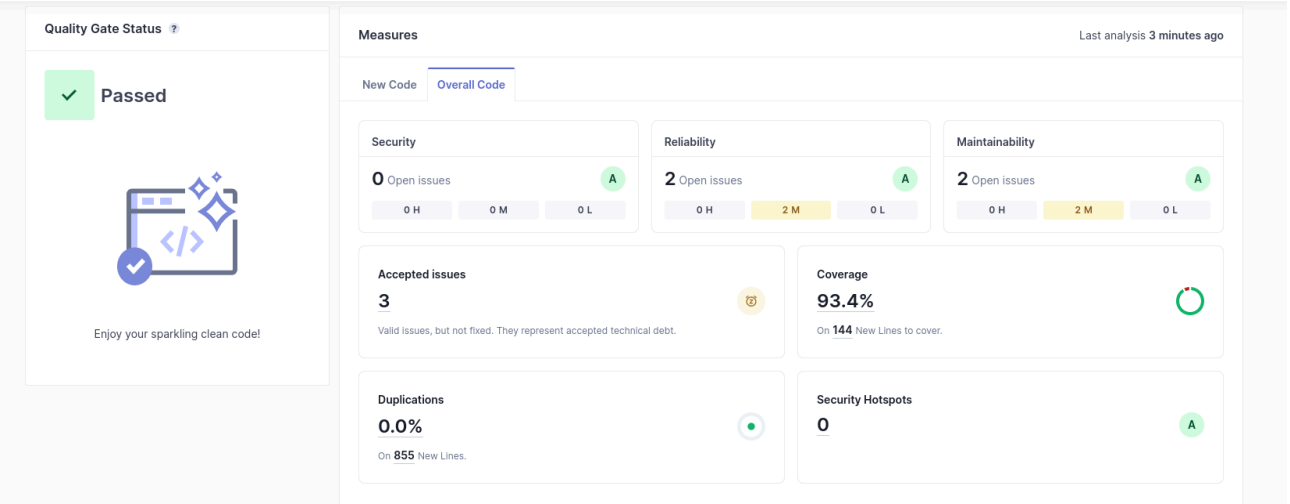

Fig. 5 - Sonarqube Quality Overview

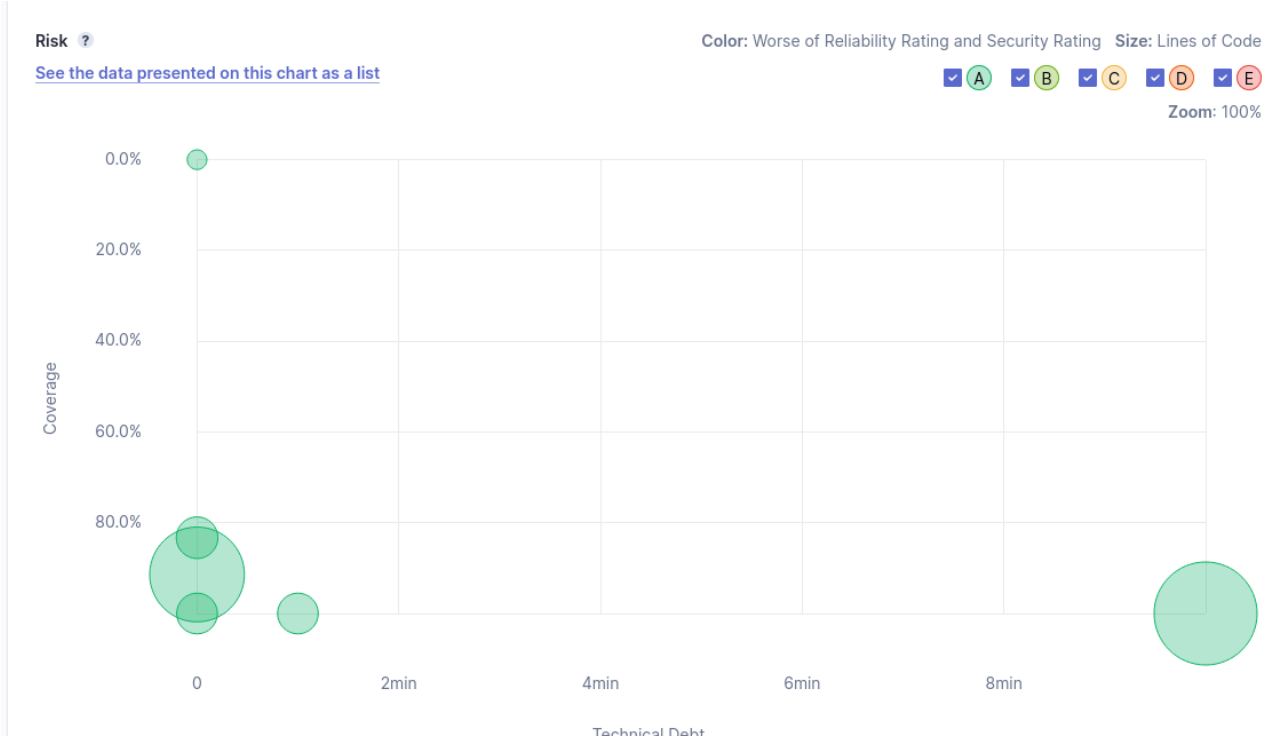

Fig. 6 - Sonarqube Measure Tabs

Here is an example of a code smell that might not have been detected without the warning from SonarQube:

Fig. 7 - Code smell example

3.5 Continuous integration pipeline [optional]

A CI pipeline was established using GitHub Actions, executing all created tests except for integration tests and Selenium tests, ensuring continuous validation of code pushed directly to the main branch or through pull requests.

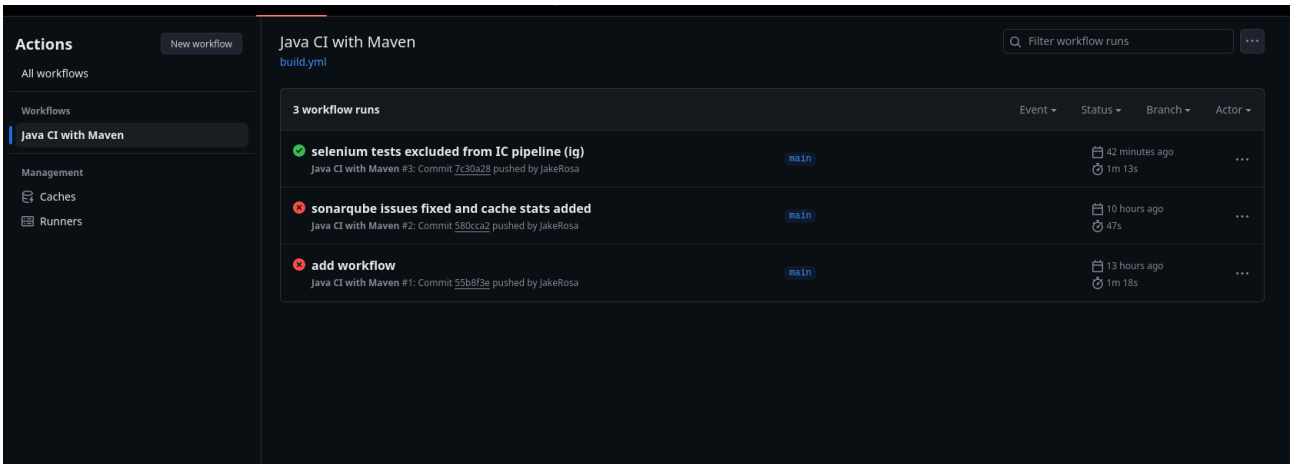

Fig. 8 - CI workflows

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    name: Build
    runs-on: ubuntu-latest

    defaults:
      run:
        working-directory: hw1/BusBooking

    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'
          cache: maven      You, 13 hours ago • add workflow
      - name: Build with Maven with Unit Tests
        run: mvn -B package --file pom.xml -Punit-tests
```

Fig. 9 - GitHub Actions config file

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/JakeRosa/TQS_109089
Video demo	https://github.com/JakeRosa/TQS_109089/blob/main/hw1/demo.webm
CI/CD pipeline	https://github.com/JakeRosa/TQS_109089/actions

Reference materials

https://rapidapi.com/twelve-data1-twelve-data-default/api/twelve-data1
https://www.baeldung.com/spring-redirect-and-forward
https://www.baeldung.com/spring-rest-openapi-documentation
https://www.baeldung.com/jacoco-report-exclude
https://www.baeldung.com/spring-controllers
https://spring.io/guides/gs/testing-web

https://www.baeldung.com/spring-boot-testing
https://www.baeldung.com/thymeleaf-in-spring-mvc
https://www.geeksforgeeks.org/redis-cache-in-java-using-jedis/
https://www.baeldung.com/java-spring-mockito-mock-mockbean
https://www.baeldung.com/injecting-mocks-in-spring