HW1: Mid-term assignment report

José Gameiro [108840], v09-04-20204

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.

This report also outlines the development process and key features of our bus ticket selling application. The primary objective was to create a robust services API facilitating the purchase of bus tickets, focusing on two core functionalities: searching for bus connections between cities and booking reservations for passengers. To showcase the application's capabilities, it was also developed a simplified web app demonstrating these essential use cases. The product's name is Bus Wise.

To ensure the reliability and quality of the application, we incorporated various types of tests:

A) Unit tests: Implemented for critical components such as booking logic, cache behavior, and utility functions like validators and converters.

B) Service level tests: Conducted with dependency isolation using mocks, particularly testing scenarios detached from external data providers.
C) Integration tests: Utilized Spring Boot MockMvc and REST-Assured to verify the functionality of the API endpoints.
D) Functional testing: Employed BDD with Selenium WebDriver to validate the web interface's behavior.
Furthermore, we integrated code quality metrics into the development process by utilizing SonarQube for analysis. This integration ensures adherence to coding standards and identifies areas for improvement. Integration with SonarQube (or Codacy) was established, enabling continuous analysis of the project's codebase. For public Git projects, SonarCloud was utilized for analysis.

1.2 Current limitations

There exists some problems with the search method in the front-end, like in the select departure city and destination city it may not work very well, and the currency exchange options only has 2 options EUR and USD available.
There's also missing some tests to cover some possible outcomes, like with functional tests I only developed one.
There are also missing some functionalities when dealing with the data received from the external API like how many hits/misses.

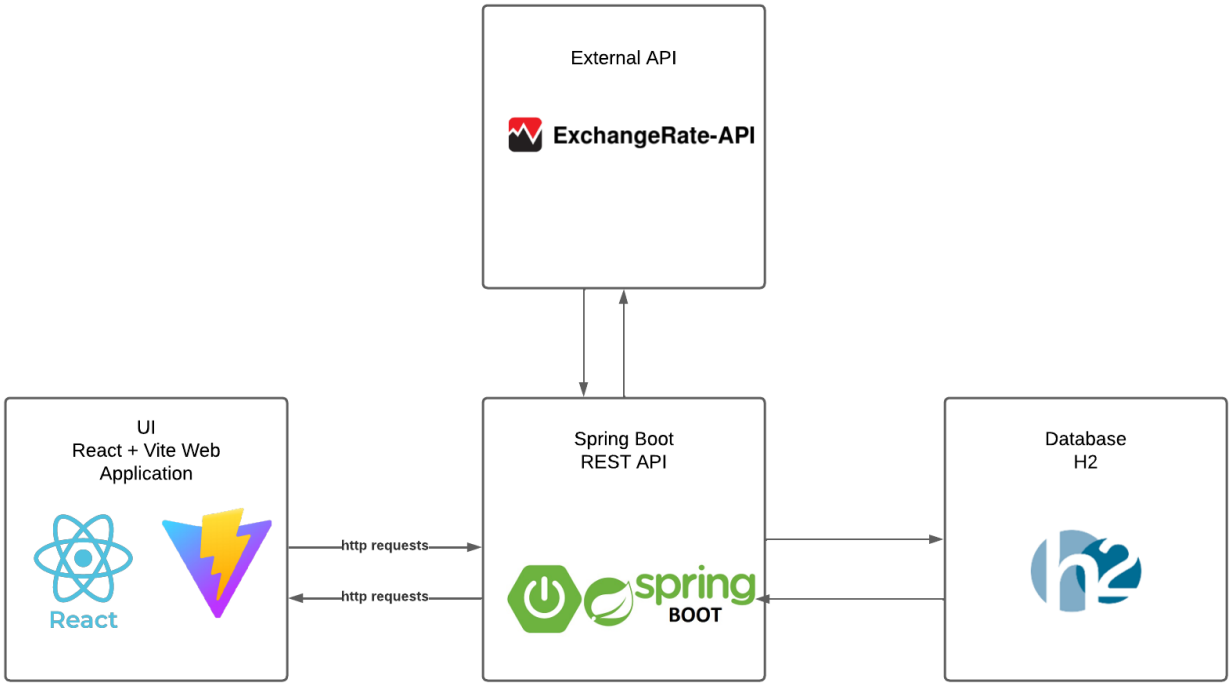
2 Product specification

2.1 Functional scope and supported interactions

The application targets two primary categories of users: passengers and administrators.

Passengers:

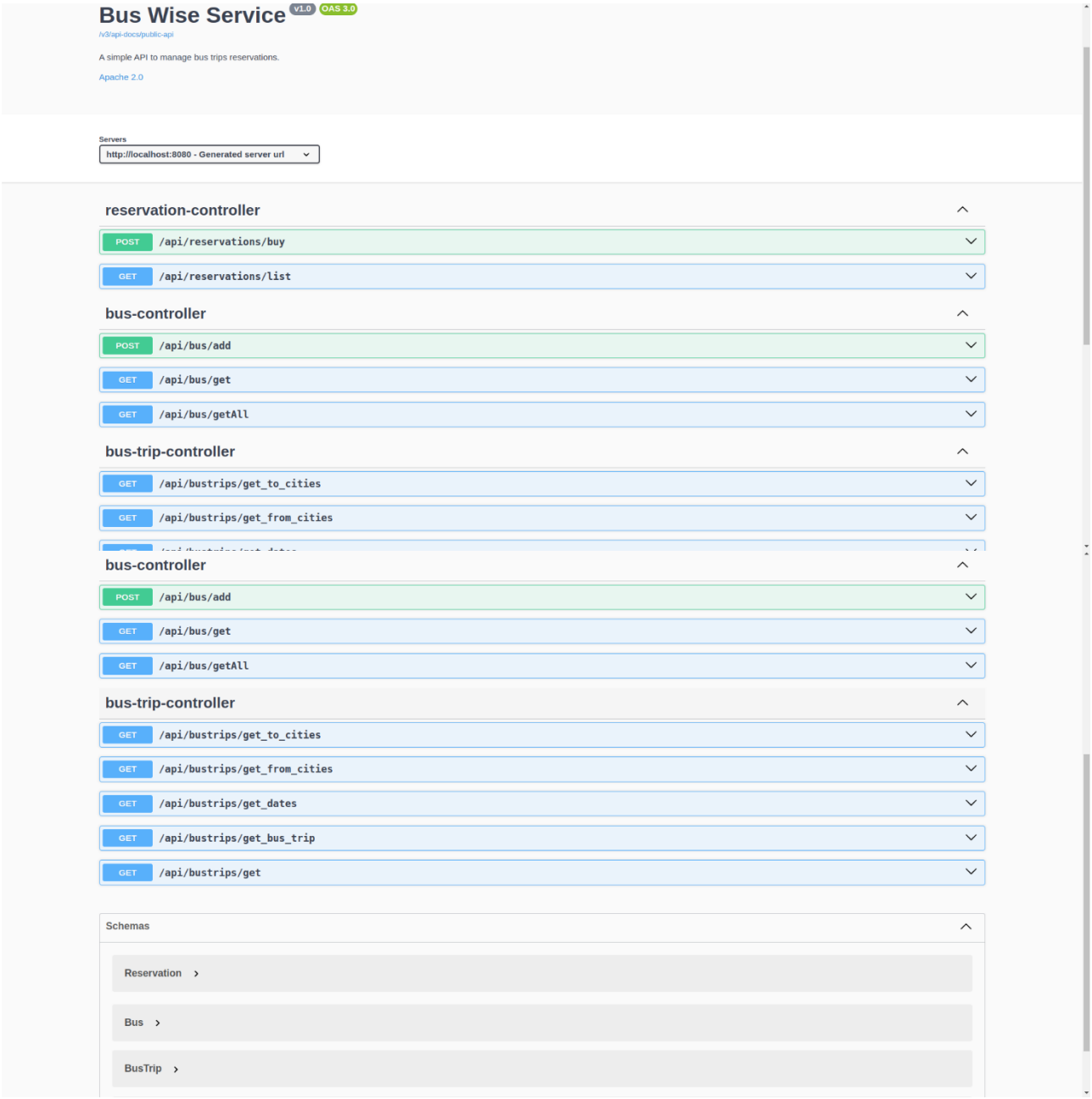
Actors: Individuals seeking to travel by bus.
Purpose: Passengers use the application to search for bus connections between cities and book reservations for their journeys.
Main Usage Scenario: A passenger accesses the application's web interface, enters the desired origin and destination cities, along with the travel date. The application then presents available bus connections meeting the criteria. The passenger selects a preferred trip, provides passenger details, and completes the booking process. Upon successful reservation, the passenger receives a confirmation of their ticket.



In this architecture it exists 4 main components:

- The Web Application, a web application developed with React + Vite, with some other libraries, like tailwindcss, react-router-dom, react-icons, etc;
- The REST API a simple API built using Spring Boot;
- A database in H2;
- And an external API which provides exchange rates;

2.3 API for developers



3 Quality assurance

3.1 Overall strategy for testing

The overall test development strategy employed in the project was centered around Behavior-Driven Development (BDD) using several tools like Cucumber, Junit, Mockito, etc. The strategy Test-Driven Development (TDD) was not adopted for this project.

For service level testing, we utilized Mockito to create mocks for isolating dependencies, particularly to test scenarios detached from external data providers.

In terms of integration testing, we relied on Spring Boot MockMvc to perform tests on our API endpoints. While REST-Assured was suggested, we chose not to use it in favor of maintaining consistency with our selected tools and frameworks, which aligned well with our testing approach.

3.2 Unit and integration testing

In our project, we implemented both unit and integration tests to ensure the functionality and reliability of the application.

Unit tests were created for critical components such as input validation and some cache behaviors. These tests validate the behavior of individual units of code in isolation to ensure they function as expected.

Examples:

In the TestInputValidators class, we validate input fields such as email and phone numbers using the ReservationFormValidator. These tests ensure that the validation logic returns the correct outcomes for different input scenarios.

```java
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import deti.tqs.backend.services.ReservationFormValidator;

public class TestInputValidators {

  @Test
  @DisplayName("Check if the email has a valid format it should return true")
  void whenValidEmail_ReturnTrue() {
    ReservationFormValidator validator = new ReservationFormValidator();
    assertThat(validator.validateEmail("adriana@gmail.com")).isTrue();
  }

  @Test
  @DisplayName("Check if the email has a invalid format it should return false")
  void whenInvalidEmail_ReturnFalse() {
    ReservationFormValidator validator = new ReservationFormValidator();
    assertThat(validator.validateEmail("adriana@gmail")).isFalse();
    assertThat(validator.validateEmail("adriana.pt")).isFalse();
  }

  @Test
  @DisplayName("Check if the phone has a valid size")
  void whenValidPhone_ReturnTrue() {
    ReservationFormValidator validator = new ReservationFormValidator();
    assertThat(validator.validatePhone("912345678")).isTrue();
  }

  @Test
  @DisplayName("Check if the phone has a invalid size")
  void whenInvalidPhone_ReturnFalse() {
    ReservationFormValidator validator = new ReservationFormValidator();
    assertThat(validator.validatePhone("91234567")).isFalse();
    assertThat(validator.validatePhone("9123456789")).isFalse();
  }
```

In the `TestCache` class, the behavior of the cache was verified in the `CurrencyExchangeService` class, ensuring it expires after a specified time interval.

```java

public class TestCache {

  private static CurrencyExchangeService currencyService = new CurrencyExchangeService(6000);

  @Test
  @DisplayName("Check when time to live expires the cache is invalid, otherwise the cache must be valid")
  void testCacheAfterTimetoLiveExpires() throws Exception {
    currencyService.exchange("EUR", "USD");
    assertThat(currencyService.checkCache()).isTrue();

    Thread.sleep(7000);
    assertThat(currencyService.checkCache()).isFalse();
  }

  @Test
  @DisplayName("Check if the list of currencies has the right values EUR and USD")
  void testListCurrencies() throws Exception {
    assertThat(currencyService.listCurrencies()).contains("EUR", "USD");
  }

}

```

One example of integration test is this one, it verifies if the /api/bus/getAll endpoint returns a JSON array containing information about all available buses. It checks the HTTP response status, content type, and the presence of specific bus names in the response body.

```java

  @Test
  @DisplayName("Test endpoint get a list containing all the buses")
  void testGetAllBuses() throws Exception {
    mvc.perform(get("/api/bus/getAll").contentType("application/json"))
      .andExpect(status().isOk())
      .andExpect(content().contentType("application/json"))
      .andExpect(jsonPath("$", hasSize(4)))
      .andExpect(jsonPath("$[0].name", is("Flix Bus 1")))
      .andExpect(jsonPath("$[1].name", is("Flix Bus 2")))
      .andExpect(jsonPath("$[2].name", is("Flix Bus 3")))
      .andExpect(jsonPath("$[3].name", is("Flix Bus 4")));
  }

```

Additionally, integration tests can also verify the behavior of service layer logic. In this test, it was created a mock of the busRepository to simulate saving a bus object. Then it's call the

addBus method of the busService and verify that the saved bus object matches the expected result.

```java
@Test
@DisplayName("Check save bus")
void checkSaveBus() {
  Bus bus = new Bus();
  bus.setName("Flix Bus 6");
  bus.setCapacity(60);

  when(busRepository.save(bus)).thenReturn(bus);

  assertThat(busService.addBus(bus)).isEqualTo(bus);

  verify(busRepository, times(1)).save(bus);
}
```

Lastly, integration tests can also ensure that database interactions work correctly. Here's an example of a repository integration test:

```java
@Test
@DisplayName("When a new bus trip is saved, then it should be found it by its ID")
void whenNewBusTripSaved_thenFindById() {
  BusTrip busTrip = new BusTrip();
  busTrip.setFromCity("Lagos");
  busTrip.setToCity("Santarém");
  busTrip.setDate("2021-06-01");
  busTrip.setTime("11h32");
  busTrip.setPrice(12.45);
  busTrip.setBusId(1);

  busTripRepository.save(busTrip);

  BusTrip found = busTripRepository.findById(busTrip.getId());

  assertThat(found.getId()).isEqualTo(busTrip.getId());
  assertThat(found).isEqualTo(busTrip);
}
```

3.3 Functional testing

For user-facing test cases, it was considered only one scenario which was a user buys a ticket without errors or bugs, for that it was defined the following file with a feature:

```
Feature: Use BusWise to buy a ticket

Scenario: User buys a ticket

  Given a user enters on the website and sees the homepage
  When the user selects the departure city 'Castelo Branco'
  And the user selects the destination city 'Figueira da Foz'
  And clicks on the search button
  Then the user sees the available trips
  And clicks on the button choose trip
  Then the user confirms the price of the trips
  And fills in the form with the personal data
  And chooses seat number 14
  And clicks on the button buy reservation
```

3.4 Code quality analysis

For static code analysis, it was utilized SonarCloud, a powerful tool that provides automated code review to detect bugs, vulnerabilities, and code smells in a codebase. Interpreting the results from SonarCloud allows us to gain insights into the overall health of our codebase and identify areas for improvement. By addressing the issues reported by SonarCloud, we can enhance the maintainability, reliability, and security of our application. One particular code smell that it wasn't solved and I also didn't understand it well was field injection with the @Autowired element, Sonar Cloud advertise and suggest use constructor injection instead,

3.5 Continuous integration pipeline [optional]

Yes, we implemented a Continuous Integration (CI) pipeline for our project using GitHub Actions. The CI pipeline automates the process of building, testing, and analyzing our codebase with every change pushed to the repository. We created two GitHub Actions workflows to achieve this.

Java CI with Maven: This workflow is triggered on every push to the main branch of the repository. It builds the project and runs unit tests using Maven with JDK 17. Here's an illustration of the setup:

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'

    - name: Build and run unit tests with Maven
      run: cd HW1/backend && mvn clean test
      continue-on-error: false
```

SonarCloud: This workflow is triggered on every push to the main branch and on pull requests. It performs a code analysis using SonarCloud to evaluate the quality of our codebase. Here's an illustration of the setup:

```yaml
name: SonarCloud
on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and perform a code analysis
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: 17
          distribution: "zulu"
      - name: Cache packages from SonarCloud
        uses: actions/cache@v3
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache packages from Maven
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom-xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Build project and perform analysis
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: cd HW1/backend && mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=zegameiro_TQS_108840
```

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/zegameiro/TQS_108840
Video demo	Video available in the personal repository
QA dashboard (online)	https://sonarcloud.io/project/overview?id=zegameiro_TQS_108840

Reference materials

https://www.baeldung.com/junit-5

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

https://docs.cucumber.io/

https://www.selenium.dev/documentation/

https://site.mockito.org/

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/