HW1: Mid-term assignment report

Ana Alexandra Antunes [876543], v2022-04-07

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 project centers around "Pickabus," a web-based application designed to simplify bus ticket bookings. Pickabus serves as a platform for users to search and book bus tickets for various routes, offering a user-friendly interface for selecting trip details, such as origin, destination, date, and number of tickets. The application aims to streamline the process of finding and securing bus seats, enhancing the travel planning experience for its users.

Application Purpose

Pickabus is developed with the goal of providing a comprehensive and accessible service for bus travelers. By aggregating information on available trips from multiple bus operators, it allows users to:

- Search for bus trips across different cities and countries.
- Compare prices and schedules.
- Book tickets instantly with the option to select their preferred currency.
- Manage their reservations effectively.

The application is envisioned to cater to both occasional travelers and regular commuters, facilitating a smoother and more efficient ticket booking process. It addresses common challenges in bus travel, such as the difficulty in comparing prices, the inconvenience of visiting multiple websites to find suitable trips, and the complexity of booking tickets for international travel due to currency differences.

1.2 Current limitations

The biggest limitation found was concerning the SonarQube because it was impossible to get it to work properly.

2Product specification

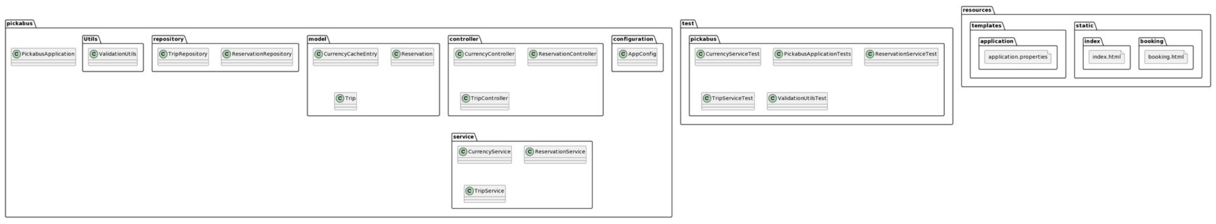
2.1 Functional scope and supported interactions

Pickabus is a web-based application designed to streamline the process of booking bus tickets. It serves various actors, including passengers seeking to reserve seats on bus trips and administrators managing trip schedules and bookings. The application allows users to search for available bus routes based on origin, destination, and date, view detailed trip information, and proceed with booking tickets.

Main Usage Scenario:

- Searching for Trips: Users can search for bus trips by specifying the origin, destination, and departure date. The system then presents a list of available options that match the criteria.
- Selecting a Trip: Users can view detailed information about each trip, including time, price, and the number of available seats, and then select the most suitable option.
- Booking Tickets: After selecting a trip, users proceed to book their tickets by providing passenger details and selecting the number of seats. They can also choose their preferred currency for payment.
- Confirmation: Users receive a confirmation of their booking, including a reservation token, which can be used to retrieve the booking details later.

2.2 System architecture



The architecture of the "Pickabus" application is structured following a typical multi-layer design, ensuring separation of concerns and high maintainability. The diagram below illustrates the overall architecture of the system:

Configuration:

- AppConfig: Centralizes the application's configuration settings, ensuring easy management and updates as needed.

Model:

- CurrencyCacheEntry, Reservation, Trip: These classes represent the domain model, encapsulating the core business logic and data structure.

Repository:

- TripRepository, ReservationRepository: Serve as data access layers that communicate with the database to fetch and persist application data.

Service:

- CurrencyService, ReservationService, TripService: Contain business logic and service-related operations, acting as intermediaries between the controllers and repositories.

Controller:

- CurrencyController, ReservationController, TripController: Handle incoming HTTP requests, invoke service layer operations, and return appropriate responses.

Utils:

- ValidationUtils: Provides utility methods for common validations across the application.

Resources:

- static and templates directories contain static resources and view templates, respectively. These are used for the UI layer where index.html and booking.html are located.

Tests:

- CurrencyServiceTest, PickabusApplicationTests, ReservationServiceTest, TripServiceTest, ValidationUtilsTest: These classes are dedicated to testing the application's functionality, ensuring reliability and robustness.

Technologies/Frameworks:

- Spring Boot: Serves as the foundation for creating stand-alone, production-grade Spring-based applications with minimal configuration.
- Spring MVC: Provides a model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- Spring Data JPA: Simplifies data access within the Spring application, interacts with the database, and provides CRUD operations.
- JUnit: Facilitates unit testing of the application, ensuring that code changes do not break existing functionality.
- Mockito: Offers mocking frameworks for unit tests to simulate the behavior of complex, real objects.
- PlantUML: Used to generate the visual diagram representing the project's structure, aiding in documentation and understanding of the system's architecture.

2.3 API for developers

Our application exposes a set of RESTful APIs that allow developers to programmatically interact with the platform. Below is a documentation of the available endpoint groups.

Problem Domain Endpoints
These endpoints provide access to core functionalities related to trip management and reservation services:

- POST /api/trips/booking: Allows booking a trip by specifying the trip details and passenger information.
- GET /api/trips/search: Searches for available trips based on origin and destination.
- GET /api/trips/cities: Retrieves a list of cities where trips are available.

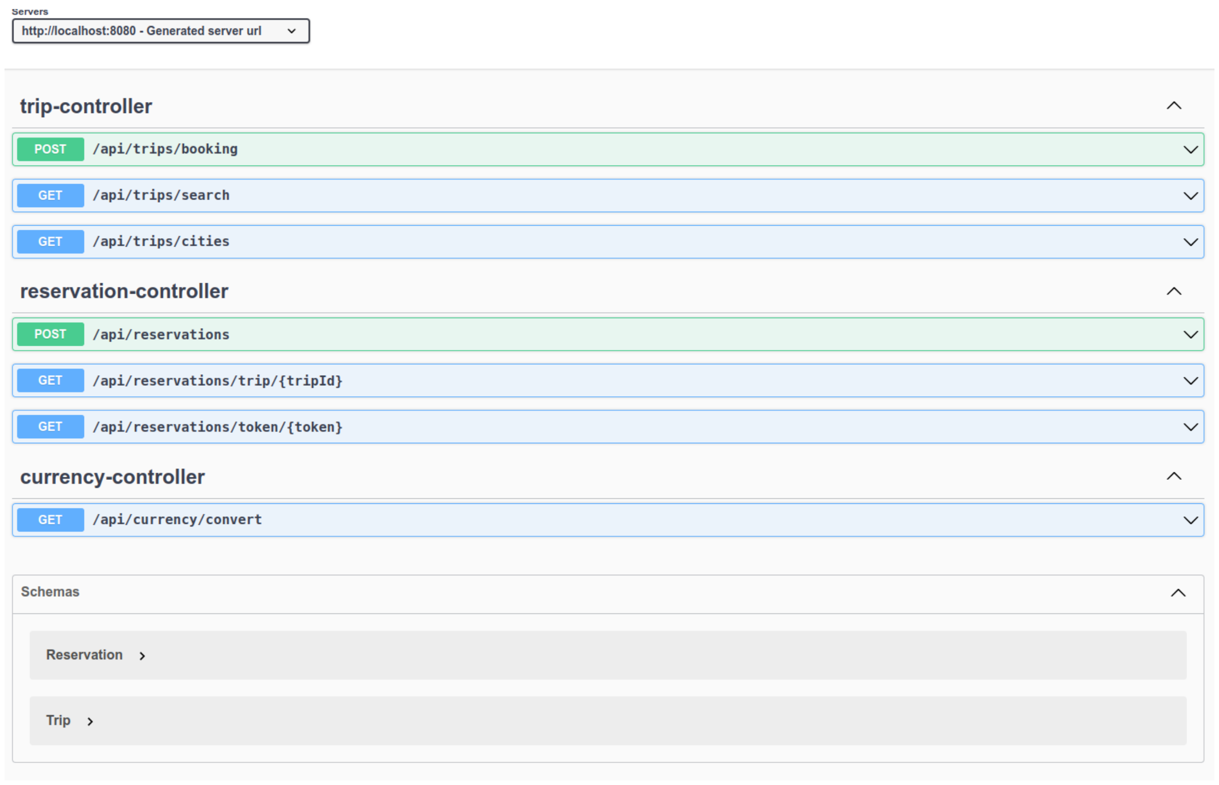
Reservation Management Endpoints
These endpoints are used for managing reservations:

- POST /api/reservations: Submits a new reservation.
- GET /api/reservations/trip/{tripId}: Fetches all reservations made for a specific trip.
- GET /api/reservations/token/{token}: Retrieves a reservation by its unique token.

Currency Conversion Endpoint
This endpoint allows for currency conversion which is vital for international users:

- GET /api/currency/convert: Converts a given amount from one currency to another based on real-time exchange rates.

Cache Usage Statistics (TBD)
Endpoints related to cache statistics are yet to be implemented. They will provide insights into cache performance, including hit and miss rates, which can be essential for optimizing the application.



Servers

http://localhost:8080 - Generated server url

trip-controller

POST /api/trips/booking

GET /api/trips/search

GET /api/trips/cities

reservation-controller

POST /api/reservations

GET /api/reservations/trip/{tripId}

GET /api/reservations/token/{token}

currency-controller

GET /api/currency/convert

Schemas

Reservation

Trip

The image above displays the Swagger UI documentation for our API, offering a user-friendly interface for developers to test and understand the available endpoints.

3Quality assurance

3.1 Overall strategy for testing

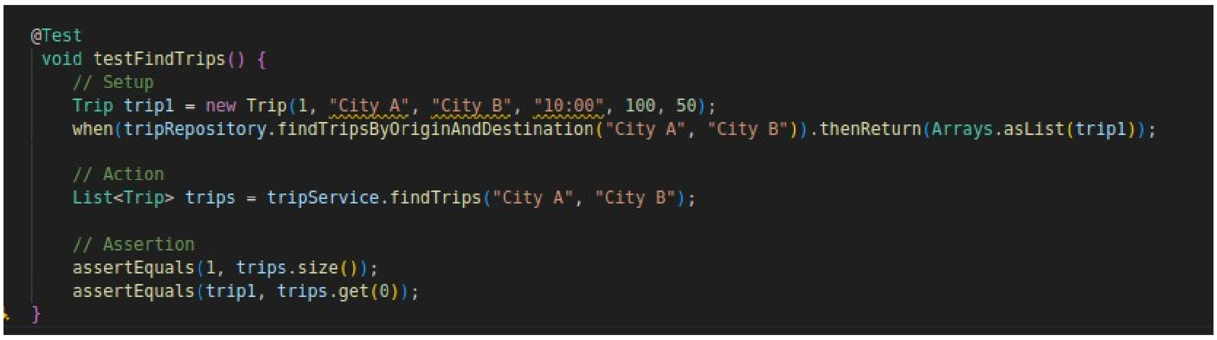
Our test development strategy was a hybrid approach, combining elements of Test-Driven Development (TDD), Behavior-Driven Development (BDD) using Cucumber, and integration testing with REST-Assured. This mix allowed us to ensure the functionality at the unit level, the service level, and the end-user experience.

- TDD: We began with unit tests for the core functionality, ensuring that our codebase was robust from the start.
- Cucumber and BDD: For user-facing features, we employed BDD to write tests in a language that non-technical stakeholders could understand, aligning test cases with user stories.
- REST-Assured: To test our RESTful services, we used REST-Assured, which allowed us to easily simulate HTTP requests and assert the responses.

3.2 Unit and integration testing

Unit and integration tests were used extensively across all microservices. Unit tests focused on individual components, while integration tests verified the interactions between components and with the database.

Implementation Strategy: For unit testing, we mocked dependencies using Mockito to isolate the component being tested. Integration tests were run against a test database to ensure that our services interacted with the database as expected.



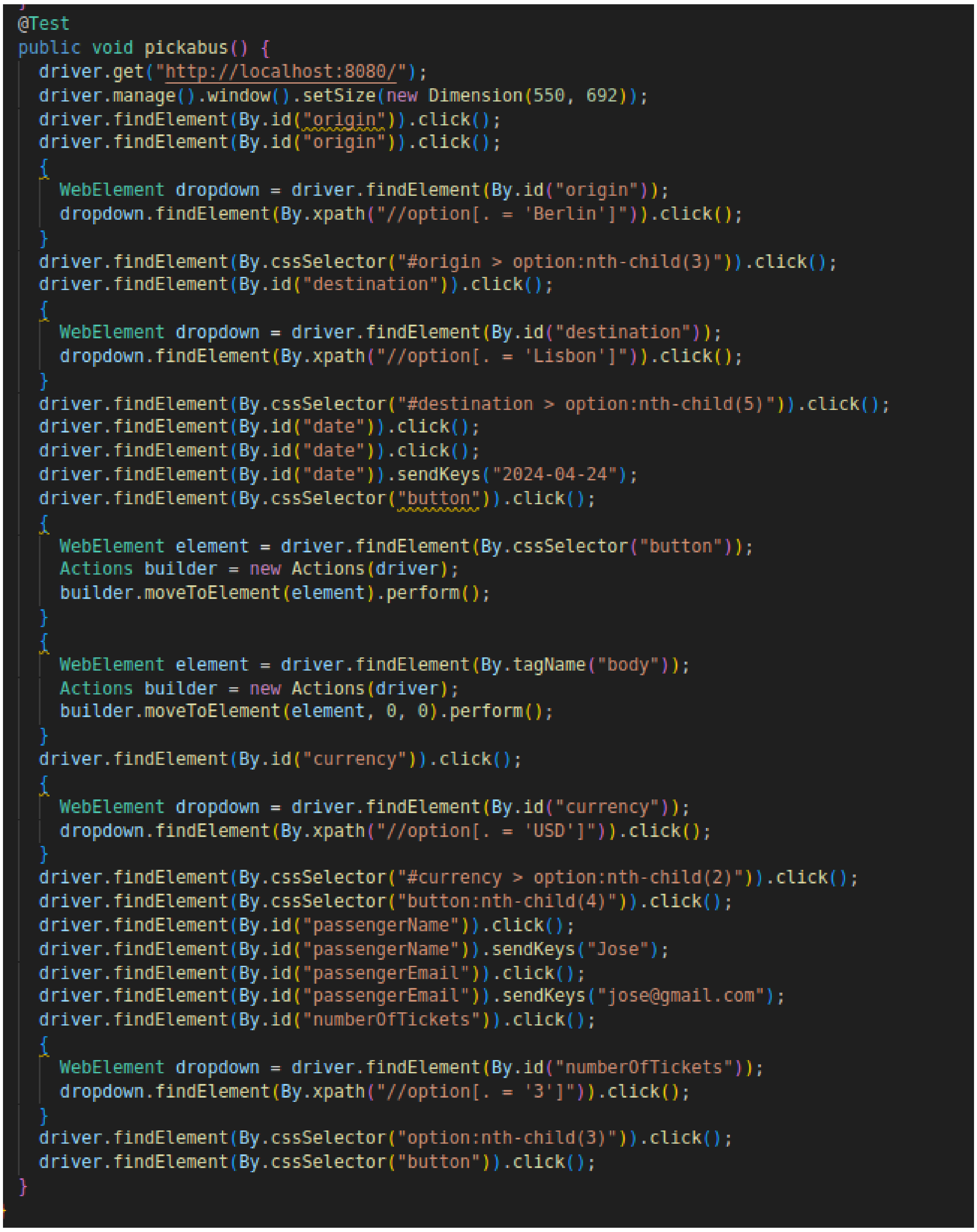
```
@Test
void testFindTrips() {
    // Setup
    Trip trip1 = new Trip(1, "City A", "City B", "10:00", 100, 50);
    when(tripRepository.findTripsByOriginAndDestination("City A", "City B")).thenReturn(Arrays.asList(trip1));

    // Action
    List<Trip> trips = tripService.findTrips("City A", "City B");

    // Assertion
    assertEquals(1, trips.size());
    assertEquals(trip1, trips.get(0));
}
```

3.3 Functional testing

For user-facing test cases, we focused on the end-to-end booking process, from searching for trips to making a reservation. These tests were implemented using Selenium WebDriver for automated browser interactions.



```java
@Test
public void pickabus() {
  driver.get("http://localhost:8080/");
  driver.manage().window().setSize(new Dimension(550, 692));
  driver.findElement(By.id("origin")).click();
  driver.findElement(By.id("origin")).click();
  {
    WebElement dropdown = driver.findElement(By.id("origin"));
    dropdown.findElement(By.xpath("//option[. = 'Berlin']")).click();
  }
  driver.findElement(By.cssSelector("#origin > option:nth-child(3)")).click();
  driver.findElement(By.id("destination")).click();
  {
    WebElement dropdown = driver.findElement(By.id("destination"));
    dropdown.findElement(By.xpath("//option[. = 'Lisbon']")).click();
  }
  driver.findElement(By.cssSelector("#destination > option:nth-child(5)")).click();
  driver.findElement(By.id("date")).click();
  driver.findElement(By.id("date")).click();
  driver.findElement(By.id("date")).sendKeys("2024-04-24");
  driver.findElement(By.cssSelector("button")).click();
  {
    WebElement element = driver.findElement(By.cssSelector("button"));
    Actions builder = new Actions(driver);
    builder.moveToElement(element).perform();
  }
  {
    WebElement element = driver.findElement(By.tagName("body"));
    Actions builder = new Actions(driver);
    builder.moveToElement(element, 0, 0).perform();
  }
  driver.findElement(By.id("currency")).click();
  {
    WebElement dropdown = driver.findElement(By.id("currency"));
    dropdown.findElement(By.xpath("//option[. = 'USD']")).click();
  }
  driver.findElement(By.cssSelector("#currency > option:nth-child(2)")).click();
  driver.findElement(By.cssSelector("button:nth-child(4)")).click();
  driver.findElement(By.id("passengerName")).click();
  driver.findElement(By.id("passengerName")).sendKeys("Jose");
  driver.findElement(By.id("passengerEmail")).click();
  driver.findElement(By.id("passengerEmail")).sendKeys("jose@gmail.com");
  driver.findElement(By.id("numberOfTickets")).click();
  {
    WebElement dropdown = driver.findElement(By.id("numberOfTickets"));
    dropdown.findElement(By.xpath("//option[. = '3']")).click();
  }
  driver.findElement(By.cssSelector("option:nth-child(3)")).click();
  driver.findElement(By.cssSelector("button")).click();
}
```

3.4 Code quality analysis

We used SonarQube for static code analysis to identify code smells, bugs, and security vulnerabilities. (In this part there were dificulties launching SonarQube from docker)

4References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/TiagoC18/TQS_108615
Video demo	Included in the git repository

Reference materials

https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api
https://app.currencyapi.com/
https://docs.sonarsource.com/sonarqube/latest/try-out-sonarqube/
https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner-for-maven/#fix-version
https://www.plantuml.com/