deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Miguel Cruzeiro [107660], v2024-04-09

Introduction

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 is a travel reservation system designed to facilitate trip planning and booking. It allows users to search for trips between cities, make reservations, and view trip details.

Current limitations

All the required features are implemented; however, this is a simple web application that lacks some advanced aspects like filtering trips by departure/arrival date or duration, user authentication so the user could see all his reservations and some error messages in the frontend are not implemented.

Product specification

Functional scope and supported interactions

The application targets travelers that want to plan and book bus trips between cities. Users can perform the following actions:

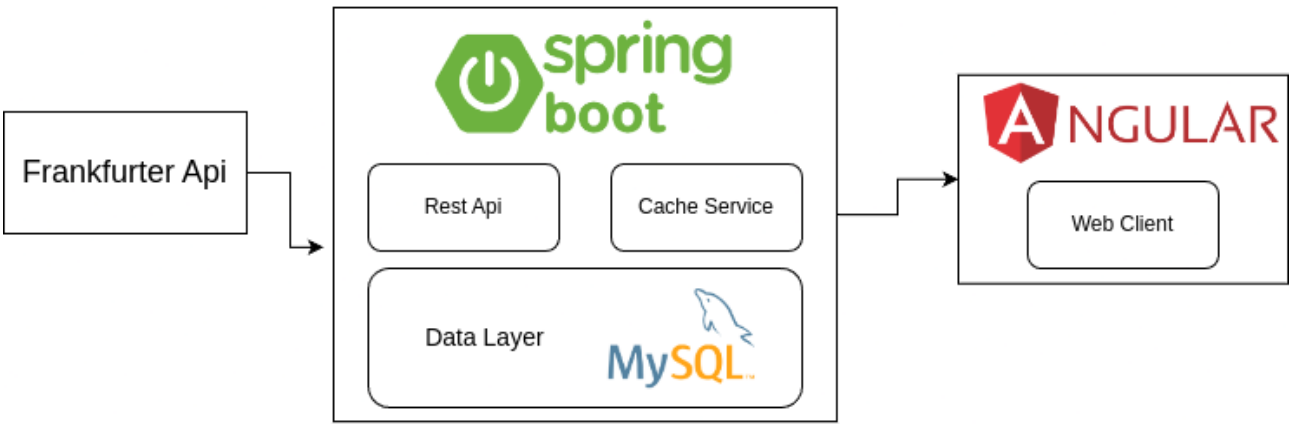
- View Trip Details: Users can view details of available trips, including departure and arrival cities, departure time, and available seats.
- Make Reservations: Users can reserve seats for a selected trip, specifying the number of seats required.
- Search for Trips: Users can search for available trips between two cities, specifying departure and arrival cities.

Main Usage Scenario:

- The user enters the origin city and destination city.
- The system displays a list of available bus trips matching the criteria.
- User selects a trip and provides passenger details (name, and number of seats).
- System calculates the price in the user's preferred currency.
- A reservation is created, and the reservation details are shown on a confirmation page.

System architecture

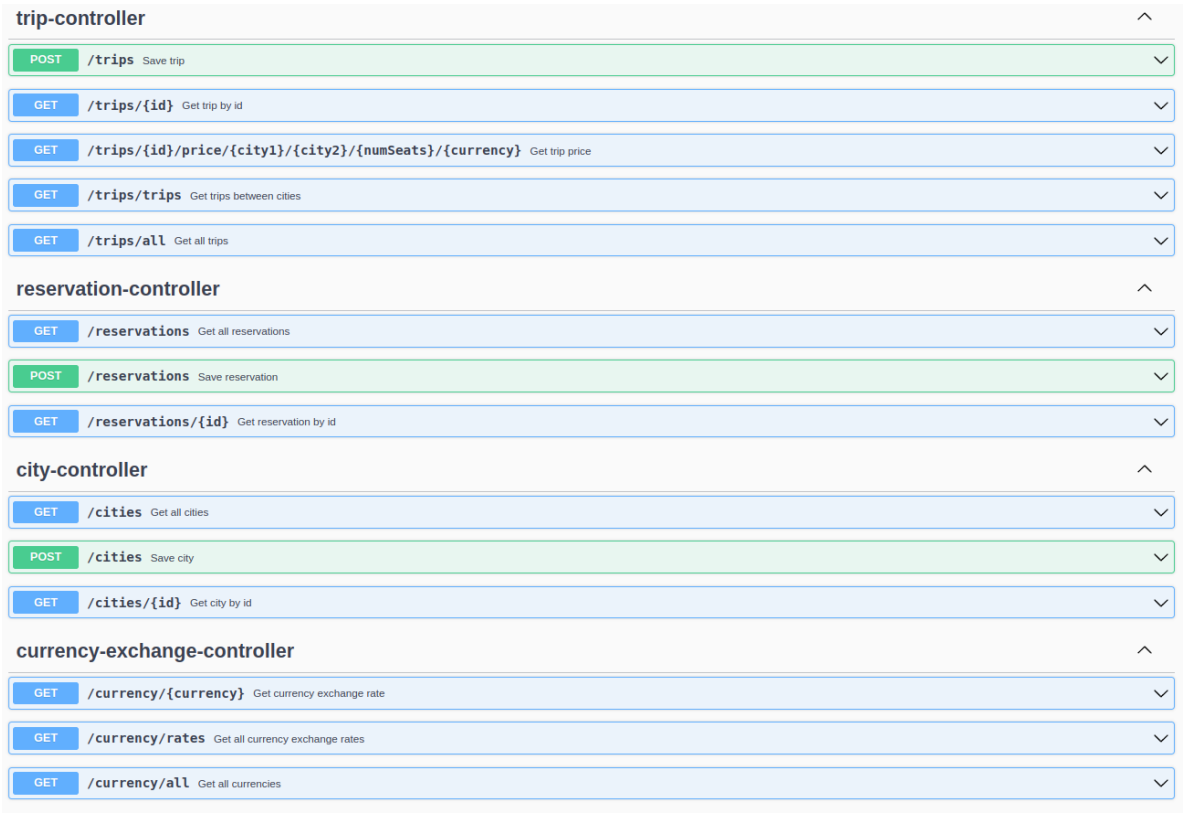
The system makes use of an external Api called “Frankfurter” to get the exchange rates to a list of currencies.
The backend and rest Api were built in Spring Boot and the frontend was developed in Angular.
There is a docker compose file in the root of the project that runs the database, the Api with the port 8080 exposed and the frontend with port 4200 exposed.



API for developers

The Api is documented and can be accessed by the link:

http://localhost:8080/swagger-ui/index.html#/



Quality assurance

Overall strategy for testing

The testing approach focused on ensuring the reliability and effectiveness of the application. A simple yet effective strategy was followed throughout the development process. Starting by developing some basic features like getting a reservation for example, then i developed the test for that same functionality. That allowed me to catch some potential issues when developing the rest of the application.

To cover all aspects of the application, a combination of different techniques was used:

- Unit tests to test individual components.
- Integration tests to check how different parts of the application interact.
- Functional tests to see if the application functions as expected in real world scenarios.

During the development of the tests, SonarQube was used as a static code analysis tool to ensure code quality, identify potential issues and check code coverage to see if any more tests are needed. SonarQube provided valuable insights into various aspects of our codebase, including code smells, bugs, vulnerabilities, and code duplication.

Unit and integration testing

Each unit test focused on testing a specific method or functionality within a class, ensuring that it behaved as expected. Unit tests were implemented using JUnit and Mockito frameworks. Mockito was used to mock an object and simulate their behavior without interacting with the real database and to inject the needed dependencies.
Here's an example of a unit test that ensures that a reservation is returned when a valid id is provided.

```
@Test
void whenValidId_thenReservationShouldBeFound() {
    when(reservationRepository.findById(reservation1.getId())).thenReturn(Optional.of(reservation1));

    Optional<Reservation> found = Optional.ofNullable(reservationService.getReservationById(reservation1.getId()));

    assertTrue(found.isPresent());
    assertEquals(reservation1.getId(), found.get().getId());

    verify(reservationRepository, times(wantedNumberOfInvocations:1)).findById(reservation1.getId());

}
```

After all the components needed are developed, integration tests were done to ensure the application behaves correctly. For these tests, real calls were made to the Api and checking if it retrieves the correct response. These tests use a Spring Framework that configures the tests to run in a real web environment with a random port and uses restTemplate to make http requests. Here's an example of an integration test that checks if the price of a trip between two cities matches the expected value.

```
@Test
void testGetTripPrice() {
    ResponseEntity<Double> response = restTemplate.exchange("http://localhost:" + randomServerPort
            + "/trips/1/price/Porto/Lisboa/1/EUR",
            HttpMethod.GET, requestEntity:null, new ParameterizedTypeReference<Double>() {
            });

    Double price = response.getBody();

    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(expected:10.0, price);
}
```

Some tests were also made to test the cache functionality. It invokes the method to get all the currency exchanges twice and it verifies that in the second call there were no more interactions with the restTemplate.

```
@Test
void testCacheUsage() {
    when(restTemplate.getForObject(url:"https://api.frankfurter.app/latest", responseType:Map.class))
            .thenReturn(exchangeRates);

    Map<String, Double> result1 = currencyExchangeService.getAllCurrencyExchanges();
    assertThat(result1).containsEntry(key:"EUR", value:1.00);
    verify(restTemplate, times(wantedNumberOfInvocations:1)).getForObject(anyString(), eq(value:Map.class));

    Map<String, Double> result2 = currencyExchangeService.getAllCurrencyExchanges();
    assertThat(result2).containsEntry(key:"EUR", value:1.00);
    verifyNoMoreInteractions(restTemplate);
}
```

Functional testing

Functional tests were done using Selenium that allows to navigate to the web page and Cucumber to define the test steps. A functional test was made simulating the normal behavior of a person reserving a bus trip.
Here are some illustrative code snippets from the test:

```
@And("I choose my departure city {string} and my destination city {string}")
public void iChooseDepartureAndDestinationCities(String departureCity, String destinationCity) {
    log.debug(format:"Choose departure city {} and destination city {}", departureCity, destinationCity);
    driver.findElement(By.id(id:"initialCity")).sendKeys(departureCity);
    driver.findElement(By.name(name:"finalCity")).sendKeys(destinationCity);
}

@And("I fill in my number of seats as {int}")
public void iFillInMyCityAs(Integer seats) {
    log.debug(format:"Fill in number of seats as {}", seats);
    driver.findElement(By.id(id:"numSeats")).sendKeys(seats.toString());
}
```

Code quality analysis

To analyze the code quality SonarQube was used. SonarQube is useful for detecting potential bugs, code smells and other issues. It was also used during the development of the tests to check the coverage and if any more tests were needed.
With this tool i was able to remove all the issues and code smells.
Here's an example of a code smell that i would never consider if it wasn't SonarQube:

```
        Map<String, Double> rates = new HashMap<>();
        Map<String, Object> ratesMap = exchangeRates.get(RATES_STRING);
        for (Map.Entry<String, Object> entry : ratesMap.entrySet()) {
            String currency = entry.getKey();
            Object rateObject = entry.getValue();
            if (rateObject instanceof Number) {
```

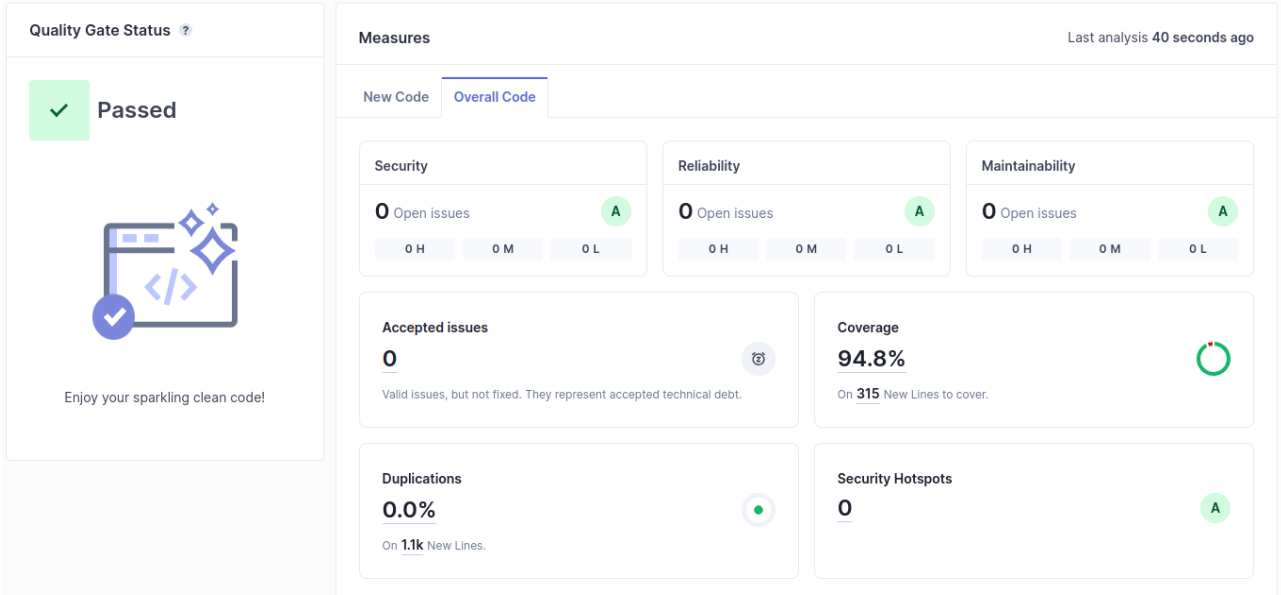
Replace this instanceof check and cast with 'instanceof Number number'

```
                rates.put(currency, ((Number) rateObject).doubleValue());
            } else {
                logger.error("Non-numeric value found for currency {}: {}", currency, rateObject);
            }
        }
        return rates;
    }
```

The solution was this:

```java
Map<String, Double> rates = new HashMap<>();
Map<String, Object> ratesMap = exchangeRates.get(RATES_STRING);
for (Map.Entry<String, Object> entry : ratesMap.entrySet()) {
    String currency = entry.getKey();
    Object rateObject = entry.getValue();
    if (rateObject instanceof Number number) {
        rates.put(currency, (number).doubleValue());
    } else {
        logger.error(format:"Non-numeric value found for currency {}: {}", currency, rateObject);
    }
}
return rates;
```

This is a screenshot of SonarQube before the submission.



References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/MiguelCruzeiro/TQS_107660/tree/main/HW1
Video demo	In the repository

Reference materials

Currency data Api:

https://github.com/hakanensari/frankfurter