HW1: Mid-term assignment report

Miguel Belchior Figueiredo [108287], 2024-06-09

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 designed system helps users search and buy bus tickets between the desired cities/bus stops. The system is designed in a way that when a user queries for a trip between two cities it returns not only the direct trips that comply with the origin city and destination but also other indirect trips. The system allows to book reservations and specify the seat on the trip bus that the user would like to sit in.

The target of this assignment was to develop such an application and at the same time test that application with different kinds of tests that differ in their purposes, scopes and loaded environments: unit tests, service level tests, integration tests and functional tests. Personally, I followed a TDD (Test Driven Development) process while following a top-down approach: start from the controller, then the service, then the repository.

1.2 Current limitations

In the frontend, it is assumed that the calls to the backend layer and currency API never fail which can result in parsing errors. In the frontend, it's only considered the success case where the user makes a reservation. If he can't because of trying to make a reservation on a full bus then the frontend will get an error. However, all these details are accounted for, in the backend layer of this project despite not being able to fully implement the solution on the frontend due to time constraints.
At the backend layer, one of the tests regarding the trip repository would not pass if run together with other tests (only passed running it alone). The reason for this behavior is unknown, especially because the corresponding repository seems to be well implemented.

2 Product specification

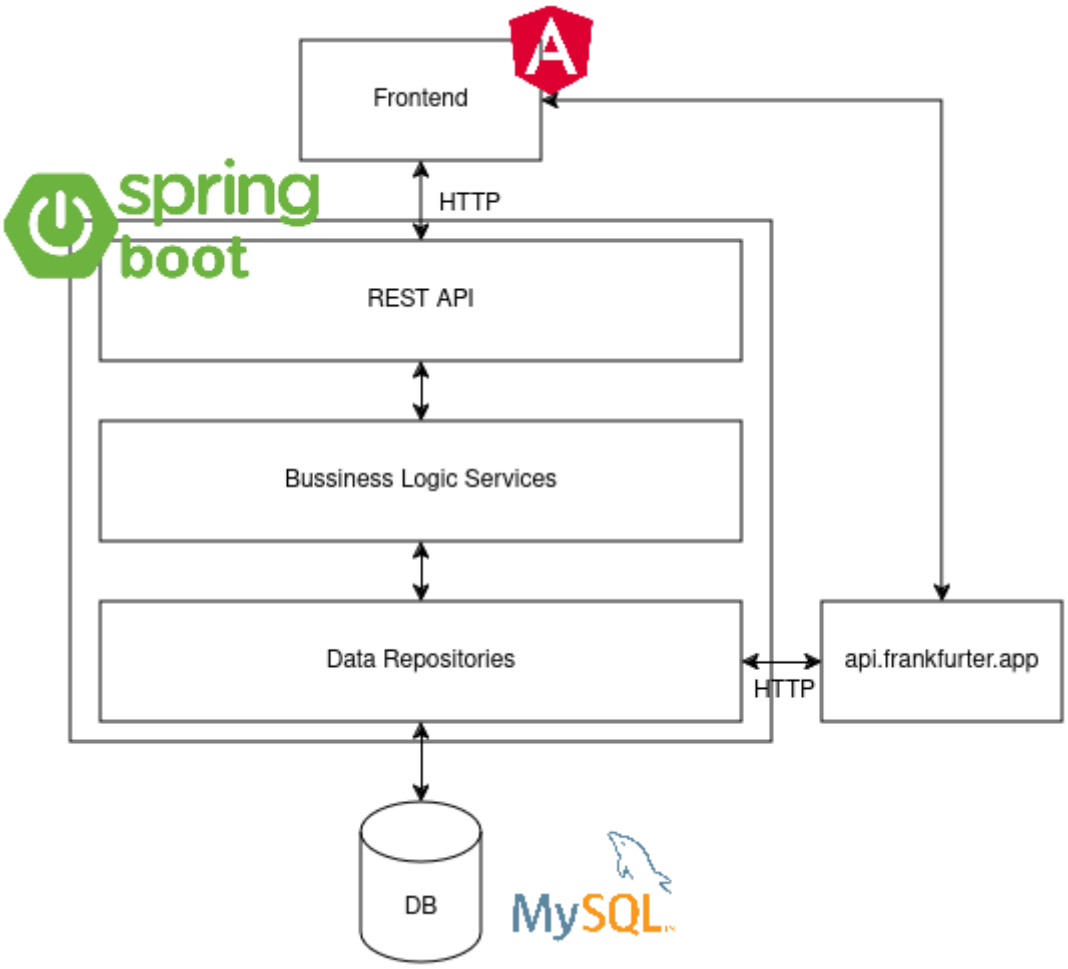
2.1 Functional scope and supported interactions

There is only one user in this application which encompasses the entirety of the following usage scenario

1. The user enters the main page and selects from a list of the system bus stops, the desired origin and destination city for his trip. Furthermore, the user must specify the date and desired currency in which he wishes to receive the matching trips price.
2. Then the system shows the user the corresponding trips (along with the relevant information) for which he can go from his origin city to his destination city. As mentioned previously, this may be a direct or indirect trip leaving the user the choice between all possible trips.
3. After choosing one trip, then the user must provide the details of the reservation he desires to make: seat number on the bus and his name and address.
4. If the backend validates the desired reservation, the newly made reservation is returned to the frontend that shows a successful message with the reservation details.

2.2 System architecture

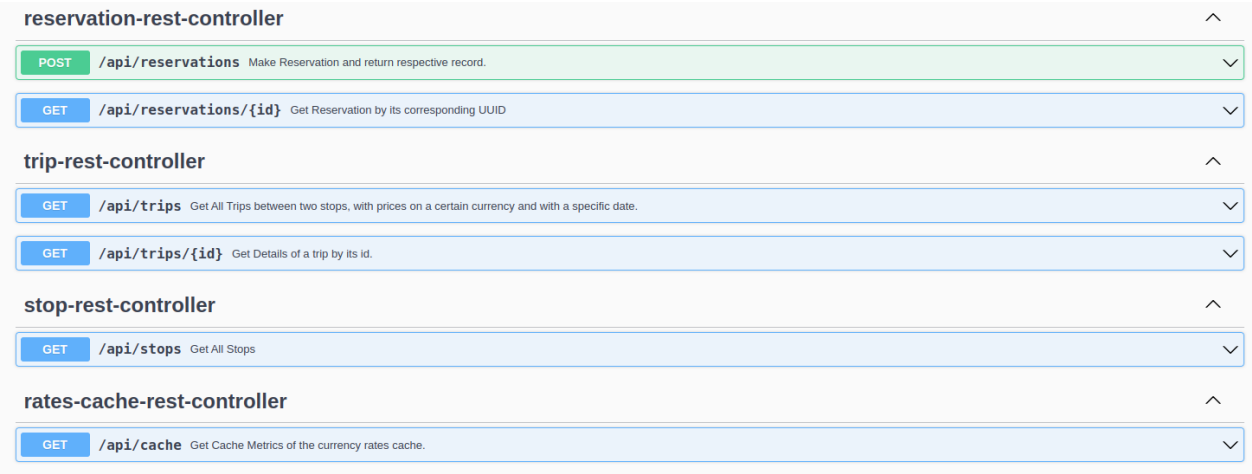
The system architecture has a frontend which communicates with the backend through a rest API using HTTP. The frontend was built using angular and boostrap and communicated with the backend for trip, bus stops and reservation related data and with an external api to query for a list of currencies (which I could then use to ask for the backend for trips with prices in a certain currency). The backend was built using spring boot which communicated through repositories to fetch and persist data on a mysql database. It also communicated with the external api to ask for currency conversion rates. Despite in the diagram the API and the data repositories communicate with the external api this data is only saved on the requested cache (not in the persistent database). The diagram was designed this way to emphasize separation between business logic services and data access repositories. In this way, there is a repository that wraps the API and returns it to the desired services decoupling the logic that handles the data and the logic of fetching such data.



2.3 API for developers

The API documentation can be found on http://localhost:8080/swagger-ui/index.html when running the spring application. There are 4 sets of endpoints:

1. **Reservation Controller** – Handles the creation of reservation and fetching of reservations by its unique UUID;
2. **Trip Controller** – Handles the search of all trips between two stops, with a certain currency and with a specific date. It also allows to search a trip by its id;
3. **Stop Controller** – Allows querying the system for all of its bus stops;
4. **Rates Controller** – contains an endpoint that returns the statistic of the cache that caches the currency conversion rates;



3 Quality assurance

3.1 Overall strategy for testing

As previously mentioned, I followed a TDD (Test Driven Development) process while following a top-down approach:

1. **Unit Tests** - Performed unit tests on individual components like the rates cache and on DTO classes functions that convert entities to and from the corresponding DTO (data transfer object);
2. **Controller With Mock Service Test** - Tested controllers using RestAssured mockMvc while mocking required services to perform bussiness logic for the controller (using Mockito);
3. **Service With Mock Repository Test** - Tested services with assertJ while mocking services and data repositories which the service being tested depends on;
4. **Repository Test –** Tested custom repository queries;
5. **Integration Tests** – Tested the entirety of the backend in a fully loaded environment using test containers and spring context data loading to instantly start the testing database with valid and production like data. This way I don't have to worry about manually setting up the database with data on the tests (Also used spring context data loading when running the spring boot application). In this case, the tests were realized using RestAssured.

To test the overall system from the user's perspective, I used cucumber for BDD testing and selenium to automatically interact with an instance of a browser, simulating the user.

3.2 Unit and integration testing

As previously mentioned, the followed implementation approach was TDD. The implemented unit tests encompass verifying the rates cache component and ensuring the enforcement of a time-to-live in all its entries. The unit tests also tested the conversion of data entities to and from DTO's. The following print display the testing of the previously mentioned cache:

```
class RatesCacheTest {

  static final Logger logger = getLogger(lookup().lookupClass());
  RatesCache ratesCache;

  @BeforeEach
  void setup(){
    ratesCache = new RatesCache();
  }

  @Test
  void testGetAfterPut() {
    ratesCache.put("USD", 10f);
    assertEquals(10f, ratesCache.get("USD"));
    assertEquals(1, ratesCache.getMetrics().getCacheHits());
    assertEquals(0, ratesCache.getMetrics().getCacheMisses());
  }

  @Test
```

```
  void testGetNoPut() {
    assertNull(ratesCache.get("USD"));
    assertEquals(0, ratesCache.getMetrics().getCacheHits());
    assertEquals(1, ratesCache.getMetrics().getCacheMisses());
  }

  @Test
  void testGetAfterTtl(){
    ratesCache.setTtl(1);
    ratesCache.put("ALL", 15f);

    // Compliant with sonar (instead of sleep)
    await().atMost(2, TimeUnit.SECONDS).until(() -> ratesCache.get("ALL") == null);
    assertNull(ratesCache.get("ALL"));
    assertEquals(2, ratesCache.getMetrics().getCacheMisses());
  }

  @Test
  void testGetMetrics() {
    ratesCache.get("naoexiste");
    ratesCache.get("naoexiste");
    ratesCache.get("naoexiste");
    ratesCache.put("USD", 10f);
    ratesCache.get("USD");

    assertEquals(1, ratesCache.getMetrics().getCacheHits());
    assertEquals(3, ratesCache.getMetrics().getCacheMisses());
  }
}
```

The previous methods all test different aspects of a general cache with a string-float mapping:

1. **TestGetAfterPut()** - tests if the previously inserted value is saved on the cache;
2. **TestGetNoPut()** - tests the cache metrics after a cache miss;
3. **TestGetAfterTtl()** - tests that the time to live logic is being enforced and the value disappears after that time;
4. **TestGetMetrics()** - tests cache metrics for a series of cache interactions.

The integration tests test all controllers of the application on a fully loaded environment. An example of that use case are the reservation controller integration tests which focuses mainly on the endpoint responsible for making reservations and its error prone scenarios like reservating a seat on a full trip, reservating an occupied seat. It is worth noting the use of test containers and spring data loading for simplifying the test development and execution process.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Testcontainers
@TestPropertySource(locations = "classpath:application-it.properties")
public class ReservationControllerIT {

  static final Logger logger = getLogger(lookup().lookupClass());

  @Container
  public static MySQLContainer container = new MySQLContainer<>(DockerImageName.parse("mysql:latest"))
      .withUsername("miguel")
```

```
        .withPassword("password")
        .withDatabaseName("test");
@DynamicPropertySource
static void properties(DynamicPropertyRegistry registry) {
  registry.add("spring.datasource.url", container::getJdbcUrl);
  registry.add("spring.datasource.password", container::getPassword);
  registry.add("spring.datasource.username", container::getUsername);
}
@LocalServerPort
int randomServerPort;
@BeforeEach
public void setUp(){
  RestAssured.port = randomServerPort;
  // data inserted through data_test.sql -> no setup required
}

@Test
void givenInvalidTrip_whenMakeReservation_thenReturnStatus404() {
  given().
        contentType(ContentType.JSON).
        body(new ReservationDTO(1132312L, 1, "Cliente", "Morada")).
  when().
        post("api/reservations").
  then().
        statusCode(HttpStatus.SC_NOT_FOUND).
        body(is(Matchers.emptyOrNullString()));
}

@Test
void givenValidTrip_whenMakeReservationOnInvalidSeat_thenReturnStatus409() {
  given().
            contentType(ContentType.JSON).
            body(new ReservationDTO(1L, 1312321, "Cliente", "Morada")).
        when().
            post("api/reservations").
        then().
            statusCode(HttpStatus.SC_CONFLICT).
            body(is(Matchers.emptyOrNullString()));
}

@Test
void givenValidFullTrip_whenMakeReservation_thenReturnStatus409() {
  given().
        contentType(ContentType.JSON).
        body(new ReservationDTO(2L, 1, "Cliente", "Morada")).
  when().
        post("api/reservations").
  then().
        statusCode(HttpStatus.SC_CONFLICT).
        body(is(Matchers.emptyOrNullString()));
}
```

```
@Test
void givenValidNotFullTrip_whenMakeReservationOnOccupiedSeat_thenReturnStatus409() {
  given().
        contentType(ContentType.JSON).
        body(new ReservationDTO(1L, 1, "Cliente", "Morada")).
  when().
        post("api/reservations").
  then().
        statusCode(HttpStatus.SC_CONFLICT).
        body(is(Matchers.emptyOrNullString()));
}
@Test
void givenValidNotFullTrip_whenMakeReservation_thenReturnReservation() {
  given().
        contentType(ContentType.JSON).
        body(new ReservationDTO(3L, 2, "Cliente", "Morada")).
  when().
        post("api/reservations").
  then().
        statusCode(HttpStatus.SC_OK).
        body("trip.id", is(3));

  // verify available seat numbers on reserved trip
  given().
  when().
        get("api/trips/3").
  then().
        statusCode(HttpStatus.SC_OK).
        body("id", is(3)).
        body("availableSeats", not(contains(2))).
        body("availableSeats", contains(1,3,4));
}
@Test
void givenValidUUID_whenSearchReservation_thenReturnReservation() {
  given().
  when().
        get("api/reservations/aa0f7252-309c-11ea-a72a-0242ac130002").
  then().
        statusCode(HttpStatus.SC_OK).
        body("trip.id", is(1));
}
@Test
void givenInvalidUUID_whenSearchReservation_thenReturnStatus404() {
  given().
  when().
        get("api/reservations/aaaaaaaa-aaac-11ea-a72a-0242ac130002").
  then().
        statusCode(HttpStatus.SC_NOT_FOUND).
        body(is(Matchers.emptyOrNullString()));
}
```

3.3 Functional testing

There were written user-facing tests using cucumber following a BDD approach which emphasizes the previously mentioned main use case: a user searches for trips, selects a trip and makes a reservation.

```
@bustickets
Feature: Bustickets Flow
  To allow a customer to find his and buy a bus ticket.

  Scenario: Reserving a bus ticket
    Given I am on "http://localhost:4200/"
    When I choose my flight from "Lisboa" to "Aveiro"
      And I choose date 2024-06-02
      And I choose currency USD
      And I click search trips
    Then 2 trips should be found
      And found trip 1 should be presented with Route, Bus Capacity and Available Seats "Lisboa -> Coimbra -> Aveiro" "50" "2,3,4,5,6,7,8,9,10,11,12,13,1
      And I select trip 1
      And selected trip should have form to fill in for seatNumber clientName clientAddress
      And I fill in my reservation information 3 "ClienteCucumber" "MoradaCucumber"
      And make my reservation
    Then should be presented successful reservation message
```

If it weren't the time constraint, I would have made more tests regarding the following use cases:

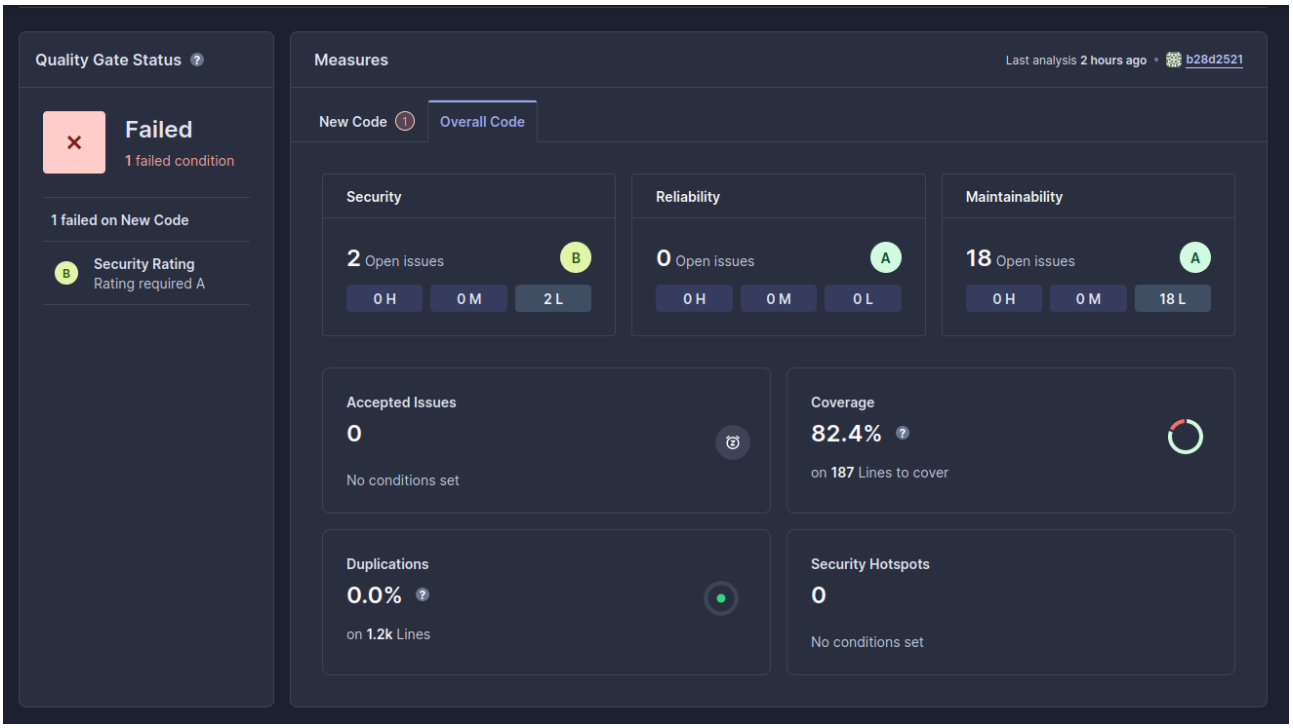
1. Get reservation details by the reservation id that was returned on the previous use case;
2. Use cases regarding the form handling errors on the frontend to ensure that API calls were made only after all arguments being non null and selected by the user.

3.4 Code quality analysis

For static code analysis I used sonar cloud with a custom quality gate that only differs from the default one by allowing a coverage of more than 80% of coverage. From the static code analysis, some of the annoying reported issues by the tool were:

- The logging of “user data” when logging asked currency for a trip which breaks the security rating (the 2 open issues regarding security are because of this);
- Asking for removal of unused imports and public modifiers of classes.

Regarding the results, they are good despite not passing the quality gate because of the previously mentioned security issues that break down the security rating to a B. Due to the time constraint I couldn’t complete the correction of the reported issues by the tool. Regarding coverage 82.4% is a pretty positive value as most of the code is covered by the implemented tests. The nonexistence of duplications and security hotspots is also a positive remark.



3.5 Continuous integration pipeline [optional]

I implemented a CI pipeline to run all unit tests and do static code analysis to sonar cloud every time a commit is done to the main branch of the backend folder. Two workflow files were done:

1. The sonar cloud build.yaml file that does the static code analysis. Furthermore, also runs all unit tests (-DskiplTs skip all integration tests);

```yaml
name: SonarCloud
on:
  push:
    branches:
      - main
    paths:
      - "hw1/**"
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and analyze
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0  # Shallow clones should be disabled for a better relevancy of analysis
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: 17
          distribution: 'zulu' # Alternative distribution options are available.
      - name: Cache SonarCloud packages
        uses: actions/cache@v3
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=Migas77_TQS_108287 -DskipITs
        working-directory: hw1/hw1_bustickets
```

2. A maven workflow that also runs verify and tests all unit tests only. It wasn't required as the last workflow already ran the tests, but it was also configured.

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
    paths:
      - "hw1/**"
  pull_request:
    branches: [ "main" ]
    paths:
      - "hw1/**"

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'
        cache: maven
    - name: Build with Maven
      run: mvn -B package --file pom.xml -DskipITs
      working-directory: hw1/hw1_bustickets
```

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/Migas77/TQS_108287
Video demo	https://github.com/Migas77/TQS_108287/tree/main/hw1/hw1_bustickets/solution_video
QA dashboard (online)	https://sonarcloud.io/summary/new_code?id=Migas77_TQS_108287&branch=main
CI/CD pipeline	https://github.com/Migas77/TQS_108287/tree/main/.github/workflows

Reference materials

https://github.com/rest-assured/rest-assured
https://www.baeldung.com/spring-boot-data-sql-and-schema-sql
https://getbootstrap.com/docs/5.0/getting-started/introduction/