

HW1: Mid-term assignment report

Rodrigo Silva Aguiar [108969], v2023-03-24

1	Introduction	1
1.1	Overview of the work	1
1.2	Current limitations	2
2	Product specification	3
2.1	Functional scope and supported interactions	3
2.2	System architecture	3
2.3	API for developers	5
3	Quality assurance	5
3.1	Overall strategy for testing	5
3.2	Unit and integration testing	5
3.3	Functional testing	6
3.4	Code quality analysis	8
3.5	Continuous integration pipeline [optional]	10
4	References & resources	12

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.

<briefly introduce your application: name the product, if applicable; what is its purpose?>

The application consists of a webpage where users can buy bus tickets for various trips between cities (origin and destination) for a given date and time and check reservations made for trips.

To support this webpage, a spring-boot backend is present, along with a H2 in-memory database.

In order to assure the correct functioning of both the backend and the frontend components, several types of tests were developed for the application, such as :

- Unit Tests (with Junit 5)
- Service Level Tests (with Mockito)
- Integration Tests (with SpringBootTest + MockMvc + TestRestTemplate)

- Functional Tests on the UI (Cucumber + Selenium Web Driver)

To further assure code quality and maintainability, SonarCloud was also used for static code analysis.

A CI flow was implemented using Github Actions, such that every time code is pushed into the repository, it is automatically tested and analyzed by SonarCloud.

The API is supplied with automatic documentation generated by SwaggerUI, which may be accessed at localhost:8080/docs or {baseApiPath:port/docs}, if not using the default Spring Boot host and port.

1.2 Current limitations

<explain the known limitations → unimplemented or faulty (but expected) features>

Given the project requirements, the implemented solution respects the wanted use cases and implementation features:

There is a REST API implemented in Spring-Boot which can be used by external clients (assuming it is deployed outside of localhost), where clients can search for trips between cities, buy tickets for trips and check ticket reservations made.

There is a simplistic webpage made with React + Tailwind CSS + React Query for API requests that presents the users with a UI to facilitate interacting with the backend.

Price currency may be selected by the user, reflecting current exchange rates. For this, an external API was used, <https://www.exchangerate-api.com>. This API provides real time currency exchange rates for more than 50 currencies, which may then be selected by users in the frontend.

Ticket reservations are assigned with a ID, which may be used to further check the reservation details.

The backend has logging support to the terminal, implemented with SLF4J Api + Logback, registering the various operations that are done in the API.

The current service doesn't have support for:

- The booking of return trips.
- Cancel existing bookings.
- Alter dates in an existing booking.

The caching of exchange rates is currently done using a Java HashMap along with a TTL variable, which works perfectly for the scope of the project, but an upgrade to a proper caching mechanism with cache invalidation should be considered if this was a bigger project.

2 Product specification

2.1 Functional scope and supported interactions

<functional description of the application: who (actors) will use the application and for what? Briefly explain the main **usage scenario**. >

The main usage scenario consists in the user searching for bus trips between two cities, selecting the wanted trip and then buying a trip, with a selected seat.

After this, the user may check his tickets bought through the ticket ID provided on the purchase act and given details.

2.2 System architecture

<briefly present the software architecture. Include one or more diagrams.>

<detail the specific technologies/frameworks that were used>

The application may be divided into two big parts, a frontend component and a backend component.

The frontend component is made with React, using Tailwind CSS and DaisyUI for the component styling, TanStack Query (formerly React Query) for API requests and caching, and Vite in order to provide a server and environment in which to run the application.

The backend component was developed with Spring Boot and may be divided into the following parts:

- Controller Layer

The Controller Layer exposes the backend to the outside world.

It is responsible for receiving HTTP requests, processing them, interacting with the Service Layer to find a response and then return it to the requester.

- Service Layer

The Service Layer is responsible for handling the business logic of the application.

In the service layer, operations such as validating user inputted data and dealing with currency exchange rates are done.

Its also responsible for delegating data saving or retrieving operations to the repository layer and returning the response back to the controller layer.

- Repository Layer

This layer is responsible for interacting with the database and retrieving or saving wanted data.

This layer is implemented with the help of Spring data JPA + Hibernate to automatically generate SQL queries based on a predefine function naming structure.

- Entity Layer

In this layer, entities are stored, which are mapped to tables in the database.

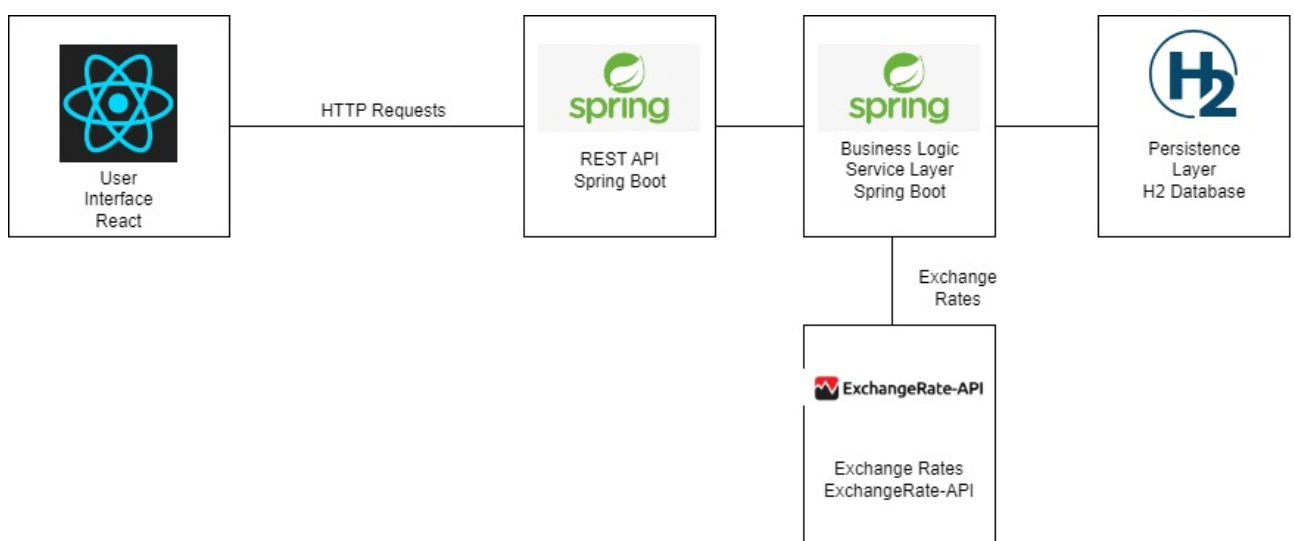
This layer is supported once again by Spring Data JPA which allows direct serialization from POJO (plain old java objects) to SQL tables and entries.

- Persistence Layer

This layer consists of a database, which is used to guarantee the persistence of data such as Buses, Tickets and Trips.

Due to the relatively small size of the project, H2 was chosen as a database, given its ease of implementation within Spring Boot and its speed in queries, due to it being a in-memory database.

The System Architecture may be viewed in the following image:



2.3 API for developers

<what services/resources can a developer obtain from your project? document your API endpoints>

<note: for the homework, you are expected to expose two “groups” of endpoints:

- Problem domain: get the environmental data data by region/city, etc.
- Cache usage statistics: how many hits/misses,... >.

[Base URL: localhost:8080/weather]

client	Regular user of the weather forecast API	▼
GET	/now/{latitude},{longitude}	get weather forecast of the current day for the given coordinates
GET	/recent/{latitude},{longitude}/{days}	get weather forecast of the next days starting from today until the given number of days for the given coordinates
GET	/period/{latitude},{longitude}/{start},{end}	get weather forecast of the given time period for the given coordinates
GET	/cached	get weather forecasts previously requested and still present in cache

3 Quality assurance

3.1 Overall strategy for testing

[what was the overall test development strategy? E.g.: did you do TDD? Did you choose to use Cucumber and BDD? Did you mix different testing tools, like REST-Assured and Cucumber?...]

Firstly, most of the application code was developed, given the vision of the application and the given requirements.

Afterwards, tests were developed for it, considering its expected functionality, and the code was refactored if any tests failed.

Given this, TDD was not done as the code was developed prior to the tests but was changed after if it didn't pass the tests.

Mixing of testing tools was only done in the Functional Testing of the User Interface, mixing Cucumber and Selenium Web Driver.

3.2 Unit and integration testing

[where did you use unit and integration test? for what? which was the implementation strategy?]

[may add some screenshots/code snippets for clarification]

Unit testing was used to test the TTL component of the exchange rate cache and the email / phone validator functions. It was not deemed necessary anywhere else due to the implementation of the architecture and the nature of spring boot applications, which are composed of components that are supposed to work together.

Other candidates for unit tests such as for example checking if a bus is full is done in the controller layer, and therefore, tested there.

```

@Test
void testResultAfterTTLExpire() throws Exception {

    currencyExchangeService.exchange(from:"EUR", to:"USD");
    assertThat(currencyExchangeService.isCacheValid()).isTrue();

    Thread.sleep(millis:6000);
    assertThat(currencyExchangeService.isCacheValid()).isFalse();

}

@Test
void testResultBeforeTTLExpire() throws Exception {

    currencyExchangeService.exchange(from:"EUR", to:"USD");
    assertThat(currencyExchangeService.isCacheValid()).isTrue();

    Thread.sleep(millis:4000);

    assertThat(currencyExchangeService.isCacheValid()).isTrue();

}

```

Integration testing was done with the use of TestRestTemplate to check that all the layers in the backend we're functioning and communicating properly. Some examples of code may be seen below.

```

@Test
void whenPostTicket_thenReturn200() {

    ResponseEntity<Ticket> response =
restTemplate.postForEntity("/tickets/buy", ticket, Ticket.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(response.getBody().getEmail()).isEqualTo("josecalcas@gmail.com")
;

}

```

```

@Test
void whenHaveBuses_thenGetBuses() {
    Bus bus = new Bus();
    bus.setName("bus bue fixe");
    bus.setTotalSeats(50);
    Bus bus2 = new Bus();
    bus2.setName("bus bue fixe 2");
}

```

```
bus2.setTotalSeats(39);
busRepository.saveAndFlush(bus);
busRepository.saveAndFlush(bus2);
ResponseEntity<Bus[]> response = restTemplate.getForEntity("/bus/list",
Bus[].class);
assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
assertThat(response.getBody()).hasSize(2);
assertThat(response.getBody()[0].getName()).isEqualTo("bus bue fixe");
assertThat(response.getBody()[0].getTotalSeats()).isEqualTo(50);
}
```

3.3 Functional testing

[which user-facing test cases did you considered? How were they implemented?]
[may add some screenshots/code snippets]

Functional testing was done utilizing Cucumber + Selenium Web Driver.

The Gherkin Reference for the tests may be seen below:

Feature: Using Bus Ticket Service

Scenario: User can buy a bus Ticket

Given the user entered in the website
When the user searches for trips
And selects the first trip
And selects the seat 26
And the seat number 26 is not taken yet
And the user inputs his information
And the user clicks on the buy button
Then the user should receive a confirmation message

Scenario: User cannot buy a bus Ticket

Given the user entered in the website
When the user searches for trips
And selects the first trip
And selects the seat 26
And the seat number 26 is already taken
Then the buy button should say "Seat Taken"

These natural language tests were then converted to real tests with Cucumber and the use of Selenium Web Driver to manipulate the browser.

To aid with the implementation and readability of these tests, the Page Object Model was used, providing representation of the website pages in java classes.

```
@When("the user searches for trips")
public void the_user_searches_for_trips() {
    homePage.clickSeeTripsButton() ;
}

@And("selects the first trip")
public void selects_the_first_trip() throws InterruptedException {
    driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(2));
    listTripsPage.selectFirstTrip();
}

@Then("the user should receive a confirmation message")
public void the_user_should_receive_a_confirmation_message() {
    int size = buyTicketPage.getConfirmationMessage();
    assert(size > 0);
}
```

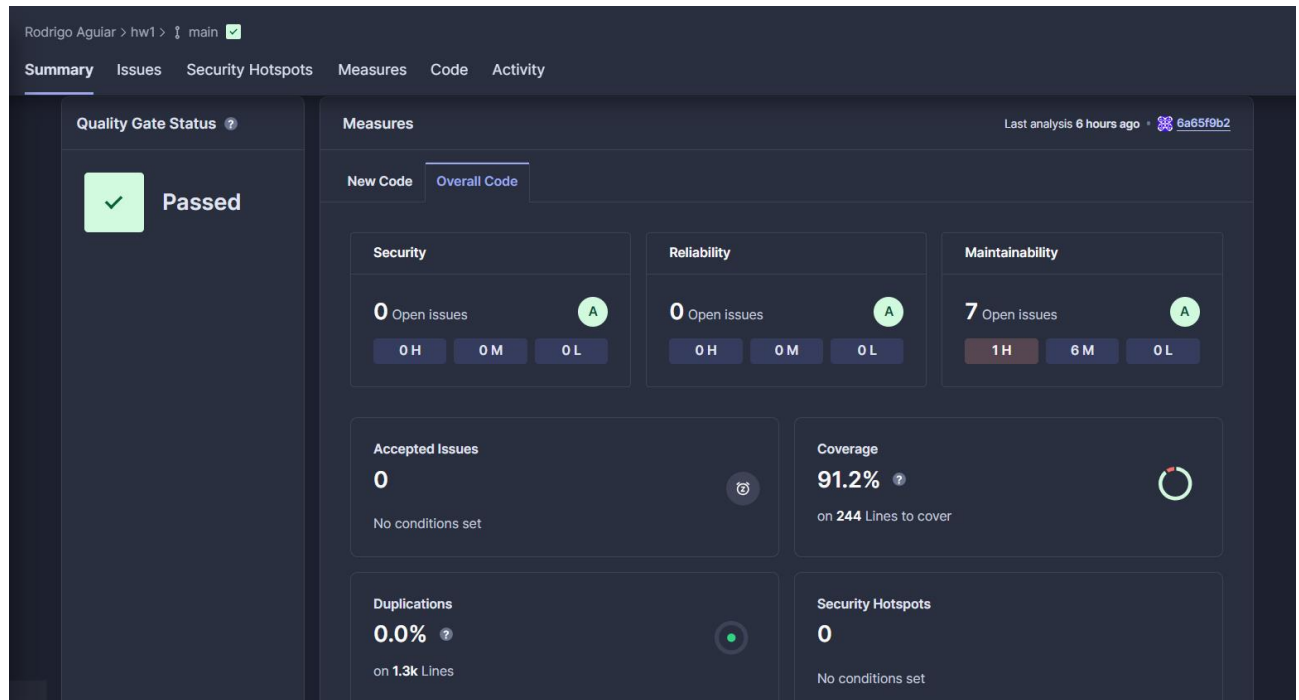
In total, 58 normal tests were developed and 12 integration tests, concluding with a total of 70 tests developed for the application.

3.4 Code quality analysis

[which tools/workflow did you use to for static code analysis? Show and interpret the results.]
[you may add some interesting lessons learned, e.g., some code smell reported by the tool that was difficult to spot and otherwise you wouldn't address it]

Code quality analysis was done through SonarCloud with Jacoco for the coverage reports.

The results may be viewed in the below image.



There are only 7 issues, from where 5 are asking to “Define and throw a dedicated exception instead of using a generic one.”, which we’re not deemed as important.

The other 2 errors are one from using `Thread.sleep()` in a test (which did not seem to give problems at all) and the other because the `CucumberTest` class does not contain any tests (which is not supposed to).

Given that, all the other issues we’re solved, being most of them wrong naming conventions for variables or packages.

There was a security hotspot in dealing with the external exchange rate API, in where the user provided data (in this case the currency “from”) was used to construct the API request URL, which was dealt with by defining a allow list of currencies and checking if the given currency is in that allow list.

The overall code coverage is 91%, being pretty good given the project contains 945 lines of code.

hw1





View as


List

14 files

Coverage 91.2% [See history](#)

New code: since previous version

	Coverage	Uncovered Lines	Uncovered Conditions
 src/main/java/tqs/deti/services/TripService.java	66.7%	9	5
 src/main/java/tqs/deti/services/CurrencyExchangeService.java	88.6%	5	3
 src/main/java/tqs/deti/services/TicketService.java	95.7%	0	1
 src/main/java/tqs/deti/controllers/TicketController.java	96.5%	1	1


 There are 10 hidden components with a score of 100%.

Show them

Out of the 14 classes, 10 have 100% coverage.

TripService is the one with the lowest coverage since I had to remove a test from there that was giving errors that I wasn't sure what was causing them.

In the other 3 classes, the uncovered lines are mostly conditions that weren't tested in both cases.

In general, SonarCloud static code analysis was pretty useful in order to improve the quality of the code of the application and also to learn about some security vulnerabilities that I wasn't aware of before.

3.5 Continuous integration pipeline [optional]

[did you implement a CI pipeline? What was the setup? Illustrate with screenshots, if applicable]

A CI pipeline was implemented for both automatic code testing and static analysis using Github Actions.

It consists of two actions with the following jobs:

- Code testing -> runs mvn test and mvn failsafe integration-test.
- Static code analysis -> static analysis using SonarCloud + Jacoco.

The code for the actions is as follows:

```
name: Maven Test

on:
  push:
    branches:
      - main

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Set up JDK 17
        uses: actions/setup-java@v2
        with:
          distribution: 'adopt'
          java-version: '17'

      - name: Build and run unit tests with Maven
        run: cd hw1/backend && mvn test
        continue-on-error: false

      - name: Run integration tests with Maven
        run: cd hw1/backend && mvn failsafe:integration-test
        continue-on-error: false
```

```
name: SonarCloud

on:
  push:
    branches:
      - main
  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
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
```

```

    java-version: 17
    distribution: "zulu"
  - 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 }
      SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
    run: cd hw1/backend && mvn -B verify jacoco:report
org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -
Dsonar.projectKey=FiNeX96_TQS_108969 -
Dsonar.coverage.jacoco.xmlReportPaths=target/site/jacoco/jacoco.xml

```

These 2 GitHub actions significantly facilitated the testing of the code, not needing to run the test commands every time changes were made on the code.

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/FiNeX96/TQS_108969
Video demo	< short video demonstration of your solution; consider including in the Git repository>
QA dashboard (online)	https://sonarcloud.io/summary/new_code?id=FiNeX96_TQS_108969
CI/CD pipeline	Only defined in Github Actions, that are present in the repository under <code>.github/workflows/</code>
Deployment ready to use	The solution runs locally through Docker, I don't have access to a server :/

Reference materials

<https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api>

<https://www.baeldung.com/spring-boot>

<https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven>

<https://docs.sonarsource.com/sonarcloud/enriching/test-coverage/java-test-coverage/>

<https://docs.sonarsource.com/sonarcloud/advanced-setup/ci-based-analysis/sonarscanner-for-maven/>