HW1: Mid-term assignment report

Rodrigo Silva Aguiar [108969], v2024-04-09

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.
<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.
Users can also check ticket reservations made given the ticket ID.

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

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.

Functional testing may fail sometimes because of the load time of the pages. Waits were introduced to try to remedy this problem, but it may happen sometimes.

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 application is designed to accommodate the needs of a user which intends to search for bus trip's between two cities, see a list of the available trips, choose one, buy a ticket (selecting the desired seat), and then if wanted check the ticket details.

The whole application flow and concept was inspired by Rede Expressos, which I personally use.

The application doesn't support administrative features currently, so the only actor is a user who wants to buy a ticket for a bus trip.

For this, the user follows these steps:

1. Enter the website and click the button "Search for Trips."
2. Select the wanted origin and destination cities, date, and preferred currency.
3. Select the wanted trip.
4. Select the wanted seat in which to travel.
5. Input user details (name, email, phone) and select preferred currency for payment.
6. Click on the button "Buy Ticket."
7. Save the ticket ID provided to check later the reservation details.
8. Go back to the Homepage.
9. Click the "Check Ticket Reservations" button.
10. Input the given ticket id into the search box.
11. Check ticket reservation details on the table.

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 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.
It's 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.

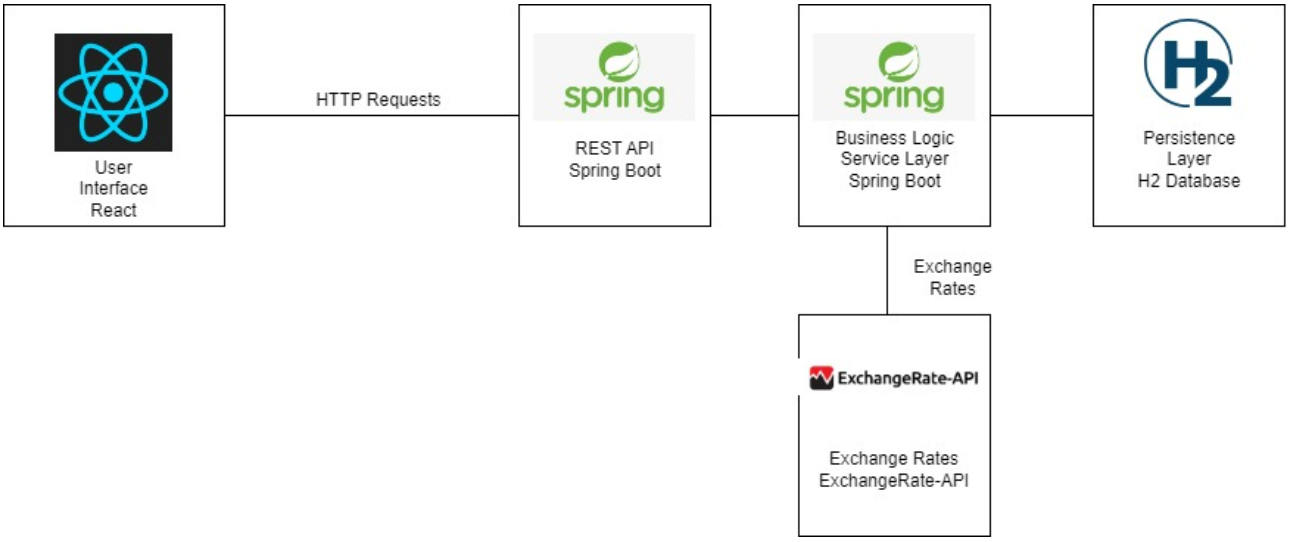
This layer is supported once again by Spring Data JPA which allows direct serialization from POJO (plain old java objects) to SQL tables and entries.

- 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 an in-memory database.

The System Architecture may be viewed in the following image:



Both the backend and frontend components were containerized using Docker and can be launched simultaneously by doing the “docker compose up” command in the hw1 folder. The frontend may be accessible at http://localhost:5173 and the backend at http://localhost:8080 .

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:

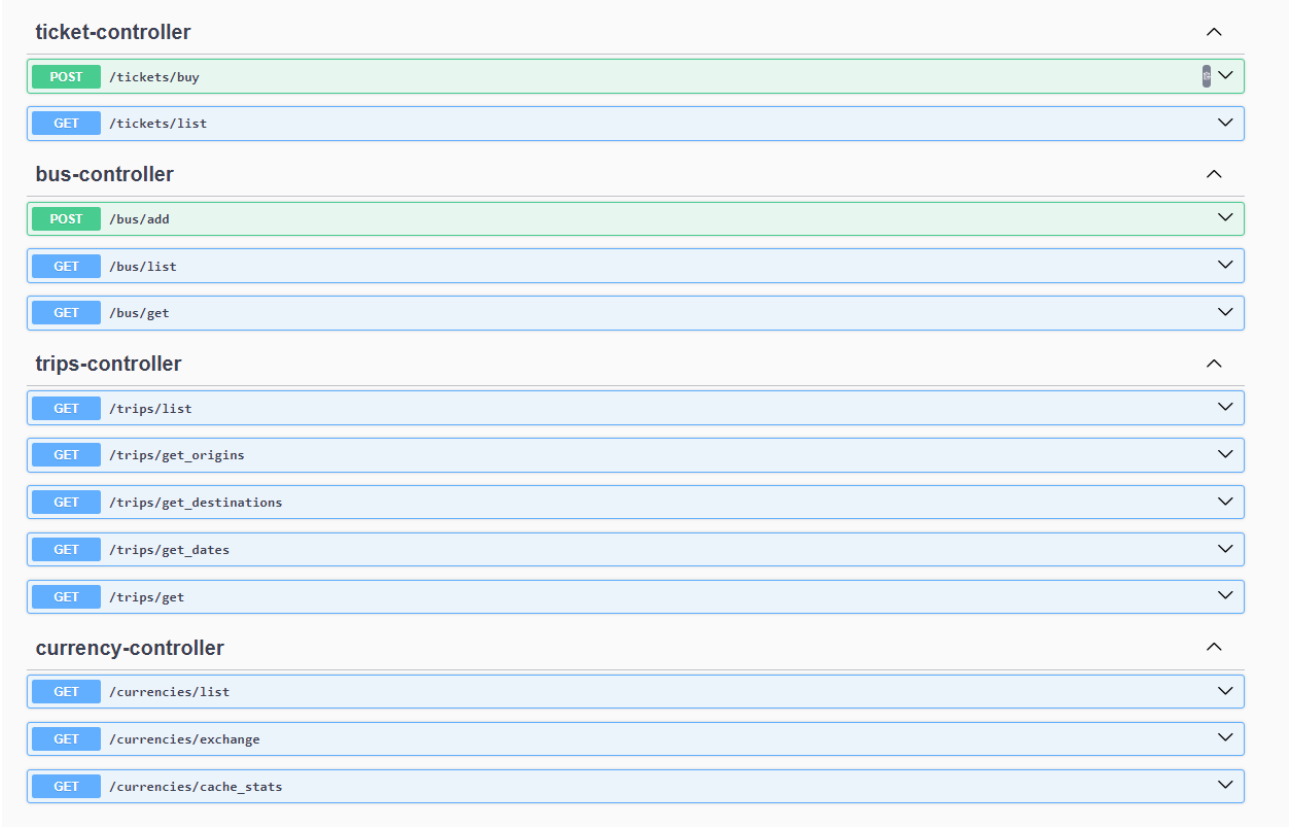
- Problem domain: get the environmental data data by region/city, etc.
- Cache usage statistics: how many hits/misses,… >.

The backend supports automatic documentation generated by SwaggerUI which may be accessed at the endpoint /docs.
Through this documentation, external developers may study the API, analyzing the present endpoints, required parameters and the response formats.

Support for various functions is present, such as:

- Buying or listing tickets
- Adding, listing or getting buses
- Listing or getting trips, or all of the trip’s origins, destinations and dates

- Listing all the present currencies supported by the system, doing a currency exchange and getting the statistics of the currency exchange rate cache.



The documentation for the used schemas (models) is also present:

Schemas

Ticket {

id	string
price	string
tripID	integer($int32)
seatNumber	integer($int32)
name	string
phone	integer($int32)
email	string

}

Bus {

id	integer($int32)
name	string
totalSeats	integer($int32)

}

Seat {

number	integer($int32)
seatType	string
taken	boolean

}

3 Quality assurance

3.1 Overall strategy for testing

Given the application requirements and vision, firstly the project structure was setup, services, repositories, controllers which were after unit-tested.

For the integration of this components together, integration tests were developed alongside the implementation to ensure that they had the required and expected functionality.
After a component passed all the tests, the implementation of the next one was started.

For the frontend component, it was first fully developed (personally a learning experience, my first time using React) and then it was tested utilizing Cucumber + Selenium functional testing.

Given this, development of tests was done alongside the development of code, to ensure that the expected functionality was met.

A total of 72 tests were developed, ensuring a great coverage of the developed code and the testing of as many use cases and edge cases as possible.

3.2 Unit and integration testing

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.

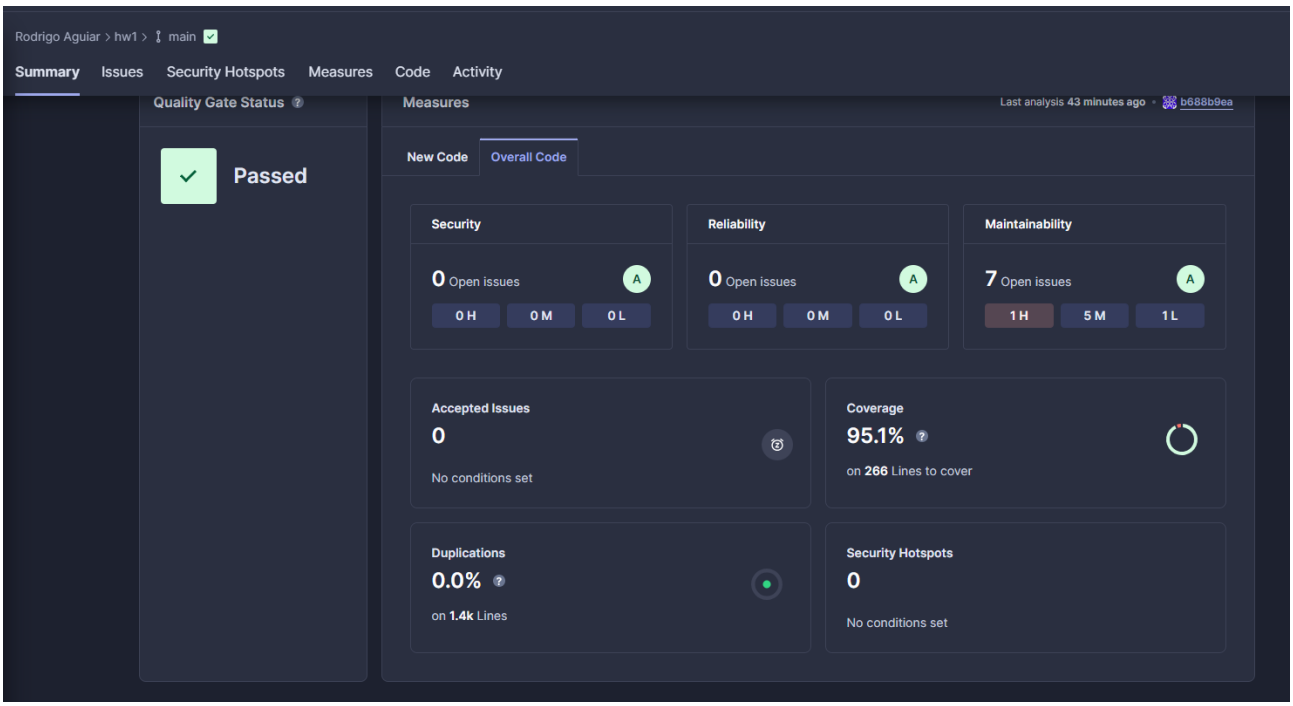
```
    @When("the user searches for trips")
    public void the_user_searches_for_trips()  {
        homePage.clickSeeTripsButton() ;
    }

    @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);
    }
```

Another test for the checking ticket reservations functionality was conceived but removed later because of too many errors resulting from page load times and selenium problems.

3.4 Code quality analysis

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

The results may be viewed in the below image.



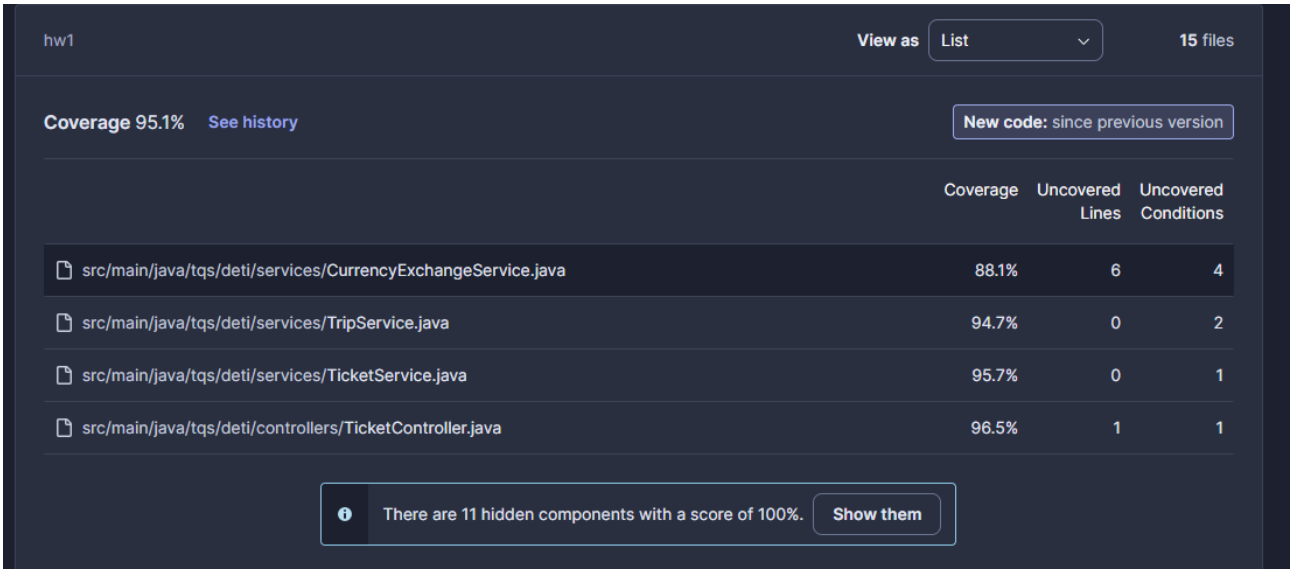
There are only 7 issues, from where 5 are asking to “Define and throw a dedicated exception instead of using a generic one.”, which we’re not deemed as important.

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 were 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 95%, resulting from a total of 75 tests, being pretty good given the project contains 716 lines of JAVA code.



Out of the 15 classes, 11 have 100% coverage.

The coverage that is missing on the other ones is mostly from uncovered conditions.
As a good practice, I usually implement error handling in the various layers of the application, making it so that I would have to target specific conditions at a specific layer in order to cover the code for them.
For example, checking if some instance of an object is null in the controller layer and also in the service layer will make the code in the service layer never be covered unless it is targeted specifically.

In general, SonarCloud static code analysis was useful 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:

```yaml
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
```

```yaml
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	Mp4 file in the repository under hw1/ directory
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 .github/workflows/
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/