HW1: Mid-term assignment report

Rúben Garrido [107927], v2024-04-09

1 Introduction **2**
1.1 Overview of the work 2
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 3

3 Quality assurance **5**
3.1 Overall strategy for testing 5
3.2 Unit and integration testing 5
3.3 Functional testing 5
3.4 Code quality analysis 6
3.5 Continuous integration 7

4 References & resources **9**
4.1 Project resources 9

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. It aims to demonstrate the application of various TQS strategies and methodologies in a real-world scenario, providing insights into the challenges and solutions encountered during the software development lifecycle.

The project involves the development of a comprehensive software solution for managing bus trips, including features for user authentication, trip listing, reservation management, and more. The software is designed with a micro-services architecture, utilizing Docker for containerization, and a REST API for backend interactions. The frontend is built with Next.js, providing a modern and responsive user interface.

GitHub Actions is used together with SonarQube, for providing Continuous Integration and static code analysis.

1.2 Current limitations

Currently, there is no admin console (admins need to use the API directly), and not all unit and functional tests are made, due to the lack of time.

2 Product specification

2.1 Functional scope and supported interactions

This application is meant for people that want to buy bus trips. The frontend is adapted to only users and not admins, however. Therefore, the only actors are the Client (aka the user), and the System admin.

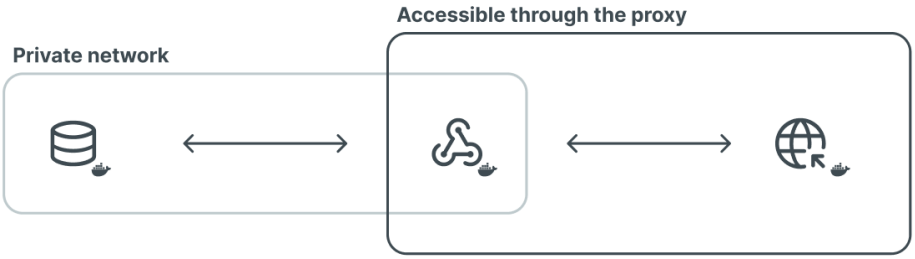
A user can authenticate in the system, list all trips, buy a new one, and check the current reservations.

2.2 System architecture

The system architecture deeply follows the one found in the IES course: a micro-services architecture with Docker.

There are four Docker containers:

- Database: a PostgreSQL DB that is connected to a backend network and no exposed ports, which means only the API can access it, internally.
- Backend: a Spring Boot app that interacts with the DB and creates a REST API.
- Frontend: a Next.js app, that provides the graphical interface for this app
- Proxy: a Nginx reverse proxy that connects both backend and frontend (running in different ports) into a single HTTP (80) port, that is the only exposed in the whole system. It connects to both frontend and backend networks.



2.3 API for developers

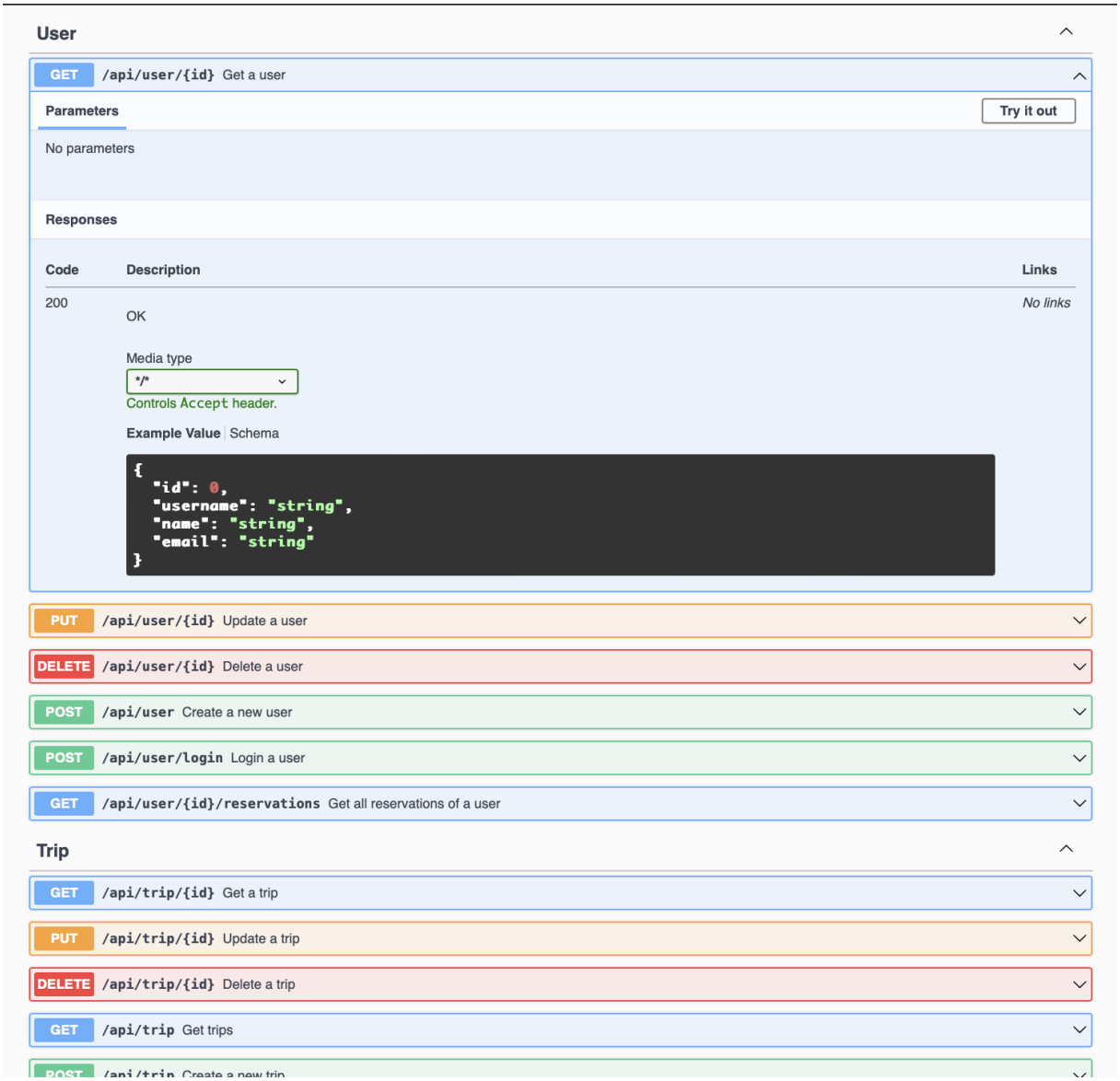
The API supports five entities: Bus, City, Trip, Reservation, and User. All of them have endpoints for the basic CRUD operations.

Regarding relationships, a Trip contains one Bus and two Cities, while a Reservation has one Trip and one User. Therefore, there are two additional endpoints:

- Get reservations associated to an user
- Get reservations associated to a trip

There is also an endpoint that shows statistics about the currency API service and its cache.

The API is fully documented, with the OpenAPI standard and Swagger as the user interface. A little demo is presented in the following print screen:


User
GET /api/user/{id} Get a user
Parameters
Try it out
No parameters
Responses
Code
Description
Links
200
OK
No links
Media type
/
Controls Accept header.
Example Value | Schema
{
"id": 0,
"username": "string",
"name": "string",
"email": "string"
}
PUT /api/user/{id} Update a user
DELETE /api/user/{id} Delete a user
POST /api/user Create a new user
POST /api/user/login Login a user
GET /api/user/{id}/reservations Get all reservations of a user
Trip
GET /api/trip/{id} Get a trip
PUT /api/trip/{id} Update a trip
DELETE /api/trip/{id} Delete a trip
GET /api/trip Get trips

3 Quality assurance

3.1 Overall strategy for testing

Due to low time available, Test-Driven Development (TDD) was not used. I decided to do all implementation first, so that I could get a working API as soon as possible.

I used Cucumber (and therefore BDD) in functional tests, since those resemble the user expected behavior. However, unit and integration tests don't have many steps each, so a BDD testing might be an overkill feature for them.

All controller tests (both unit and integration) use Rest Assured, for a consistent testing regarding the REST API. Also, Testcontainers were used in integration tests, so that the test environment is the same as the main one.

3.2 Unit and integration testing

Integration tests were made for as many scenarios as possible regarding the API controller. This ensures that, by testing the API on a full Spring environment, all layers underneath - like the repository, service and etc. - are correctly assessed.

However, unit tests were also made, for covering the most common usages of each layer. For one to have an idea, code coverage regarding unit tests is around 75%, while integration tests raise it to a massive 97%. If I was given more time, I would have made the unit tests cover the same as the integration ones. Check section 3.4 for more details.

3.3 Functional testing

For the functional tests, the main scenario was tested:

1. Sign up
2. Select cities
3. Select a trip
4. Choose seats
5. Buy

As of the time of writing this report, there are no other functional tests, because there was no time.

The Selenium WebDriver for Java is used for this test. A Firefox driver is used instead of a Chromium one, since Firefox comes bundled in most Linux distributions, which makes it easy to port the tests to any device. Also, Firefox is being run in a headless mode, so that non-GUI systems (like GitHub Actions) can run the test without breaking due to no display manager.

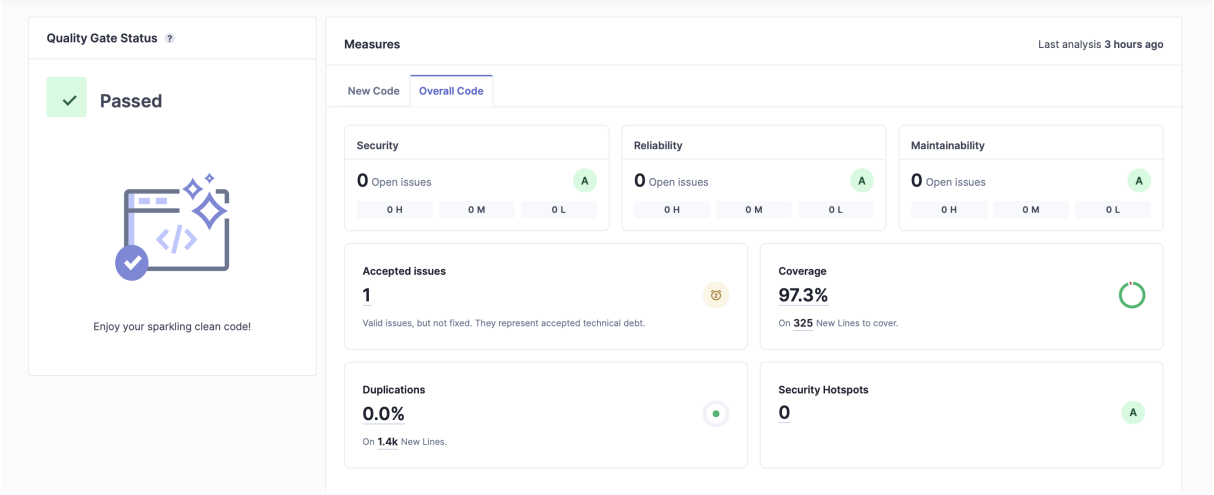
```
FirefoxOptions options = new FirefoxOptions();
options.addArguments("-headless");
WebDriver driver = new FirefoxDriver(options);
```

BDD is a good approach for tests like this one, since it tries to mimic natural language an end user uses. Because of this, Cucumber is used as the steps tool. The code is as follows:

```
Feature: Book a trip
  Scenario: Sign-up
    Given I navigate to "http://localhost"
    When I click on the signup button
    And I fill in username with "RGarrido"
    And I fill in name with "Rúben Garrido"
    And I fill in email with "rubengarrido@ua.pt"
    And I fill in password with "Ruben#78236"
    And I click on the submit button
    And I fill in departure with 1
    And I fill in arrival with 2
    And I search for trips
    And I choose trip 6
    And I book the trip
    Then I should see a success message
```

3.4 Code quality analysis

Sonarqube was used for the code analysis. A Sonarqube instance was deployed on my Raspberry Pi, for 24/7 availability while being able to use GitHub Actions in a private GitHub repository. Check section 3.5 for more details regarding the Actions workflow.



Overall code coverage is 97.3%. There is one accepted issue, regarding using `Thread.sleep()` in the functional test. This had to be marked as accepted, since there is no way for Selenium to wait for API fetching.

All other issues were solved for a better code, even though there weren't many. However, those were some tiny things that go beyond the IntelliJ real-time analysis and therefore I wouldn't be able to address them.

No security vulnerabilities were made over time. Also, technical debt was not used, due to low issues.

3.5 Continuous integration

A CI pipeline was implemented using GitHub Actions. The Sonar analysis requires tests to be run, so running tests in a separate workflow would be redundant. Therefore, a single workflow was created, reducing both the complexity and energy usage in the GitHub Actions runners, which helps the environment.

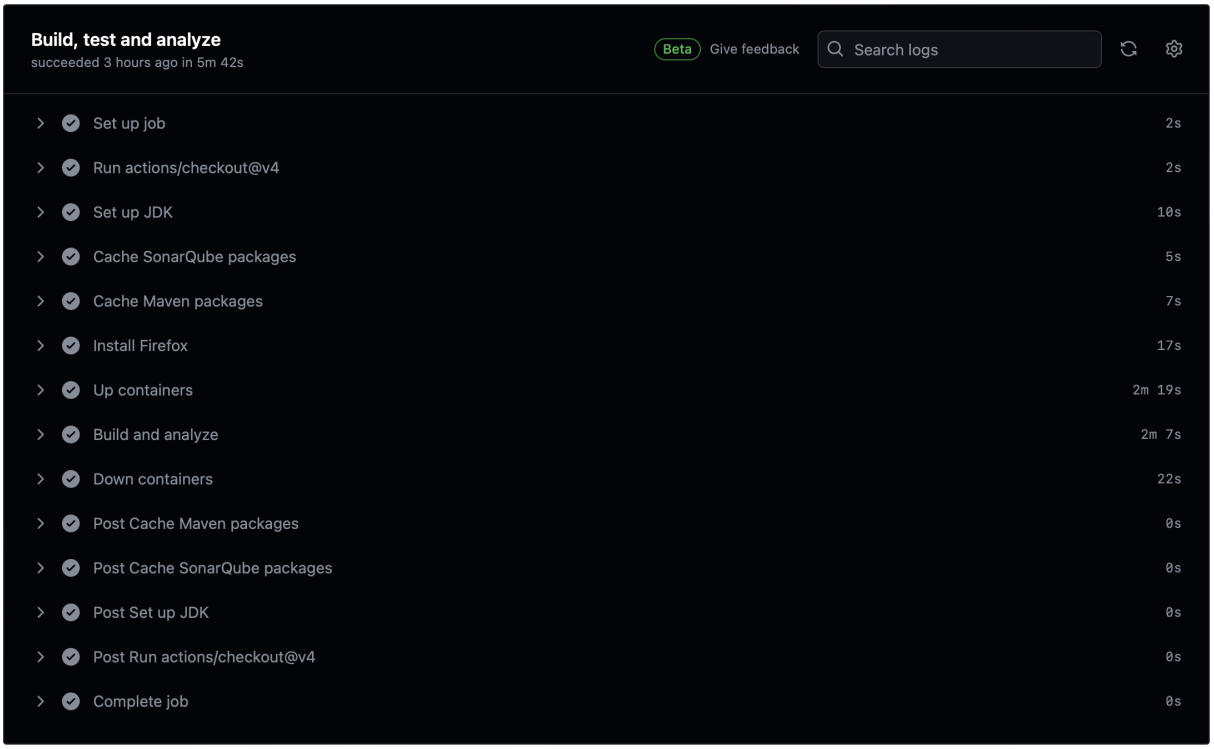
A simple SonarQube workflow is characterized by installing the JDK and running `mvn verify sonar:sonar`. However, this was not enough. Here's why:

- SonarQube and Maven packages were not cached, which makes CI slower and more energy consuming.
- Integration tests do not account for code coverage.
- Functional tests require having a full project running in the background, so that the website can be successfully opened and tested against.
- On Ubuntu, functional tests require the deb version of Firefox to be installed, instead of the default Snapcraft-based one.

So, the workflow has the following steps:

1. Install JDK
2. Get the previous job's Maven and SonarQube packages
3. Install the deb version of Firefox
4. Run `docker compose up` (production version)
5. Run tests and SonarQube analysis

This workflow runs every time there is a change in the backend folder of this project. This ensures that commits regarding the course labs do not trigger the CI pipeline, since there are no code changes in the project. The average running time is about 6 minutes, most of which spent in the `docker compose up` process.



4 References & resources

4.1 Project resources

Recurso	**URL/Localização**
Repositório Git	`https://github.com/RGarrido03/TQS_107927`
Video Demo	In the Git repository.
QA Dashboard (online)	`https://sonarqube.garridoegarridolda.pt` Login: ruben; password: Ruben#2024
CI/CD Pipeline	`https://github.com/RGarrido03/TQS_107927/blob/main/.github/workflows/tests_and_sonar.yaml`

Tabela 1: Resources