

Homework 1

Environment Monitor

7th of March 2023

Author

103154 João Fonseca

Software Testing and Quality Control 2022/23

Licenciatura em Engenharia Informática

Universidade de Aveiro

0 Index

- 0 Index.....1**
- 1 Introduction.....2**
 - 1.1 Overview of the work2
 - 1.2 Current limitations2
- 2 Product specification.....3**
 - 2.1 Functional scope and supported interactions3
 - 2.2 System architecture4
 - 2.3 API for developers4
- 3 Quality assurance.....5**
 - 3.1 Overall strategy for testing.....5
 - 3.2 Unit and integration testing.....5
 - 3.3 Functional testing.....7
 - 3.4 Code quality analysis8
 - 3.5 Continuous integration and continuous deployment pipeline.....9
- 4 References & resources.....10**

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

This Spring Boot based application provides information about the quality of the air for a specified location. It displays the current and forecast air quality index and metrics on a table, and provides a graph to visualize the fluctuation of the forecast metrics. It also provides statistics over the use of the internal cache, used to store recent made requests to the external APIs provided by OpenWeather and OpenMeteo.

The main purpose of this application is to demonstrate the application of many software testing strategies such as unit tests and integration tests in JUnit 5, functional tests with Selenium and static code analysis on Sonar Cloud. A GitHub Actions workflow was put in place to automate not only these processes, but to also deploy the solution in a Microsoft Azure Web App, implementing a full CI/CD pipeline.

1.2 Current limitations

The validation of the external API results is not properly done, neither the structure nor the data types. It is dependent on an exception being thrown by the Open Feign dependency when the returned data does not map to the defined data transfer object. In the case of it happening, another external API is always fetched.

2 Product specification

2.1 Functional scope and supported interactions

There are three main actors in the application: users, system administrators and developers.

User: interested in checking the air quality for a given location.

Upon landing on the home page, the user is presented with a simple but informative webpage, prompting the insertion of a location. The user inserts a location and submits the query. The webpage refreshes itself with the current and forecast air quality indexes and metrics.

System Administrator: interested in checking the internal cache performance.

Upon noticing a spike in the performance of the application, the system administrator checks the cache usage statistics to determine the best course of action to take. Based on these metrics, the system administrator can temper with the cache configuration (time to live and capacity) to improve the overall performance of the system, maintaining an equilibrium between the number of requests to external APIs and the timeliness of the provided information.

Developer: interested in integrating the provided Rest API in his own application.

The developer is currently developing a weather forecast application and needs some external APIs as a source for information. To fetch air quality information, the developer chose to use this API, that provides the current and forecast air quality indexes and metrics. By doing so, he is able to provide this kind of information to the users of his application.

2.2 System architecture

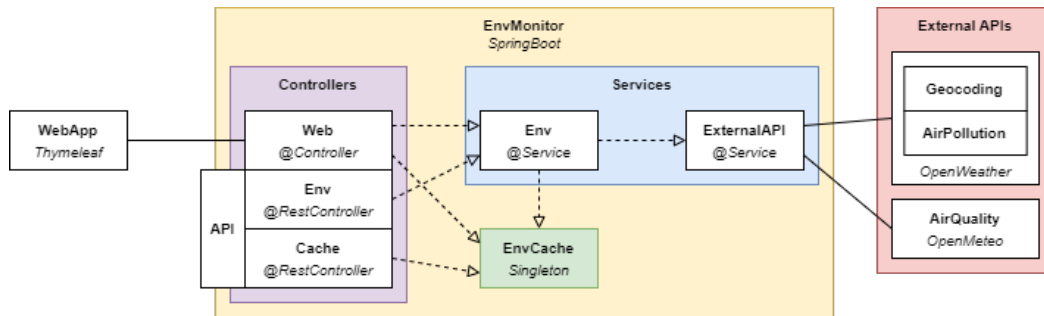


Fig. 1: Architecture diagram.

The entire architecture is based around the Spring Boot framework.

- Postman: used to test the endpoints of the external APIs.
- Open Feign: used to fetch data from the external APIs by the means of an annotated interface, which automatically maps the returned data to a specific Java data transfer object.
- Lombok: used to create constructors, getters and setters for data transfer objects by the means of annotations.
- Thymeleaf: used to create the webapp, allows the creation of a server-side rendered template that consumes the data provided by the web controller, making the front-end development easier and integrated in the entire Spring Boot ecosystem.
- EnvCache: extends a custom made abstract cache class. Follows a singleton pattern, which makes its instance unique in the entire execution environment.

2.3 API for developers

Host URL

Local: **localhost:8080**
Deployed: **env-monitor.azurewebsites.net**

Environment

Base URL: **<Host URL>/api/env**
Endpoints: **GET /current?q=<location>**
Get current air quality for the given location.
GET /forecast?q=<location>
Get forecast air quality for the given location.

Cache

Base URL: **<Host URL>/api/cache**
Endpoints: **GET /stats**

Get use statistics from the cache.

3 Quality assurance

3.1 Overall strategy for testing

Given the sheer simplicity of the application, the code skeleton and most of its implementation was developed alongside the creation of tests. A TDD (test-driven development) strategy was therefore not strictly followed. However, the implemented tests cover around 93.5% of the code (according to the Sonar Cloud analysis), giving us enough assurance on the correctness of the implementation.

The tests were developed using JUnit 5 with the extended support given by different testing tools such as Hamcrest, AssertJ, Mockito, REST-Assured and Selenium WebDriver. Adding to these tests, a logger based on the SLF4J interface was setup up to allow for an easier debugging of eventual errors along the development.

3.2 Unit and integration testing

Unit tests

<u>Where:</u>	Cache, data transfer objects (DTOs) and utility methods (Utils).
<u>Purpose:</u>	Testing the individual behaviour of component, isolated from the others.
<u>Strategy:</u>	Making sure all possible behaviours were assessed.
<u>Library:</u>	AssertJ.
<u>Test files:</u>	CacheTest CacheStatsDTOTest EnvComponentsDTOTest EnvDTOTest EnvItemDTOTest OpenMeteoAirQualityDTOTest OpenMeteoAirQualityHourlyDTOTest OpenWeatherAirPollutionComponentsDTOTest OpenWeatherAirPollutionDTOTest OpenWeatherAirPollutionItemDTOTest OpenWeatherGeocodingDTOTest ConfigUtilsTest ConverterUtilsTest EnvUtilsTest

Service tests

<u>Where:</u>	Environment and external API services.
<u>Purpose:</u>	Testing the behaviour of services with mocked versions of adjacent components.
<u>Strategy:</u>	Making sure all possible execution paths were exercised, including correct uses but also the insertion of invalid locations or unavailability of one external API.
<u>Libraries:</u>	AssertJ, Mockito.
<u>Test files:</u>	EnvServiceTest ExternalAPIServiceTest

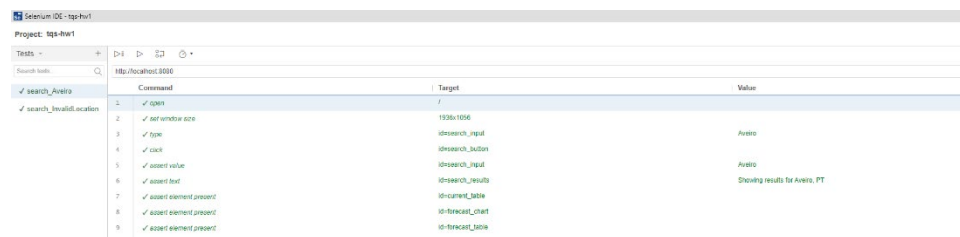
Integration tests

<u>Where:</u>	Environment and cache statistics REST controllers, and Thymeleaf web controller.
<u>Purpose:</u>	Testing the behaviour of the application endpoints according to certain scenarios.
<u>Strategy:</u>	Making sure all possible responses from the endpoints were exercised. Two variations of similar tests were developed: one using mocked services, the other loading the full application context. Using mocked services, it was possible to validate the controllers on their own, followed by a complete validation of the application from the endpoints to the invocation of the external APIs.
<u>Libraries:</u>	Hamcrest, Mockito, RestAssured (including RestAssuredMockMvc).
<u>Test files:</u>	CacheRestControllerTest CacheRestControllerIT EnvRestControllerTest EnvRestControllerIT WebControllerTest WebControllerIT

3.3 Functional testing

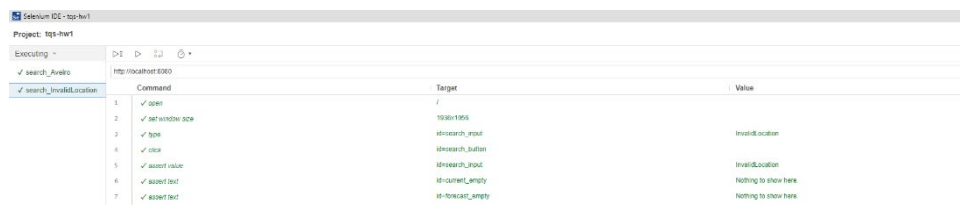
For the webpage, two test cases were considered: searching for a valid location and an invalid location. In the case of a valid location, the current and forecast air quality data should be displayed on the page; otherwise, no air quality data is displayed, only a message indicating no available information.

These functional tests were recorded using Selenium IDE, allowing for an easier creation and editing of the tests steps. After making sure the tests had the minimum number of needed steps, they were exported to Java source code using the built-in feature of Selenium IDE for that purpose.



Step	Command	Target	Value
1	open	/	
2	set window size	1024x1024	
3	type	id=search_input	Aveiro
4	click	id=search_button	
5	assert value	id=search_input	Aveiro
6	assert text	id=search_results	Showing results for Aveiro, PT
7	assert element present	id=current_table	
8	assert element present	id=forecast_chart	
9	assert element present	id=forecast_table	

Fig. 2: Selenium IDE search for Aveiro.



Step	Command	Target	Value
1	open	/	
2	set window size	1024x1024	
3	type	id=search_input	InvalidLocation
4	click	id=search_button	
5	assert value	id=search_input	InvalidLocation
6	assert text	id=current_empty	Nothing to show here.
7	assert text	id=forecast_empty	Nothing to show here.

Fig. 3: Selenium IDE search for invalid location.

Moving on to Java, the generated code was cleaned up and refactored according to the Page Object pattern. Two pages were created: the IndexPage (landing page of the website containing no air quality data) and the SearchPage (similar to the previous page, but containing methods to verify the presence of the current and forecast air quality data on the page). The original exported test code studently became much cleaner, allowing for a better understanding and structuring of the test cases.

3.4 Code quality analysis

For the static code analysis, the chosen tool was Sonar Cloud. The analysis was automatically run every time there was a push to the GitHub repository and the results were sent over to the associated Sonar Cloud project. This automation was possible thanks to a workflow created on the repository using GitHub Actions.

This analysis allowed for the detection of some security vulnerabilities, such as not sanitizing the received locations before logging, or using the generalized `@RequestMapping` instead of the `@GetMapping` for a GET endpoint. Most of the code smells weren't addressed, since that criterion passed the default and recommended quality gate by Sonar Cloud used in this project.

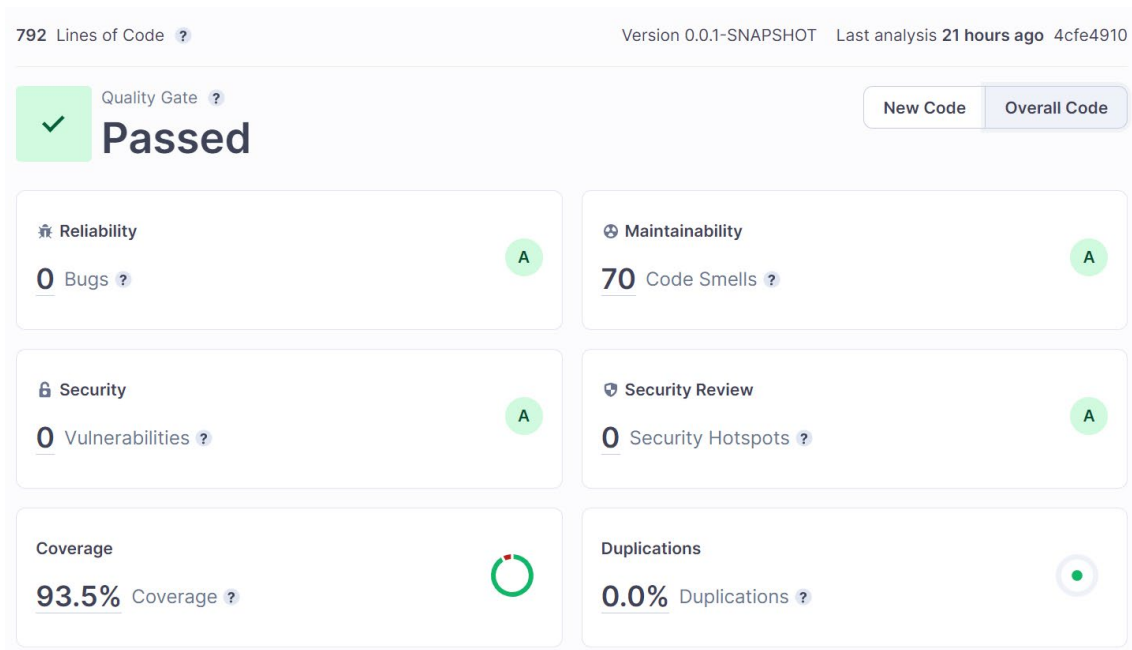


Fig. 4: Overall results after last Sonar Cloud analysis.

After fixing security issues, the finished application was able to pass the quality gate, with only 70 code smells (no critical or blocker code smells) and with a test coverage of 93.5%, greatly surpassing the required 80%.

3.5 Continuous integration and continuous deployment pipeline

For implementing a CI/CD pipeline, GitHub Actions played an important role for orchestrating this task. In a first phase, the application tests are run and the static code analysis is posted to Sonar Cloud. After that, the application JAR file is created and saved inside the repository. Last but not least, the previously created JAR file is deployed to a previously created Microsoft Azure Web App, making the application ready and available for production.

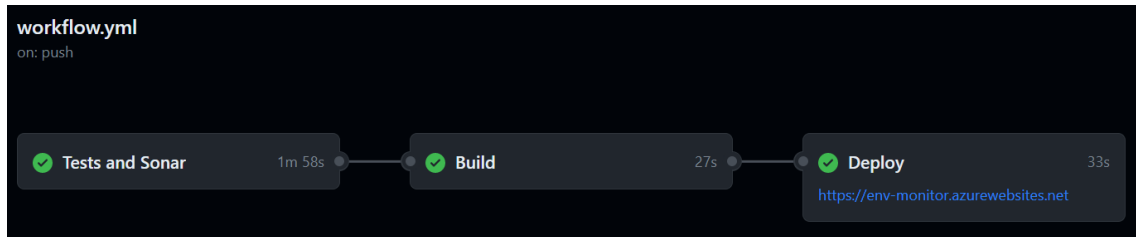


Fig. 5: CI/CD workflow overview in GitHub Actions.

4 References & resources

Project resources

Resource	URL/Location
Git repository	https://github.com/joaomfonseca/tqs-hw1
Video demo	Inside the GitHub repository.
QA dashboard (online)	https://sonarcloud.io/summary/overall?id=tqs-hw1
CI/CD pipeline	https://github.com/joaomfonseca/tqs-hw1/blob/main/.github/workflows/workflow.yml
Deployment ready to use	https://env-monitor.azurewebsites.net/

Reference materials

Maven

Java version: 11

Parent: Spring Boot Starter Parent (2.7.10)

Dependencies: Spring Web
Spring DevTools
Spring Logging
Thymeleaf
Spring Cloud Open Feign
Lombok
RestAssured
Selenium Java

Plugins: JaCoCo
Sonar

For more information, check out the Maven project's *pom.xml* file.

APIs

OpenWeather: Geocoding API
Air Pollution API

OpenMeteo: Air Quality API