HW1: Mid-term assignment report

Vasco Jorge Regal Sousa [97636], v2022-04-22

1 Introduction **1**
1.1 Overview of the work 1
1.2 Current limitations 1

2 Product specification **1**
2.1 Functional scope and supported interactions 1
2.2 System architecture 2
2.3 API for developers 2

3 Quality assurance **2**
3.1 Overall strategy for testing 2
3.2 Unit and integration testing 2
3.3 Functional testing 2
3.4 Code quality analysis 2
3.5 Continuous integration pipeline [optional] 3

4 References & resources **3**

1 Introduction

1.1 Overview of the work

This report documents the key steps that were taken to develop a simple covid metric aggregation webApp, **CovIncidence**. To achieve this, a Test Driven Development approach was used, allied with CI/CD concepts for QA and faster development cycles.

1.2 Current limitations

Since we are querying an external service, delays between responses are expected, as well as certain data being missing. In this project, we are not responsible for the source or the availability of the data.

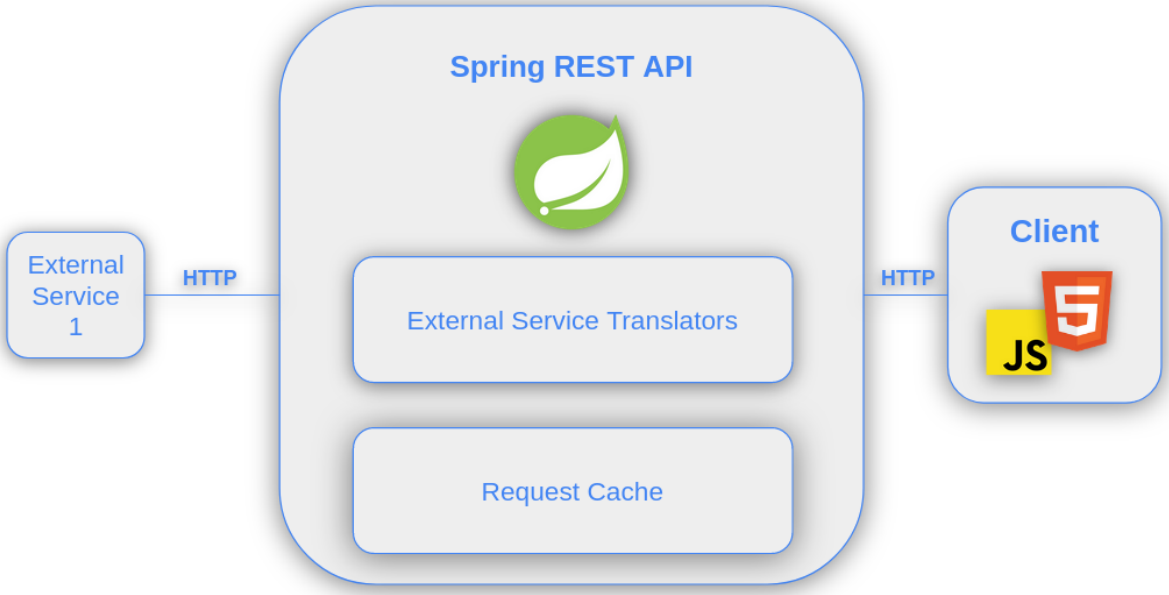
2 Product specification

2.1 Functional scope and supported interactions

CovIncidence aims to provide a simple, user-friendly app able to analyze the COVID-19 pandemic and show some interesting aggregations on linear charts. This data can easily be filtered out by country and/or time period. Besides that, there's also info related to global indicators, since the beginning of the pandemic. A simple request ttl cache is supported.
The API is also configurable to use different external APIs. In the current project, two are supported: APISports and VACCOVID, which can be setup using the **data.external.api** property of the spring project (values APISports or VACCOVID, uses APISports as default if invalid/not provided).

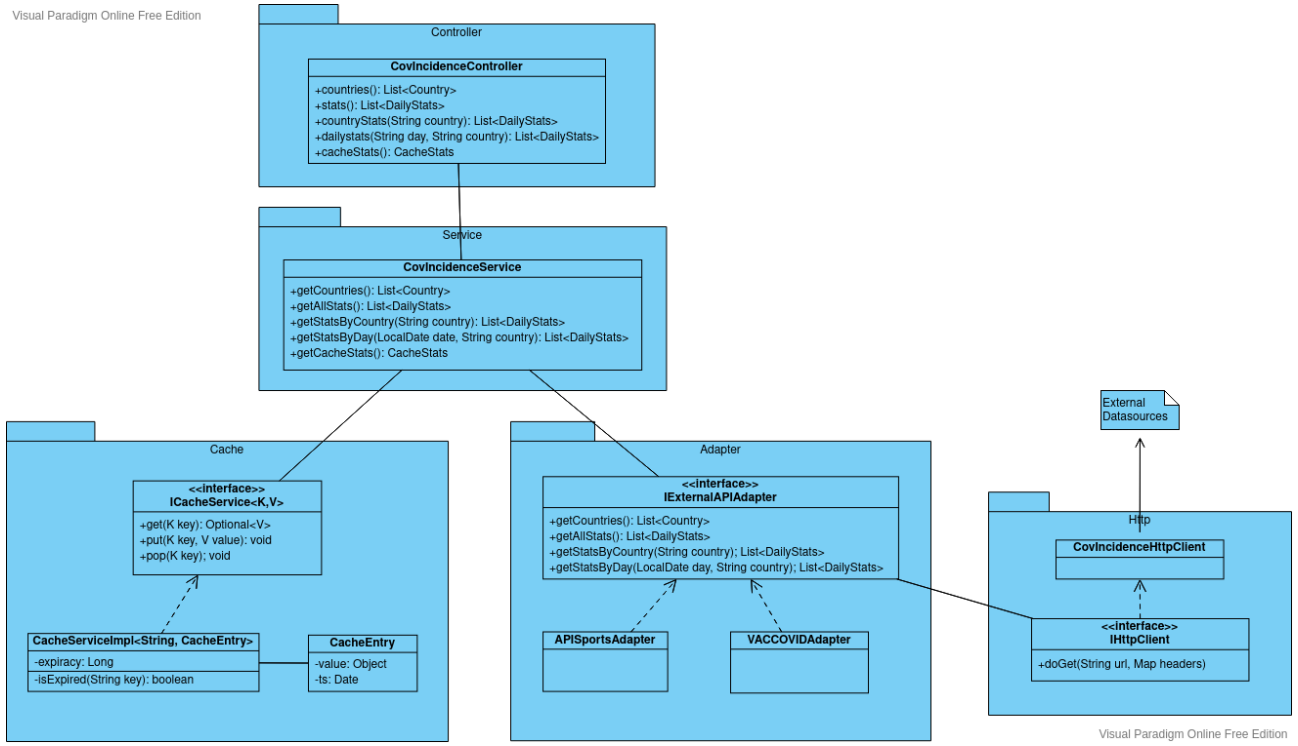
2.2 System architecture

General system architecture:



The architecture of the REST api follows a standard Controller - Service - Datasource approach. Controllers pick up the requests, call a Service function which handles logic, accesses cache and fetches the data. In this case, our data is fetched by calling a method of an adapter interface, meant to query and translate the data from the external api.

Spring Class Diagram:



2.3 API for developers

The REST API documentation was automatically generated with the maven swagger dependency. It is accessed on the **/docs** resource of the api.

3 Quality assurance

3.1 Overall strategy for testing

To test the application, two different methodologies were applied.
For the **backend** (Spring REST API), a TDD approach was used: First the “skeleton” of the architecture was designed, creating classes and method signatures (all returning null). Then, for each component, tests were written. Having these tests guiding us on how our functionalities should behave, the actual class was developed and was only considered done once every test passed. The order in which the components were tested/developed also followed the layer-oriented project structure (Controller -> Service -> APIAdapters -> Http and finally the Cache).
Due to the simplicity of the webapp, only one test was run to verify the info was being correctly displayed.

3.2 Unit and integration testing

Unit tests were used for each component, by mocking related dependencies to try to isolate the test subject as much as possible. Dependency stubbing was done with Mockito. For example, when testing the Service layer, both the API adapter (the source of our data) and the cache (which are both only interfaces) were mocked.

```
@ExtendWith(MockitoExtension.class)
public class ServiceTest {

   @Mock(lenient = true)
   private IExternalAPIAdapter externalAPIAdapter;

   @Mock(lenient = true)
   private ICacheService<String, Object> cacheService;

   @InjectMocks
   private CovIncidenceService service;

      . . .
}
```

```
      . . .

Mockito.when(externalAPIAdapter.getCountries()).thenReturn(countires);
Mockito.when(externalAPIAdapter.getAllStats()).thenReturn(all);
Mockito.when(externalAPIAdapter.getStatsByCountry("Portugal")).thenRet
urn(ptstats);
Mockito.when(externalAPIAdapter.getStatsByDay(LocalDate.parse("2021-03
-04"),                          "Portugal")).thenReturn(ptstatdaily);
Mockito.when(externalAPIAdapter.getStatsByDay(LocalDate.parse("2021-03
-04"),                             "Japan")).thenReturn(jpstatdaily);
Mockito.when(externalAPIAdapter.getStatsByDay(LocalDate.parse("2021-03
-04"), null)).thenReturn(s0304);

Mockito.when(cacheService.get("/stats")).thenReturn(Optional.empty());
Mockito.when(cacheService.get("/countries")).thenReturn(Optional.of(co
untires));
```

```
Mockito.when(cacheService.get("/countries?country=Portugal")).thenRetu
rn(Optional.empty());
Mockito.when(cacheService.get("/day?country=Portugal&date=2021-03-04")
).thenReturn(Optional.empty());
Mockito.when(cacheService.get("/day?country=Portugal&date=2021-03-04")
).thenReturn(Optional.empty());
Mockito.when(cacheService.get("/day?country=All&date=2021-03-04")).the
nReturn(Optional.empty());

      . . .
```

A similar approach was used for the rest of the project, in an attempt to cover all developed classes.

As for the integration tests, a test was created to verify the service's methods where its dependencies are implementations of previously tested components. In this case, instead of mocking the behavior of the cache and api adapter interfaces, we actually want to test the system as a whole, or at least part of it, to make sure that although unit tests may be passing, the communication between components is functional.
The setup for this IT test:

```
public class ServiceApiSportsIT {
    private IHttpClient httpClient;
    private IExternalAPIAdapter adapter;
    private CovIncidenceService service;
    private CacheServiceImpl cache;


    @BeforeEach
    public void setup() {
        this.httpClient = new CovIncidenceHttpClient();
        this.adapter = new APISportsAdapter(httpClient);
        this.cache = new CacheServiceImpl();
        this.service = new CovIncidenceService(this.adapter, cache);
    }
```

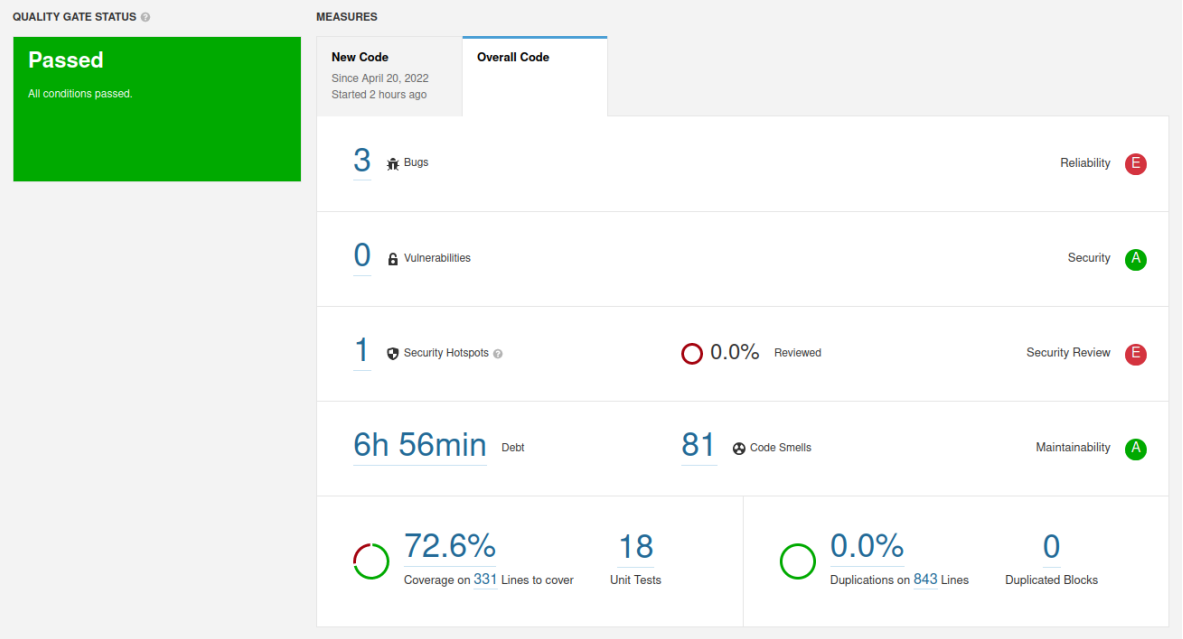
3.3 Functional testing

Frontend was tested using the PageObject pattern. Since we can't really test the filters (Chart.js charts were used to display data, which don't render html elements Selenium could

read from), we only tested the static cards showing the global stats and the cache page with the internal metrics.
Since a QA server was not setup and the frontend (external service, independent of Spring boot project) tests required a running Spring API to be run and cross check the data, the decision was made to exclude these tests from the CI pipeline (If passing these tests was a requirement to deploy the app, and the tests can only be run if the app is deployed, we put ourselves in an impossible situation without an extra testing server).

3.4 Code quality analysis

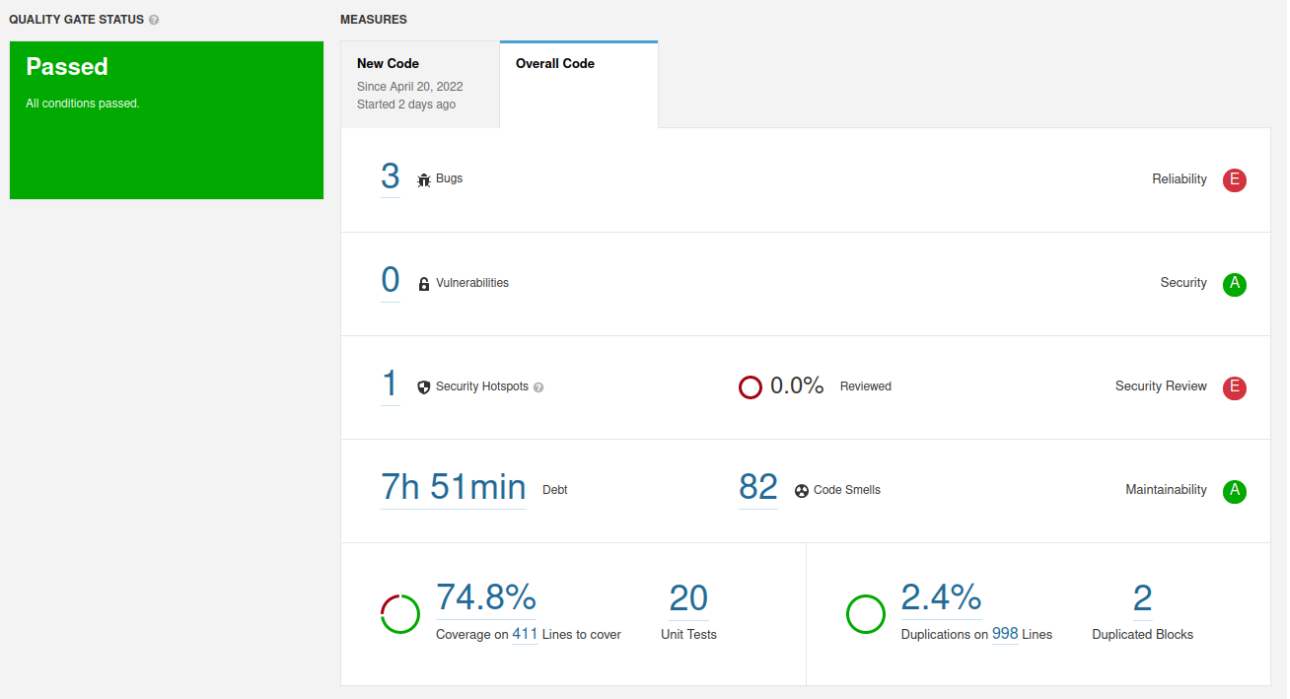
To analyze code quality, **SonarQube** was used. How and at which point of the development cycle was this tool integrated is explained in the next step (CI pipeline).
One big lesson learned is that this pipeline and static code analysis should have been implemented even before the development started. Since the pipeline was created only around the point where the Cache implementation was being tested, a lot of code smells accumulated, which could have been prevented if they were detected before committing any increment.
This was the first analysis report:



Although it passes the quality gate (the default was used), the missing coverage is due to the utils classes not having unit tests yet.

After the pipeline, the SonarQube report was checked before pushing so new code smells were quickly fixed. One of the most common code smell reported was the use of the diamond constructor with types (for example <String, String>) where the types could be omitted.

The final analysis looked like this:



3.5 Continuous integration pipeline [optional]

The CI pipeline used was created with both Git hooks and GitHub Actions.
A Git hook is a script that runs when git related events are triggered, like commits or pushes. To assure the quality of the commited code, before every commit, the maven tests were run and the SonarQube report generated. If any of these fails (tests failing or not passing the quality gate) the commit would be automatically aborted. This effectively prevented the addition of test-covered bugs and/or code smells / minor bugs that could quickly be fixed by checking the SonarQube server. **Note:** Git hooks are local only, they reside in the .git folder so they are not present in the remote repository.

```
#!/bin/bash

# location: .git/hooks/pre-commit

SRC_PATTERN="HW1/"
if git diff --cached --name-only | grep --quiet "$SRC_PATTERN"
then
```

```
      cd ./HW1/tqscovid && mvn test
      if [ $? -ne 0 ]; then
            printf "\n[ERROR] Some tests failed. Aborting commit...\n"
            exit 1
      fi
      docker start sonarqube && mvn verify sonar:sonar \
      -Dsonar.host.url=http://localhost:9000 -Dsonar.projectKey=tqs_hw1
\     -Dsonar.login=*
      if [ $? -ne 0 ]; then
            printf  "\n[ERROR]  SonarQube  Gate  not  passed.  Aborting
commit...\n"
            exit 1
      fi

fi
```

As for the GitHub Actions workflow, on push, two jobs are defined: test and deploy. Testing simply runs the maven tests (This is a bit redundant since you can't really push code with failures thanks to the hook and this is an individual project) and then the application is deployed on an Azure Virtual Machine using a docker-compose (both the api and frontend). This last job is executed on a self-hosted GitHub Runner, created on the virtual machine.

The created runner:

And the github actions workflow:

```yaml
# path: ./github/workflows/ci.yml

name: CI Pipeline

on:
 push:
   branches:
     - main
jobs:
 test:
   runs-on: ubuntu-latest
```

```
  steps:
    - uses: actions/checkout@v2
    - uses: actions/setup-java@v1
      with:
        java-version: 11
    - run: cd HW1/tqscovid && mvn test
deploy:
  runs-on: azure
  steps:
    - uses: actions/checkout@v2
    - run: cd HW1/ && sudo sh prod.sh
```

prod.sh is a script that builds and ups the compose

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/VascoRegal/TQS_97636
Video demo	https://github.com/VascoRegal/TQS_97636/raw/main/HW1/video.mkv
QA dashboard (online)	
CI/CD pipeline	https://github.com/VascoRegal/TQS_97636/blob/main/.github/workflows/ci.yml
Deployment ready to use	http://20.53.251.194/webapp/

Note: Since the Virtual Machine where the project is deployed is part of the university's free Azure plan, it is quite slow and I was unlucky enough to have a machine located in Australia (literally as physically far as possible) due to a problem when picking the vm's image (other images were locked because of the plan's policy). Non cached requests have poor performance (not only the time of reaching the machine but also the external api query). If a faster version must be used for evaluation, a docker-compose file is provided in the HW1 folder, which launches the application on localhost. Once the requests get cached, it functions pretty decently, though.
Also since we can't really load environment variables on vanilla JS, if we want to run the project in localhost, the frontend must be manually configured by changing the api URL on **frontend/js/data.js**, line 1.

Reference materials

Testing:

- **https://circleci.com/blog/unit-testing-vs-integration-testing/**
- **TQS course slides and previous labs**

CI/CD:

- **https://docs.github.com/pt/actions/hosting-your-own-runners/about-self-hosted-runners**
- **https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks**