HW1: Mid-term assignment report

Diogo Cruz [98595], v2022-05-02

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.

The purpose of this web application is to provide details on COVID incidence data. This was develop using a multi-layer web application, in Spring Boot, supplied with automated tests, integration with external sources (API) and a Cache implementation.

1.2 Current limitations

Because of the use of an external API, some data is outdated or missing completely, something we are not responsible for.

Also, some tests could have been further exploited.

The CI framework is missing the components for testing Selenium Web Drivers and there is only access to one external API.

These features where not implement due to setbacks during production and lack of time.

2 Product specification

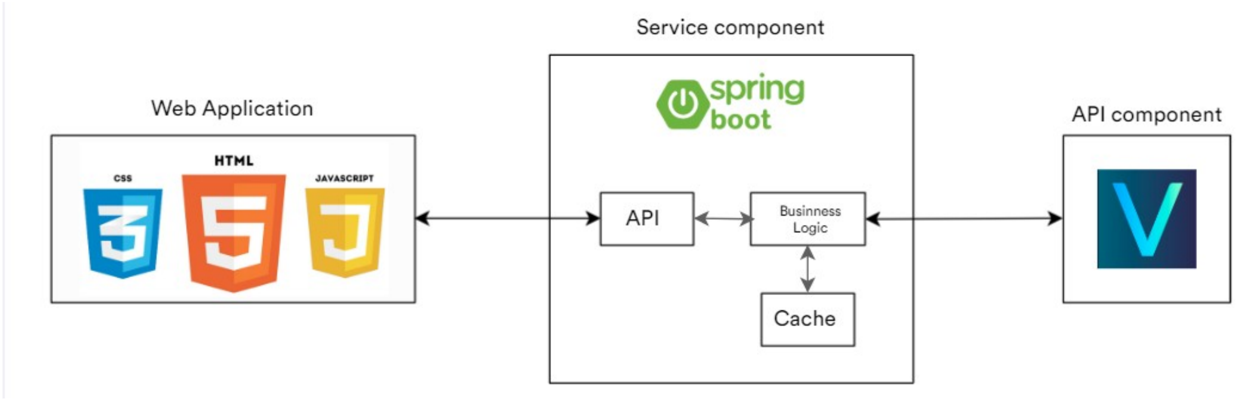
2.1 Functional scope and supported interactions

The developed web application aims to provide a simple, user-friendly interface that can show data related to the COVID-19 pandemic. When users open the webpage they can see the current global situation, as well as the top 10 most affected countries worldwide. If they want to search for a specific country, they can do so from the main page and check data related to the current day (if available) and data from the last six months. All this is provided by the API (VACCOVID)

2.2 System architecture

The architecture is divided in 3 sections:

- Service component using **Spring Boot**
- Web component using **HTML**, **JavaScript** and **CSS**
- API component using the **VACCOVID**.



2.3 API for developers

The web app contains only 3 endpoints:

- **/world** - access to total world data and top 10 most affected countries
- **/{iso}** - access to a specific country specified by the *iso* code provided in the *url* (example: /prt returns Portugal data)
- **/cache** - access to details about the implemented cache (requests, hits, misses)

Endpoint created to populate the *HTML* select option in the main page;

- **/countries** – this endpoint returns a list of only the names all the countries provided by the API.

Endpoints created for testing purposes (not available directly through the application):

- **/report/{iso}** - returns the daily data for the country specified in the *url.*
- **/report/world** – returns the total worldwide data since the start of the pandemic.

- **/report/top10** - returns the 10 countries most affected by COIVD-19 cases.
- **/report/iso/{country}** – returns the *iso code* in string format of the specified country name in the *url.*
- **/report/country/{iso}** – returns the country name in string format of the specificized *iso code* in the *url.*
- **/report/lastsixmonths/{country}** – returns a list of the daily data from the last 6 months in the specified country name in the *url.*
- **/report/cache** – returns in string format the cache details.

3 Quality assurance

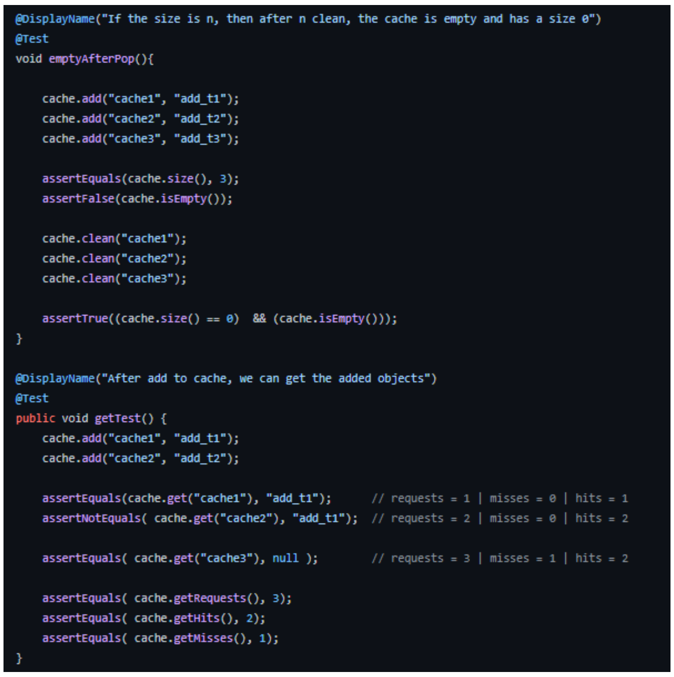
3.1 Overall strategy for testing

Before anything, the skeleton of the architecture was designed, creating all the necessary components and their relations between each other. Then, when the product idea became clearer, writing the necessary tests became a simpler task because it became more clear which tests to implement.

We tested as many components as possible: for the backend ranging from Unit tests, to Integration tests, utilizing tools such as *Mockito* and *MockMvc*; for the frontend we went with a Behavior-Driven Development, using tools such as *Cucumber* and *Selenium.*

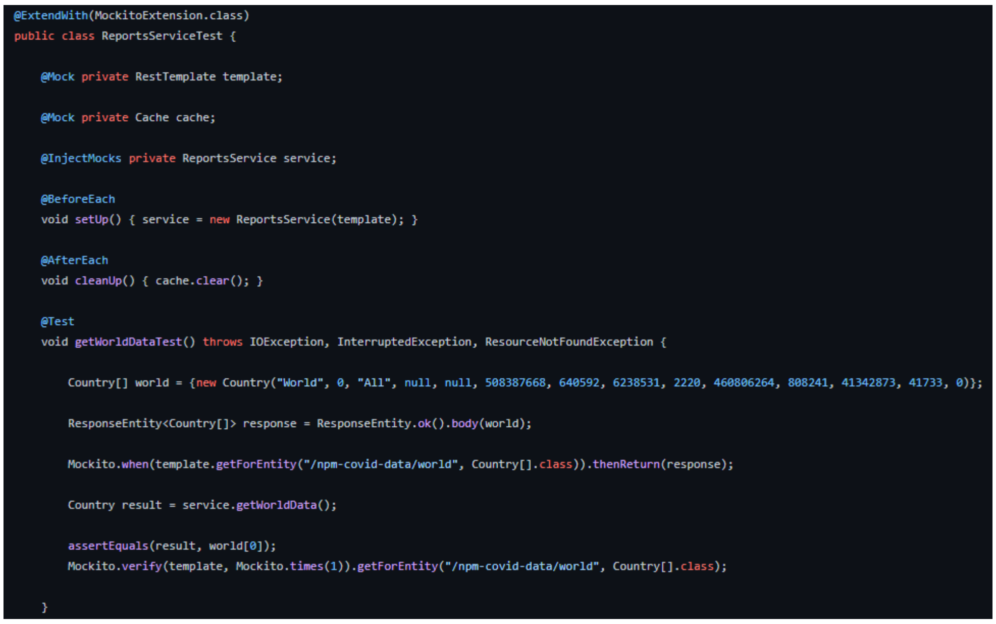
3.2 Unit and integration testing

For **Unit tests** were implemented to make sure that individual parts were working as intended. We used these tests in the *model* and *cache* components to test their functionalities. For example, in the *CacheTest.java,* elements such as the cache creation and adding or removing elements



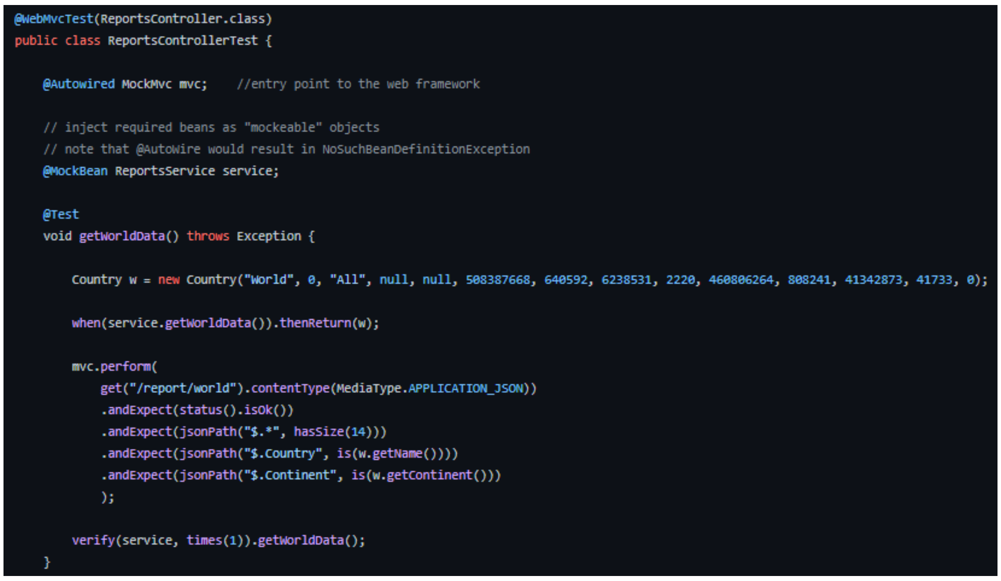
```java
@DisplayName("If the size is n, then after n clean, the cache is empty and has a size 0")
@Test
void emptyAfterPop(){

    cache.add("cache1", "add_t1");
    cache.add("cache2", "add_t2");
    cache.add("cache3", "add_t3");

    assertEquals(cache.size(), 3);
    assertFalse(cache.isEmpty());

    cache.clean("cache1");
    cache.clean("cache2");
    cache.clean("cache3");

    assertTrue((cache.size() == 0)  && (cache.isEmpty()));
}

@DisplayName("After add to cache, we can get the added objects")
@Test
public void getTest() {
    cache.add("cache1", "add_t1");
    cache.add("cache2", "add_t2");

    assertEquals(cache.get("cache1"), "add_t1");      // requests = 1 | misses = 0 | hits = 1
    assertNotEquals( cache.get("cache2"), "add_t1");  // requests = 2 | misses = 0 | hits = 2

    assertEquals( cache.get("cache3"), null );        // requests = 3 | misses = 1 | hits = 2

    assertEquals( cache.getRequests(), 3);
    assertEquals( cache.getHits(), 2);
    assertEquals( cache.getMisses(), 1);
}
```

We also used **Unit testing** in the *service* component using and injecting *Mock* objects and with the help of a *RestTemplate* to help with the API call for testing. This was done to make sure that, even though unit tests pass, the connection between components is also functional.



```java
@ExtendWith(MockitoExtension.class)
public class ReportsServiceTest {

    @Mock private RestTemplate template;

    @Mock private Cache cache;

    @InjectMocks private ReportsService service;

    @BeforeEach
    void setUp() { service = new ReportsService(template); }

    @AfterEach
    void cleanUp() { cache.clear(); }

    @Test
    void getWorldDataTest() throws IOException, InterruptedException, ResourceNotFoundException {

        Country[] world = {new Country("World", 0, "All", null, null, 508387668, 640592, 6238531, 2220, 460806264, 808241, 41342873, 41733, 0)};

        ResponseEntity<Country[]> response = ResponseEntity.ok().body(world);

        Mockito.when(template.getForEntity("/npm-covid-data/world", Country[].class)).thenReturn(response);

        Country result = service.getWorldData();

        assertEquals(result, world[0]);
        Mockito.verify(template, Mockito.times(1)).getForEntity("/npm-covid-data/world", Country[].class);

    }
```

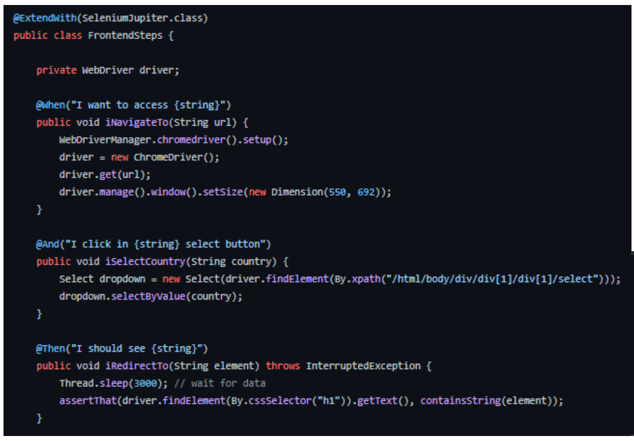
For **Integration tests** we tested the *controller* component, using tools such as *SpringBoot MockMvc*. Using the test endpoints mentioned above we can access their behaviour to make sure they are working correctly. To make that happen, a call is made to the *service* component and the answer provided must be expected by the *controller test class.*



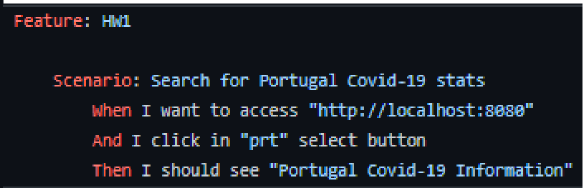
```java
@WebMvcTest(ReportsController.class)
public class ReportsControllerTest {

    @Autowired MockMvc mvc;    //entry point to the web framework

    // inject required beans as "mockeable" objects
    // note that @AutoWire would result in NoSuchBeanDefinitionException
    @MockBean ReportsService service;

    @Test
    void getWorldData() throws Exception {

        Country w = new Country("World", 0, "All", null, null, 508387668, 640592, 6238531, 2220, 460806264, 808241, 41342873, 41733, 0);

        when(service.getWorldData()).thenReturn(w);

        mvc.perform(
            get("/report/world").contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.*", hasSize(14)))
            .andExpect(jsonPath("$.Country", is(w.getName())))
            .andExpect(jsonPath("$.Continent", is(w.getContinent()))
            );

        verify(service, times(1)).getWorldData();
    }
```

3.3 Functional testing

For **functional testing** the frontend of the product was tested, using tools such as *Cucumber* and *Selenium Web Driver.* With both these tools we described scenarios with certain tasks to be performed of what should happen when using the application.



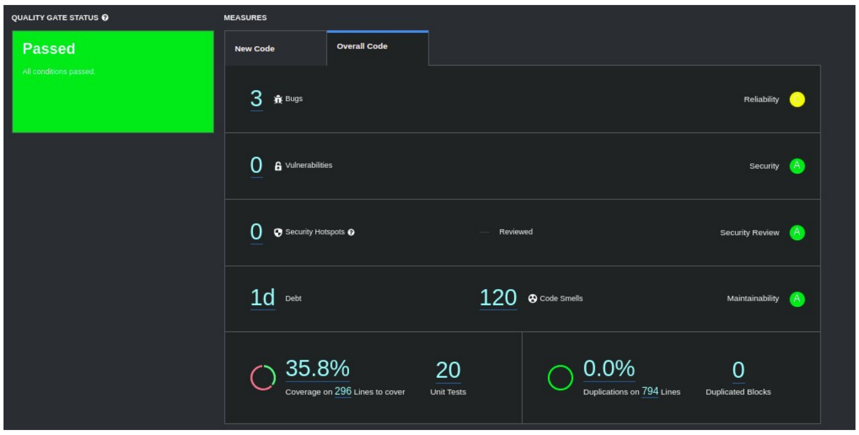
```java
@ExtendWith(SeleniumJupiter.class)
public class FrontendSteps {

    private WebDriver driver;

    @When("I want to access {string}")
    public void iNavigateTo(String url) {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
        driver.get(url);
        driver.manage().window().setSize(new Dimension(550, 692));
    }

    @And("I click in {string} select button")
    public void iSelectCountry(String country) {
        Select dropdown = new Select(driver.findElement(By.xpath("/html/body/div/div[1]/div[1]/select")));
        dropdown.selectByValue(country);
    }

    @Then("I should see {string}")
    public void iRedirectTo(String element) throws InterruptedException {
        Thread.sleep(3000); // wait for data
        assertThat(driver.findElement(By.cssSelector("h1")).getText(), containsString(element));
    }
```



```gherkin
Feature: HW1

    Scenario: Search for Portugal Covid-19 stats
        When I want to access "http://localhost:8080"
        And I click in "prt" select button
        Then I should see "Portugal Covid-19 Information"
```

3.4 Code quality analysis

Code quality analysis was made using *SonarQube.* 3 Bugs are found and there are 120 code smells.



QUALITY GATE STATUS

Passed

All conditions passed.

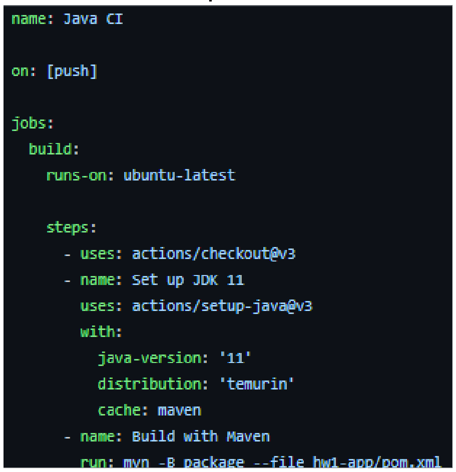
MEASURES

New Code | Overall Code

3	Bugs		Reliability C
0	Vulnerabilities		Security A
0	Security Hotspots	— Reviewed	Security Review A
1d	Debt	120 Code Smells	Maintainability A
35.8%	Coverage on 296 Lines to cover	20	Unit Tests
0.0%	Duplications on 794 Lines	0	Duplicated Blocks

3.5 Continuous integration pipeline [optional]

A CI pipeline was implemented using Github Actions. All tests pass except the *Selenium Web Driver* test which fails because a docker component is not implemented in the project.
The reason it's not implemented is simple due to setbacks during production and lack of time.



```yaml
name: Java CI

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 11
        uses: actions/setup-java@v3
        with:
          java-version: '11'
          distribution: 'temurin'
          cache: maven
      - name: Build with Maven
        run: mvn -B package --file hw1-app/pom.xml
```

4 References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/DXOGO/HW1
Video demo	Included in the repository in the *demo* folder

Reference materials

Material provided in the class by the professor.

External API VACCOVID