HW1: Mid-term assignment report

Diogo Marcelo Oliveira Monteiro [97606], v2022-04-26

Índice

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 developed product consists of a REST API that provides COVID-19 incidence data on a global scope, or per region, for a given day or period. The API resorts to an external API to obtain the data and uses an in-memory cache to reduce external API calls for frequent queries. Along side the API, a simple website was also developed to showcase the API's capabilities.

1.2 Current limitations

The API relies on only one external API to fetch data, meaning it is entirely dependent on its availability, and no alternative data sources can be chosen. A means to select different sources could have been implemented but was not.

Another limitation is that region data can only be obtained through the region’s ISO code. The region name parameter is unimplemented.
Getting incidence data for a period will return one set of values, consisting of a summary of the period. An endpoint to get a list of the individual value sets per day in the specified period may have been desirable but was not implemented.

2 Product specification

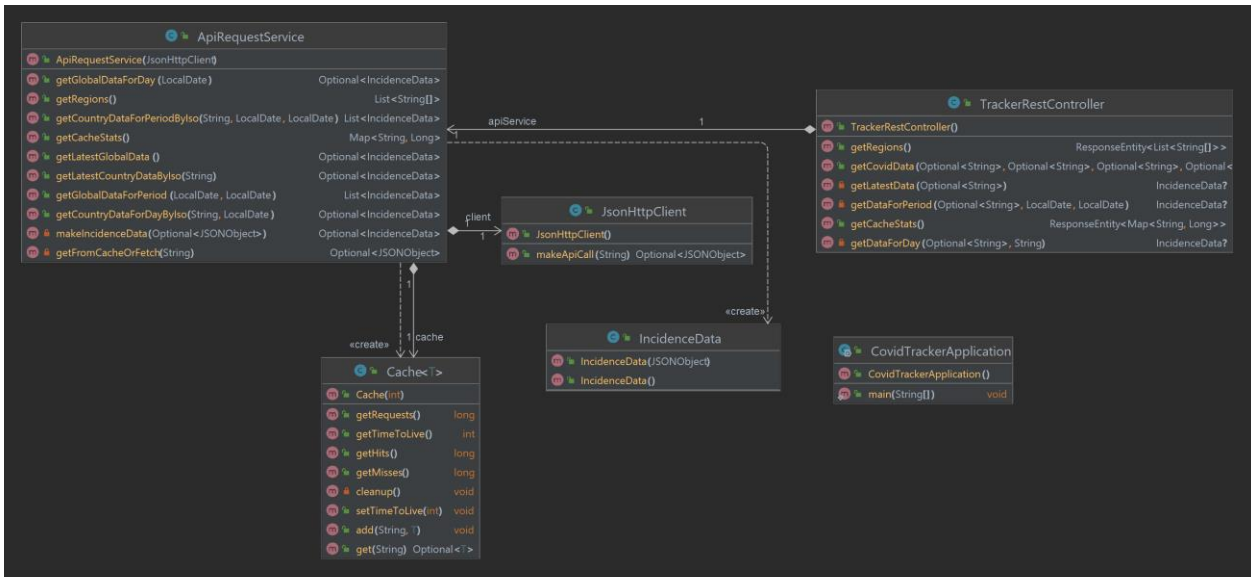
2.1 Functional scope and supported interactions

The application allows users to view COVID-19 incidence data such as total number cases, new cases, recoveries, deaths, among other stats on a global scope or for a given region, with the option to view data for specific days or periods.
Usage scenarios include:

- Comparing two regions’ COVID-19 statistics
- Seeing the evolution of one region’s statistics over time
- Obtaining the number of new cases for a three day period

2.2 System architecture

The architecture is as follows:


ApiRequestService
ApiRequestService(JsonHttpClient)
getGlobalDataForDay (LocalDate) Optional<IncidenceData>
getRegions() List<String[]>
getCountryDataForPeriodByIso(String, LocalDate, LocalDate) List<IncidenceData>
getCacheStats() Map<String, Long>
getLatestGlobalData () Optional<IncidenceData>
getLatestCountryDataByIso(String) Optional<IncidenceData>
getGlobalDataForPeriod (LocalDate, LocalDate) List<IncidenceData>
getCountryDataForDayByIso(String, LocalDate) Optional<IncidenceData>
makeIncidenceData(Optional<JSONObject>) Optional<IncidenceData>
getFromCacheOrFetch(String) Optional<JSONObject>
apiService
1
1
client
1
1
JsonHttpClient
JsonHttpClient()
makeApiCall(String) Optional<JSONObject>
TrackerRestController
TrackerRestController()
getRegions() ResponseEntity<List<String[]>>
getCovidData (Optional<String>, Optional<String>, Optional<String>, Optional<
getLatestData (Optional<String>) IncidenceData?
getDataForPeriod (Optional<String>, LocalDate, LocalDate) IncidenceData?
getCacheStats() ResponseEntity<Map<String, Long>>
getDataForDay (Optional<String>, String) IncidenceData?
1
«create»
1 cache
«create»
Cache<T>
Cache(int)
getRequests() long
getTimeToLive() int
getHits() long
getMisses() long
cleanup() void
setTimeToLive(int) void
add(String, T) void
get(String) Optional<T>
IncidenceData
IncidenceData(JSONObject)
IncidenceData()
CovidTrackerApplication
CovidTrackerApplication()
main(String[]) void
Figure 1 - API Class Diagram

For the API, the Spring Boot framework was utilized, with the Web and Devtools starter dependencies. Additionally, the “json-simple” library was used for JSON parsing.
As for the website, Flask was used alongside Bootstrap and plain HTML, CSS and JS, with the “requests” python library to make API calls.

2.3 API for developers

The API has the following endpoints, with the base URL “localhost:8080/api/v1”

- **GET:** /incidence[?[iso]&[date]&[start]&[end]]
    - **No parameters:** returns global data for current day
    - **iso:** returns data for region with given ISO code
    - **date:** returns data for given date
    - **start and end:** returns summary of data between start and end dates
- **GET:** /regions
    - Returns list of available regions and their ISO codes
- **GET:** /cache_stats
    - Returns cache statistics

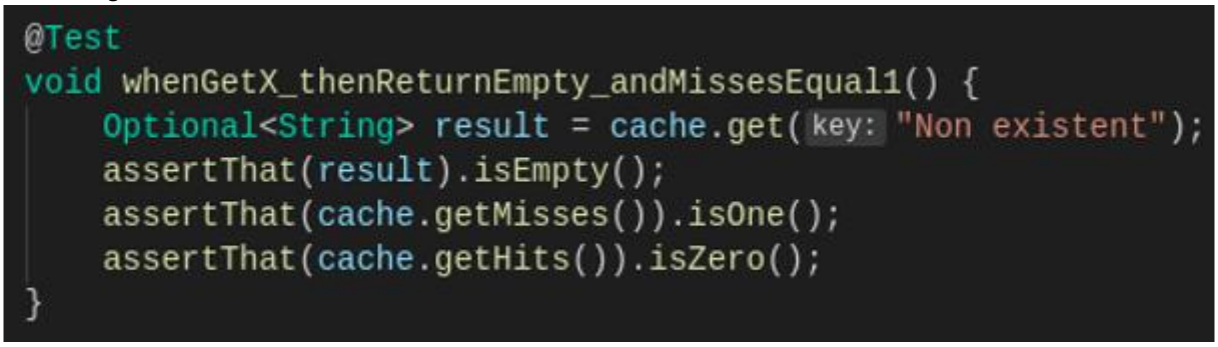
3 Quality assurance

3.1 Overall strategy for testing

To test the software, a Top-Down Development strategy was adopted. The basic skeleton for implementation, such as class and method definitions, were created but left unimplemented. After that, the appropriate tests were created by thinking of the intended functionally, focusing on what each component should do first and foremost. Then, the implementation was done by striving to pass all the previously written tests. Finally, some additional tests were written after the fact to cover some scenarios that were initially missed.

3.2 Unit and integration testing

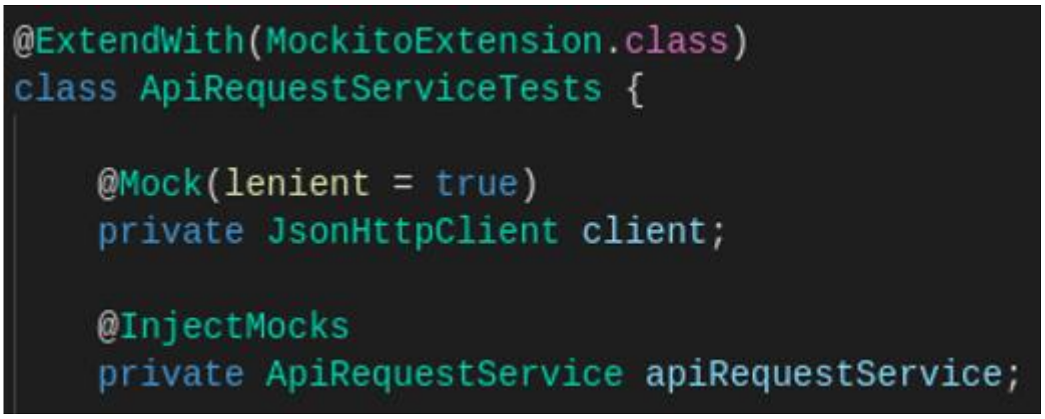
Unit tests were created for the smaller components of the solution, that could be tested in a vacuum. Said components are the "IncidenceData" class, the "Cache" class, and the "ApiRequestService" class.
The tests for the first two were rather simple, with the testing strategy adopted being the simulation of predictable scenarios and asserting that the outcomes were as expected. For example, here's a test for the Cache which describes the scenario "given an empty cache, when a get for X is made then X should not be found, and misses should be 1":



```
@Test
void whenGetX_thenReturnEmpty_andMissesEqual1() {
    Optional<String> result = cache.get( key: "Non existent");
    assertThat(result).isEmpty();
    assertThat(cache.getMisses()).isOne();
    assertThat(cache.getHits()).isZero();
}
```

Figure 2 - A cache unit test

Tests for the "ApiRequestService" class made use of mocking, as the subject under test has a dependency with may complicate predictable testing.



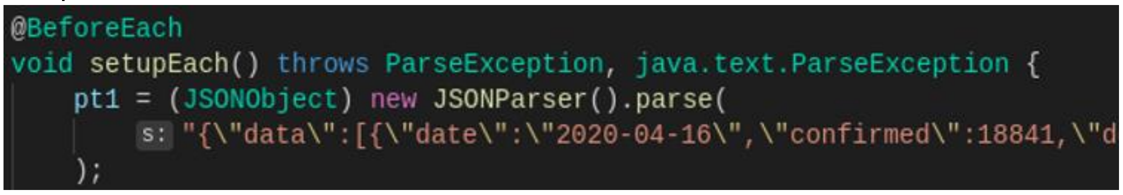
```java
@ExtendWith(MockitoExtension.class)
class ApiRequestServiceTests {

    @Mock(lenient = true)
    private JsonHttpClient client;

    @InjectMocks
    private ApiRequestService apiRequestService;
```

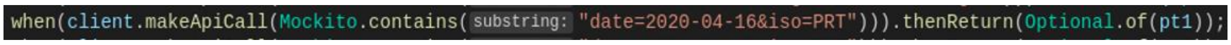
Figure 3 - Mocking Dependency Injections

This allowed us to setup predictable behavior for the dependencies, and reliably test the component.



```java
@BeforeEach
void setupEach() throws ParseException, java.text.ParseException {
    pt1 = (JSONObject) new JSONParser().parse(
        s: "{\"data\":[{\"date\":\"2020-04-16\",\"confirmed\":18841,\"d
    );
```

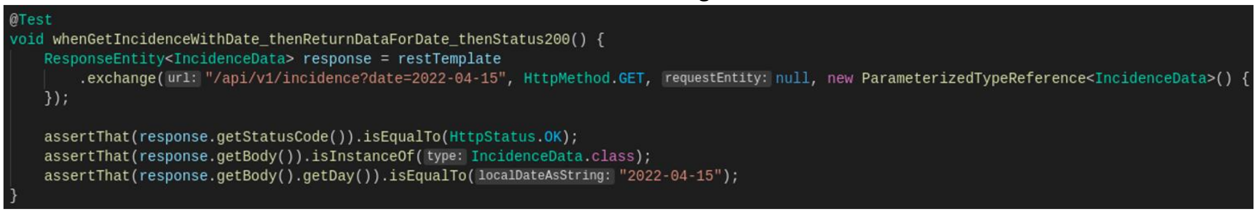
Figure 4 - A snippet of mocking setup



```java
when(client.makeApiCall(Mockito.contains( substring: "date=2020-04-16&iso=PRT"))).thenReturn(Optional.of(pt1));
```

Figure 5 - Mocking one of the dependency's methods

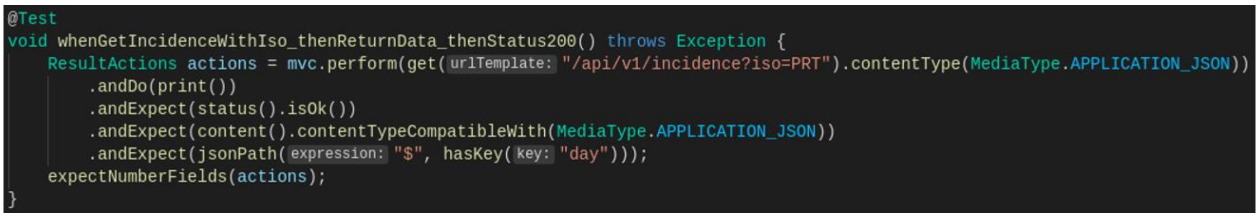
As for integration tests, the REST Controller was tested by loading the full application context, and two strategies were used to interact with the controller – MockMvc and Rest Template.
The tests were written to try different endpoints with different sets of parameters and expect certain formats and response codes.
Here follows two tests, with different interaction strategies:



```java
@Test
void whenGetIncidenceWithDate_thenReturnDataForDate_thenStatus200() {
    ResponseEntity<IncidenceData> response = restTemplate
        .exchange( url: "/api/v1/incidence?date=2022-04-15", HttpMethod.GET,  requestEntity: null, new ParameterizedTypeReference<IncidenceData>() {
    });

    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(response.getBody()).isInstanceOf( type: IncidenceData.class);
    assertThat(response.getBody().getDay()).isEqualTo( localDateAsString: "2022-04-15");
}
```

Figure 6 - Integration Test using Rest Template



```java
@Test
void whenGetIncidenceWithIso_thenReturnData_thenStatus200() throws Exception {
    ResultActions actions = mvc.perform(get( urlTemplate: "/api/v1/incidence?iso=PRT").contentType(MediaType.APPLICATION_JSON))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath( expression: "$", hasKey( key: "day")));
    expectNumberFields(actions);
}
```

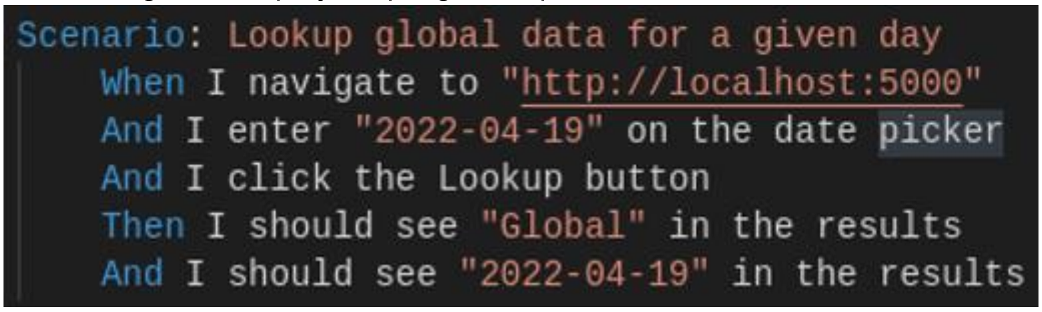
Figure 7 - Integration Test using MockMvc

3.3 Functional testing

The website was tested with following cases in mind:

- View global data for a day
- View global data for a period
- View region data for a day
- View region data for a period

To test these cases, Selenium was used. Firstly, by using Selenium IDE, each case was recorded and exported. Using Cucumber, these cases were written as features and scenarios, then coding each step by adapting the exported code from Selenium IDE.

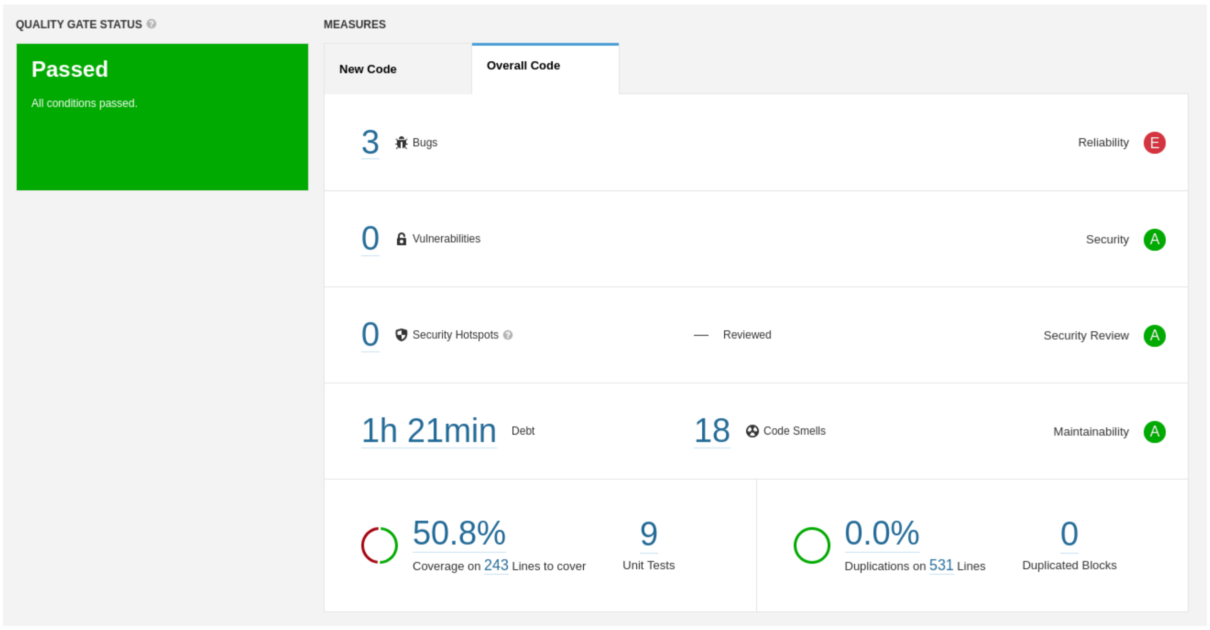


```
Scenario: Lookup global data for a given day
    When I navigate to "http://localhost:5000"
    And I enter "2022-04-19" on the date picker
    And I click the Lookup button
    Then I should see "Global" in the results
    And I should see "2022-04-19" in the results
```

Figure 8 - Scenario written in Gherkin, for use with Cucumber

3.4 Code quality analysis

For static code analysis, a local SonarQube container was used.
The first scan was made right after the initial implementation was done, and the results were as follows:



QUALITY GATE STATUS
Passed
All conditions passed.

MEASURES
New Code | Overall Code

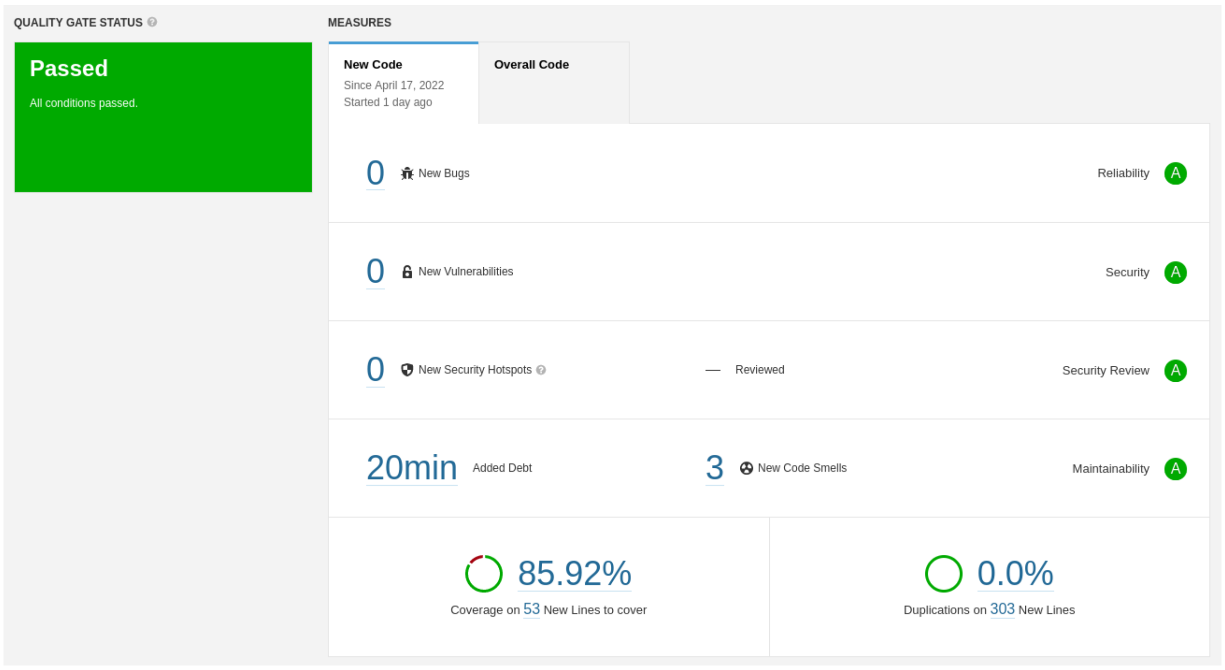
3 Bugs — Reliability E
0 Vulnerabilities — Security A
0 Security Hotspots — Reviewed — Security Review A
1h 21min Debt — 18 Code Smells — Maintainability A
50.8% Coverage on 243 Lines to cover — 9 Unit Tests
0.0% Duplications on 531 Lines — 0 Duplicated Blocks

Figure 9 - First SonarQube scan

As pictured, 3 bugs were found and quite a few code smells as well. The test coverage was also lower than expected, considering all the unit and integration tests done.
The bugs were addressed, as well as most code smells. Most of the smells were minor, but one major smell was "Excess code complexity", with one method having many "if/else" statements. To solve the issue, some parts of the code were made into support functions, to make the code more readable and easier to follow.

As for the coverage, it turns out that the integration tests were not being considered for this value, due to the test classes names not ending in "test". Renaming the classes appropriately, for example "TrackerRestControllerIT" to "TrackerRestControllerITest", fixed the issue.

A follow-up scan yielded the following:


QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code
Since April 17, 2022
Started 1 day ago

Overall Code

0 New Bugs — Reliability A

0 New Vulnerabilities — Security A

0 New Security Hotspots — — Reviewed — Security Review A

20min Added Debt — 3 New Code Smells — Maintainability A

85.92% Coverage on 53 New Lines to cover

0.0% Duplications on 303 New Lines

Figure 10 - Second SonarQube Scan

As depicted, all the bugs are resolved and so are most code smells, and coverage has reached a satisfactory value.

4 References & resources

Project resources

Resource:	**URL/location:**
Git repository	https://github.com/diomont/tqs_97606/tree/main/HW1
Video demo	https://github.com/diomont/tqs_97606/blob/main/HW1/website_demo.mp4

Reference materials

Various Baeldung guides