deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

HW1: Mid-term assignment report

Pedro Daniel Fonts Lopes [97827]

1 Introduction

1.1 Overview of the work

Covid Tracker service is a full featured Spring boot application that exposes 3 endpoints that are expected to be used with conjunction of a web application. The main objective is to present covid statics according to days and cities or countries. In order to assure the system usability some quality gates and usability parameters were set which will be presented later in section 3.

1.2 Current limitations

As of now the current limitations reside in the parameters which the available endpoint's take, they are not very intuitive since that the external API used states that Portugal is a City and for instance Aveiro has no data, this is confusing and takes its toll on the final product, on way of addressing this issue is too refactor some business logic and implement a "City" resolution service of some kind to correctly and more conveniently translate this application

endpoint parameters to the correct parameters of the external API.
On another level since this application only queries a single external API there's no fault tolerance and no redundancy built in.

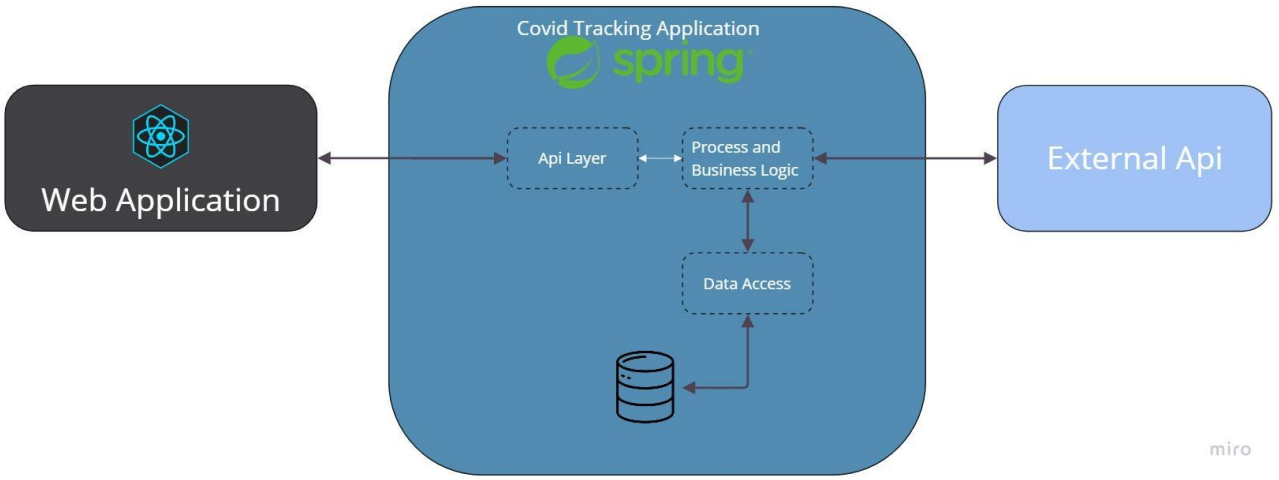
2 Product specification

2.1 Functional scope and supported interactions

The main usage of this application is to use it as a way of staying informed about the current COVID-19 situation all over the world, since that a small website is also included in the application in the for of an separate React web application, which can compile some information and plot graphs about the situation of the last few days.

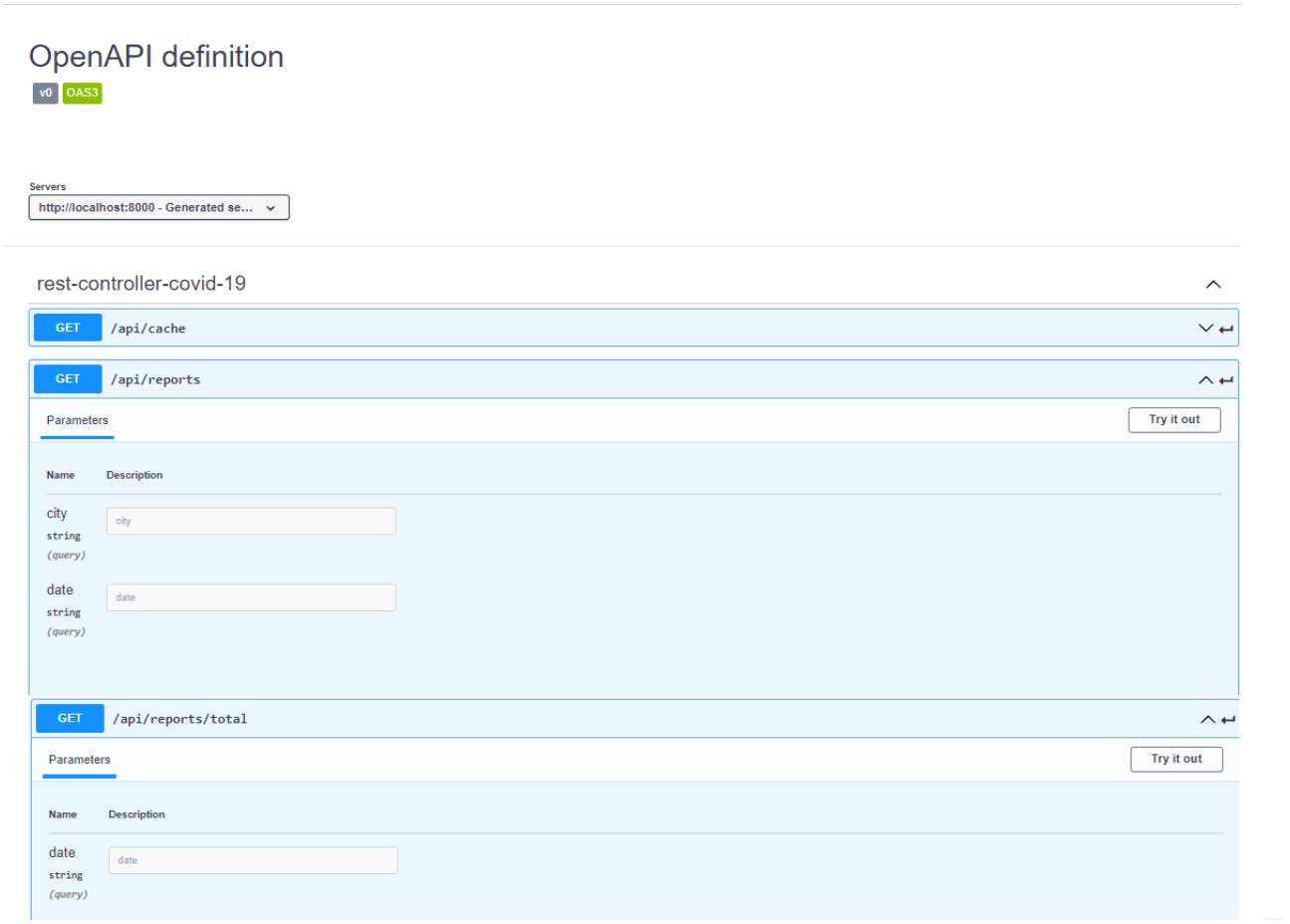
2.2 System architecture

This application consists of three actors, its core is built using Spring Boot Framework, then the presentation layer is built with React, finally the core of the system queries an external API.
Inside the core of the system, it is possible to identify 4 layers, which are, the API Layer that exposes 3 endpoints, the business and process logic that ensures the correct flow of requests and responses from the API Layer, External API, this layer also ensures that every request to the API Layer is correctly cached and destroyed after some time.
Finally we have the Data Access Layer that stores everything in a volatile H2 Database.



2.3 API for developers

As of now the application exposes 3 REST endpoints, one to query for data related to a region or city, and another one to query about global COVID statistics, both according to a date, the last one presents information about the number of Requests that have been made, how many cache hits and how many misses.



3 Quality assurance

3.1 Overall strategy for testing

The strategy of developing this application did not make use of TDD, firstly all the code was developed, and after that tests were progressively written to correctly assert the behaviour of the final product. In the end every test that failed since that was the expected behaviour the code affecting the test was modified until said test passed.
Some of the tests made use of more than one testing tool. For instance, the cucumber tests were made with selenium and **junit** extensions as well as **hamcrest** and **assertJ**.
Since this is a Spring Boot Application it was decided to take on the testing layer by layer, starting from the top, RestController, to the bottom, Data JPA Testing, this way every test was loaded with the minimum Spring Boot Context needed, since the *lower* part of each layer, which dealt with the layer under was mocked using Mockito junit5 integration.

3.2 Unit and integration testing

Integration tests were only introduced to the rest controller, since this was the only layer deemed to need this type of test, as of the rest it was only used unit testing, with the help of the above mentioned libraries.
Some tests and examples can be seen here:

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

  @Mock
  Resolver resolver;
  @Mock
  private CacheRepository repository;
  @Mock
  private CacheManager cacheManager;
  @InjectMocks
  private Covid19Service service;

  @Test
   void testReportForCachedResult() {
    ReportData reportData =
      new ReportData( "2020-04-11", 3217, 92, 0, 270, 12,
        0, "2020-04-11 22:45:33", 3125, 258, 0.0286
        , new RegionReportData() );
    ArrayList<ReportData> arrayListData = new ArrayList<>();
    arrayListData.add( reportData );
    RootReport rootReport = new RootReport( arrayListData );

    String url =
"https://covid-19-statistics.p.rapidapi.com/reports?city_name=Autauga&date=20
20-04-11";
    when( resolver.getURLForSpecificCityAndDate( Mockito.any(), Mockito.any()
) ).thenReturn( url );

    when( cacheManager.inCache( "https://covid-19-statistics.p.rapidapi" +
      ".com/reports?city_name=Autauga&date=2020-04-11" ) ).thenReturn( new
Cache( url, rootReport ) );

    ResponseEntity<Object> responseEntity = service.getReportForCityAndDate(
"Autauga", "2020-04-11" );
    assertThat( responseEntity.getStatusCode() ).isEqualTo( HttpStatus.OK );

    verify( resolver, times( 1 ) ).getURLForSpecificCityAndDate(
Mockito.any(), Mockito.any() );
```

```
    verify( resolver, times( 0 ) ).getSpecificReportFor( Mockito.any(),
Mockito.any() );
    verify( cacheManager, times( 1 ) ).inCache( Mockito.any() );
  }
```

Here we can see a test that is occurring without loading any Spring boot Context, all the mocking of the agents is conducted using the Inversion of Control of Mockito, this is clear with the annotation @Mock and @InjectMocks.

On the other hand, this an example of a test related to the RestController integration test, also loaded with the minimal Spring Boot Context which in this case is the WebMvc, with Inversion of Control and Dependency Injection managed by Spring Boot.

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK, classes
= Covid19TrackingServiceApplication.class)
@AutoConfigureMockMvc
 class RestControllerTestIT {
  @Autowired
  private MockMvc mvc;

  @Autowired
  private CacheRepository repository;

  @AfterEach
   void resetDb() {
    repository.deleteAll();
  }

  @Test
  void whenValidInput_thenReturnResponseDataShouldBeSummaryReport() throws
Exception {
    mvc.perform( get( "/api/reports/total?date=2020-04-11" ).contentType(
MediaType.APPLICATION_JSON ) )
       .andDo( print() ).andExpect( status().isOk() )
       .andExpect( content().contentTypeCompatibleWith(
MediaType.APPLICATION_JSON ) )
       .andExpect( jsonPath( "$.data.date", is( "2020-04-11" ) ) )
       .andExpect( jsonPath( "$.data.confirmed", is( 1771514 ) ) )
       .andExpect( jsonPath( "$.data.fatality_rate", is( 0.0612 ) ) );

  }
```

It is also important to notice that this integration test uses a classic approach of Spring Boot Testing with the MockMvc class in contrast of the use of Rest Assured that was used in the following example:

```
@WebMvcTest(RestControllerCovid19.class)  class RestControllerTest {
  @MockBean private Covid19Service service;

  @Autowired private MockMvc mvc;

  @BeforeEach  void setUp() throws Exception {
    RestAssuredMockMvc.mockMvc( mvc );
  }

  @AfterEach  void tearDown() throws Exception {
    RestAssuredMockMvc.reset();
  }

  @Test  void
testEndpointFor200StatusCodeAnd1callToServiceGetCacheWillResultIn1CallToServi
ceGetCache() {
    ObjectMapper objectMapper = new ObjectMapper();
    ObjectNode objectNode = objectMapper.createObjectNode();
    objectNode.put( "hits", 0 );
    objectNode.put( "misses", 0 );
    objectNode.put( "requests", 0 );
    when( service.getCache() ).thenReturn( ResponseEntity.ok( objectNode ) );
    given().get( "/api/cache" ).then().log().body().assertThat().status(
HttpStatus.OK ).and()
           .contentType( ContentType.JSON ).and().body( "hits", is( 0 )
).and().body( "misses", is( 0 ) ).and()
           .body( "requests", is( 0 ) );
    verify( service, times( 1 ) ).getCache();
 }
```

3.3 Functional testing

Functional testing was done running the website and the Test with full Spring Boot test context loaded, then there were written features in gherkin, finally the web page was tested using selenium for java, with the help of the **assertj** library.
Gherkin features were written to simulate how the user would interact with the web application.

```
@And("all of them should have the values {int}")
 public void allOfThemShouldHaveTheValues( int arg0 ) throws
```

```
InterruptedException {

    Thread.sleep( 4000 );
    WebElement requests =
      driver.findElement( By.cssSelector( "#root > div > div:nth-child(2) >
div:nth-child(1) > p:nth-child(2)" ) );
    WebElement misses =
      driver.findElement( By.cssSelector( "#root > div > div:nth-child(2) >
div:nth-child(2) > p:nth-child(2)" ) );
    WebElement hits =
      driver.findElement( By.cssSelector( "#root > div > div:nth-child(2) >
div:nth-child(3) > p:nth-child(2)" ) );
    assertThat( requests.getText() ).isEqualTo( arg0 + "" );
    assertThat( misses.getText() ).isEqualTo( arg0 + "" );
    assertThat( hits.getText() ).isEqualTo( arg0 + "" );
  }
```

3.4 Code quality analysis

All the code quality analysis was done using the platform SonarCloud with GitHub Actions. This tool analysed best practices as well as test coverage and naming conventions, for packages, classes and variables.
Some of this would never be addressed if they were not present in this analysis.
An interesting fact about java conventions is that modifier keywords such as *final*, *static* *private* and *public* have an order that should be followed.

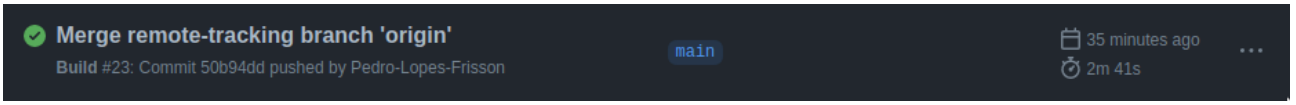


Figure 1: Github Actions build succeed

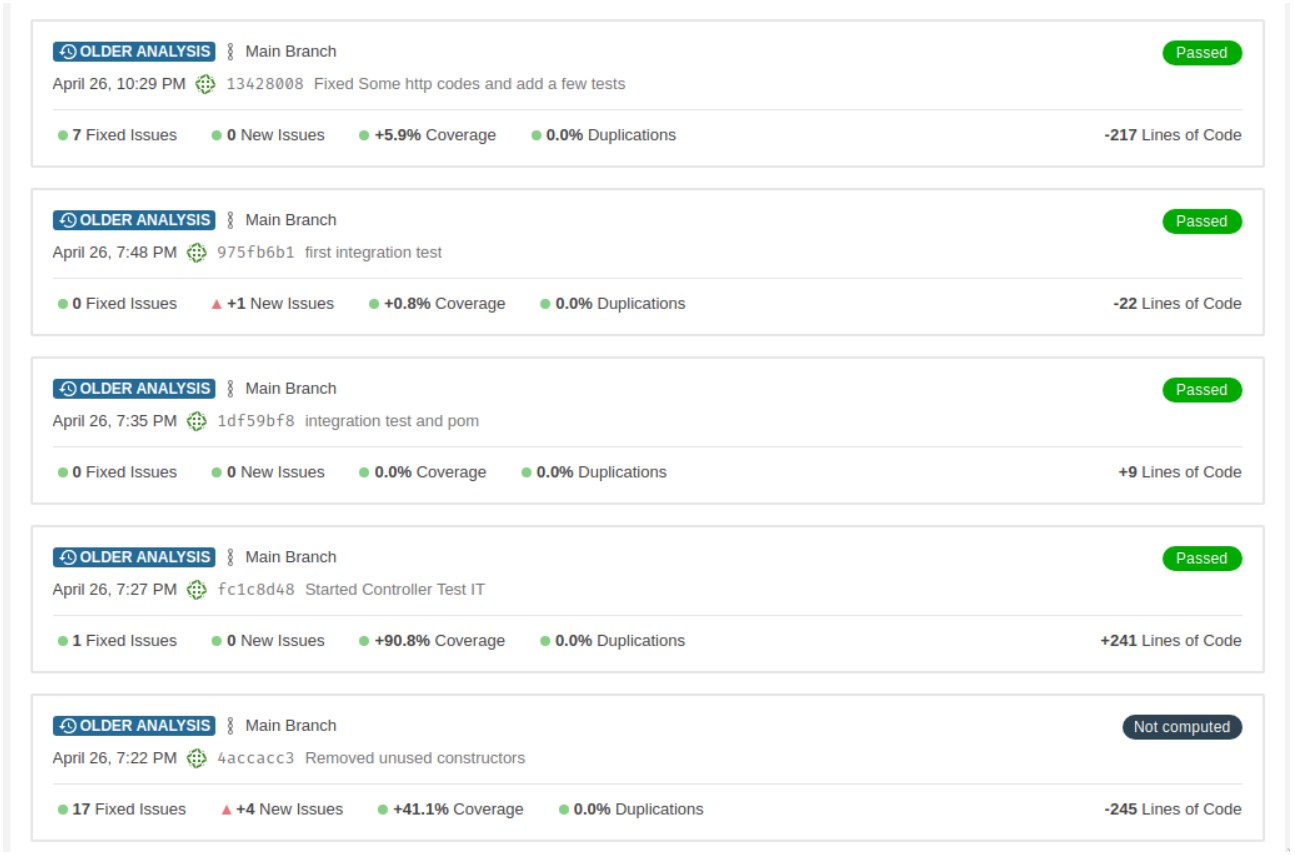


Figure 2: Some relevant commits

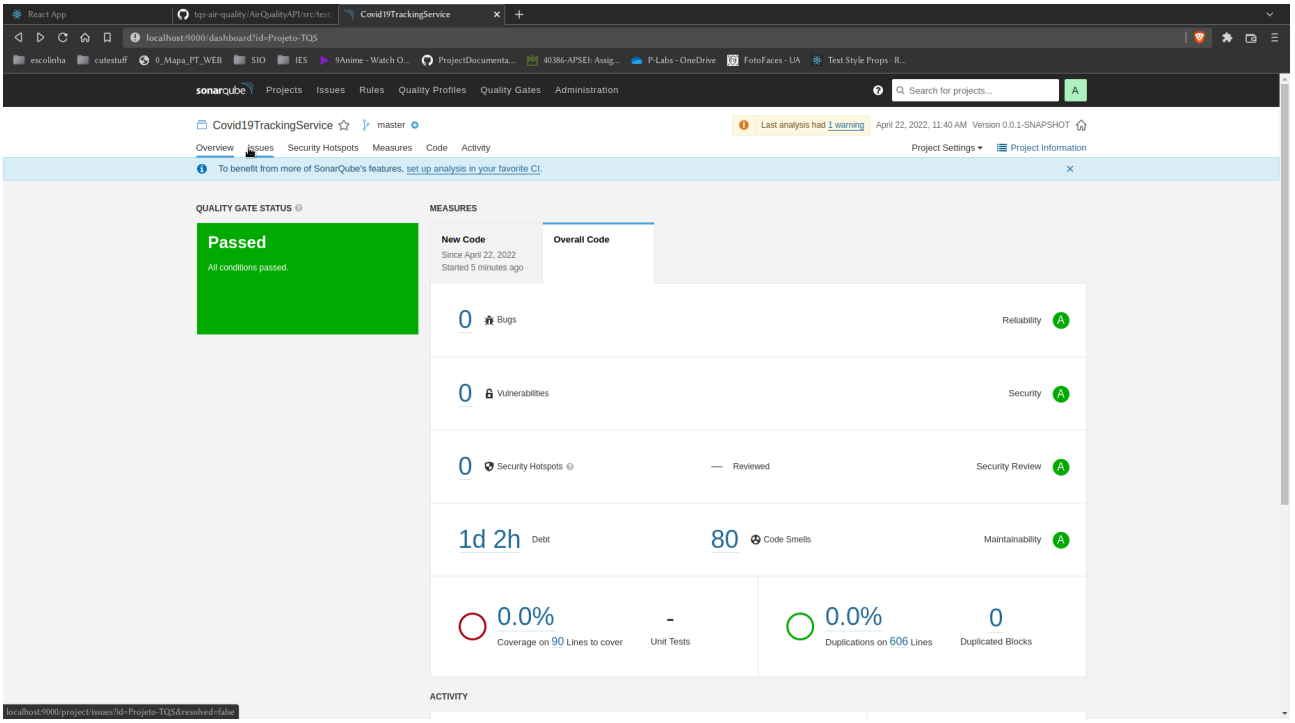


Figure 3:Initial Code Smells, no tests.

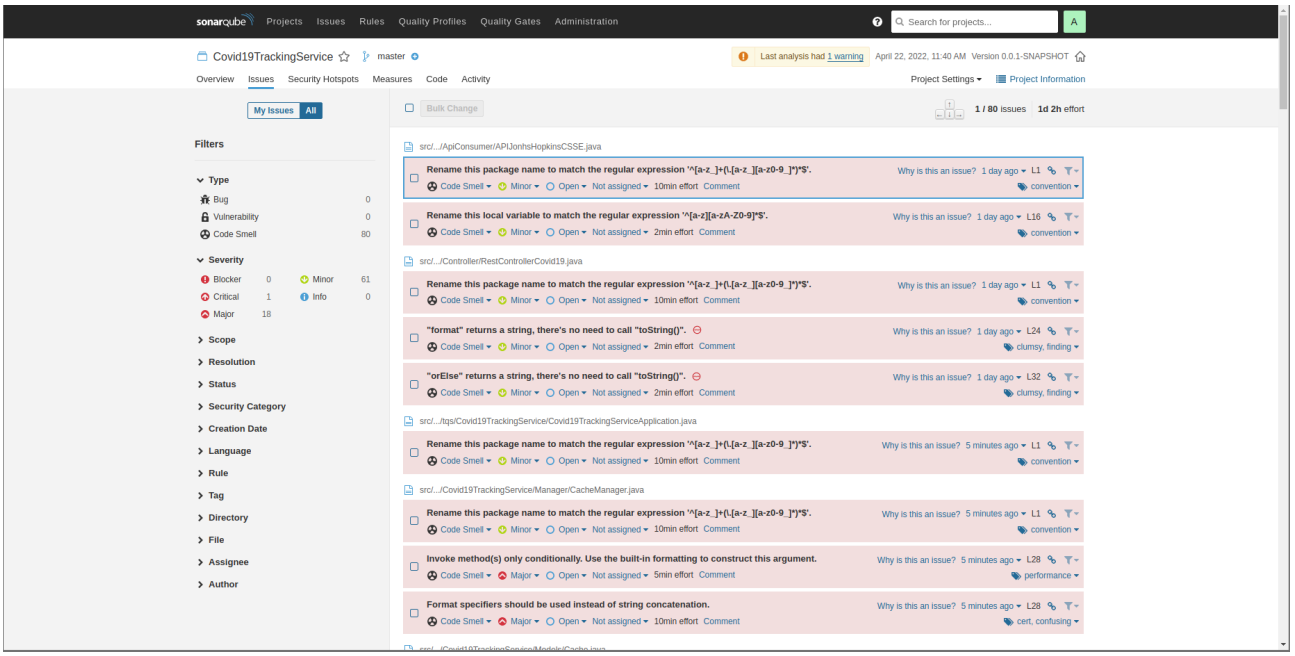


Figure 4: Some examples of the code smells

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/Pedro-Lopes-Frisson/tqs_97827/tree/main/HW1
Video demo	https://github.com/Pedro-Lopes-Frisson/tqs_97827/blob/main/HW1/2022-04-29%2023-13-45.mp4
QA dashboard (online)	https://sonarcloud.io/project/overview?id=Pedro-Lopes-Frisson_tqs_97827
Swagger UI	https://app.swaggerhub.com/apis/tqsCovidTracker/open-api_definition/v0#/

Reference materials

Used API https://rapidapi.com/axisbits-axisbits-default/api/covid-19-statistics/

Open API spring boot integration https://www.baeldung.com/spring-rest-openapi-documentation

Retrofit 2 for class instantiation and API querying https://www.baeldung.com/retrofit

MockWebServer mockable web server used to test Retrofit2
https://github.com/square/okhttp/tree/master/mockwebserver