HW1: Mid-term assignment report

Martinho Martins Bastos Tavares [98262], v2022-04-27

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 application (“Covid stats”) is essentially a REST API for providing COVID incidence data for a certain country/territory, extracting this data from external sources (other REST APIs). This application was built to support multiple APIs simultaneously, providing data on confirmed cases, deaths, recovered cases, active cases and the fatality rate, for a specified time period or a date. In terms of location, both country specific and global statistics are supported.

In its current state, the application has implemented support for 2 external APIs. The orchestration of the multiple APIs is done seamlessly at runtime, depending on the availability of the external source to fulfill the request, but preliminary configuration is also supported, allowing for enabling/disabling any sources or permitting incomplete responses (from sources that can’t provide all the data that the application presents).

The API also contains a cache to prevent multiple equal requests from being sent. Cache properties can also be set (time-to-live of the entries and maximum size), and its statistics are also provided in the API.

The application contains an extremely simple web app to interactively communicate with the API through a REST interface, allowing the user to see all COVID and cache statistics with all available query parameters.

1.2 Current limitations

In terms of implementation, there weren't any major features that came to mind that would be missing/faulty from the application. I would say some things weren't properly implemented, but due to time constraints some not ideal solutions were used. An example of this is setting the property "server.error.include-message = always" in "application.properties" in order to display in the web app the exceptions that are thrown in the REST API, which may not be secure if handled poorly.

If anything else, the implementation of some tests for the application used patterns that could have probably been avoided, as they aren't maintainable (usage of Java Reflection with "ReflectionTestUtils" from Spring).

2 Product specification

2.1 Functional scope and supported interactions

The application is built for people who wish to obtain COVID data from various sources in a simple format. API maintainers, on the other hand, can easily access the cache statistics (these can also be accessed by anyone else).

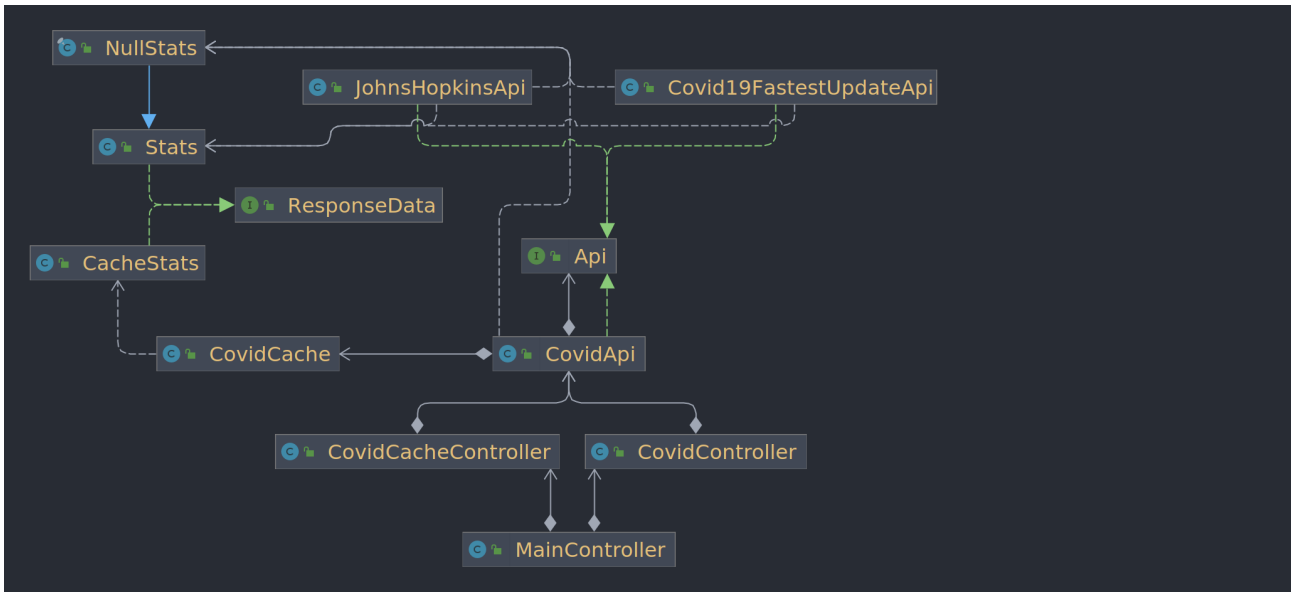
COVID data can be obtained for various scenarios: whether it's national or global statistics, and specify a type instance or interval.

The application offers a web interface for those who easily want to know the COVID statistics, and a REST API is also offered, providing the exact same data, for application developers.

Therefore, the defined actors are application developers, maintainers of the API itself, and normal people that want to check the pandemic status.

2.2 System architecture

Below is the main architecture diagram (custom exception classes, converters and the main application were excluded):



From the very beginning, the application was designed with the ability to use multiple external APIs in mind. For this, the Proxy design pattern was used, where the service layer bean, the CovidApi class, would act as a proxy to the external APIs, internally alternating between each of them depending on configuration or availability. Therefore, they all implement the same Api interface.

After some thorough surveying, the 2 chosen APIs are:

- Covid19FastestUpdate: https://documenter.getpostman.com/view/10808728/SzS8rjbc
  - This API provided data for all query parameters, but the world data is incomplete, missing one of the attributes
- JohnsHopkins: https://rapidapi.com/axisbits-axisbits-default/api/covid-19-statistics/
  - This API requires the usage of a RapidApi Key. This key is present in the application as a Spring application property, and can be provided through an environment variable (RAPID_API_KEY), so that it's not hardcoded
  - All query parameters were supported except the date range ones

Each of the APIs has its drawbacks when supporting the query parameters or providing data. Therefore, the CovidApi proxy will try to mix what each of them is capable of into a whole.

CovidApi contains a CovidCache instance and the other Api implementations, alternating between getting cached responses and requesting the APIs to fetch them.

The responses returned are implementations of the ResponseData interface, which allows the use of polymorphism to store different kinds of responses on the cache. The Stats class represents the COVID statistics and CacheStats represents the cache's. CacheStats responses are generated by the cache, while Stats are generated by the external APIs.

NullStats is a result of the Null Object design pattern, which was introduced in light of an issue faced during testing. The reason for this decision is detailed in the Unit and integration testing section of the document. In practice, NullStats represents an inexistent response, not due to unavailability, but due to configurations, and it's only to be used internally by the application, never presented to the end user.

The MainController internally uses the REST API to obtain its data (dependency on CovidCacheController and CovidController). Having it laid out like this ensures that all data that is presented on the MainController can also be obtained on the REST API.

This is a Spring Boot application. The used Spring Starter dependencies are:

- **Spring Web**: to build the Rest API and web application
- **Thymeleaf**: template engine for building the web interface
- **Spring WebFlux**: mainly for the use of WebClient, which makes calling external APIs easy

Other main dependencies used are:

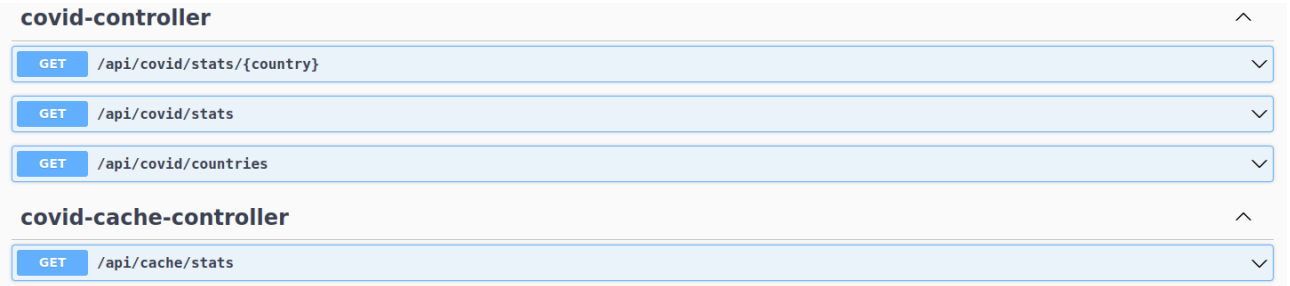
- **Gson**: for custom JSON serialization/deserialization of the responses from external APIs
- **JUnit5**: for testing
- **Cucumber**: to develop the functional tests on the web interface
- **Selenium**: *ditto*
- **HtmlUnit**: *ditto*
- **Awaitility**: for testing the cache's TTL (time-to-live) effect on the stored entries, which required rigorous timing
- **OkHttp3**: web server to "mock" the external APIs, in order to test the classes that call them
- **Lombok**: remove boilerplate code

In order to communicate unavailability of the external APIs and other errors that would be translated to HTTP statuses, various custom exceptions were developed. In order to communicate external API unavailability, the UnavailableExternalApiException was developed, and for overall API unavailability (when no external source can satisfy the request), then an UnavailableApiException is thrown, which is a ResponseStatusException. Exceptions that extended ResponseStatusException were displayed on the web app and the REST API, to communicate errors to the end users.

The logging utility used was Spring's default logger, Logback. Log statements that proved to be useful mainly for debugging are placed in important parts of the application.

2.3 API for developers

The image below details the available endpoints on the application (generated from Swagger UI):



For each controller, the available query parameters and the response schema are shown below (each of the generated schemas has a "null" attribute, which isn't actually present in the responses).

For the COVID statistics endpoints, the date parameters are optional and specify the following characteristics:

- date: the date at which to obtain the statistics
- after: obtain statistics relative to that date afterwards
- before: obtain statistics relative to that date beforehand

The *after* and *before* parameters specify the time range within which to obtain statistics, while *date* specifies a single point in time. These can be used simultaneously, but the *date* parameter will be prioritized.

Endpoint	Description	Parameters	Response Schema
/api/covid/stats/{country}	Obtain COVID statistics from a particular country		{ "confirmed": 0, "newConfirmed": 0, "deaths": 0, "newDeaths": 0, "recovered": 0, "newRecovered": 0, "active": 0, "newActive": 0, "fatalityRate": 0 }
/api/covid/stats	Obtain COVID statistics from the world		{ "confirmed": 0, "newConfirmed": 0, "deaths": 0, "newDeaths": 0, "recovered": 0, "newRecovered": 0, "active": 0, "newActive": 0, "fatalityRate": 0 }
/api/covid/countries	Obtain all supported countries, which is a union of the supported countries of all external APIs	None	["string"]
/api/cache/stats	Obtain the cache statistics	None	{ "hits": 0, "misses": 0, "stored": 0, "ttl": 0 }

3 Quality assurance

3.1 Overall strategy for testing

For testing, a TDD (test driven development) approach was used. Firstly, the main classes' interfaces were developed, with very minimal or no implementation at all, so that they could be used to develop the tests. After tests were fully implemented, the application was built in order to pass those tests.

The only exception to this rule were the functional tests on the web interface. The web interface was the last thing that was implemented in the application (priority was given to the REST API), so its tests with Cucumber were done later. Due to the simplicity of the web interface, the web app was developed first and the Cucumber tests then analyzed it.

3.2 Unit and integration testing

Unit tests were used for the following classes:

- **Covid19FastestUpdateApi**: external API test. The external source is mocked with a MockWebServer
- **JohnsHopkinsApi**: *ditto*
- **CovidCache**: the cache doesn't depend on any other component, and its functionality can be tested in isolation
- **CovidCacheController**: the CovidApi dependency is mocked (through @MockBean), mainly tests receiving requests and sending responses
- **CovidController**: *ditto*

As integrations tests, the application has:

- **Covid IT**: tests for functionality of the whole application (excluding the web app), with mocking of the external API beans
- **CovidController IT**: test exceptions that are mapped to responses with HTTP status codes (subclasses of ResponseStatusException). This is an integration test since it requires that the Spring Boot application be initialized in order to have proper exception handling
- **CovidApi IT**: test the functionality of the CovidApi **in tandem** with the CovidCache, with the external API beans mocked. Proper CovidApi testing requires the CovidCache as well due to the convenience method that was introduced to the cache, which has a functional interface as an argument:

```
public ResponseData getOrStore(ApiQuery apiQuery,
Function<ApiQuery, ResponseData> responseProvider) {
    if (stale(apiQuery)) {
        misses++;
        ResponseData response = responseProvider.apply(apiQuery);
        store(apiQuery, response);
        return response;
    }
    hits++;
    return get(apiQuery);
}
```

This is the method that should be called by users of the cache, since it's the one that updates the cache's stats. Callers provide a "responseProvider" that fetches the response in case it isn't cached or is expired. However, in testing, this means that the CovidApi's main method, "getStats", which returns the stats for a given query, is dependent on the implementation of this cache's method, since effectively the fetching of a new response is only applied within the cache's method. Therefore, what was once a CovidApi unit test, ended up being an integration test, and CovidCache went from being a Mock to being a Spy, which allowed analyzing the number of calls done to its methods (with Mockito) while maintaining its original implementation.

During the test driven development approach, one problem worth mentioning that ended up positively influencing the design was the introduction of the Null Object pattern presented in the System architecture section. Below is the final version of the method that presented the problem ("getStats" from CovidApi):

```
private Stats queryApis(ApiQuery query) {
    Stats response = new NullStats();
    int initialApiIdx = chosenApiIdx;
    if (!supportedApis.isEmpty())
        do {
            Api chosenApi = supportedApis.get(chosenApiIdx);
            try {
                response = chosenApi.getStats(query);
            } catch (UnavailableExternalApiException ex) {
                log.debug("Api {} could not fulfill request: {}",
chosenApi, ex.getMessage());
                chosenApiIdx = (++chosenApiIdx) %
supportedApis.size();
            }
        } while (response.isNull() && initialApiIdx != chosenApiIdx);

    if (response.isNull()) {
        log.error("No external API could fulfill the request: {}",
query);
        throw new UnavailableApiException();
    }

    return response;
}
```

Before the NullStats class was introduced, this method used the "null" value in its place. Instead of `response.isNull()` calls, checks for null `response == null` were made before. The "null" value ideally had the meaning of "inexistent response", but when mocking was used on the external API implementations, the default result returned from `response = chosenApi.getStats(query)` was "null" by default. In this case, "null" had both the meaning of "inexistent response" and "unimplemented/unmocked method", which was confusing and interfered with the method's logic. Therefore, the NullStats class was introduced to explicitly define the desired meaning, which is generally a good practice, to avoid the ambiguous meaning of "null".

The tests also propelled the definition of application properties to easily configure the application in a desired way (for example, enabling/disabling external API implementations or disabling the auto fetching of supported countries on instantiation).

It's worth noting that, for all tests, the default properties specify that the external APIs do **not** automatically fetch the list of countries they support when they are initialized, since tests should not be dependent on external services. Below is the full default test configuration.

application-test.properties

```
rapid-api.key = ""
api.auto-fetch-countries = false
api.covid-fu.enabled = true
api.covid-fu.incomplete-responses = false
api.johns-hopkins.enabled = true
covid-cache.ttl = 60
covid-cache.max-size = 6000
```

3.3 Functional testing

The Functional tests on the web interface used Cucumber with feature files. Two features were considered, one for each type of statistical data (COVID and cache). For the COVID stats feature, one scenario was developed for obtaining world data, and another for obtaining country specific data. Below are the feature files:

```
Feature: Check Covid-19 stats

 Scenario: Obtain world data
   Given I am in the Home page
   When I check to obtain world data
   And I choose the date 2021-01-01
   And I click 'Submit' under the covid section
   Then I should receive covid stats from the 'world'
   And the date 'at' field should be 2021-01-01
   And no other date field should appear


 Scenario: Obtain country data
   Given I am in the Home page
   When I uncheck to obtain world data
   * I choose stats after 2021-12-12
   * I choose the country 'Portugal'
   * I click 'Submit' under the covid section
   Then I should receive covid stats from 'Portugal'
   And the date 'after' field should be 2021-12-12
   And no other date field should appear
```

```
Feature: Check API cache stats

 Scenario: Obtain cache stats
   Given I am in the Home page
   When I click 'Submit' under the cache section
   Then I should receive cache stats
```

The scenario writing was done in a dynamic but rigorous way through “ParameterTypes” in order to allow future additions to be easily made (for example, dynamic specification of the data section, which can be “covid” or “cache”) and provide feedback of when a written action is, while syntactically correct, not “semantically implemented” (for example, specification of the starting page, which only supports the home page).

For the step definitions, the Page Object pattern was used, in order to easily write them in a maintainable way. In total, 3 concrete Page Objects were developed. The code used in these page objects was extracted from a project created on Selenium IDE, which replicated the steps detailed in the feature files (the project is available in the file “TQS-HW1.side”).

For the WebDriver, the HtmlUnitDriver was used due to its simple usage. JavaScript had to be enabled for the HtmlUnitDriver, since some visual feedback logic is implemented in the home page.

One of the challenges faced while preparing Cucumber tests for a Spring Boot environment was running these tests in parallel with the whole Spring Boot Web application, **while** some dependencies had to be configured in order to have predictable test behavior. After many hiccups, the following setup had to be applied:

- Creation of a Spring Configuration class specifically for the Cucumber tests, which needed the @SpringBootTest and @CucumberContextConfiguration annotations. The web environment needed to have a defined port, so that the driver in the CucumberSteps step definition class could use it to access the web application (knowing the value of a random port in the CucumberSteps class proved to be extremely difficult, if not impossible).
  The external APIs were disabled, since the cache was used as the holder of the responses.

```
@CucumberContextConfiguration
@SpringBootTest(
      webEnvironment =
      SpringBootTest.WebEnvironment.DEFINED_PORT,
      properties = {
              "api.covid-fu.enabled=false",
              "api.johns-hopkins.enabled=false"
      })
@TestPropertySource("/application-test.properties")
public class CucumberSpringConfiguration {}
```

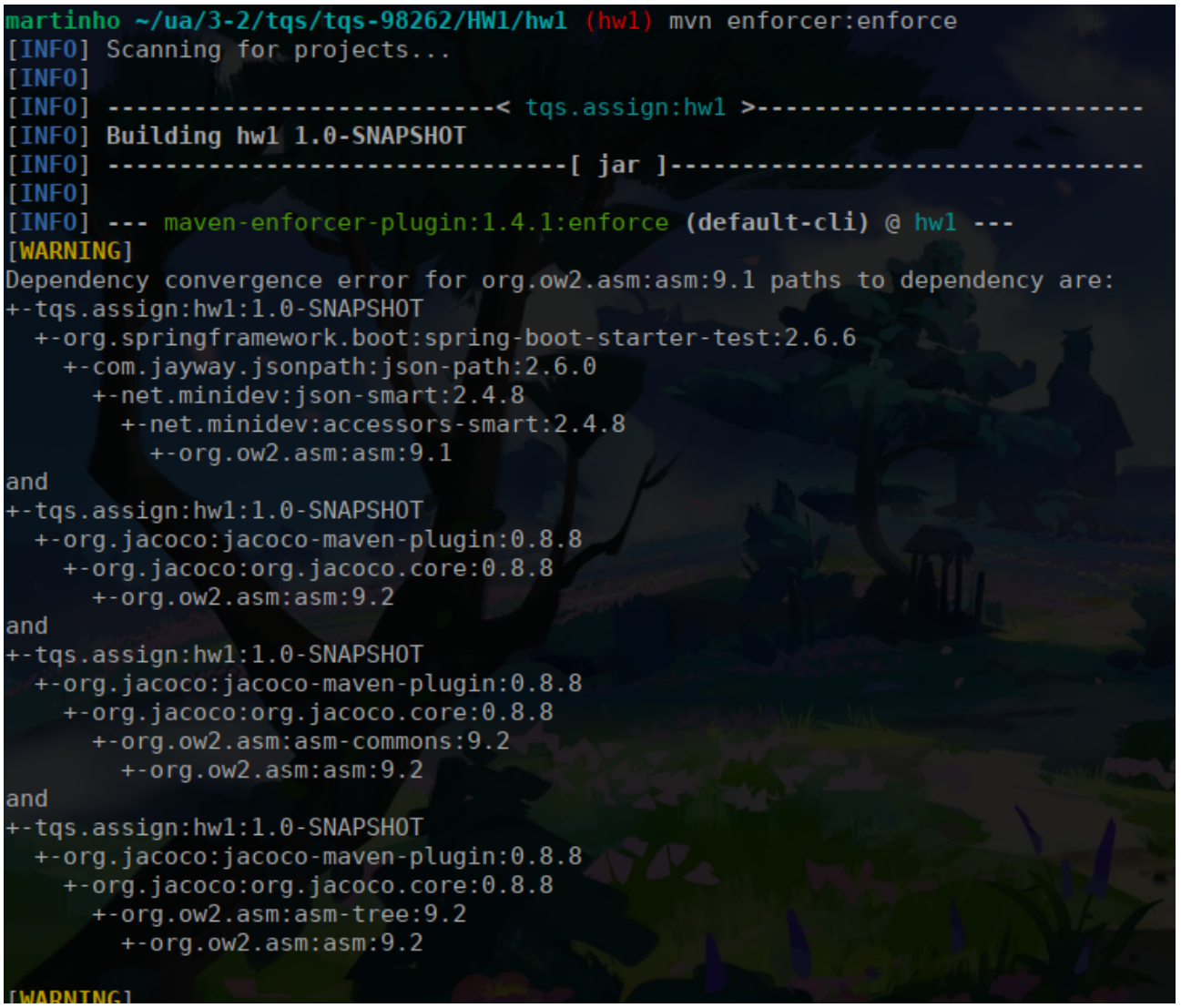
- Addition of the @ContextConfiguration annotation to the CucumberSteps step definition class, so that Spring beans could be injected. The @Component annotation can’t be used, since the steps definition class can’t be a Spring-managed bean, as the Cucumber

framework manually manages the instantiation of this class. The autowired dependencies were the CovidApi and the CovidCache classes, to prepare responses for requests and populate the list of supported countries. This way, the API that the CucumberTests will run with is predictable.

```
@ContextConfiguration
public class CucumberSteps {

    // Javascript is used to enable/disable the countries dropdown
    private final WebDriver driver = new HtmlUnitDriver(true);

    private WebPage currentPage;

    @Autowired
    public CucumberSteps(CovidApi covidApi, CovidCache
covidCache)
```

One of the problems faced at this stage that deserves mention is conflict of dependencies with differing versions. The exceptions thrown were very specific and of generic Java classes (with no online resources mentioning them), which pointed to this being the cause.

To analyze dependency conflicts, the Maven Enforcer goal was run. Below is an example output:

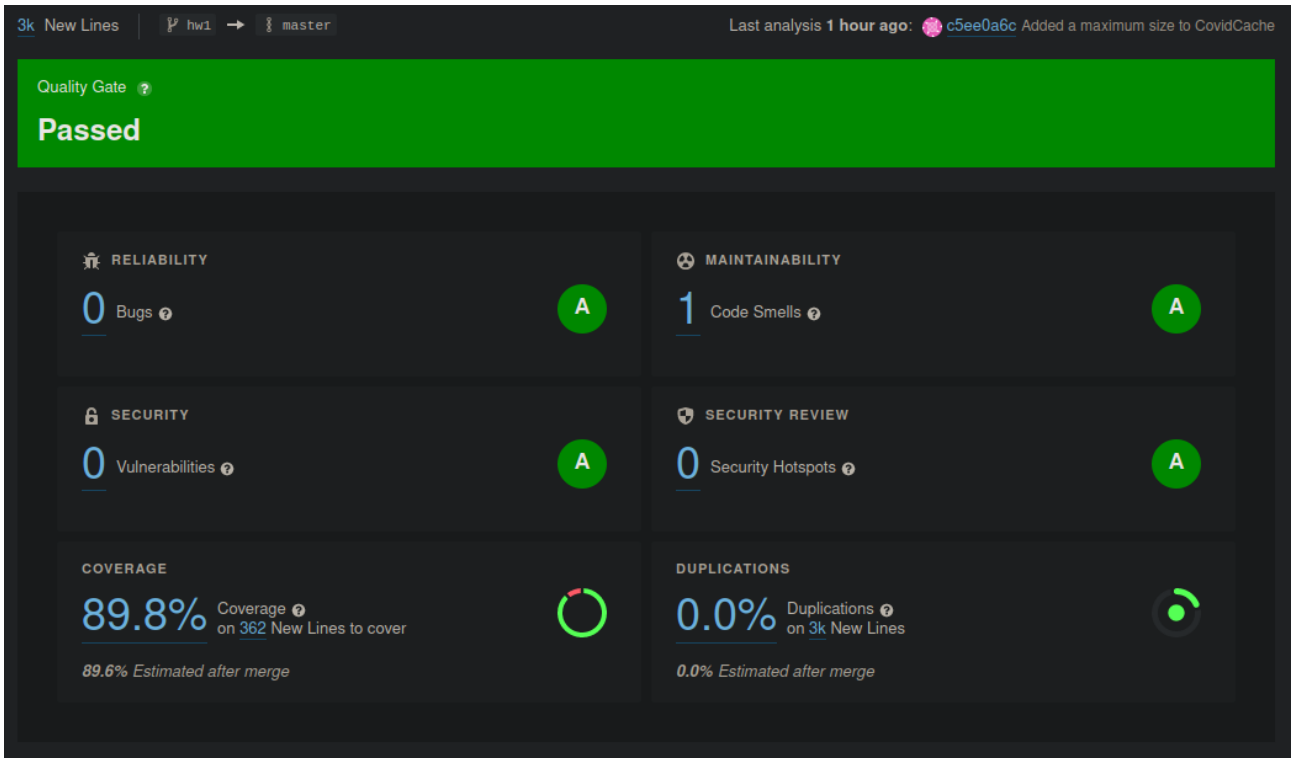


Some dependencies, related to selenium and okhttp, used differing versions of the okio dependency, which was suspected to be the problem. After downgrading some of the dependencies on the pom file, essentially only JaCoCo related problems were reported by the Enforcer goal.

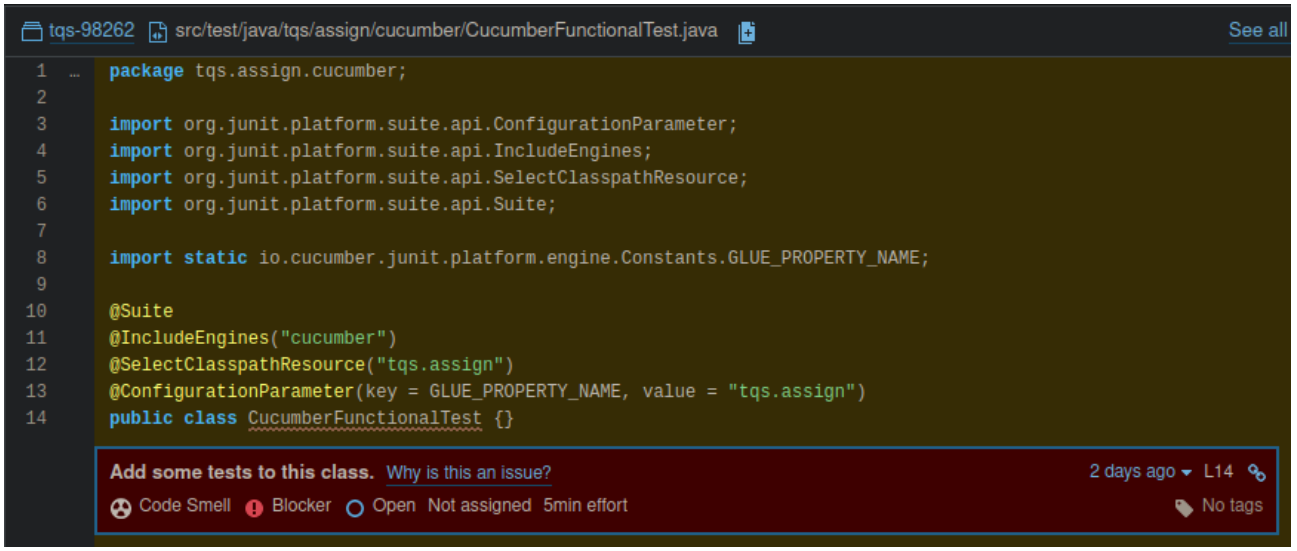
3.4 Code quality analysis

For static code analysis, the Codacy tool was considered to be included in the CI pipeline (explained in the next section). The local text output that it provides by default on GitHub is cumbersome to analyze, so a cloud dashboard was sought after. However, its cloud capabilities were seemingly more aimed towards organizations, so the SonarCloud tool was used instead.

Below is the final report of the SonarCloud dashboard:



The single code smell that was noted by the platform isn't actually something to consider, as it is a product of how Cucumber testing with JUnit 5 is laid out. The platform is asking to include test methods in the "CucumberFunctionalTest" class, but considering how Cucumber works, that shouldn't be the case:



It's worth noting that the code coverage could not have this value if not for the "lombok.config" file that was added to the root of the project, which prevents SonarCloud from analyzing the coverage on code added by Lombok's annotations, which are simply used to remove boilerplate code.

```
config.stopBubbling = true
lombok.addLombokGeneratedAnnotation = true
```

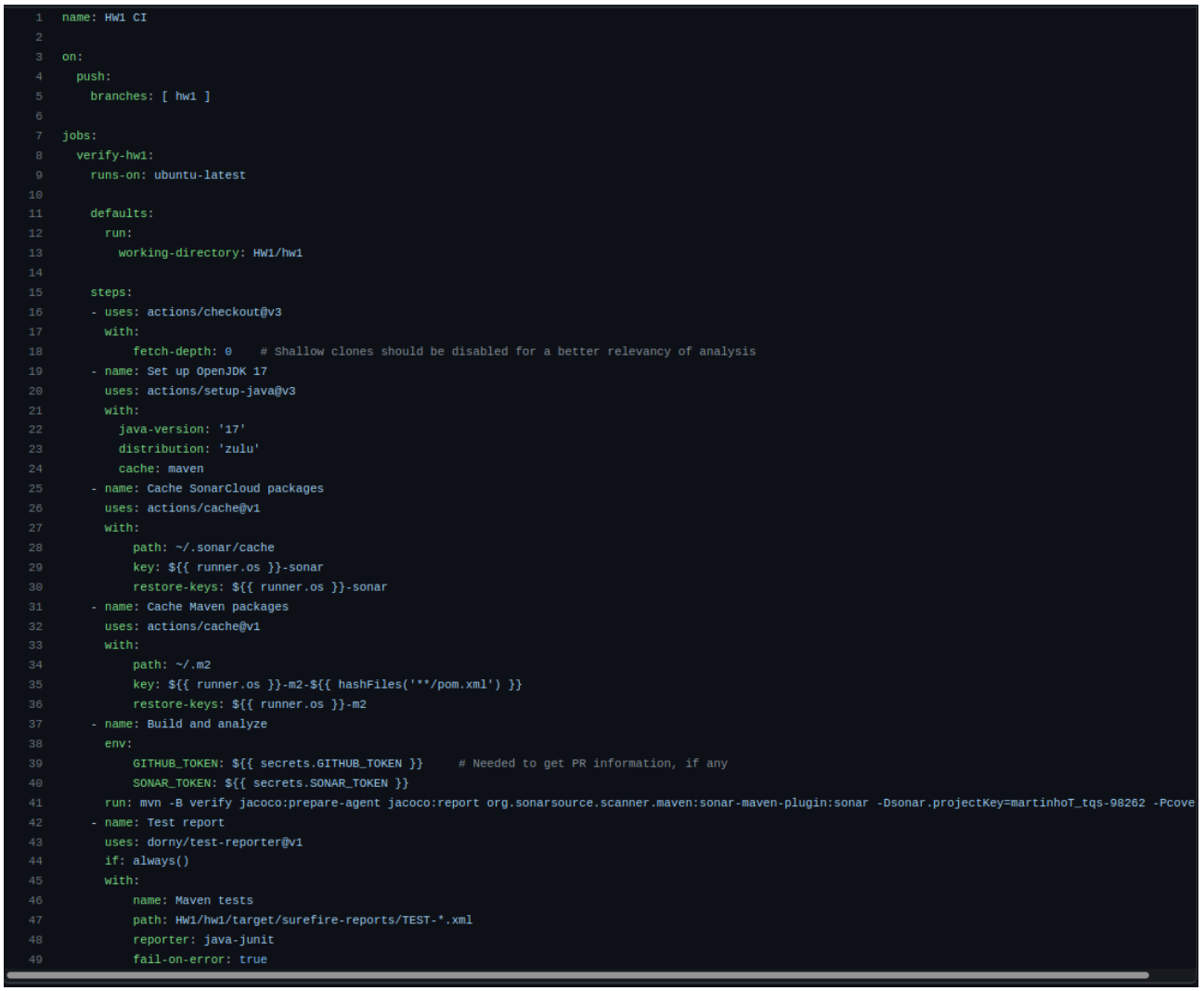
The risk graph is shown below:

The class with the lowest code coverage value was “Utils”, which essentially built a Gson instance for JSON serialization/deserialization. This class defined TypeAdapters for the Gson instance to convert date strings into their respective Date classes, and SonarCloud expects these conversions, and all of their possible conditions, to be included in the tests:

```
24      private static class LocalDateTimeAdapter extends TypeAdapter<LocalDateTime> {
25
26          @Override
27          public void write(JsonWriter jsonWriter, LocalDateTime localDateTime) throws IOException {
28              if (localDateTime == null) {
29                  jsonWriter.nullValue();
30                  return;
31              }
32              jsonWriter.value( localDateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME) );
33          }
34
35          @Override
36          public LocalDateTime read(JsonReader jsonReader) throws IOException {
37              if (jsonReader.peek() == JsonToken.NULL) {
38                  jsonReader.nextNull();
39                  return null;
40              }
41              return LocalDateTime.parse(jsonReader.nextString(), DateTimeFormatter.ISO_LOCAL_DATE_TIME);
42          }
43
44      }
```

3.5 Continuous integration pipeline

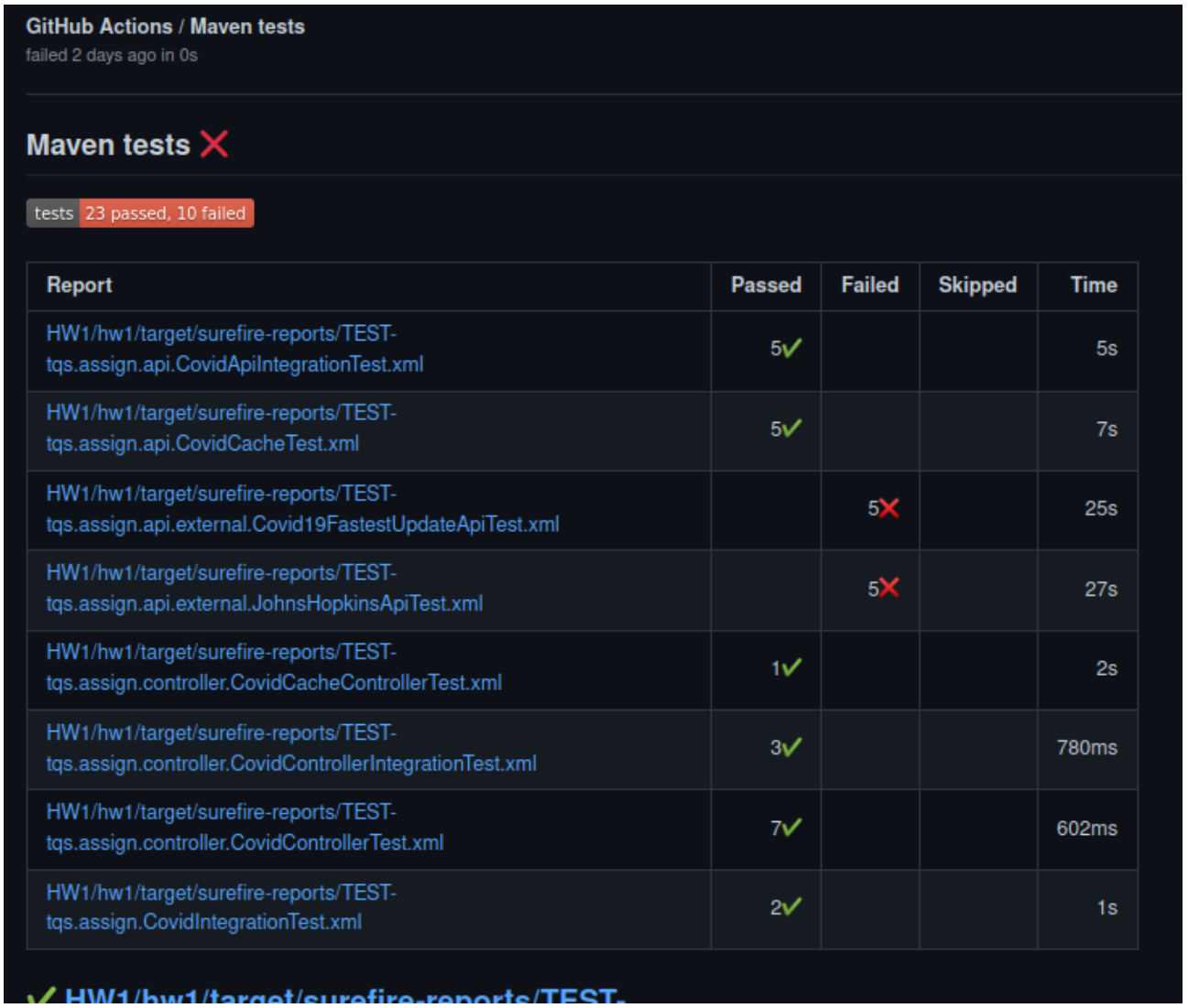
Since the project repository is located on GitHub, the Continuous Integration framework used was GitHub Actions for simplicity. Below is the final version of the defined workflow file:



```yaml
name: HW1 CI

on:
  push:
    branches: [ hw1 ]

jobs:
  verify-hw1:
    runs-on: ubuntu-latest

    defaults:
      run:
        working-directory: HW1/hw1

    steps:
    - uses: actions/checkout@v3
      with:
          fetch-depth: 0    # Shallow clones should be disabled for a better relevancy of analysis
    - name: Set up OpenJDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'zulu'
        cache: maven
    - name: Cache SonarCloud packages
      uses: actions/cache@v1
      with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
    - name: Cache Maven packages
      uses: actions/cache@v1
      with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
    - name: Build and analyze
      env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}     # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
      run: mvn -B verify jacoco:prepare-agent jacoco:report org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=martinhoT_tqs-98262 -Pcove
    - name: Test report
      uses: dorny/test-reporter@v1
      if: always()
      with:
          name: Maven tests
          path: HW1/hw1/target/surefire-reports/TEST-*.xml
          reporter: java-junit
          fail-on-error: true
```

A single job was defined, which verifies the Maven Java 17 project when pushes to the homework branch (hw1) are done. SonarCloud integration is present here, which will automatically update the project's dashboard online with the report results. Since code coverage was also desired for these reports, the "Build and analyze" step includes the required JaCoCo goals.

For convenience, a "Test reporter" step was added which uses the "dorny/test-reporter@v1" action. This step collects the test reports generated by the "mvn verify" command previously run and neatly presents them directly on GitHub for easy visualization. An example portion of this report is shown below:



GitHub Actions / Maven tests

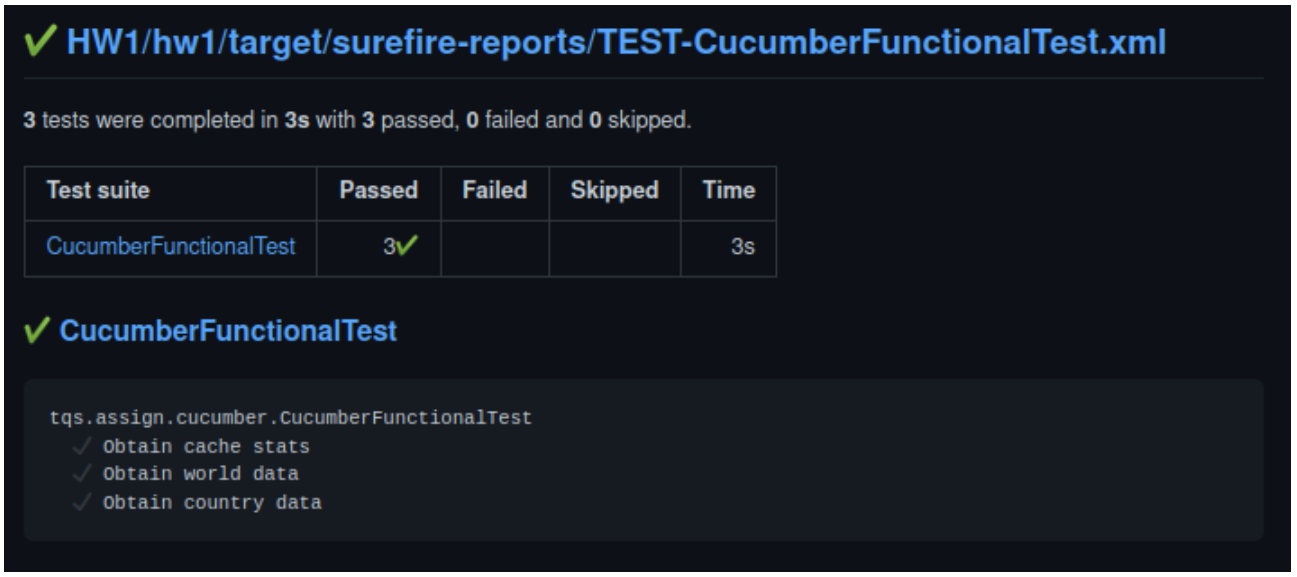
failed 2 days ago in 0s

Maven tests ❌

tests 23 passed, 10 failed

Report	Passed	Failed	Skipped	Time
HW1/hw1/target/surefire-reports/TEST-tqs.assign.api.CovidApiIntegrationTest.xml	5✅			5s
HW1/hw1/target/surefire-reports/TEST-tqs.assign.api.CovidCacheTest.xml	5✅			7s
HW1/hw1/target/surefire-reports/TEST-tqs.assign.api.external.Covid19FastestUpdateApiTest.xml		5❌		25s
HW1/hw1/target/surefire-reports/TEST-tqs.assign.api.external.JohnsHopkinsApiTest.xml		5❌		27s
HW1/hw1/target/surefire-reports/TEST-tqs.assign.controller.CovidCacheControllerTest.xml	1✅			2s
HW1/hw1/target/surefire-reports/TEST-tqs.assign.controller.CovidControllerIntegrationTest.xml	3✅			780ms
HW1/hw1/target/surefire-reports/TEST-tqs.assign.controller.CovidControllerTest.xml	7✅			602ms
HW1/hw1/target/surefire-reports/TEST-tqs.assign.CovidIntegrationTest.xml	2✅			1s

✅ HW1/hw1/target/surefire-reports/TEST-



✅ HW1/hw1/target/surefire-reports/TEST-CucumberFunctionalTest.xml

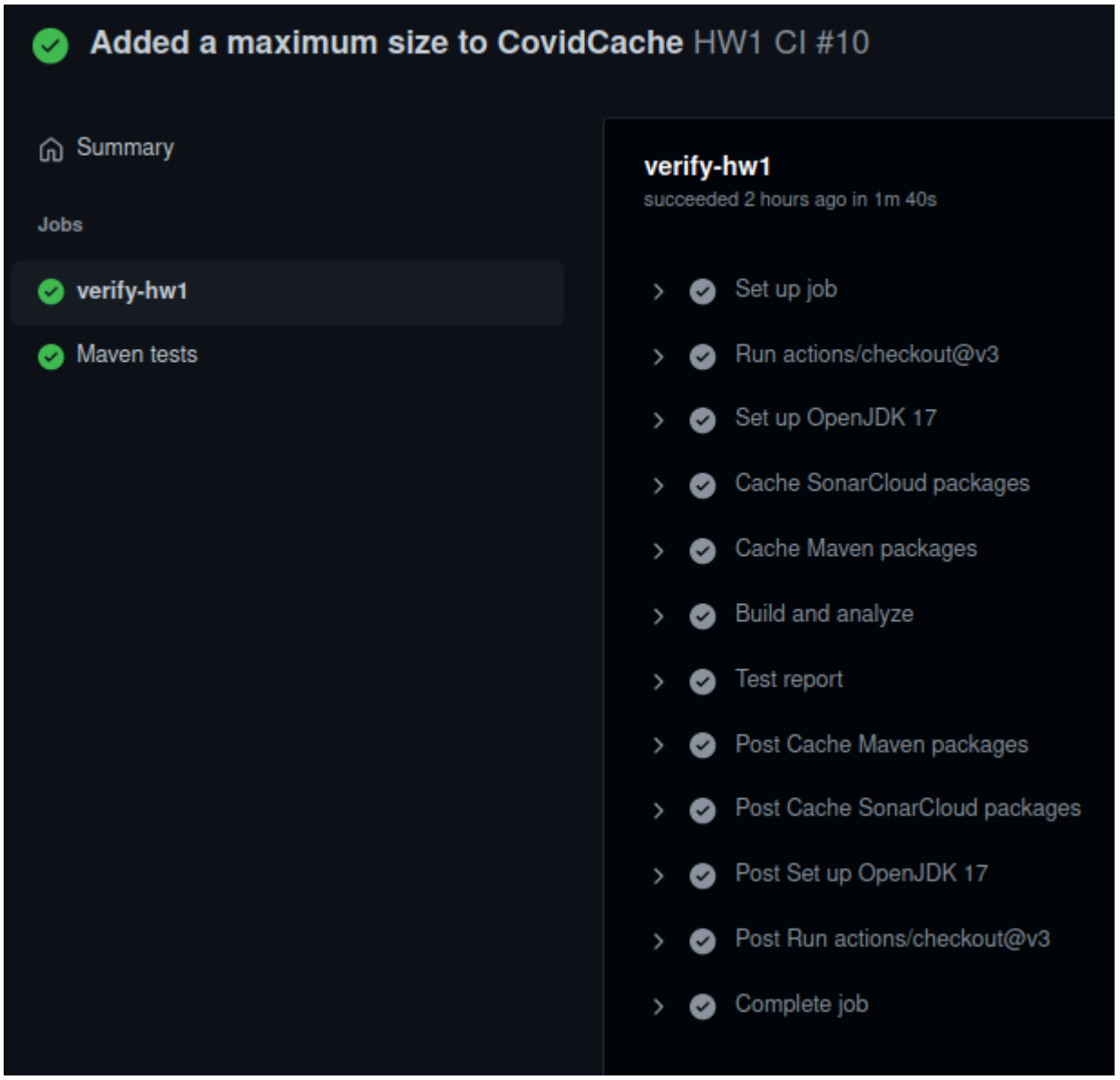
3 tests were completed in **3s** with **3** passed, **0** failed and **0** skipped.

Test suite	Passed	Failed	Skipped	Time
CucumberFunctionalTest	3✅			3s

✅ CucumberFunctionalTest

```
tqs.assign.cucumber.CucumberFunctionalTest
  ✓ Obtain cache stats
  ✓ Obtain world data
  ✓ Obtain country data
```

Below is the example of a CI run on a push to the “hw1” branch:



4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/martinhoT/tqs-98262
Video demo	https://www.youtube.com/watch?v=T__9HZteREA
QA dashboard (online)	https://sonarcloud.io/summary/new_code?id=martinhoT_tqs-98262&branch=hw1
CI/CD pipeline	https://github.com/martinhoT/tqs-98262/blob/master/.github/workflows/hw1.yml
Deployment ready to use	-

Reference materials

During the development of the application, the following resources were consulted and proved to be very useful:

GitHub Actions setup:

- Learn GitHub Actions
- GitHub Marketplace · Actions to improve your workflow

Spring Web exception handling:

- Error Handling for REST with Spring | Baeldung
- Spring ResponseStatusException | Baeldung
- Exception Handling in Spring MVC
- Using Spring @ResponseStatus to Set HTTP Status Code

Dependency conflicts:

- Solving Dependency Conflicts in Maven - DZone Java

Spring Boot properties:

- Properties with Spring and Spring Boot | Baeldung
- Override Properties in Spring's Tests | Baeldung

Logging:

- Logging in Spring Boot | Baeldung

Cucumber with Spring:

- Cucumber Spring Integration | Baeldung
- cucumber-jvm/spring at main · cucumber/cucumber-jvm