DEPENDABILITY ANALYSIS OF A JAVA OPEN-SOURCE PROJECT

commons-dbutils

10 OF JANUARY OF 2025

SOFTWARE DEPENDABILITY

Alejandro Jesús Medina Fernández | Pablo Acinas Suárez

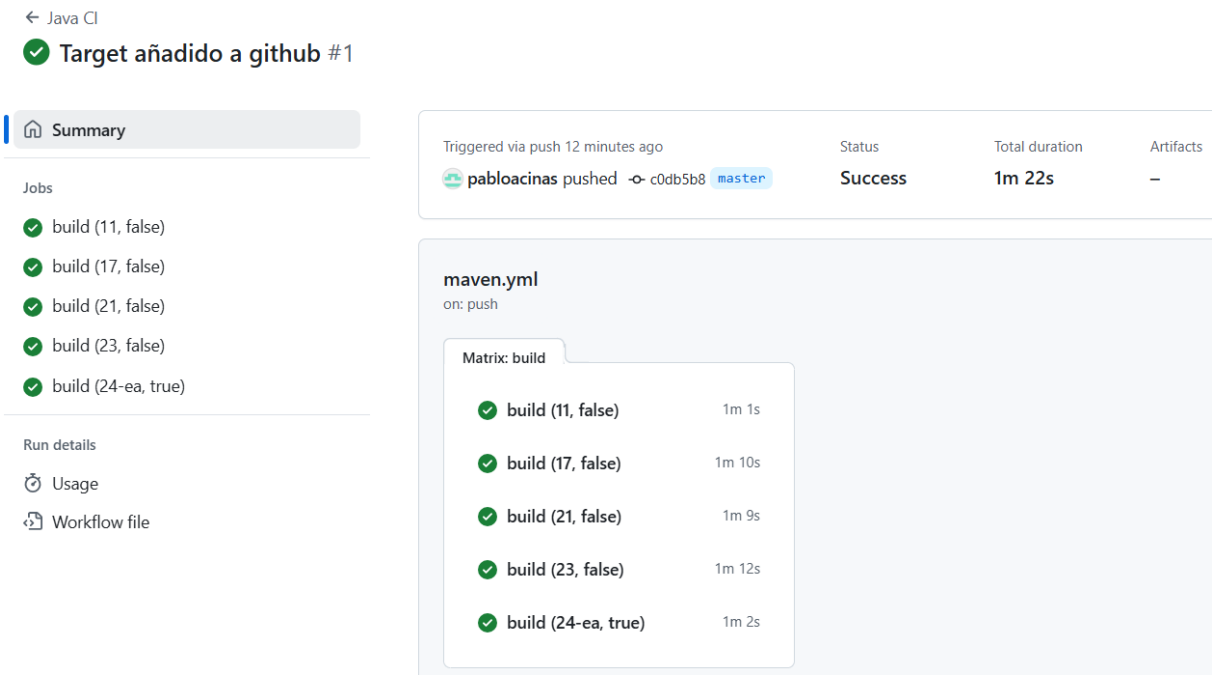
INDEX

This project focuses on analyzing the **Commons DbUtils** library, a part of the Apache Commons family. Commons DbUtils is a Java library designed to simplify the use of JDBC (Java Database Connectivity) by providing utility classes to handle database connections, queries, and result set processing. Its primary goal is to reduce the boilerplate code involved in database operations, improving code readability and maintainability.

Project Build

i. Verification of successful build in CI/CD and local environments.

The first part of the project was forking the original project in our own github repository. It was very to ensure the inclusion of CI/CD. To achieve this, we utilized **GitHub Actions**, taking advantage of the existing YAML files that were already created in the original project. At first, in the forked repository we had to enable workflows in order to use the original ones. The maven.yaml is configured to build the project every time a push occurs, so the first we did was a push to ensure it runs all the jobs without problems.



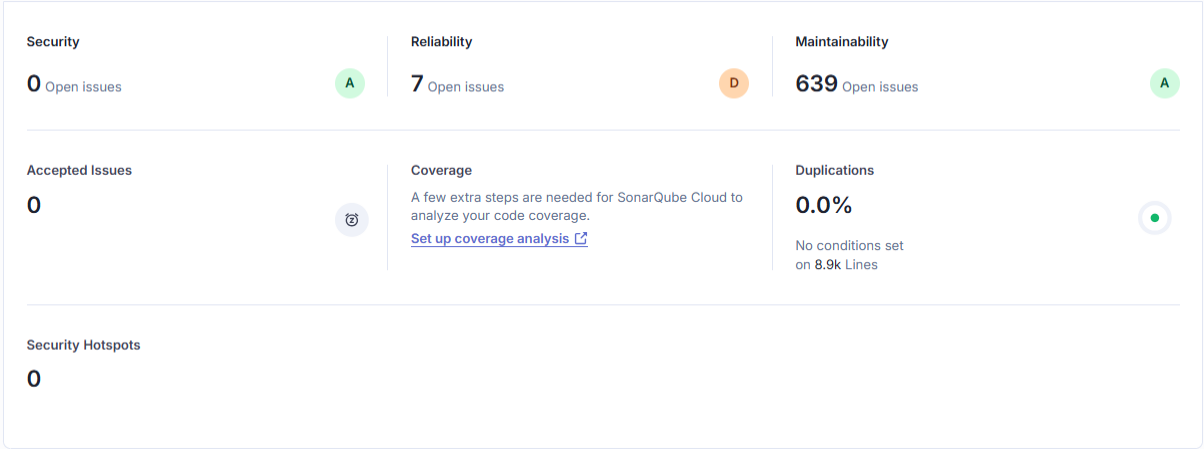
In the **local machine** we also had to ensure the project is built correctly with maven. We added the project from github to the local machine using git clone. Then we use the command “maven clean install” to run all the test cases and compile the project.

```
[INFO] ------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------
[INFO] Total time:  19.321 s
[INFO] Finished at: 2025-01-04T11:47:02+01:00
[INFO] ------------------------------------------
PS C:\Users\Usuario\git\commons-dbutils>
```

Software Quality Analysis

i. Comprehensive evaluation using SonarCloud.

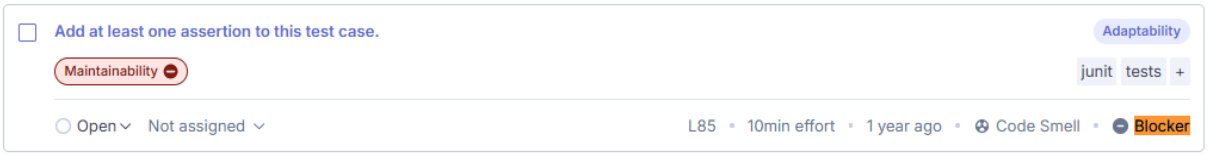
We used SonarCloud, which is a cloud-based service for continuous code quality inspection. It helps in identifying bugs, vulnerabilities, and code smells by analyzing the codebase. After the analysis we saw that our project has 639 maintainability issues. We will focus on the blocker and critical ones.



ii. Categorization of issues identified by SonarCloud.

Going deeply into the issues. There are 16 **blocker** issues, 17 **critical** issues, 54 **major** issues,58 **minor** issues **502** info.

The **blocker** issues are all the same:



This happens because if there is no assertion, the test is not verifying nothing

iii. Resolution of issues through refactoring or rationale provided for skipped fixes.

14 of the 16 blocker issues are in **DbUtilsTest.java**
The resolution of the blocker issues in this project is to add an assertion in the methods that don't have. In the L85 blocker issue this was the code before resolution:

```
@Test
public void testCloseNullConnection() throws Exception {
    Add at least one assertion to this test case.
    DbUtils.close((Connection) null);
}
```

To solve the problem we used `assertDoesNotThrow`: This verifies that no exception is thrown when DbUtils.close() receives a null.

```
    @Test
-       public void testCloseNullConnection() throws Exception {
+   public void testCloseNullConnection() throws Exception {
        assertDoesNotThrow(() -> DbUtils.close((Connection) null));
    }
```

We applied the same resolution to all issues in this file.

In the **QueryRunnerTest.java** class we have the same issue:

```
    @Test
    public void testFillStatementWithBean() throws Exception {
```

Add at least one assertion to this test case.

```
        final MyBean bean = new MyBean();
        when(meta.getParameterCount()).thenReturn(3);
        runner.fillStatementWithBean(prepStmt, bean, "a", "b", "c");
    }
```

We solved the problem using an assertion which checks that the number of parameters used is 3.

```
@Test
public void testFillStatementWithBean() throws Exception {
    final MyBean bean = new MyBean();
    when(meta.getParameterCount()).thenReturn(3);
    runner.fillStatementWithBean(prepStmt, bean, "a", "b", "c");
    assertEquals(3, meta.getParameterCount());
}
```

In the **ResultSetIteratorTest.java** class we have the same issue:

```
    @Test
    public void testCreatesResultSetIteratorTakingThreeArgumentsAndCallsRemove() {
```

Add at least one assertion to this test case.

```
        final ResultSet resultSet = mock(ResultSet.class);
        final ResultSetIterator resultSetIterator = new ResultSetIterator(
resultSet, null);
        resultSetIterator.remove();

    }
```

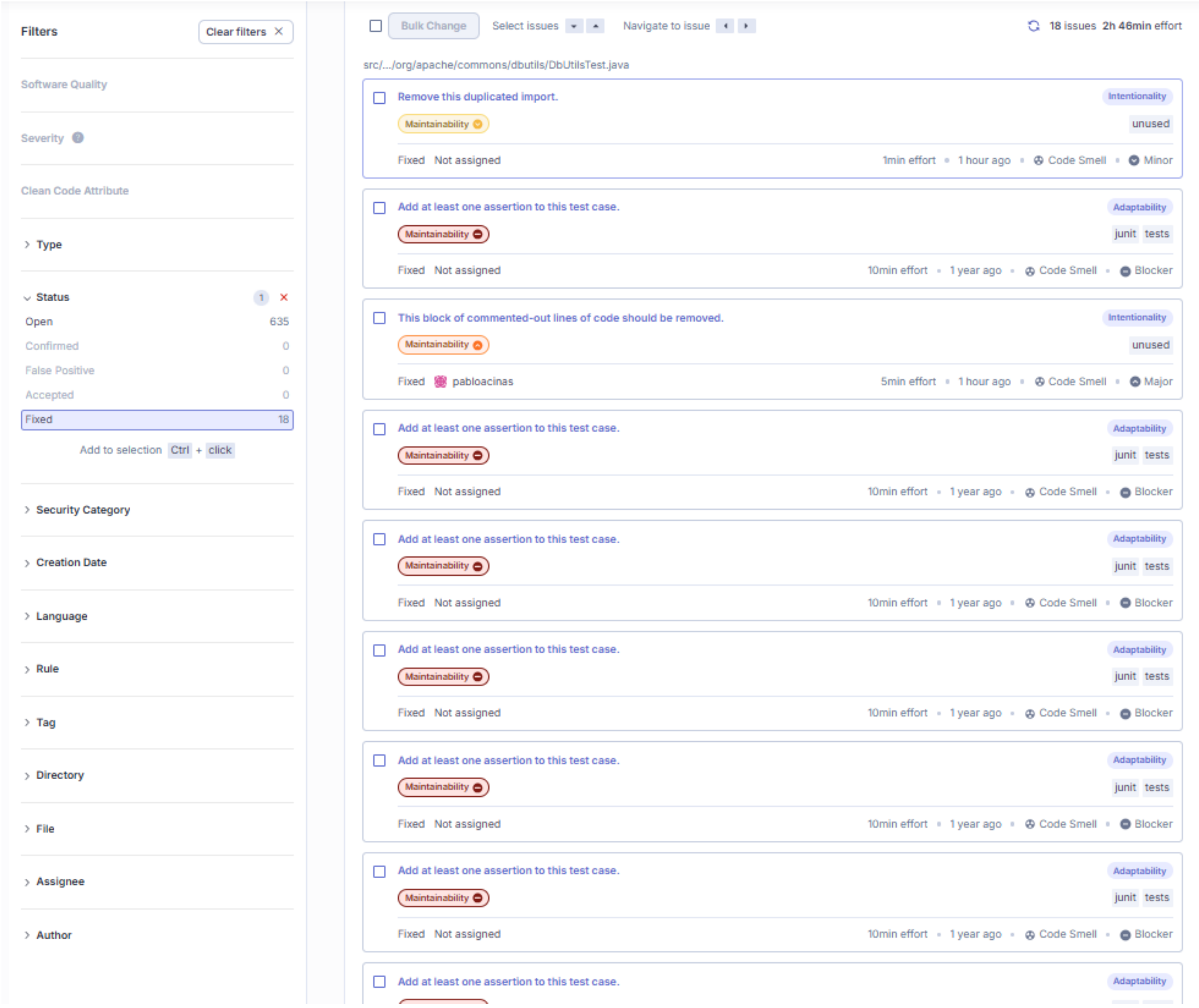
We solved the problem using an assertion that checks that the method `resultSetIterator.remove` worked without errors, if not, it throws an exception

```
  @Test
  public void testCreatesResultSetIteratorTakingThreeArgumentsAndCallsRemove() {

      final ResultSet resultSet = mock(ResultSet.class);
      final ResultSetIterator resultSetIterator = new ResultSetIterator(resultSet, null);
      resultSetIterator.remove();
      try {
          resultSetIterator.remove();
          assertTrue(true);
      } catch (Exception e) {
      fail("remove() method threw an exception: " + e.getMessage());
  }

  }
```

We checked that in the SonarCloud all the issues we solve are fixed



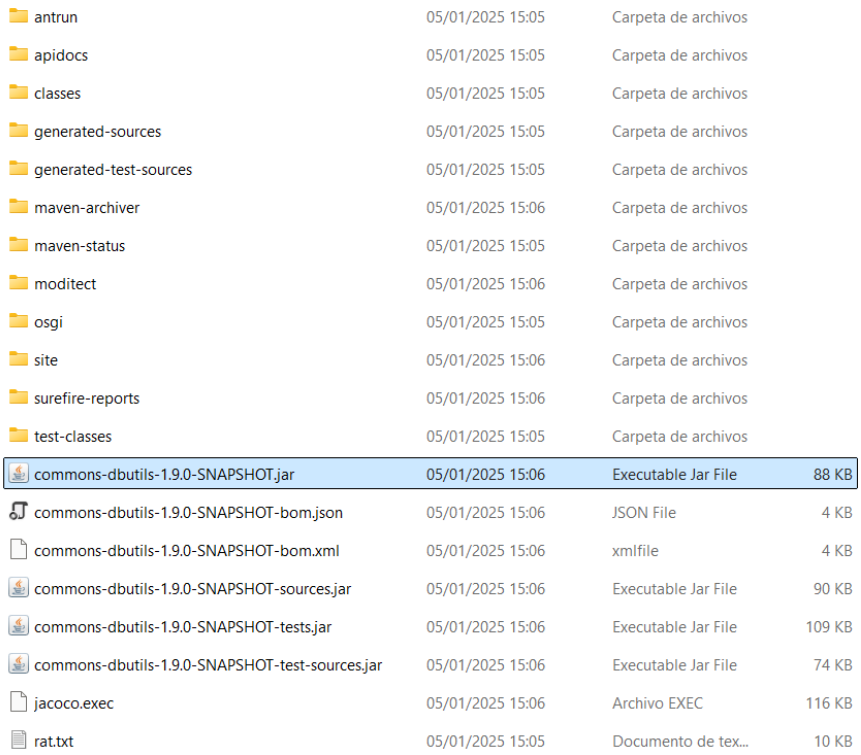
<u>Project Containerization</u>

i. Creation of our application

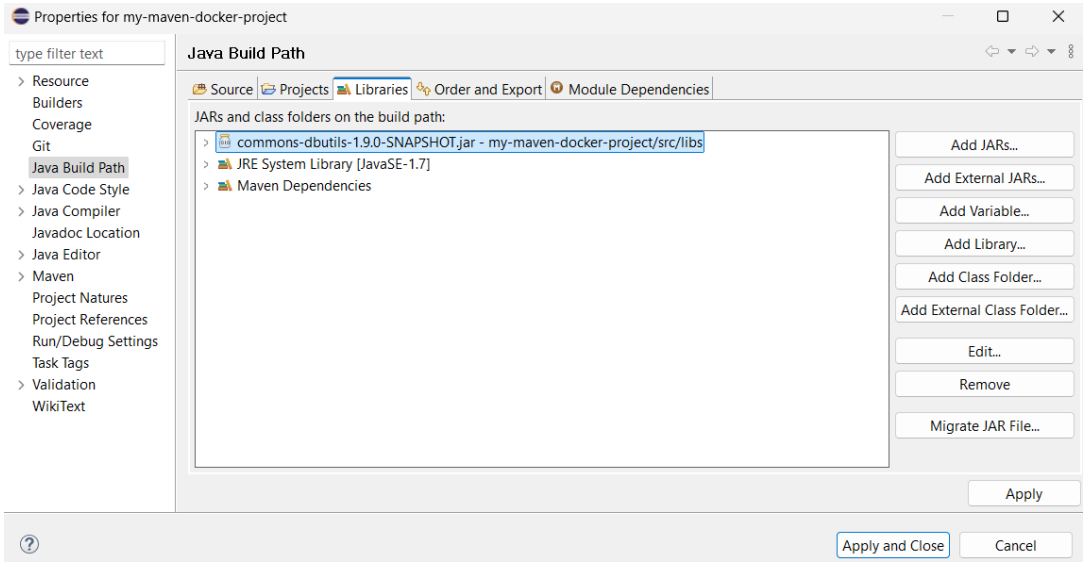
At this point, we have decided to create an **external application** that imports the .jar generated when we packaged our commons-dbutils project. The goal of this external application is to test that after the modifications made in the previous point it still works correctly.

The first part of this process is to use the command mvn clean package in order to execute the entire build process, including compiling the source code, running tests and packaging the application into a JAR file.

We will use commons-dbutils-1.9.0-SNAPSHOT.jar



After that we've created a new Maven Java Application and added the .jar created previously



We created a little class which connects to a local database called prueba.db and uses the classes `org.apache.commons.dbutils.QueryRunner` and `org.apache.commons.dbutils.DbUtils`

```
package com.mycompany.app;

import org.apache.commons.dbutils.DbUtils;
import org.apache.commons.dbutils.QueryRunner;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class App {

    // Ruta de la base de datos
    private static final String URL = "jdbc:sqlite:C:/Users/Usuario/Desktop/SOFTWARE_DEPENDABILITY/BD/prueba.db";

    // Método para conectar a la base de datos
    public static Connection conectar() {
        Connection conn = null;
        try {
            // Estableciendo la conexión
            conn = DriverManager.getConnection(URL);
            System.out.println("Conexión establecida.");
        } catch (SQLException e) {
            System.out.println("Error al conectar a la base de datos: " + e.getMessage());
        }
        return conn;
    }

    public static void main(String[] args) {
        // Probar la conexión
        Connection conn = conectar();
        if (conn != null) {
            // Insertar usuarios en la tabla 'usuarios'
            try {
                QueryRunner runner = new QueryRunner();
                String sql = "INSERT INTO usuarios (Nombre, Apellidos) VALUES (?, ?)";

                // Lista de usuarios para insertar
                Object[][] usuarios = {
                    {"Juan", "Pérez"},
                    {"María", "López"},
                    {"Carlos", "González"}
                };

                runner.batch(conn, sql, usuarios);
                System.out.println("Usuarios insertados exitosamente.");
            } catch (SQLException e) {
                System.out.println("Error al insertar usuarios: " + e.getMessage());
            } finally {
                // Cerrar la conexión
                DbUtils.closeQuietly(conn);
                System.out.println("Conexión cerrada.");
            }
        }
```

Here we can see that it works correctly

```
Conexión establecida.
Usuarios insertados exitosamente.
Conexión cerrada.
```

	Nombre	Apellidos
	Filtro	Filtro
1	Juan	Pérez
2	María	López
3	Carlos	González

ii. Dockerizing our application

Then, the next step is to create a Dockerfile , which is a text/script configuration file that contains collections of commands that will be automatically executed, in sequence, in the Docker environment to build a new Docker image.

```
FROM openjdk:17

WORKDIR /app

COPY target/my-maven-docker-project-1.0-SNAPSHOT.jar app.jar

EXPOSE 8080

CMD ["java", "-jar", "app.jar"]
```

FROM openjdk:17
This specifies the base image for the Docker image. We use **OpenJDK 17** image, which includes the Java Development Kit (JDK) for Java 17.

WORKDIR /app
Sets the working directory inside the Docker container.

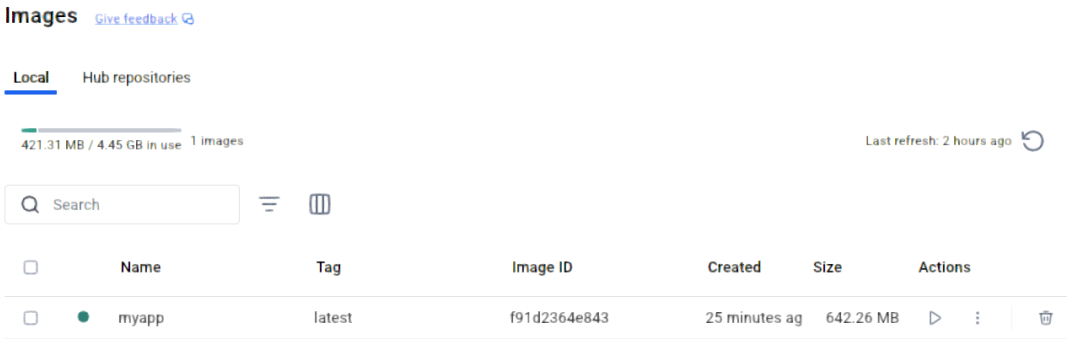
COPY target/commons-dbutils-1.9.0-SNAPSHOT.jar app.jar

The COPY command takes a file from the local machine and places it into the container's file system. It renames commons-dbutils-1.9.0-SNAPSHOT.jar into app.jar

CMD ["java", "-jar", "app.jar"]

Those are the instructions necessaries to run the app

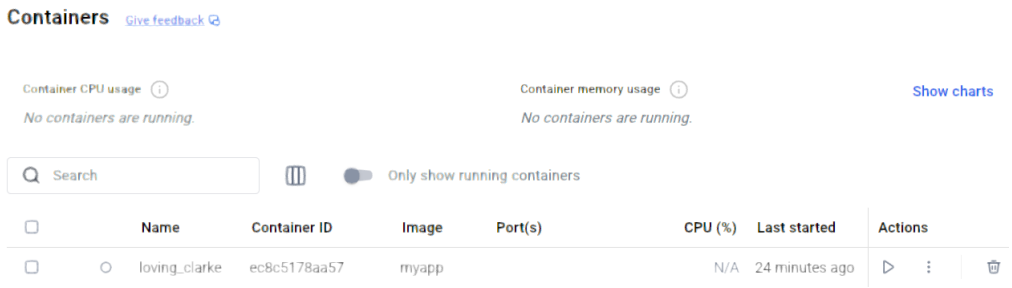
After the creation of the docker file, we build the image with the command `docker build -t myapp .` and we can see it in Docker Desktop



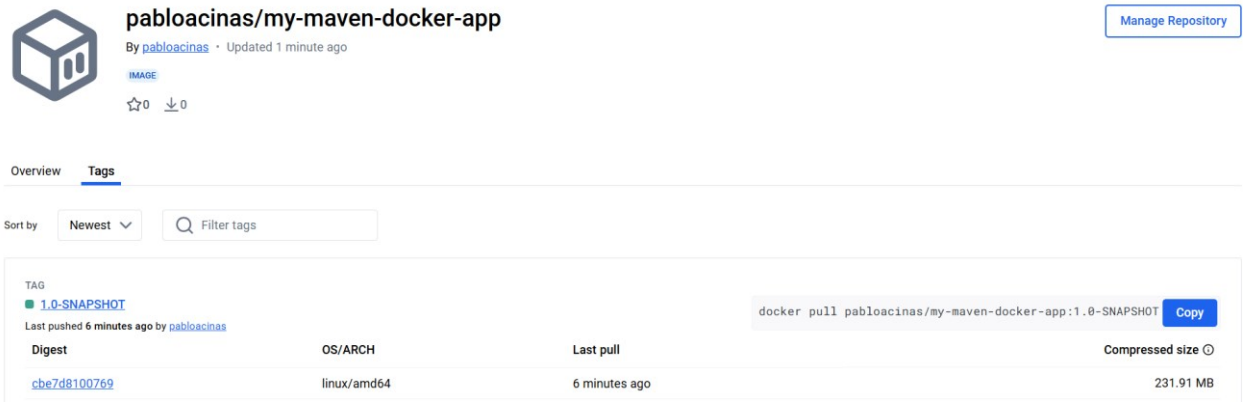
The next step is run the image with the command `docker run myapp`

```
PS C:\Users\Usuario\git\commons-dbutils> docker run myapp
commons-dbutils library is ready for use
```

We can also see that when we run the image a container has been created



At this moment, our application is running locally, the next step is to upload the app to DockerHub. At first, we execute the command `docker login.` After that, we use `docker build -t pabloacinas/my-maven-docker-app:1.0-SNAPSHOT .` to build the docker image associated with an username and then we use `docker push pabloacinas/my-maven-docker-app:1.0-SNAPSHOT` in order to upload the docker image into the DockerHub



Code Coverage

i. Code coverage analysis using JaCoCo.

We used JaCoCo, which is a code coverage tool for Java applications. Its main purpose is to measure how well the code is being tested through unit tests. It provides metrics such as line coverage, branch coverage, and method coverage, helping to identify areas of the code that are not being tested and improving the overall quality and reliability of the software.

We didn't have to modify de pom.xml file of our Java Application because it was already implemented. We use the command `maven jacoco:report` to generate the report of our project.

Here there are the results of the report:

Apache Commons DbUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.dbutils		59%		77%	276	552	468	1.114	225	418	3	19
org.apache.commons.dbutils.handlers		100%		100%	0	58	0	114	0	48	0	12
org.apache.commons.dbutils.wrappers		100%		93%	2	64	0	101	0	49	0	2
org.apache.commons.dbutils.handlers.columns		100%		82%	5	44	0	30	0	30	0	10
org.apache.commons.dbutils.handlers.properties		100%		94%	1	16	0	24	0	6	0	2
Total	1.721 of 5.221	67%	68 of 364	81%	284	734	468	1.383	225	551	3	45

The JaCoCo report shows an overall instruction coverage of 67% and branch coverage of 81%. Out of 5221 instructions, 1721 remain uncovered, and 68 out of 364 branches are missed.

- The main package (org.apache.commons.dbutils) has 59% instruction coverage and 77% branch coverage.
- The package org.apache.commons.dbutils.handlers has a 100% instruction coverage and 100% branch coverage.
- org.apache.commons.dbutils.handlers.wrapples , org.apache.commons.dbutils.handlers.columns and org.apache.commons.dbutils.handlers.properties both have 100% instruction coverage but they have lower branch coverage.
- A total of 1383 lines, 551 methods, and 45 classes are uncovered across the project.

org.apache.commons.dbutils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BaseResultSetHandler		2%		50%	188	193	283	291	187	192	0	1
QueryRunner		78%		82%	19	65	30	188	8	34	0	1
AbstractQueryRunner		79%		76%	20	57	39	165	6	27	0	1
DbUtils		47%		54%	10	31	31	82	5	20	0	1
AsyncQueryRunner.QueryCallableStatement		0%		0%	3	3	19	19	2	2	1	1
BeanProcessor		88%		85%	9	48	15	122	0	17	0	1
AsyncQueryRunner.BatchCallableStatement		0%		0%	3	3	15	15	2	2	1	1
AsyncQueryRunner.UpdateCallableStatement		0%		0%	3	3	15	15	2	2	1	1
AsyncQueryRunner		87%		n/a	5	41	4	32	5	41	0	1
ResultSetIterator		66%		100%	2	10	9	21	2	8	0	1
StatementConfiguration		82%		87%	3	22	2	23	1	14	0	1
OutParameter		76%		100%	3	10	3	21	3	9	0	1
BasicRowProcessor		94%		66%	4	15	1	28	1	9	0	1
QueryLoader		95%		83%	1	9	1	22	0	6	0	1
DbUtils.DriverProxy		89%		n/a	1	8	1	10	1	8	0	1
StatementConfiguration.Builder		98%		50%	1	9	0	14	0	8	0	1
GenerousBeanProcessor		100%		91%	1	8	0	19	0	2	0	1
BasicRowProcessor.CaseInsensitiveHashMap		100%		n/a	0	6	0	15	0	6	0	1
ProxyFactory		100%		n/a	0	11	0	12	0	11	0	1
Total	1.721 of 4.245	59%	60 of 268	77%	276	552	468	1.114	225	418	3	19

The JaCoCo report for the org.apache.commons.dbutil package shows an overall instruction coverage of 77% and branch coverage of 62%.

- **Strengths:**

  Classes like QueryRunner and AbstractQueryRunner show a good coverage of more than 78% and more than 76% branch coverage, demonstrating strong test case execution and validation. Other classes like GenerousBeanProcessor have 100% coverage, but they are so small compared to the other ones.

- **Weaknesses:**

  BaseResultSetHandler: Very poor instruction coverage of **2%** and branch coverage of **50%**. DbUtils: Instruction coverage of **47%** and branch coverage of **54%**. This is bad because there are 2 of the 4 biggest classes.

org.apache.commons.dbutils.handlers

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BeanMapHandler		100%		100%	0	8	0	18	0	7	0	1
KeyedHandler		100%		100%	0	8	0	17	0	7	0	1
ScalarHandler		100%		100%	0	7	0	15	0	5	0	1
ColumnListHandler		100%		100%	0	6	0	13	0	5	0	1
ArrayHandler		100%		100%	0	5	0	8	0	4	0	1
BeanHandler		100%		100%	0	4	0	7	0	3	0	1
AbstractKeyedHandler		100%		100%	0	4	0	6	0	3	0	1
BeanListHandler		100%		n/a	0	3	0	7	0	3	0	1
MapHandler		100%		100%	0	4	0	6	0	3	0	1
AbstractListHandler		100%		100%	0	3	0	5	0	2	0	1
MapListHandler		100%		n/a	0	3	0	6	0	3	0	1
ArrayListHandler		100%		n/a	0	3	0	6	0	3	0	1
Total	0 of 365	100%	0 of 20	100%	0	58	0	114	0	48	0	12

It shows a perfect coverage among all the classes.

org.apache.commons.dbutils.handlers

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
SqlNullCheckedResultSet		100%		95%	1	58	0	91	0	46	0	1
StringTrimmedResultSet		100%		83%	1	6	0	10	0	3	0	1
Total	0 of 348	100%	2 of 30	93%	2	64	0	101	0	49	0	2

SqlNullCheckedResultSet has an excellent coverage. StringTrimmedResultSet has an excellent coverage but it has a lower branch coverage.

org.apache.commons.dbutils.handlers.columns

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BooleanColumnHandler		100%		75%	1	5	0	3	0	3	0	1
LongColumnHandler		100%		75%	1	5	0	3	0	3	0	1
ByteColumnHandler		100%		75%	1	5	0	3	0	3	0	1
FloatColumnHandler		100%		75%	1	5	0	3	0	3	0	1
DoubleColumnHandler		100%		100%	0	5	0	3	0	3	0	1
IntegerColumnHandler		100%		100%	0	5	0	3	0	3	0	1
ShortColumnHandler		100%		75%	1	5	0	3	0	3	0	1
StringColumnHandler		100%		n/a	0	3	0	3	0	3	0	1
SQLXMLColumnHandler		100%		n/a	0	3	0	3	0	3	0	1
TimestampColumnHandler		100%		n/a	0	3	0	3	0	3	0	1
Total	0 of 173	100%	5 of 28	82%	5	44	0	30	0	30	0	10

All classes (BooleanColumnHandler, LongColumnHandler, ByteColumnHandler, etc.) have **100% instruction coverage.** Several classes (BooleanColumnHandler, LongColumnHandler, ShortColumnHandler, etc.) have **75% branch coverage**, leaving **5 branches** untested in total.

org.apache.commons.dbutils.handlers.columns.properties

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
DatePropertyHandler		100%		92%	1	11	0	21	0	3	0	1
StringEnumPropertyHandler		100%		100%	0	5	0	3	0	3	0	1
Total	0 of 90	100%	1 of 18	94%	1	16	0	24	0	6	0	2

Almost excellent coverage. It only miss 1 branch of 18.

The conclusion of the Jacoco report is that the majority of the project has a very well coverage. However, BaseResultSetHandler and DbUtils are some of the most important classes in the project and they have a poor coverage

Mutation Testing

i. Mutation testing campaign to evaluate test cases using PiTest.

Mutation testing is a software testing technique used to evaluate the quality of unit tests. It works by introducing small modifications (**mutations**) to the source code, such as changing operators or values, and then running the tests to see if they detect the changes. If the tests fail, they are considered effective, if not, it indicates gaps in the test suite.

In this project we've used PITest, which is a popular mutation testing tool for Java. It automatically generates mutations in the code, runs the tests, and reports which mutations were "**killed**" (detected) and which survived. It helps developers identify weaknesses in their test cases and improve overall test coverage.

To use this plugin we had to modify the pom.xml file and add the following lines to use the PITest plugin adapted to JUnit 5 in our Maven project.

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.15.2</version>
    <dependencies>
      <dependency>
        <groupId>org.pitest</groupId>
        <artifactId>pitest-junit5-plugin</artifactId>
        <version>1.2.1</version>
      </dependency>
    </dependencies>
  </plugin>
<plugin>
```

To generate the PITest mutation coverage report we run this command:

```
mvn test-compile org.pitest:pitest-maven:mutationCoverage
```

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
39	67%	946/1419	51%	377/734	84%	377/449

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.dbutils	13	59%	677/1150	43%	267/617	80%	267/332
org.apache.commons.dbutils.handlers	12	100%	114/114	100%	26/26	100%	26/26
org.apache.commons.dbutils.handlers.columns	10	100%	30/30	87%	33/38	87%	33/38
org.apache.commons.dbutils.handlers.properties	2	100%	24/24	100%	14/14	100%	14/14
org.apache.commons.dbutils.wrappers	2	100%	101/101	95%	37/39	95%	37/39

Report generated by PIT 1.15.2

Enhanced functionality available at arcmutate.com

Analysis of the mutation report

- **Number of lines:**

  The plugin generated mutations in the code of **39** classes

- **Line Coverage**:

  67% of the code's lines (946 out of 1419) were executed during the tests. This measures how much of the code is being covered by the test suite.
  A good result would be over 80%

- **Mutation Coverage**:

  51% of the mutations (377 out of 734) were killed by the tests. It means that the test suite successfully detected the mutations and test have failed as expected when the behaviour of them was altered.
  There is a big margin to improve the quality and the quantity of tests.

- **Test Strength**:

  84% of the test outcomes are "strong," meaning they successfully identified mutations when expected. This metric evaluates the ability of the tests to correctly validate the code's behavior against changes or errors.
  It's a good value because it's over 80%

org.apache.commons.dbutils

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
13	59%	677/1150	43%	267/617	80%	267/332

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AbstractQueryRunner.java	76%	126/165	63%	42/67	70%	42/60
AsyncQueryRunner.java	33%	28/86	24%	16/67	50%	16/32
BaseResultSetHandler.java	3%	8/292	2%	5/210	83%	5/6
BasicRowProcessor.java	98%	42/43	81%	17/21	89%	17/19
BeanProcessor.java	88%	107/122	86%	43/50	90%	43/48
DbUtils.java	66%	65/98	46%	25/54	74%	25/34
GenerousBeanProcessor.java	100%	19/19	100%	11/11	100%	11/11
OutParameter.java	86%	18/21	57%	4/7	100%	4/4
ProxyFactory.java	100%	12/12	100%	9/9	100%	9/9
QueryLoader.java	96%	22/23	75%	6/8	86%	6/7
QueryRunner.java	87%	183/211	78%	58/74	84%	58/69
ResultSetIterator.java	57%	12/21	36%	4/11	80%	4/5
StatementConfiguration.java	95%	35/37	96%	27/28	96%	27/28

- **Critical Classes**
  - **AsyncQueryRunner.java:**
    - **Line Coverage:** 33% (28/86)
    - **Mutation Coverage:** 24% (16/67)
    - **Test Strength:** 50%
    - This class has poor coverage across all metrics. It should be prioritized for adding or improving test cases.
  - **BaseResultSetHandler.java:**
    - **Line Coverage:** 3% (8/292)
    - **Mutation Coverage:** 2% (5/210)
    - **Test Strength:** 83%
    - This class is almost entirely untested. It needs significant attention to add tests for both line and mutation coverage.
- **Good Line Coverage but Bad Mutation Coverage**:
  - **DbUtils.java:**
    - **Line Coverage:** 66% (65/98)
    - **Mutation Coverage:** 46% (25/54)
    - **Test Strength:** 74%
    - While line coverage is quite good, the test cases aren't effectively detecting many mutations. Test quality should be reviewed and improved.
- **Excellent Classes**:

- **BeanProcessor.java:**
  - **Line Coverage:** 88% (43/50)
  - **Mutation Coverage:** 86% (43/50)
  - **Test Strength:** 90%
  - This class has excellent coverage and strong tests.

org.apache.commons.dbutils.handlers

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
12	100%	114/114	100%	26/26	100%	26/26

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AbstractKeyedHandler.java	100%	6/6	100%	3/3	100%	3/3
AbstractListHandler.java	100%	5/5	100%	2/2	100%	2/2
ArrayHandler.java	100%	8/8	100%	2/2	100%	2/2
ArrayListHandler.java	100%	6/6	100%	1/1	100%	1/1
BeanHandler.java	100%	7/7	100%	1/1	100%	1/1
BeanListHandler.java	100%	7/7	100%	1/1	100%	1/1
BeanMapHandler.java	100%	18/18	100%	3/3	100%	3/3
ColumnListHandler.java	100%	13/13	100%	3/3	100%	3/3
KeyedHandler.java	100%	17/17	100%	3/3	100%	3/3
MapHandler.java	100%	6/6	100%	2/2	100%	2/2
MapListHandler.java	100%	6/6	100%	1/1	100%	1/1
ScalarHandler.java	100%	15/15	100%	4/4	100%	4/4

This package showcases excellent test coverage and strength, with 100% in all metrics for every class.

org.apache.commons.dbutils.handlers.columns

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
10	100%	30/30	87%	33/38	87%	33/38

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
BooleanColumnHandler.java	100%	3/3	60%	3/5	60%	3/5
ByteColumnHandler.java	100%	3/3	100%	4/4	100%	4/4
DoubleColumnHandler.java	100%	3/3	100%	4/4	100%	4/4
FloatColumnHandler.java	100%	3/3	75%	3/4	75%	3/4
IntegerColumnHandler.java	100%	3/3	100%	4/4	100%	4/4
LongColumnHandler.java	100%	3/3	75%	3/4	75%	3/4
SQLXMLColumnHandler.java	100%	3/3	100%	3/3	100%	3/3
ShortColumnHandler.java	100%	3/3	75%	3/4	75%	3/4
StringColumnHandler.java	100%	3/3	100%	3/3	100%	3/3
TimestampColumnHandler.java	100%	3/3	100%	3/3	100%	3/3

This package has perfect line coverage but it doesn't have perfect mutation coverage and test strength. This indicates that while all lines of code are executed by the tests, a small

number of mutations are not being detected, which suggests areas where the test cases could be improved.

org.apache.commons.dbutils.handlers.properties

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	100%	24/24	100%	14/14	100%	14/14

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
DatePropertyHandler.java	100%	21/21	100%	10/10	100%	10/10
StringEnumPropertyHandler.java	100%	3/3	100%	4/4	100%	4/4

In this package only 2 classes are covered, anyway, all metrics are excellent.

org.apache.commons.dbutils.wrappers

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
2	100%	101/101	95%	37/39	95%	37/39

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
SqlNullCheckedResultSet.java	100%	91/91	94%	32/34	94%	32/34
StringTrimmedResultSet.java	100%	10/10	100%	5/5	100%	5/5

In this case, we have perfect line coverage and very high mutation coverage and test strength, only 2 mutations are not killed.

The conclusion is that classes like **BaseResultSetHandler.java** and **BaseResultSetHandler.java** should be prioritized for improvement. Poor results are due a very low number of tests or due the bad quality of the tests, so they don't detect the mutation.

Performance Testing

i. Implementation of performance tests with **JMH** to stress the project's most critical components.

What is Performance Testing?
We use performance testing to evaluate the speed, scalability, and stability of a system under specific workloads. It helps identify bottlenecks and ensures the system meets performance requirements. Key metrics include **throughput** (operations per second), **latency** (response time), and **resource usage** (CPU, memory, etc.).

JMH (Java Microbenchmark Harness)
For benchmarking we use JMH which is a framework from the OpenJDK team for precise microbenchmarking in Java. It handles JVM complexities like JIT optimizations and Garbage Collection to provide reliable performance measurements. With annotations like **@Benchmark**, JMH allows developers to test specific methods, measure metrics such as throughput or latency, and ensure high performance in Java applications.

The first step was to download the jhm_lessons folder, since the pom.xml contained the dependencies and plugins that I needed to add to my pom.xml:

```
<dependencies>

    <!-- Dipendenze JMH -->
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>1.37</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>1.37</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

```xml
<plugins>
    <!-- Plugin per JMH -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
            <annotationProcessorPaths>
                <path>
                    <groupId>org.openjdk.jmh</groupId>
                    <artifactId>jmh-generator-annprocess</artifactId>
                    <version>1.37</version>
                </path>
            </annotationProcessorPaths>
        </configuration>
    </plugin>

    <!-- Plugin Maven Shade per creare un JAR eseguibile -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.4.1</version>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>shade</goal>
                </goals>
                <configuration>
                    <createDependencyReducedPom>true</createDependencyReducedPom>
                    <transformers>
                        <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                            <mainClass>org.openjdk.jmh.Main</mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        </executions>
    </plugin>
```

Once copied and pasted, we had to create the benchmark in the src/main/java/org/apache/commons/dbutils folder:

```java
 2⊕ * Licensed to the Apache Software Foundation (ASF) under one or more…
17
18 package org.apache.commons.dbutils;
19
20⊕ import java.util.concurrent.TimeUnit;…
37
38 @State(org.openjdk.jmh.annotations.Scope.Thread)
39 @Fork(1)
40 @Threads(1)
41 @Warmup(iterations = 5)
42 @Measurement(iterations = 5)
43 @OutputTimeUnit(TimeUnit.MILLISECONDS)
44 public class MyBenchmark {
45
46     // Variable de control para imprimir el mensaje solo una vez
47     private boolean printedThroughput;
48
49⊖    /**
50      * Configuración inicial antes de ejecutar las iteraciones del benchmark.
51      */
52⊖    @Setup(Level.Trial)
53     public void setup() {
54         if (!printedThroughput) {
55             System.out.println("Executing Throughput");
56             printedThroughput = true;
57         }
58     }
```

```
59
60      /**
61       * Benchmark que mide el throughput (operaciones por segundo).
62       */
63      @Benchmark
64      @BenchmarkMode(Mode.Throughput)
65      @OutputTimeUnit(TimeUnit.SECONDS)
66      public void measureThroughput() {
67          // Simulando una operación costosa
68          int result = 0;
69          for (int i = 0; i < 1000; i++) {
70              result += i;
71          }
72      }
73
74      /**
75       * Método principal para ejecutar el benchmark.
76       */
77      public static void main(String[] args) throws RunnerException {
78          Options options = new OptionsBuilder()
79              .include(MyBenchmark.class.getSimpleName()) // Ejecutar este benchmark específico
80              .forks(1) // Ejecutar una sola vez
81              .build();
82
83          new Runner(options).run(); // Ejecutar el benchmark
84      }
85  }
86
```

This benchmark measures the throughput (operations per second) of a block of code that performs a simulated operation: adding the numbers 0 to 999 in a loop..

How it works:

Before the benchmark runs, the setup() method prints a message once to indicate that the throughput is being measured.

The measureThroughput() method performs the simulated addition operation repeatedly during the benchmark iterations.

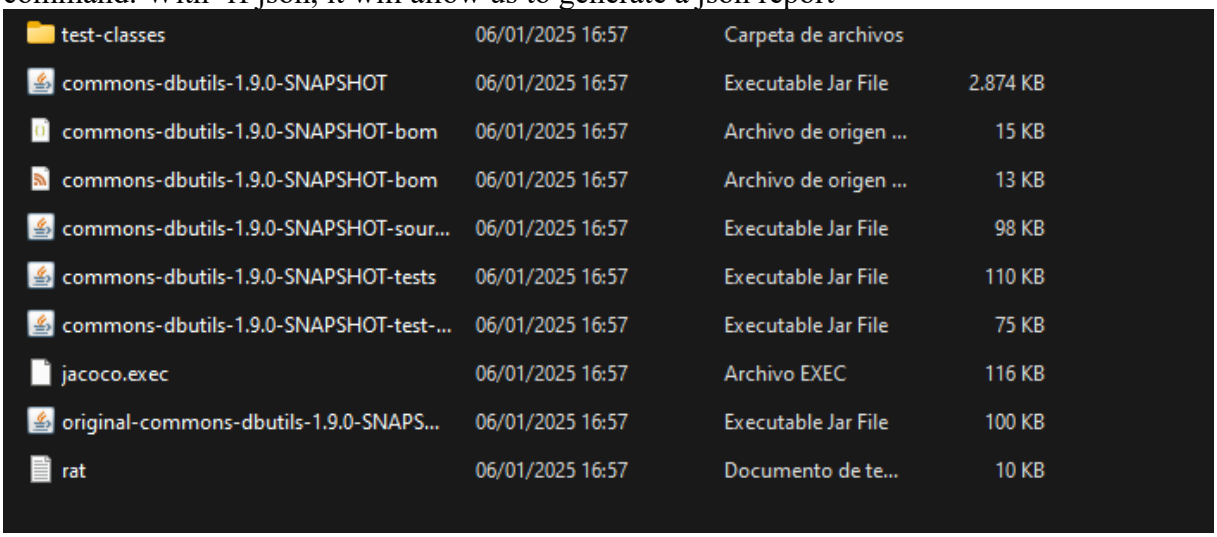
The benchmark mode set to Mode.Throughput measures how many times this method can be run per second.

The configuration specifies 1 fork, 5 warmup iterations to "warm up" the JVM, and 5 measurement iterations, ensuring accurate results.

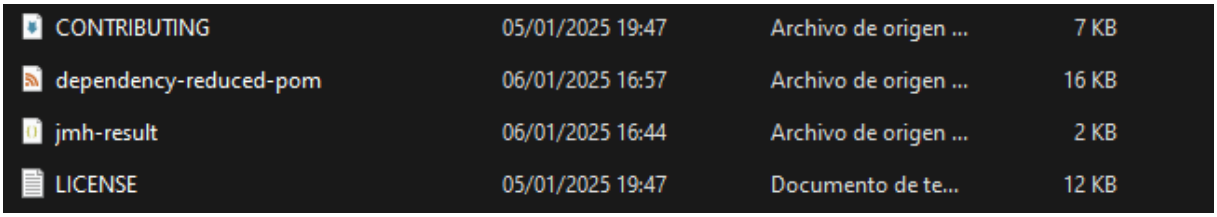
We first use the mvn clean install command, but later we use mvn clean package. The reason for using mvn clean package instead of mvn clean install is that we don't need to install the packaged artifact to the local repository when you just want to generate the packaged archive (JAR).

Once the command is used, a series of snapshots are generated, of which we must use the first one using the java -jar .\target\commons-dbutils-1.9.0-SNAPSHOT.jar -rf json

command. With -rf json, it will allow us to generate a json report



test-classes	06/01/2025 16:57	Carpeta de archivos	
commons-dbutils-1.9.0-SNAPSHOT	06/01/2025 16:57	Executable Jar File	2.874 KB
commons-dbutils-1.9.0-SNAPSHOT-bom	06/01/2025 16:57	Archivo de origen ...	15 KB
commons-dbutils-1.9.0-SNAPSHOT-bom	06/01/2025 16:57	Archivo de origen ...	13 KB
commons-dbutils-1.9.0-SNAPSHOT-sour...	06/01/2025 16:57	Executable Jar File	98 KB
commons-dbutils-1.9.0-SNAPSHOT-tests	06/01/2025 16:57	Executable Jar File	110 KB
commons-dbutils-1.9.0-SNAPSHOT-test-...	06/01/2025 16:57	Executable Jar File	75 KB
jacoco.exec	06/01/2025 16:57	Archivo EXEC	116 KB
original-commons-dbutils-1.9.0-SNAPS...	06/01/2025 16:57	Executable Jar File	100 KB
rat	06/01/2025 16:57	Documento de te...	10 KB



CONTRIBUTING	05/01/2025 19:47	Archivo de origen ...	7 KB
dependency-reduced-pom	06/01/2025 16:57	Archivo de origen ...	16 KB
jmh-result	06/01/2025 16:44	Archivo de origen ...	2 KB
LICENSE	05/01/2025 19:47	Documento de te...	12 KB

The jmh result is the created inform which has the following results:


```
[
    {
        "jmhVersion" : "1.37",
        "benchmark" : "org.apache.commons.dbutils.MyBenchmark.measureThroughput",
        "mode" : "thrpt",
        "threads" : 1,
        "forks" : 1,
        "jvm" : "C:\\Program Files\\Java\\jdk-17\\bin\\java.exe",
        "jvmArgs" : [
        ],
        "jdkVersion" : "17.0.6",
        "vmName" : "Java HotSpot(TM) 64-Bit Server VM",
        "vmVersion" : "17.0.6+9-LTS-190",
        "warmupIterations" : 5,
        "warmupTime" : "10 s",
        "warmupBatchSize" : 1,
        "measurementIterations" : 5,
        "measurementTime" : "10 s",
        "measurementBatchSize" : 1,
        "primaryMetric" : {
            "score" : 4.3143388020112835E7,
            "scoreError" : 2479147.3070332874,
            "scoreConfidence" : [
                4.066424071307955E7,
                4.562253532714612E7
            ],
```

```
            ],
            "scorePercentiles" : {
                "0.0" : 4.2560031962168895E7,
                "50.0" : 4.2893480426032886E7,
                "90.0" : 4.395804673631519E7,
                "95.0" : 4.395804673631519E7,
                "99.0" : 4.395804673631519E7,
                "99.9" : 4.395804673631519E7,
                "99.99" : 4.395804673631519E7,
                "99.999" : 4.395804673631519E7,
                "99.9999" : 4.395804673631519E7,
                "100.0" : 4.395804673631519E7
            },
            "scoreUnit" : "ops/s",
            "rawData" : [
                [
                    4.3696497070435405E7,
                    4.2560031962168895E7,
                    4.395804673631519E7,
                    4.2893480426032886E7,
                    4.26088839056118E7
                ]
            ]
        },
        "secondaryMetrics" : {
        }
    }
]

```


These results show us an average performance of 43,143,388 operations per second, with a margin of error of approximately 2.4 million operations. This indicates that the code under test is highly efficient, and the confidence interval and percentiles show that the results are consistent.

When pushing, we tried to compile it on github, giving us errors in both maven.yml and codeql-analysis.yml.

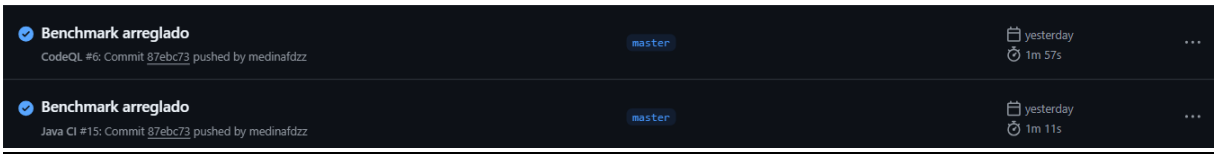
For cdql-analysis.yml the main problem was that we don´t have the @state annotation in our code:

```
[2025-01-06 03:27:19] [autobuild] [INFO]
[2025-01-06 03:27:19] [autobuild] [INFO] --- maven-compiler-plugin:3.13.0:testCompile (default-testCompile) @ commons-dbutils ---
[2025-01-06 03:27:19] [autobuild] [INFO] Recompiling the module because of changed dependency.
[2025-01-06 03:27:19] [autobuild] [INFO] Compiling 47 source files with javac [debug release 11] to target/test-classes
[2025-01-06 03:27:19] [autobuild] [INFO] -------------------------------------------------------------
Error: 1-06 03:27:19] [autobuild] [ERROR] COMPILATION ERROR :
[2025-01-06 03:27:19] [autobuild] [INFO] -------------------------------------------------------------
Error: 1-06 03:27:19] [autobuild] [ERROR] /home/runner/work/commons-dbutils/commons-dbutils/src/test/java/org/apache/commons/dbutils/MyBenchmark.java:[49,17] @Setup annotation is placed within a class not having
@State annotation. This is prohibited because it would have no effect.
[2025-01-06 03:27:19] [autobuild] [INFO] 1 error
[2025-01-06 03:27:19] [autobuild] [INFO] -------------------------------------------------------------
[2025-01-06 03:27:19] [autobuild] [INFO] ------------------------------------------------------------------------
[2025-01-06 03:27:19] [autobuild] [INFO] BUILD FAILURE
[2025-01-06 03:27:19] [autobuild] [INFO] ------------------------------------------------------------------------
```

And for Maven.yml the main problema was the imports order:

```
[INFO] There are 901 errors reported by Checkstyle 10.18.2 with /home/runner/work/commons-dbutils/commons-dbutils/src/conf/checkstyle/checkstyle.xml ruleset.
Error:  src/test/java/org/apache/commons/dbutils/MyBenchmark.java:[24,1] (imports) ImportOrder: Wrong order for 'org.openjdk.jmh.annotations.Measurement' import.
Error:  src/test/java/org/apache/commons/dbutils/MyBenchmark.java:[25,1] (imports) ImportOrder: Wrong order for 'org.openjdk.jmh.annotations.Fork' import.
Error:  src/test/java/org/apache/commons/dbutils/MyBenchmark.java:[27,1] (imports) ImportOrder: Wrong order for 'org.openjdk.jmh.annotations.Setup' import.
Error:  src/test/java/org/apache/commons/dbutils/MyBenchmark.java:[29,1] (imports) ImportOrder: Wrong order for 'org.openjdk.jmh.annotations.Level' import.
Error:  src/test/java/org/apache/commons/dbutils/MyBenchmark.java:[36,1] (imports) ImportOrder: Wrong order for 'java.util.concurrent.TimeUnit' import.
```

So we just put he @state annotation and ordered the imports in our benchmark



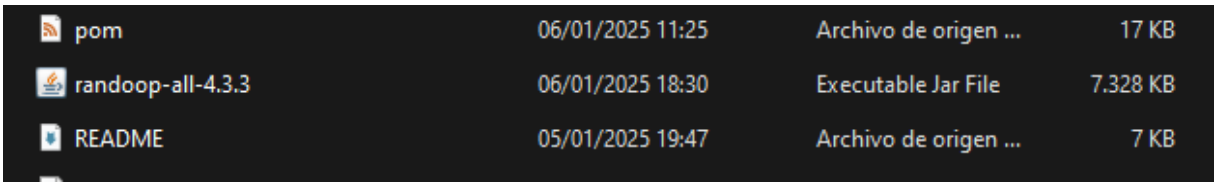
Automated Test Generation

i. Use of automated tools to create tests covering poorly tested code components.

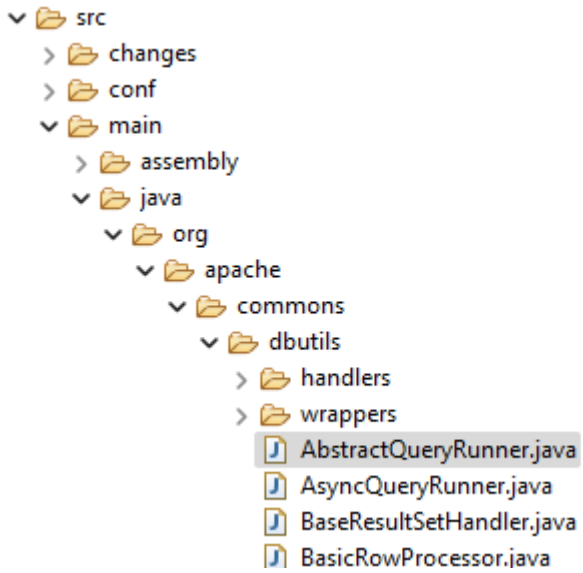
What is randoop?
Randoop is an automatic unit test generation tool based on random testing techniques. Its main purpose is to create JUnit test cases that cover different code scenarios, including complex execution paths that might be difficult to identify manually.

Steps to follow:

1) First, we download the archive called randoop-all-4.3.3.jar (This file was placed in the project path: .\commons-dbutils but it is not necessary.to put here), the purpose of this JAR is to ensure that Randoop is accessible from the command line.



2) We chose the class on which to generate tests, in our case we opted for the class org.apache.commons.dbutils.AbstractQueryRunner.



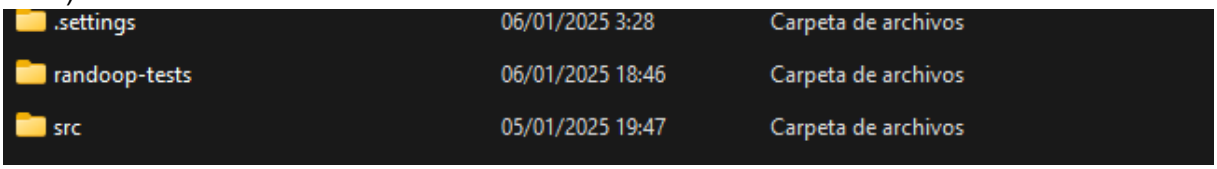
3) The command was defined with the necessary parameters for Randoop:

java -cp "C:\Users\suso1\git\commons-dbutils\randoop-all-4.3.3.jar;C:\Users\suso1\git\commons-dbutils\target\classes" randoop.main.Main gentests --testclass=org.apache.commons.dbutils.AbstractQueryRunner --time-limit=20 --junit-output-dir="C:\Users\suso1\git\commons-dbutils\randoop-tests"

We adjusted the values of:
--testclass: Class to analyze.
--time-limit: Time limit in seconds to generate tests.
--junit-output-dir: Folder where the generated cases were saved.

4) Initially we had the problem: ClassNotFoundException: randoop.main.Main, since we had specified the wrong route

5) Once the command was correctly configured, it was executed to generate automatic tests. The generated tests were saved in the specified folder (randoop-tests) in JUnit format.




```
Progress update: steps=4546000, test inputs generated=2, failing inputs=0      (2025-01-07T17:37:05.536130900Z     240M
used)
Progress update: steps=4546202, test inputs generated=2, failing inputs=0      (2025-01-07T17:37:05.537128400Z     240M
used)
Normal method executions: 20
Exceptional method executions: 0

Average method execution time (normal termination):      0,00463
Average method execution time (exceptional termination): NaN
Approximate memory usage 240M
Explorer = ForwardGenerator(steps: 4546202, null steps: 4546200, num_sequences_generated: 2;
    allSequences: 2, regresson seqs: 1, error seqs: 0=0=0, invalid seqs: 0, subsumed_sequences: 0, num_failed_output_tes
t: 1;
    sideEffectFreeMethods: 1113, runtimePrimitivesSeen: 38)

No error-revealing tests to output.

About to look for failing assertions in 1 regression sequences.

Regression test output:
Regression test count: 1
Writing regression JUnit tests...
Created file C:\Users\suso1\git\commons-dbutils\randoop-tests\RegressionTest0.java
Created file C:\Users\suso1\git\commons-dbutils\randoop-tests\RegressionTest.java
Wrote regression JUnit tests.
About to look for flaky methods.

Invalid tests generated: 0
```

This Randoop report shows a summary of the generated test execution:

- Normal method executions: 20 methods have been executed with successful completion.

- Exceptional executions: No exceptional executions (errors) have been generated during the execution of the methods.

- Average method execution time:
  Normal completion: The average execution time for methods with successful completion was 0.00463 seconds.

- Exceptional completion: No exceptional completion has been recorded, so no data is available.

- Approximate memory usage: 240 MB.

- Sequence generator:
  4,546,202 steps were generated, of which 4,546,200 were valid steps.

- 2 sequences were generated in total, with one regression sequence and no erroneous or invalid sequences.

- Methods with no side effects: 1,113 methods have no identified side effects.

- Runtime primitives seen: 38 primitives.

- Regression tests: A regression test was generated, and the corresponding files were written to the randoop-tests directory:
  RegressionTest0.java
  RegressionTest.java

Security Analysis

i. Security evaluation using OWASP FindSecBugs and OWASP Dependency-Check.

We have started using FindSecBugs to analyze the security of our code. This tool, which is based on SpotBugs, helps us detect potential vulnerabilities and improve the security of the application. This way, we can automatically audit the code and find potential flaws that could put the project at risk. It is one more way to ensure that the code is clean and more secure.

1) First step was use this commands to install FindSecBugs
git clone -b "version-1.12.0" --single-branch --depth 1 https://github.com/find-sec-bugs/find-sec-bugs

cd find-sec-bugs

cd cli

gradle packageCli

After this we get:

```
C:\Users\suso1\find-sec-bugs\cli>gradle packageCli

BUILD SUCCESSFUL in 7s
2 actionable tasks: 2 executed

C:\Users\suso1\find-sec-bugs\cli>
```

We have some problems because the Maven versión was above 7.0 so me need to modify the gradle.build.changing the next:

- compile -> implementation
- runtime -> runtimeClasspath
- archiveName -> archiveFileName
- destinationDir -> destinationDirectory

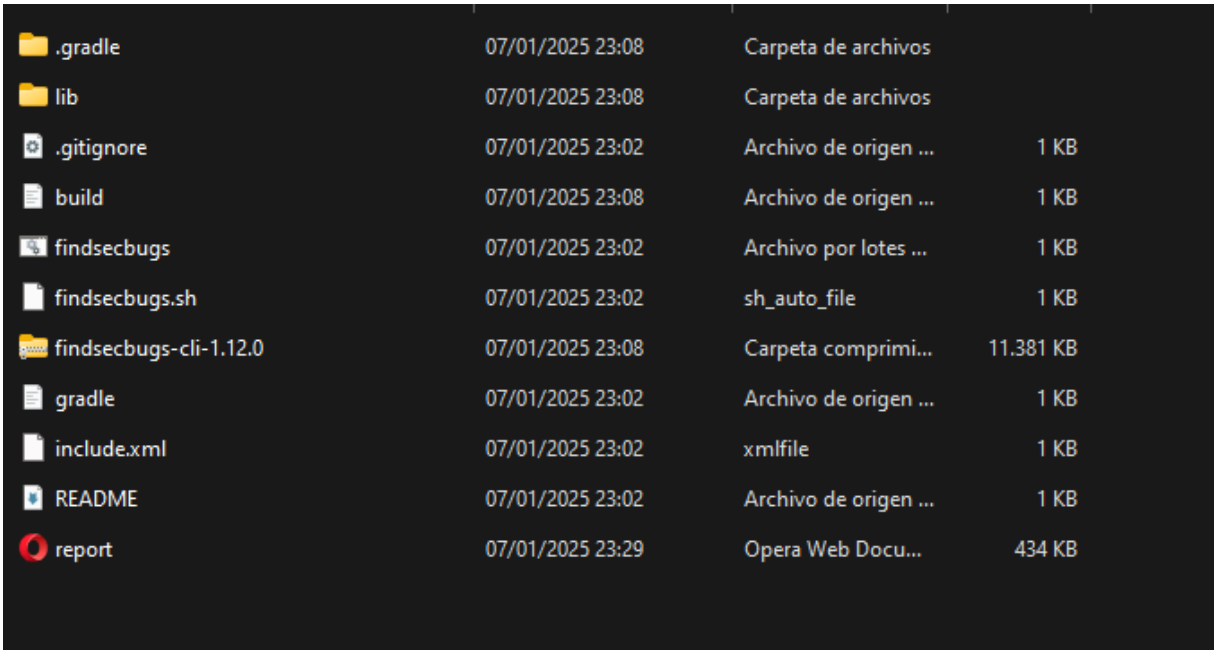
After this we need to use Git Bash because we are going to use the file called findsecbugs.sh to generate a problems security report about our Project:

```
suso1@LaptopAM MINGW64 ~/find-sec-bugs/cli ((version-1.12.0))
$ ./findsecbugs.sh -progress -html -output report.html /c/Users/suso1/git/commons-dbutils
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#noProviders for further details.
Scanning archives (1 / 1)
2 analysis passes to performWARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by edu.umd.cs.findbugs.ba.jsr305.TypeQualifierValue (file:/C:/Users/
suso1/find-sec-bugs/cli/lib/spotbugs-4.5.3.jar)
WARNING: Please consider reporting this to the maintainers of edu.umd.cs.findbugs.ba.jsr305.TypeQualifierValue
WARNING: System::setSecurityManager will be removed in a future release

Pass 1: Analyzing classes (5243 / 5243) - 100% complete
Pass 2: Analyzing classes (4246 / 4246) - 100% complete
Done with analysis

suso1@LaptopAM MINGW64 ~/find-sec-bugs/cli ((version-1.12.0))
```

It creates a report in the directory that we use the command:



This report is upload at the GitHub Project as reportFindSecBugs.

This report shows:

Summary of vulnerabilities: The report will include a summary of the security vulnerabilities found, categorizing them by severity (e.g. Critical, High, Medium, Low). This helps prioritize corrective actions.

Detailed description of issues: Each vulnerability found is detailed with a full description. This includes:

The type of security issue (e.g. SQL injection, incorrect use of encryption, exposure of sensitive data, etc.).

A reference to the source or vulnerability technique used.
Affected source code, with the specific lines where the issue was detected.
Vulnerability categories: Issues are grouped by categories, allowing you to identify common types of vulnerabilities in your code.

Recommendations: In most cases, FindSecBugs suggests how to fix or mitigate the vulnerabilities found. This may include better coding practices, use of more secure functions, or recommended security libraries.

Statistics: Some reports include charts or tables showing the number of vulnerabilities by severity, category, or issue type, making it easier to understand the overall impact of security issues on the project.

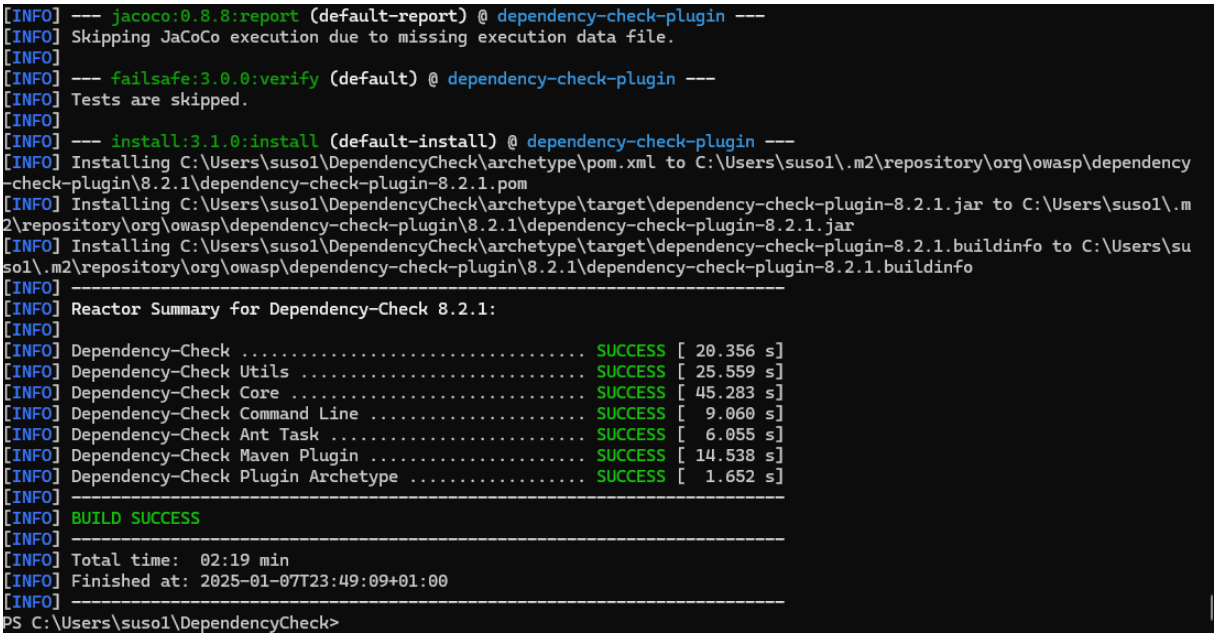
Next, we use OWASP Dependency-Check (DC) to improve the security of our applications. This scan helps us detect potential vulnerabilities in our project dependencies, and we are integrating it into the workflow to ensure that there are no security issues in the code we are using. Through the tool, we can identify risks and take action before they affect the project.

1) The first command we use was:
   git clone -b "v8.2.1" --single-branch --depth 1https://github.com/jeremylong/DependencyCheck

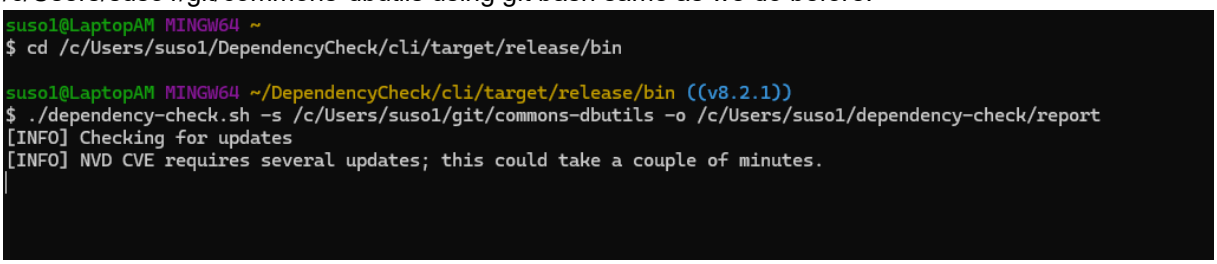
   cd DependencyCheck

   mvn -s settings.xml install -DskipTests=true

   After use this command we see this:



   Then, we run OWASP with the command: ./dependency-check.sh -s /c/Users/suso1/git/commons-dbutils using git bash same as we do before:



We use the first command to move to the dependency-check.sh location and the second command to to run the dependency security scan.

The report is upload to the gitHubProject as dependency-check-report.

The report shows:

Vulnerability Summary: A listing of vulnerabilities found in the project's dependencies, including their severity (high, medium, low).

Vulnerability Dependencies: A breakdown of the specific dependencies that contain vulnerabilities, with details about each vulnerability, such as the vulnerability name, its CVE (Common Vulnerabilities and Exposures), and its description.

Vulnerability Severity: A visual or tabular summary of the severity of the vulnerabilities found (e.g., "Critical," "High," "Medium," "Low").

Recommendations: Suggestions on how to mitigate or resolve the vulnerabilities, such as updating a version of the dependency or applying security patches.

Vulnerability Distribution: Charts or tables showing the distribution of vulnerabilities by type (e.g., by severity or by component).