Quarkus - Measuring the coverage of your tests

Learn how to measure the test coverage of your application. This guide covers:

- Measuring the coverage of your Unit Tests

- Measuring the coverage of your Integration Tests

- Separating the execution of your Unit Tests and Integration Tests

- Consolidating the coverage for all your tests

Please note that code coverage is not supported in native mode.

1. Prerequisites

To complete this guide, you need:

- less than 15 minutes

- an IDE

- JDK 1.8+ installed with JAVA_HOME configured appropriately

- Apache Maven 3.6.3

- Having completed the Testing your application guide

2. Architecture

The application built in this guide is just a JAX-RS endpoint (hello world) that relies on dependency injection to use a service. The service will be tested with JUnit 5 and the endpoint will be annotated via a `@QuarkusTest` annotation.

3. Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example. Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `tests-with-coverage-quickstart` directory.

4. Starting from a simple project and two tests

Let's start from an empty application created with the Quarkus Maven plugin:

```
mvn io.quarkus:quarkus-maven-plugin:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=tests-with-coverage-quickstart
cd tests-with-coverage-quickstart
```

Now we'll be adding all the elements necessary to have an application that is properly covered with tests.

First, an application serving a hello endpoint:

```
package org.acme.testcoverage;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class GreetingResource {

    private final GreetingService service;

    @Inject
    public GreetingResource(GreetingService service) {
        this.service = service;
    }

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    public String greeting(@PathParam("name") String name) {
        return service.greeting(name);
    }

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

This endpoint uses a greeting service:

```
package org.acme.testcoverage;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class GreetingService {

    public String greeting(String name) {
        return "hello " + name;
    }

}
```

The project will also need some tests. First a simple JUnit:

```
package org.acme.testcoverage;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class GreetingServiceTest {

    @Test
    public void testGreetingService() {
        GreetingService service = new GreetingService();
        Assertions.assertEquals("hello Quarkus",
service.greeting("Quarkus"));
    }
}
```

But also a @QuarkusTest:

```java
package org.acme.testcoverage;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Tag;

import java.util.UUID;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
@Tag("integration")
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
          .when().get("/hello")
          .then()
             .statusCode(200)
             .body(is("hello"));
    }

    @Test
    public void testGreetingEndpoint() {
        String uuid = UUID.randomUUID().toString();
        given()
          .pathParam("name", uuid)
          .when().get("/hello/greeting/{name}")
          .then()
            .statusCode(200)
            .body(is("hello " + uuid));
    }
}
```

The first one will be our example of a Unit Test and the second one will be our example of Integration Test.

5. Separating executions of Unit Tests and Integration Tests

You may want to consider that JUnits and QuarkusTests are two different kind of tests and that they deserve to be separated. This way you could run them separately, in different cases or some more often than the others. In order to do so, we'll use a feature of JUnit 5 that allows us to tag some tests. Let's tag `GreetingResourceTest.java` and specify that it is an Integration Test:

```
import org.junit.jupiter.api.Tag;
...

@QuarkusTest
@Tag("integration")
public class GreetingResourceTest {
    ...
}
```

We're now able to distinguish unit tests and integration tests. Now, let's bind them to different Maven lifecycle phases. Let's use surefire to bind unit tests to the **test** phase and the integration tests to the **integration-test** phase.

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>${surefire-plugin.version}</version>
                <configuration>
                    <excludedGroups>integration</excludedGroups>
                    <systemPropertyVariables>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.ut
il.logging.manager>
                        <maven.home>${maven.home}</maven.home>
                    </systemPropertyVariables>
                </configuration>
                <executions>
                    <execution>
                        <id>integration-tests</id>
                        <phase>integration-test</phase>
                        <goals>
                            <goal>test</goal>
                        </goals>
                        <configuration>

<excludedGroups>!integration</excludedGroups>
                            <groups>integration</groups>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

This way, the `QuarkusTest` instances will be executed as part of the `integration-test` build phase while the other JUnit tests will still be ran during the `test` phase. You can run all the tests with the command `./mvnw clean verify` (and you will notice that two tests are ran in different phases).

6. Measuring the coverage of JUnit tests using JaCoCo

It is now time to introduce JaCoCo to measure the coverage. The straightforward way to add JaCoCo

to your build is to reference the plugin in your `pom.xml`.

```xml
    <properties>
    ...
        <jacoco.version>0.8.4</jacoco.version>
    </properties>

    <build>
        <plugins>
        ...
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>${jacoco.version}</version>
                <executions>
                    <execution>
                        <id>default-prepare-agent</id>
                        <goals>
                            <goal>prepare-agent</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>default-report</id>
                        <goals>
                            <goal>report</goal>
                        </goals>
                        <configuration>

<dataFile>${project.build.directory}/jacoco.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco</outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

ℹ️ If you run `./mvnw clean test` the coverage information will be collected during the execution of the unit tests in the file `jacoco.exec`.

7. Measuring separately the coverage of each test type

It is not strictly necessary, but let's distinguish the coverage brought by each test type. To do so, we`ll just output the coverage info in two different files, one in `jacoco-ut.exec` and one in `jacoco-it.exec`. We also need to generate a separate report for each test execution. Let's adjust the Jacoco configuration for that:

```
<properties>
...
    <jacoco.version>0.8.4</jacoco.version>
</properties>

<build>
    <plugins>
    ...
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>${jacoco.version}</version>
            <executions>
                <execution>
                    <id>prepare-agent-ut</id>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                    <configuration>

<destFile>${project.build.directory}/jacoco-ut.exec</destFile>
                    </configuration>
                </execution>
                <execution>
                    <id>prepare-agent-it</id>
                    <phase>pre-integration-test</phase>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                    <configuration>

<destFile>${project.build.directory}/jacoco-it.exec</destFile>
                    </configuration>
                </execution>
                <execution>
                    <id>report-ut</id>
                    <goals>
                        <goal>report</goal>
                    </goals>
```

```
                    <configuration>

<dataFile>${project.build.directory}/jacoco-ut.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
ut</outputDirectory>
                    </configuration>
                </execution>
                <execution>
                    <id>report-it</id>
                    <phase>post-integration-test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                    <configuration>

<dataFile>${project.build.directory}/jacoco-it.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
it</outputDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

8. The coverage does not seem to correspond to the reality

You can now run the tests: `./mvnw clean verify` As explained earlier, it will run the unit tests first, then the integration tests. And finally, it will generate two separate reports. First a report of the coverage of the unit tests in `target/site/jacoco-ut` then a report of the coverage of the integration tests in `target/site/jacoco-it`.

Given the content of `GreetingResourceTest`, `GreetingResource` should have been covered. But when we open the report `target/site/jacoco-it/index.html`, the class `GreetingResource` is reported with 0% of coverage. But the fact that `GreetingService` is reported as covered shows that the test execution was actually recorded. How come?

During the report generation, you may have noticed a warning:

```
[WARNING] Classes in bundle '***' do no match with execution data.
For report generation the same class files must be used as at
runtime.
[WARNING] Execution data for class
org/acme/testcoverage/GreetingResource does not match.
```

It seems that Quarkus and JaCoCo step on each other's toes. What happens is that Quarkus transforms the JAX-RS resources (and also the Panache files). You may have noticed that GreetingResource was not written in the simplest way like:

```
...
@Path("/hello")
public class GreetingResource {

    @Inject
    GreetingService service;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("/greeting/{name}")
    public String greeting(@PathParam("name") String name) {
        return service.greeting(name);
    }

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

Above, the constructor is implicit and we use injection to have an instance of GreetingService. Note that, with this code relying on an implicit constructor, the coverage would have been reported properly by JaCoCo. Instead, we introduced a constructor based injection:

```
...
@Path("/hello")
public class GreetingResource {

    private final GreetingService service;

    @Inject
    public GreetingResource(GreetingService service) {
        this.service = service;
    }
...
}
```

Some might say that this approach is preferable since the field can be **final** like this. Anyway, in some cases you might need an explicit constructor. And, in that case, the coverage is not reported properly by JaCoCo. This is because Quarkus generates a constructor without any parameter and does some bycode manipulations in order to add it to the class. That is what happened here, just before the execution of the integration tests:

```
[INFO] --- quarkus-maven-plugin:0.16.0:build (default) @ getting-
started-testing ---
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Beginning quarkus
augmentation
...
```

As a consequence, JaCoCo does not recognize the classes when it wants to create its report. But wait… there is a solution.

9. Instrumenting the classes instead

JaCoCo has two modes. The first one is based on an agent and instruments classes on-the-fly. Unfortunately, this is incompatible with the dynamic classfile transformations that Quarkus does. The second mode is called offline instrumentation. Classes are pre-instrumented in advance via the **jacoco:instrument** Maven goal and during their usage (when the tests are ran), **jacocoagent.jar** must be added to the classpath. Once the tests have been executed, it is recommended to restore the original classes using the **jacoco:restore-instrumented-classes** Maven goal.

Let's first add the dependency on **jacocoagent.jar**:

```
<project>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>org.jacoco</groupId>
            <artifactId>org.jacoco.agent</artifactId>
            <classifier>runtime</classifier>
            <scope>test</scope>
            <version>${jacoco.version}</version>
        </dependency>
    </dependencies>
</project>
```

Then let's configure three jacoco plugin goals for unit tests:

- One to instrument the classes during the **process-classes** phase

- One to restore the original classes during the **prepare-package** phase (after the tests are ran)

- One to generate the report during the **verify** phase (the report generation requires the original classes to have been restored)

and a similar setup for the integration tests too:

```
<project>
    ...
    <build>
        <plugins>
            ...
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>${jacoco.version}</version>
                <executions>
                    <execution>
                        <id>instrument-ut</id>
                        <goals>
                            <goal>instrument</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>restore-ut</id>
                        <goals>
                            <goal>restore-instrumented-
classes</goal>
                        </goals>
                    </execution>
```

```xml
                            <execution>
                                <id>report-ut</id>
                                <goals>
                                    <goal>report</goal>
                                </goals>
                                <configuration>

<dataFile>${project.build.directory}/jacoco-ut.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
ut</outputDirectory>
                                </configuration>
                            </execution>
                            <execution>
                                <id>instrument-it</id>
                                <phase>pre-integration-test</phase>
                                <goals>
                                    <goal>instrument</goal>
                                </goals>
                            </execution>
                            <execution>
                                <id>restore-it</id>
                                <phase>post-integration-test</phase>
                                <goals>
                                    <goal>restore-instrumented-
classes</goal>
                                </goals>
                            </execution>
                            <execution>
                                <id>report-it</id>
                                <phase>post-integration-test</phase>
                                <goals>
                                    <goal>report</goal>
                                </goals>
                                <configuration>

<dataFile>${project.build.directory}/jacoco-it.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
it</outputDirectory>
                                </configuration>
                            </execution>
                        </executions>
                    </plugin>
                </plugins>
            </build>
</project>
```

It also requires a small change in the Surefire configuration. Note below that we specified `jacoco-agent.destfile` as a system property in the default case (unit tests) and for the integration tests.

```xml
<project>
    ...
    <build>
        <plugins>
            ...
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>${surefire-plugin.version}</version>
                <configuration>
                    <excludedGroups>integration</excludedGroups>
                    <systemPropertyVariables>
                        <jacoco-agent.destfile>${project.build.directory}/jacoco-ut.exec</jacoco-agent.destfile>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
                        <maven.home>${maven.home}</maven.home>
                    </systemPropertyVariables>
                </configuration>
                <executions>
                    <execution>
                        <id>integration-tests</id>
                        <phase>integration-test</phase>
                        <goals>
                            <goal>test</goal>
                        </goals>
                        <configuration>

<excludedGroups>!integration</excludedGroups>
                            <groups>integration</groups>
                            <systemPropertyVariables>
                                <jacoco-agent.destfile>${project.build.directory}/jacoco-it.exec</jacoco-agent.destfile>
                            </systemPropertyVariables>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Let's now check the generated report that can be found in `target/site/jacoco-`

`it/index.html`. The report now shows that `GreetingResource` is actually properly covered! Yay!

10. Bonus: Building a consolidated report for Unit Tests and Integration Tests

So, finally, let's improve the setup even further and let's merge the two execution files (**jacoco-ut.exec** and **jacoco-it.exec**) into one consolidated report and generate a consolidated report that will show the coverage of all your tests combined.

You should end up with something like this (note the addition of the `merge-results` and `post-merge-report` executions):

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>${surefire-plugin.version}</version>
                <configuration>
                    <excludedGroups>integration</excludedGroups>
                    <systemPropertyVariables>
                        <jacoco-
agent.destfile>${project.build.directory}/jacoco-ut.exec</jacoco-
agent.destfile>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
                        <maven.home>${maven.home}</maven.home>
                    </systemPropertyVariables>
                </configuration>
                <executions>
                    <execution>
                        <id>integration-tests</id>
                        <phase>integration-test</phase>
                        <goals>
                            <goal>test</goal>
                        </goals>
                        <configuration>

<excludedGroups>!integration</excludedGroups>
                            <groups>integration</groups>
                            <systemPropertyVariables>
                                <jacoco-
agent.destfile>${project.build.directory}/jacoco-it.exec</jacoco-
agent.destfile>
                            </systemPropertyVariables>
```

```xml
            </configuration>
          </execution>
        </executions>
      </plugin>
      ...
      <plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>${jacoco.version}</version>
        <executions>
          <execution>
            <id>instrument-ut</id>
            <goals>
              <goal>instrument</goal>
            </goals>
          </execution>
          <execution>
            <id>restore-ut</id>
            <goals>
              <goal>restore-instrumented-
classes</goal>
            </goals>
          </execution>
          <execution>
            <id>report-ut</id>
            <goals>
              <goal>report</goal>
            </goals>
            <configuration>

<dataFile>${project.build.directory}/jacoco-ut.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco-
ut</outputDirectory>
            </configuration>
          </execution>
          <execution>
            <id>instrument-it</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>instrument</goal>
            </goals>
          </execution>
          <execution>
            <id>restore-it</id>
            <phase>post-integration-test</phase>
            <goals>
              <goal>restore-instrumented-
classes</goal>
```

```xml
                    </goals>
                </execution>
                <execution>
                    <id>report-it</id>
                    <phase>post-integration-test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                    <configuration>

<dataFile>${project.build.directory}/jacoco-it.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco-it</outputDirectory>
                    </configuration>
                </execution>
                <execution>
                    <id>merge-results</id>
                    <phase>verify</phase>
                    <goals>
                        <goal>merge</goal>
                    </goals>
                    <configuration>
                        <fileSets>
                            <fileSet>

<directory>${project.build.directory}</directory>
                                <includes>
                                    <include>*.exec</include>
                                </includes>
                            </fileSet>
                        </fileSets>

<destFile>${project.build.directory}/jacoco.exec</destFile>
                    </configuration>
                </execution>
                <execution>
                    <id>post-merge-report</id>
                    <phase>verify</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                    <configuration>

<dataFile>${project.build.directory}/jacoco.exec</dataFile>

<outputDirectory>${project.reporting.outputDirectory}/jacoco</outputDirectory>
                    </configuration>
```

```
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
...
<dependencies>
    ...
    <dependency>
        <groupId>org.jacoco</groupId>
        <artifactId>org.jacoco.agent</artifactId>
        <classifier>runtime</classifier>
        <scope>test</scope>
        <version>${jacoco.version}</version>
    </dependency>
</dependencies>
</project>
```

11. Conclusion

You now have all the information you need to study the coverage of your tests! But remember, some code that is not covered is certinaly not well tested. But some code that is covered is not necessarily **well** tested. Make sure to write good tests!