Quarkus - Testing Your Application

Learn how to test your Quarkus Application. This guide covers:

- Testing in JVM mode

- Testing in native mode

- Injection of resources into tests

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

- The completed greeter application from the Getting Started Guide

2. Architecture

In this guide, we expand on the initial test that was created as part of the Getting Started Guide. We cover injection into tests and also how to test native executables.

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 `getting-started-testing` directory.

This guide assumes you already have the completed application from the `getting-started` directory.

4. Recap of HTTP based Testing in JVM mode

If you have started from the Getting Started example you should already have a completed test, including the correct `pom.xml` setup.

In the `pom.xml` file you should see 2 test dependencies:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

quarkus-junit5 is required for testing, as it provides the @QuarkusTest annotation that controls the testing framework. rest-assured is not required but is a convenient way to test HTTP endpoints, we also provide integration that automatically sets the correct URL so no configuration is required.

Because we are using JUnit 5, the version of the Surefire Maven Plugin must be set, as the default version does not support Junit 5:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${surefire-plugin.version}</version>
    <configuration>
        <systemPropertyVariables>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
            <maven.home>${maven.home}</maven.home>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

We also set the java.util.logging.manager system property to make sure tests will use the correct logmanager and maven.home to ensure that custom configuration from ${maven.home}/conf/settings.xml is applied (if any).

The project should also contain a simple test:

```java
package org.acme.getting.started.testing;

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

import java.util.UUID;

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

@QuarkusTest
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));
    }

}
```

This test uses HTTP to directly test our REST endpoint. When the test is run the application will be started before the test is run.

4.1. Controlling the test port

While Quarkus will listen on port 8080 by default, when running tests it defaults to 8081. This allows you to run tests while having the application running in parallel.

Quarkus also provides RestAssured integration that updates the default port used by RestAssured before the tests are run, so no additional configuration should be required.

4.2. Injecting a URI

It is also possible to directly inject the URL into the test which can make is easy to use a different client. This is done via the `@TestHTTPResource` annotation.

Let's write a simple test that shows this off to load some static resources. First create a simple HTML file in `src/main/resources/META-INF/resources/index.html`:

```
<html>
    <head>
        <title>Testing Guide</title>
    </head>
    <body>
        Information about testing
    </body>
</html>
```

We will create a simple test to ensure that this is being served correctly:

```
package org.acme.getting.started.testing;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.nio.charset.StandardCharsets;

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

import io.quarkus.test.common.http.TestHTTPResource;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
public class StaticContentTest {

    @TestHTTPResource("index.html") ①
    URL url;

    @Test
    public void testIndexHtml() throws Exception {
        try (InputStream in = url.openStream()) {
            String contents = readStream(in);
            Assertions.assertTrue(contents.contains("<title>Testing
Guide</title>"));
        }
    }

    private static String readStream(InputStream in) throws
IOException {
        byte[] data = new byte[1024];
        int r;
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        while ((r = in.read(data)) > 0) {
            out.write(data, 0, r);
        }
        return new String(out.toByteArray(),
StandardCharsets.UTF_8);
    }
}
```

① This annotation allows you to directly inject the URL of the Quarkus instance, the value of the annotation will be the path component of the URL

For now @TestHTTPResource allows you to inject URI, URL and String representations of the URL.

5. Testing a specific endpoint

Both RESTassured and @TestHTTPResource allow you to specify the endpoint class you are testing rather than hard coding a path. This currently supports both JAX-RS endpoints, Servlets and Reactive Routes. This makes it a lot easier to see exactly which endpoints a given test is testing.

For the purposes of these examples I am going to assume we have an endpoint that looks like the following:

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

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

> **ℹ** This currently does not support the @ApplicationPath() annotation to set the JAX-RS context path. Use the quarkus.resteasy.path config value instead if you want a custom context path.

5.1. TestHTTPResource

You can the use the io.quarkus.test.common.http.TestHTTPEndpoint annotation to specify the endpoint path, and the path will be extracted from the provided endpoint. If you also specify a value for the TestHTTPResource endpoint it will be appended to the end of the endpoint path.

```java
package org.acme.getting.started.testing;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.nio.charset.StandardCharsets;

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

import io.quarkus.test.common.http.TestHTTPEndpoint;
import io.quarkus.test.common.http.TestHTTPResource;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
public class StaticContentTest {

    @TestHTTPEndpoint(GreetingResource.class)  ①
    @TestHTTPResource
    URL url;

    @Test
    public void testIndexHtml() throws Exception {
        try (InputStream in = url.openStream()) {
            String contents = readStream(in);
            Assertions.assertTrue(contents.equals("hello"));
        }
    }

    private static String readStream(InputStream in) throws
IOException {
        byte[] data = new byte[1024];
        int r;
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        while ((r = in.read(data)) > 0) {
            out.write(data, 0, r);
        }
        return new String(out.toByteArray(),
StandardCharsets.UTF_8);
    }
}
```

① Because `GreetingResource` is annotated with `@Path("/hello")` the injected URL will end with `/hello`.

5.2. RESTassured

To control the RESTassured base path (i.e. the default path that serves as the root for every request) you can use the `io.quarkus.test.common.http.TestHTTPEndpoint` annotation. This can be applied at the class or method level. To test out greeting resource we would do:

```java
package org.acme.getting.started.testing;

import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.test.common.http.TestHTTPEndpoint;
import org.junit.jupiter.api.Test;

import java.util.UUID;

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

@QuarkusTest
@TestHTTPEndpoint(GreetingResource.class)  ①
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
          .when().get()      ②
          .then()
             .statusCode(200)
             .body(is("hello"));
    }
}
```

① This tells RESTAssured to prefix all requests with `/hello`.

② Note we don't need to specify a path here, as `/hello` is the default for this test

6. Injection into tests

So far we have only covered integration style tests that test the app via HTTP endpoints, but what if we want to do unit testing and test our beans directly?

Quarkus supports this by allowing you to inject CDI beans into your tests via the `@Inject` annotation (in fact, tests in Quarkus are full CDI beans, so you can use all CDI functionality). Let's create a simple test that tests the greeting service directly without using HTTP:

```
package org.acme.getting.started.testing;

import javax.inject.Inject;

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

import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
public class GreetingServiceTest {

    @Inject  ①
    GreetingService service;

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

① The `GreetingService` bean will be injected into the test

7. Applying Interceptors to Tests

As mentioned above Quarkus tests are actually full CDI beans, and as such you can apply CDI interceptors as you would normally. As an example, if you want a test method to run within the context of a transaction you can simply apply the `@Transactional` annotation to the method and the transaction interceptor will handle it.

In addition to this you can also create your own test stereotypes. For example we could create a `@TransactionalQuarkusTest` as follows:

```
@QuarkusTest
@Stereotype
@Transactional
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface TransactionalQuarkusTest {
}
```

If we then apply this annotation to a test class it will act as if we had applied both the `@QuarkusTest` and `@Transactional` annotations, e.g.:

```
@TransactionalQuarkusTest
public class TestStereotypeTestCase {

    @Inject
    UserTransaction userTransaction;

    @Test
    public void testUserTransaction() throws Exception {
        Assertions.assertEquals(Status.STATUS_ACTIVE,
userTransaction.getStatus());
    }

}
```

8. Enrichment via QuarkusTest*Callback

Alternatively or additionally to an interceptor, you can enrich **all** your @QuarkusTest classes by implementing the following callback interfaces:

- io.quarkus.test.junit.callback.QuarkusTestBeforeAllCallback

- io.quarkus.test.junit.callback.QuarkusTestBeforeEachCallback

- io.quarkus.test.junit.callback.QuarkusTestAfterEachCallback

Such a callback implementation has to be registered as a "service provider" as defined by java.util.ServiceLoader.

E.g. the following sample callback:

```
package org.acme.getting.started.testing;

import
io.quarkus.test.junit.callback.QuarkusTestBeforeEachCallback;
import io.quarkus.test.junit.callback.QuarkusTestMethodContext;

public class MyQuarkusTestBeforeEachCallback implements
QuarkusTestBeforeEachCallback {

    @Override
    public void beforeEach(QuarkusTestMethodContext context) {
        System.out.println("Executing " + context.getTestMethod());
    }
}
```

has to be registered via src/main/resources/META-INF/services/io.quarkus.test.junit.callback.QuarkusTestBeforeEachCallback as follows:

```
org.acme.getting.started.testing.MyQuarkusTestBeforeEachCallback
```

It is possible to read annotations from the test class or method to control what the callback shall be doing.

While it is possible to use JUnit Jupiter callback interfaces like `BeforeEachCallback`, you might run into classloading issues because Quarkus has to run tests in a custom classloader which JUnit is not aware of.

9. Testing Different Profiles

So far in all our examples we only start Quarkus once for all tests. Before the first test is run Quarkus will boot, then all tests will run, then Quarkus will shutdown at the end. This makes for a very fast testing experience however it is a bit limited as you can't test different configurations.

To get around this Quarkus supports the idea of a test profile. If a test has a different profile to the previously run test then Quarkus will be shut down and started with the new profile before running the tests. This is obviously a bit slower, as it adds a shutdown/startup cycle to the test time, but gives a great deal of flexibility.

In order to reduce the amount of times Quarkus needs to restart it is recommended that you place all tests that need a specific profile into their own package, and then run tests alphabetically.

9.1. Writing a Profile

To implement a test profile we need to implement `io.quarkus.test.junit.QuarkusTestProfile`:

```
package org.acme.getting.started.testing;

import java.util.Collections;
import java.util.Map;
import java.util.Set;

import io.quarkus.test.junit.QuarkusTestProfile;

public class MockGreetingProfile implements QuarkusTestProfile {

    @Override
    public Map<String, String> getConfigOverrides() { ①
        return
Collections.singletonMap("quarkus.resteasy.path","/api");
    }

    @Override
    public Set<Class<?>> getEnabledAlternatives() { ②
        return Collections.singleton(MockGreetingService.class);
    }


    @Override
    public String getConfigProfile() { ③
        return "test";
    }
}
```

① This method allows us to override configuration properties. Here we are changing the JAX-RS root path.

② This method allows us to enable CDI `@Alternative` beans. This makes it easy to mock out certain beans functionality.

③ This can be used to change the config profile. As this default is `test` this does nothing, but is included for completeness.

Now we have defined our profile we need to include it on our test class. We do this with `@TestProfile(MockGreetingProfile.class)`.

All the test profile config is stored in a single class, which makes it easy to tell if the previous test ran with the same configuration.

10. Mock Support

Quarkus supports the use of mock objects using two different approaches. You can either use CDI alternatives to mock out a bean for all test classes, or use `QuarkusMock` to mock out beans on a per test basis.

10.1. CDI `@Alternative` mechanism.

To use this simply override the bean you wish to mock with a class in the `src/test/java` directory, and put the `@Alternative` and `@Priority(1)` annotations on the bean. Alternatively, a convenient `io.quarkus.test.Mock` stereotype annotation could be used. This built-in stereotype declares `@Alternative`, `@Priority(1)` and `@Dependent`. For example if I have the following service:

```
@ApplicationScoped
public class ExternalService {

    public String service() {
        return "external";
    }

}
```

I could mock it with the following class in `src/test/java`:

```
@Mock
@ApplicationScoped ①
public class MockExternalService extends ExternalService {

    @Override
    public String service() {
        return "mock";
    }
}
```

① Overrides the `@Dependent` scope declared on the `@Mock` stereotype.

It is important that the alternative be present in the `src/test/java` directory rather than `src/main/java`, as otherwise it will take effect all the time, not just when testing.

Note that at present this approach does not work with native image testing, as this would required the test alternatives to be baked into the native image.

10.2. Mocking using QuarkusMock

The `io.quarkus.test.junit.QuarkusMock` class can be used to temporarily mock out any normal scoped bean. If you use this method in a `@BeforeAll` method the mock will take effect for all tests on the current class, while if you use this in a test method the mock will only take effect for the duration of the current test.

This method can be used for any normal scoped CDI bean (e.g. `@ApplicationScoped`, `@RequestScoped` etc, basically every scope except `@Singleton` and `@Dependent`).

An example usage could look like:

```
@QuarkusTest
public class MockTestCase {

    @Inject
    MockableBean1 mockableBean1;

    @Inject
    MockableBean2 mockableBean2;

    @BeforeAll
    public static void setup() {
        MockableBean1 mock = Mockito.mock(MockableBean1.class);
        Mockito.when(mock.greet("Stuart")).thenReturn("A mock for
Stuart");
        QuarkusMock.installMockForType(mock, MockableBean1.class);
①
    }

    @Test
    public void testBeforeAll() {
        Assertions.assertEquals("A mock for Stuart",
mockableBean1.greet("Stuart"));
        Assertions.assertEquals("Hello Stuart",
mockableBean2.greet("Stuart"));
    }

    @Test
    public void testPerTestMock() {
        QuarkusMock.installMockForInstance(new BonourGreeter(),
mockableBean2); ②
        Assertions.assertEquals("A mock for Stuart",
mockableBean1.greet("Stuart"));
        Assertions.assertEquals("Bonjour Stuart",
mockableBean2.greet("Stuart"));
    }

    @ApplicationScoped
    public static class MockableBean1 {

        public String greet(String name) {
            return "Hello " + name;
        }
    }

    @ApplicationScoped
    public static class MockableBean2 {
```

```
        public String greet(String name) {
            return "Hello " + name;
        }
    }

    public static class BonourGreeter extends MockableBean2 {
        @Override
        public String greet(String name) {
            return "Bonjour " + name;
        }
    }
}
```

① As the injected instance is not available here we use `installMockForType`, this mock is used for both test methods

② We use `installMockForInstance` to replace the injected bean, this takes effect for the duration of the test method.

Note that there is no dependency on Mockito, you can use any mocking library you like, or even manually override the objects to provide the behaviour you require.

10.2.1. Further simplification with `@InjectMock`

Building on the features provided by `QuarkusMock`, Quarkus also allows users to effortlessly take advantage of Mockito for mocking the beans supported by `QuarkusMock`. This functionality is available via the `@io.quarkus.test.junit.mockito.InjectMock` annotation which is available in the `quarkus-junit5-mockito` dependency.

Using `@InjectMock`, the previous example could be written as follows:

```
@QuarkusTest
public class MockTestCase {

    @InjectMock
    MockableBean1 mockableBean1; ①

    @InjectMock
    MockableBean2 mockableBean2;

    @BeforeEach
    public void setup() {
        Mockito.when(mockableBean1.greet("Stuart")).thenReturn("A
mock for Stuart"); ②
    }

    @Test
    public void firstTest() {
        Assertions.assertEquals("A mock for Stuart",
mockableBean1.greet("Stuart"));
```

```
            Assertions.assertEquals(null,
mockableBean2.greet("Stuart")); ③
    }

    @Test
    public void secondTest() {

Mockito.when(mockableBean2.greet("Stuart")).thenReturn("Bonjour
Stuart"); ④
        Assertions.assertEquals("A mock for Stuart",
mockableBean1.greet("Stuart"));
        Assertions.assertEquals("Bonjour Stuart",
mockableBean2.greet("Stuart"));
    }

    @ApplicationScoped
    public static class MockableBean1 {

        public String greet(String name) {
            return "Hello " + name;
        }
    }

    @ApplicationScoped
    public static class MockableBean2 {

        public String greet(String name) {
            return "Hello " + name;
        }
    }
}
```

① `@InjectMock` results in a mock being and is available in test methods of the test class (other test classes are **not** affected by this)

② The `mockableBean1` is configured here for every test method of the class

③ Since the `mockableBean2` mock has not been configured, it will return the default Mockito response.

④ In this test the `mockableBean2` is configured, so it returns the configured response.

Although the test above is good for showing the capabilities of `@InjectMock`, it is not a good representation of a real test. In a real test we would most likely configure a mock, but then test a bean that uses the mocked bean. Here is an example:

```
@QuarkusTest
public class MockGreetingServiceTest {

    @InjectMock
    GreetingService greetingService;

    @Test
    public void testGreeting() {
        when(greetingService.greet()).thenReturn("hi");
        given()
                .when().get("/greeting")
                .then()
                .statusCode(200)
                .body(is("hi")); ①
    }

    @Path("greeting")
    public static class GreetingResource {

        final GreetingService greetingService;

        public GreetingResource(GreetingService greetingService) {
            this.greetingService = greetingService;
        }

        @GET
        @Produces("text/plain")
        public String greet() {
            return greetingService.greet();
        }
    }

    @ApplicationScoped
    public static class GreetingService {
        public String greet(){
            return "hello";
        }
    }
}
```

① Since we configured `greetingService` as a mock, the `GreetingResource` which uses the `GreetingService` bean, we get the mocked response instead of the response of the regular `GreetingService` bean

10.2.2. Using Spies instead of Mocks with `@InjectSpy`

Building on the features provided by `InjectMock`, Quarkus also allows users to effortlessly take advantage of Mockito for spying on the beans supported by `QuarkusMock`. This functionality is

available via the `@io.quarkus.test.junit.mockito.InjectSpy` annotation which is available in the `quarkus-junit5-mockito` dependency.

Sometimes when testing you only need to verify that a certain logical path was taken, or you only need to stub out a single method's response while still executing the rest of the methods on the Spied clone. Please see Mockito documentation for more details on Spy partial mocks. In either of those situations a Spy of the object is preferable. Using `@InjectSpy`, the previous example could be written as follows:

```
@QuarkusTest
public class SpyGreetingServiceTest {

    @InjectSpy
    GreetingService greetingService;

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

        Mockito.verify(greetingService, Mockito.times(1)).greet();
①
    }

    @Test
    public void testOverrideGreeting() {
        when(greetingService.greet()).thenReturn("hi"); ②
        given()
                .when().get("/greeting")
                .then()
                .statusCode(200)
                .body(is("hi")); ③
    }

    @Path("greeting")
    public static class GreetingResource {

        final GreetingService greetingService;

        public GreetingResource(GreetingService greetingService) {
            this.greetingService = greetingService;
        }

        @GET
        @Produces("text/plain")
        public String greet() {
```

```
                return greetingService.greet();
        }
    }

    @ApplicationScoped
    public static class GreetingService {
        public String greet(){
            return "hello";
        }
    }
}
```

① Instead of overriding the value, we just want to ensure that the greet method on our `GreetingService` was called by this test.

② Here we are telling the Spy to return "hi" instead of "hello". When the `GreetingResource` requests the greeting from `GreetingService` we get the mocked response instead of the response of the regular `GreetingService` bean

③ We are verifying that we get the mocked response from the Spy.

10.2.3. Using `@InjectMock` with `@RestClient`

The `@RegisterRestClient` registers the implementation of the rest-client at runtime, and because the bean needs to be a regular scope, you have to annotate your interface with `@ApplicationScoped`.

```
@Path("/")
@ApplicationScoped
@RegisterRestClient
public interface GreetingService {

    @GET
    @Path("/hello")
    @Produces(MediaType.TEXT_PLAIN)
    String hello();
}
```

For the test class here is an example:

```
@QuarkusTest
public class GreetingResourceTest {

    @InjectMock
    @RestClient ①
    GreetingService greetingService;

    @Test
    public void testHelloEndpoint() {
        Mockito.when(greetingService.hello()).thenReturn("hello
from mockito");

        given()
          .when().get("/hello")
          .then()
            .statusCode(200)
            .body(is("hello from mockito"));
    }

}
```

① Indicate that this injection point is meant to use an instance of `RestClient`.

10.3. Mocking with Panache

If you are using the `quarkus-hibernate-orm-panache` or `quarkus-mongodb-panache` extensions, check out the Hibernate ORM with Panache Mocking and MongoDB with Panache Mocking documentation for the easiest way to mock your data access.

11. Testing Security

If you are using Quarkus Security, check out the Testing Security section for information on how to easily test security features of the application.

12. Starting services before the Quarkus application starts

A very common need is to start some services on which your Quarkus application depends, before the Quarkus application starts for testing. To address this need, Quarkus provides `@io.quarkus.test.common.QuarkusTestResource` and `io.quarkus.test.common.QuarkusTestResourceLifecycleManager`.

By simply annotating any test in the test suite with `@QuarkusTestResource`, Quarkus will run the corresponding `QuarkusTestResourceLifecycleManager` before any tests are run. A test suite is also free to utilize multiple `@QuarkusTestResource` annotations, in which case all the corresponding `QuarkusTestResourceLifecycleManager` objects will be run before the tests.

Quarkus provides a few implementations of `QuarkusTestResourceLifecycleManager` out of the box (see `io.quarkus.test.h2.H2DatabaseTestResource` which starts an H2 database, or `io.quarkus.test.kubernetes.client.KubernetesMockServerTestResource` which starts a mock Kubernetes API server), but it is common to create custom implementations to address specific application needs. Common cases include starting docker containers using Testcontainers (an example of which can be found here), or starting a mock HTTP server using Wiremock (an example of which can be found here).

13. Native Executable Testing

It is also possible to test native executables using `@NativeImageTest`. This supports all the features mentioned in this guide except injecting into tests (and the native executable runs in a separate non-JVM process this is not really possible).

This is covered in the Native Executable Guide.

14. Running `@QuarkusTest` from an IDE

Most IDEs offer the possibility to run a selected class as JUnit test directly. For this you should set a few properties in the settings of your chosen IDE:

- `java.util.logging.manager` (see Logging Guide)

- `maven.home` (only if there are any custom settings in `${maven.home}/conf/settings.mxl`, see Maven Guide)

- `maven.settings` (in case a custom version of `settings.xml` file should be used for the tests)

14.1. Eclipse separate JRE definition

Copy your current "Installed JRE" definition into a new one, where you will add the properties as a new VM arguments:

- `-Djava.util.logging.manager=org.jboss.logmanager.LogManager`

- `-Dmaven.home=<path-to-your-maven-installation>`

Use this JRE definition as your Quarkus project targeted runtime and the workaround will be applied to any "Run as JUnit" configuration.

14.2. VSCode "run with" configuration

The `settings.json` placed in the root of your project directory or in the workspace will need the workaround in your test configuration:

```
"java.test.config": [
    {
        "name": "quarkusConfiguration",
        "vmargs": [ "-
Djava.util.logging.manager=org.jboss.logmanager.LogManager
-Dmaven.home=<path-to-your-maven-installation> ..." ],
        ...
    },
  ...
]
```

14.3. IntelliJ JUnit template

Nothing needed in IntelliJ because the IDE will pick the `systemPropertyVariables` from the surefire plugin configuration in `pom.xml`.