Red Hat build of Quarkus 1.11

Compiling your Quarkus applications to native executables

Last Updated: 2021-06-30

Legal Notice

Abstract

This guide shows you how to compile the Quarkus Getting Started project into a native executable and how to configure and test the native executable.

Table of Contents

PREFACE

As an application developer, you can use Red Hat build of Quarkus to create microservices written in Java that run on OpenShift and serverless environments. Applications compiled to native executables have small memory footprints and fast startup times.

This guide shows you how to compile the Quarkus Getting Started project into a native executable and how to configure and test the native executable. You will need the application created in *Getting started with Quarkus*.

Building a native executable with Red Hat build of Quarkus covers:

- Building a native executable with a single command using a container runtime such as Podman or Docker
- Creating a custom container image using the produced native executable
- Creating a container image using the OpenShift Docker build strategy
- Deploying the Quarkus native application to OpenShift
- Configuring the native executable
- Testing the native executable

Prerequisites

- Have OpenJDK (JDK) 11 installed and the **JAVA_HOME** environment variable set to specify the location of the Java SDK.
    - Log in to the Red Hat Customer Portal to download Red Hat build of Open JDK from the Software Downloads page.
- An OCI (Open Container Initiative) compatible container runtime, such as Podman or Docker.
- A completed Quarkus Getting Started project.
    - To learn how to build the Quarkus Getting Started project, see *Getting started with Quarkus*.
    - Alternatively, you can download the Quarkus quickstart archive or clone the **Quarkus Quickstarts** Git repository. The sample project is in the **getting-started** directory.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our technical content and encourage you to tell us what you think. If you'd like to add comments, provide insights, correct a typo, or even ask a question, you can do so directly in the documentation.

NOTE

You must have a Red Hat account and be logged in to the customer portal.

To submit documentation feedback from the customer portal, do the following:

1. Select the **Multi-page HTML** format.
2. Click the **Feedback** button at the top-right of the document.
3. Highlight the section of text where you want to provide feedback.
4. Click the **Add Feedback** dialog next to your highlighted text.
5. Enter your feedback in the text box on the right of the page and then click **Submit**.

We automatically create a tracking issue each time you submit feedback. Open the link that is displayed after you click **Submit** and start watching the issue or add more comments.

Thank you for the valuable feedback.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. PRODUCING A NATIVE EXECUTABLE

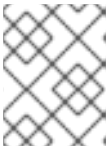
You can produce a native executable from your Quarkus application using a container runtime such as Podman or Docker. Quarkus produces a binary executable using a builder image, which you can use together with the Red Hat Universal Base Images RHEL8-UBI and RHEL8-UBI minimal. Red Hat build of Quarkus 1.11 uses **registry.access.redhat.com/quarkus/mandrel-20-rhel8:20.3** as a default for the **quarkus.native.builder-image** property.

The native executable for your application contains the application code, required libraries, Java APIs, and a reduced version of a virtual machine (VM). The smaller VM base improves the startup time of the application and produces a minimal disk footprint.

Procedure

1. Open the Getting Started project **pom.xml** file and verify that it includes the **native** profile:

```
<profiles>
  <profile>
    <id>native</id>
    <properties>
      <quarkus.package.type>native</quarkus.package.type>
    </properties>
  </profile>
</profiles>
```



NOTE

Using Quarkus **native** profile allows you to run both the native executable and the native image tests.

2. Build a native executable using one of the following methods:

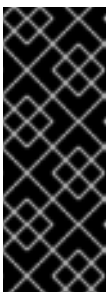
   a. Build a native executable with Docker:

```
./mvnw package -Pnative -Dquarkus.native.container-build=true
```

   b. Build a native executable with Podman:

```
./mvnw package -Pnative -Dquarkus.native.container-build=true -
Dquarkus.native.container-runtime=podman
```

   These commands create the **getting-started-*-runner** binary in the **target** directory.



IMPORTANT

Compiling a Quarkus application to a native executable consumes a lot of memory during analysis and optimization. You can limit the amount of memory used during native compilation by setting the **quarkus.native.native-image-xmx** configuration property. Setting low memory limits might increase the build time. For more details, refer to Native executable configuration properties.

3. Run the native executable:

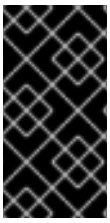
```
./target/getting-started-*-runner
```

When you build the native executable the **prod** profile is enabled and the Quarkus native tests run using the **prod** profile. You can change this using the **quarkus.test.native-image-profile** property.

CHAPTER 2. CREATING A CUSTOM CONTAINER IMAGE

You can create a container image from your Quarkus application using one of the following methods:

- Creating a container manually
- Creating a container using the OpenShift Docker build



IMPORTANT

Compiling a Quarkus application to a native executable consumes a lot of memory during analysis and optimization. You can limit the amount of memory used during native compilation by setting the **quarkus.native.native-image-xmx** configuration property. Setting low memory limits might increase the build time.

2.1. CREATING A CONTAINER MANUALLY

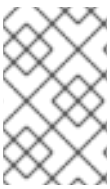
This section shows you how to manually create a container image with your application for Linux X86_64. When you produce a native image using the Quarkus Native container it creates an executable that targets the Linux X86_64 operating system. If your host operating system is different from this, you will not be able to run the binary directly and you will need to create a container manually.

Your Quarkus Getting Started project includes a **Dockerfile.native** in the **src/main/docker** directory with the following content:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.3
WORKDIR /work/
RUN chown 1001 /work \
    && chmod "g+rwX" /work \
    && chown 1001:root /work
COPY --chown=1001:root target/*-runner /work/application

EXPOSE 8080
USER 1001

CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```



UNIVERSAL BASE IMAGE (UBI)

The **Dockerfiles** use UBI as a base image. This base image was designed to work in containers. The **Dockerfiles** use the *minimal* version of the base image to reduce the size of the produced image.

Procedure

1. Build a native Linux executable using one of the following methods:
   a. Build a native executable with Docker:

      ```
      ./mvnw package -Pnative -Dquarkus.native.container-build=true
      ```

   b. Build a native executable with Podman:

```
./mvnw package -Pnative -Dquarkus.native.container-build=true -
Dquarkus.native.container-runtime=podman
```

2. Build the container image using one of the following methods:
   a. Build the container image with Docker:

```
docker build -f src/main/docker/Dockerfile.native -t quarkus-quickstart/getting-started .
```

   b. Build the container image with Podman

```
podman build -f src/main/docker/Dockerfile.native -t quarkus-quickstart/getting-started .
```

3. Run the container:
   a. Run the container with Docker:

```
docker run -i --rm -p 8080:8080 quarkus-quickstart/getting-started
```

   b. Run the container with Podman:

```
podman run -i --rm -p 8080:8080 quarkus-quickstart/getting-started
```

2.2. CREATING A CONTAINER USING THE OPENSHIFT DOCKER BUILD

You can create a container image for your Quarkus application using the OpenShift Docker build strategy. This strategy creates a container using a build configuration in the cluster.

Prerequisites

- You have access to a Red Hat OpenShift Container Platform cluster and the latest version of the OpenShift CLI (oc) is installed. For information about installing oc, see the "Installing the CLI" section of the *Installing and configuring OpenShift Container Platform clusters* guide.
- A URL for the OpenShift API endpoint.

Procedure

1. Log in to the OpenShift CLI:

```
oc login -u <username_url>
```

2. Create a new project in OpenShift:

```
oc new-project <project_name>
```

3. Create a build config based on the **src/main/docker/Dockerfile.native** file:

```
cat src/main/docker/Dockerfile.native | oc new-build --name <build_name> --strategy=docker
--dockerfile -
```

4. Build the project:

```
oc start-build <build_name> --from-dir .
```

5. Deploy the project to OpenShift:

```
oc new-app <build_name>
```

6. Expose the services:

```
oc expose svc/<build_name>
```

CHAPTER 3. NATIVE EXECUTABLE CONFIGURATION PROPERTIES

Configuration properties define how the native executable is generated. You can configure your Quarkus application using the **application.properties** file.

Configuration properties

The following table lists the configuration properties that you can set to define how the native executable is generated:

Property	Description	Type	Default
quarkus.native.additional-build-args	Additional arguments to pass to the build process.	list of string	
quarkus.native.enable-http-url-handler	Enables HTTP URL handler. This allows you to do URL.openConnection() for HTTP URLs.	boolean	**true**
quarkus.native.enable-https-url-handler	Enables HTTPS URL handler. This allows you to do URL.openConnection() for HTTPS URLs.	boolean	**false**
quarkus.native.enable-all-security-services	Adds all security services to the native image.	boolean	**false**
quarkus.native.add-all-charsets	Adds all character sets to the native image. This increases image size.	boolean	**false**
quarkus.native.graalvm-home	Contains the path of the Graal distribution.	string	**${GRAALVM_HOME:}**
quarkus.native.java-home	Contains the path of the JDK.	File	**${java.home}**
quarkus.native.native-image-xmx	The maximum Java heap used to generate the native image.	string	
quarkus.native.debug-build-process	Waits for a debugger to attach to the build process before running the native image build. This is an advanced option for those familiar with GraalVM internals.	boolean	**false**
quarkus.native.publish-debug-build-process-port	Publishes the debug port when building with docker and debug-build-process is true.	boolean	**true**

Property	Description	Type	Default
quarkus.native.cleanup-server	Restarts the native image server.	boolean	**false**
quarkus.native.enable-isolates	Enables isolates to improve the memory management.	boolean	**true**
quarkus.native.enable-fallback-images	Creates a JVM based fallback image if native image fails.	boolean	**false**
quarkus.native.enable-server	Uses native image server. This can speed up compilation but can cause changes to drop due to cache invalidation issues.	boolean	**false**
quarkus.native.auto-service-loader-registration	Automatically registers all META-INF/services entries.	boolean	**false**
quarkus.native.dump-proxies	Dumps the bytecode of all proxies for inspection.	boolean	**false**
quarkus.native.container-build	Builds using a container runtime. Docker is used by default.	boolean	**false**
quarkus.native.builder-image	The docker image to build the image.	string	**registry.access.redhat.com/quarkus/mandrel-20-rhel8:20.3**
quarkus.native.container-runtime	The container runtime used build the image. For example, Docker.	string	
quarkus.native.container-runtime-options	Options to pass to the container runtime.	list of string	
quarkus.native.enable-vm-inspection	Enables VM introspection in the image.	boolean	**false**
quarkus.native.full-stack-traces	Enables full stack traces in the image.	boolean	**true**
quarkus.native.enable-reports	Generates reports on call paths and included packages/classes/methods.	boolean	**false**
quarkus.native.report-exception-stack-traces	Reports exceptions with a full stack trace.	boolean	**true**

Property	Description	Type	Default
quarkus.native.report-errors-at-runtime	Reports errors at runtime. This may cause your application to fail at runtime if you are using unsupported feature.	boolean	**false**
quarkus.native.resources.includes	A comma separated list of globs to match resource paths that should be added to the native image. Use slash (/) as a path separator on all platforms. Globs must not start with slash. For example you have **src/main/resources/ignored.png** and **src/main/resources/foo/selected.png** in your source tree and one of your dependency JARs contains **bar/some.txt** file, with the following configuration quarkus.native.resources.includes = foo/**,bar//*.txt the files **src/main/resources/foo/selected.png** and **bar/some.txt** will be included in the native image, while **src/main/resources/ignored.png** will not be included. To find out more about the glob features see the Supported glob features and its description.	list of string	
quarkus.native.debug.enabled	Enables debug and generates debug symbols in a separate .debug file. When used with **quarkus.native.container-build**, Red Hat build of Quarkus only supports Red Hat Enterprise Linux or other Linux distributions as they contain the **binutils** package that installs the **objcopy** utility to split the debug info from the native image.	boolean	**false**

Supported glob features and its description

The following table lists the supported glob features and its description:

Character	Feature description
*	Matches a possibly empty sequence of characters that does not contain slash (/).

****	Matches a possibly empty sequence of characters that might contain slash (/).
?	Matches one character, but not slash.
[abc]	Matches one character from the range specified in the bracket, but not slash.
[a-z]	Matches one character from the range specified in the bracket, but not slash.
[!abc]	Matches one character not specified in the bracket; does not match slash.
[a-z]	Matches one character outside the range specified in the bracket; does not match slash.
{one,two,three}	Matches any of the alternating tokens separated by comma; the tokens may contain wildcards, nested alternations and ranges.
\	The escape character. There are three levels of escaping: **application.properties** parser, MicroProfile Config list converter, and Glob parser. All three levels use the backslash as the escaping character.

Additional resources

- Configuring your Quarkus applications

3.1. CONFIGURING MEMORY CONSUMPTION FOR QUARKUS NATIVE COMPILATION

Compiling a Quarkus application to a native executable consumes a lot of memory during analysis and optimization. You can limit the amount of memory used during native compilation by setting the **quarkus.native.native-image-xmx** configuration property. Setting low memory limits might increase the build time.

Procedure

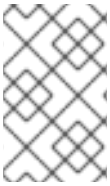
- Use one of the following methods to set a value for the **quarkus.native.native-image-xmx** property to limit the memory consumption during the native image build time:
  - Using the **application.properties** file:

    ```
    quarkus.native.native-image-xmx=<maximum_memory>
    ```

  - Setting system properties:

    ```
    mvn -Pnative -Dquarkus.native.container-build=true -Dquarkus.native.native-image-xmx=<maximum_memory>
    ```

    This command builds the native executable with Docker. Add **-Dquarkus.native.container-runtime=podman** argument to use Podman.



NOTE

For example, to set the memory limit to 6 GB, enter **quarkus.native.native-image-xmx=6g**. The value must be a multiple of 1024 greater than 2MB. Append the letter m or M to indicate megabytes, or g or G to indicate gigabytes.

CHAPTER 4. TESTING THE NATIVE EXECUTABLE

Test the application running in the native mode to test the functionality of the native executable. Use **@NativeImageTest** annotation to build the native executable and run test against the http endpoints.

Procedure

1. Open the **pom.xml** file and verify that the **native** profile contains the following elements:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>${surefire-plugin.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
      <configuration>
        <systemPropertyVariables>
          <native.image.path>${project.build.directory}/${project.build.finalName}-
runner</native.image.path>

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

   The **failsafe-maven-plugin** runs integration test and indicates the location of the produced native executable.

2. Open the **src/test/java/org/acme/quickstart/NativeGreetingResourceIT.java** file and verify that it includes the following content:

```
package org.acme.quickstart;


import io.quarkus.test.junit.NativeImageTest;

@NativeImageTest (1)
public class NativeGreetingResourceIT extends GreetingResourceTest { (2)

    // Run the same tests

}
```

(1) Use another test runner that starts the application from the native file before the tests. The executable is retrieved using the **native.image.path** system property configured in the *Failsafe Maven Plugin*.

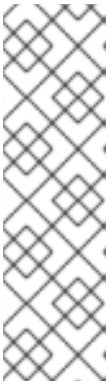
(2) This example extends the **GreetingResourceTest**, but you can also create a new test.

3. Run the test:

```
./mvnw verify -Pnative
```

The following example shows the output of this command:

```
./mvnw verify -Pnative
...
[getting-started-1.0-SNAPSHOT-runner:18820]     universe:     587.26 ms
[getting-started-1.0-SNAPSHOT-runner:18820]      (parse):   2,247.59 ms
[getting-started-1.0-SNAPSHOT-runner:18820]     (inline):   1,985.70 ms
[getting-started-1.0-SNAPSHOT-runner:18820]    (compile):  14,922.77 ms
[getting-started-1.0-SNAPSHOT-runner:18820]      compile:  20,361.28 ms
[getting-started-1.0-SNAPSHOT-runner:18820]        image:   2,228.30 ms
[getting-started-1.0-SNAPSHOT-runner:18820]        write:     364.35 ms
[getting-started-1.0-SNAPSHOT-runner:18820]      [total]:  52,777.76 ms
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.1:integration-test (default) @ getting-started ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.acme.quickstart.NativeGreetingResourceIT
Executing [/data/home/gsmet/git/quarkus-quickstarts/getting-started/target/getting-started-
1.0-SNAPSHOT-runner, -Dquarkus.http.port=8081, -Dtest.url=http://localhost:8081, -
Dquarkus.log.file.path=build/quarkus.log]
2019-04-15 11:33:20,348 INFO  [io.quarkus] (main) Quarkus 999-SNAPSHOT started in
0.002s. Listening on: http://[::]:8081
2019-04-15 11:33:20,348 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.387 s - in
org.acme.quickstart.NativeGreetingResourceIT
...
```



NOTE

Quarkus waits for 60 seconds for the native image to start before automatically failing the native tests. You can change this duration using the **quarkus.test.native-image-wait-time** system property.

You can extend the wait time using the following command where **<duration>** is the wait time in seconds:

```
./mvnw verify -Pnative -Dquarkus.test.native-image-wait-time=<duration>
```

4.1. EXCLUDING TESTS WHEN RUNNING AS A NATIVE EXECUTABLE

When you run tests against your native application, you can only interact with its HTTP endpoints. Tests do not run natively, therefore they cannot link against your application's code like they can when running on the JVM.

You can share your test class between JVM and native executions and exclude certain tests with the **@DisabledOnNativeImage** annotation to run them only on the JVM.

4.2. TESTING AN EXISTING NATIVE EXECUTABLE

You can test against the existing executable build. This allows you to run multiple sets of tests in stages on the binary after it has been build.

Procedure

- Run a test against an already built native executable:

```
./mvnw test-compile failsafe:integration-test
```

  This command runs the test against the existing native image using *Failsafe Maven Plugin*.

- Alternatively, you can specify the path to the native executable with the following command where **<path>** is the native image path:

```
./mvnw test-compile failsafe:integration-test -Dnative.image.path=<path>
```

CHAPTER 5. ADDITIONAL RESOURCES

- Testing your Quarkus applications
- Deploying your Quarkus applications to OpenShift
- Developing and compiling your Quarkus applications with Apache Maven
- Apache Maven Project
- The UBI Image Page
- The *UBI-minimal* Image Page
- The List of *UBI-minimal* Tags

Revised on 2021-06-30 13:18:34 UTC