

# Quarkus - Building a Native Executable

This guide covers:

- Compiling the application to a native executable
- Packaging the native executable in a container
- Debugging native executable

This guide takes as input the application developed in the [Getting Started Guide](#).

## GraalVM

Building a native executable requires using a distribution of GraalVM. There are three distributions: Oracle GraalVM Community Edition (CE), Oracle GraalVM Enterprise Edition (EE) and Mandrel. The differences between the Oracle and Mandrel distributions are as follows:

- Mandrel is a downstream distribution of the Oracle GraalVM CE. Mandrel's main goal is to provide a way to build native executables specifically designed to support Quarkus.
- Mandrel releases are built from a code base derived from the upstream Oracle GraalVM CE code base, with only minor changes but some significant exclusions that are not necessary for Quarkus native apps. They support the same capabilities to build native executables as Oracle GraalVM CE, with no significant changes to functionality. Notably, they do not include support for polyglot programming. The reason for these exclusions is to provide a better level of support for the majority of Quarkus users. These exclusions also mean Mandrel offers a considerable reduction in its distribution size when compared with Oracle GraalVM CE/EE.
- Mandrel is built slightly differently to Oracle GraalVM CE, using the standard OpenJDK project. This means that it does not profit from a few small enhancements that Oracle have added to the version of OpenJDK used to build their own GraalVM downloads. These enhancements are omitted because upstream OpenJDK does not manage them, and cannot vouch for. This is particularly important when it comes to conformance and security.
- Mandrel is currently only recommended for building native executables that target Linux containerized environments. This means that Mandrel users should use containers to build their native executables. If you are building native executables for macOS or Windows target platforms, you should consider using Oracle GraalVM instead, because Mandrel does not currently target these platforms. Building native executables directly on bare metal Linux is possible, with details available in the [Mandrel README](#) and [Mandrel releases](#).

The prerequisites vary slightly depending on whether you are using Oracle GraalVM CE/EE or Mandrel.



### *Install the Java 11 version of GraalVM*

While Oracle GraalVM is available for both Java 8 and Java 11 (Mandrel only supports Java 11), Quarkus only works with the Java 11 version. If you use the Oracle distribution, make sure to install the Java 11 version.

## Prerequisites for Mandrel

To complete this guide using Mandrel, you need:

- less than 15 minutes
- an IDE
- JDK 11 installed with `JAVA_HOME` configured appropriately
- A working container runtime (Docker, podman)
- The code of the application developed in the [Getting Started Guide](#).

Skip to [this section](#) to continue with the guide for Mandrel, and follow the Mandrel-specific instructions in that section.

## Prerequisites for Oracle GraalVM CE/EE

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 11 installed with `JAVA_HOME` configured appropriately
- A [working C development environment](#)
- GraalVM version 20.2.0 (be sure to install the Java 11 support) installed and [configured appropriately](#)
- A working container runtime (Docker, podman)
- The code of the application developed in the [Getting Started Guide](#).

### Supporting native compilation in C

What does having a working C developer environment mean?

- On Linux, you will need GCC, and the glibc and zlib headers. Examples for common distributions:

```
# dnf (rpm-based)
sudo dnf install gcc glibc-devel zlib-devel
libstdc++-static
# Debian-based distributions:
sudo apt-get install build-essential libz-dev
zlib1g-dev
```



- XCode provides the required dependencies on macOS:

```
xcode-select --install
```

- On Windows, you will need to install the [Visual Studio 2017 Visual C++ Build Tools](#)

## Configuring GraalVM



If you cannot install GraalVM, you can use a multi-stage Docker build to run Maven inside a Docker container that embeds GraalVM. There is an explanation of how to do this at the end of this guide.

Version 20.2.0 is required. Using the community edition is enough.

1. Install GraalVM (pick the java 11 version) if you haven't already. You have a few options for this:
  - Use platform-specific install tools like [homebrew](#), [sdkman](#), or [scoop](#).
  - Download the appropriate Community Edition archive from <https://github.com/graalvm/graalvm-ce-builds/releases>, and unpack it like you would any other JDK. Make sure to download and install at Java 11 version.
2. Configure the runtime environment. Set `GRAALVM_HOME` environment variable to the GraalVM installation directory, for example:

```
export GRAALVM_HOME=$HOME/Development/graalvm/
```

On macOS, point the variable to the `Home` sub-directory:

```
export GRAALVM_HOME=$HOME/Development/graalvm/Contents/Home/
```

On Windows, you will have to go through the Control Panel to set your environment variables.



Installing via scoop will do this for you.

3. Install the `native-image` tool using `gu install`:

```
`${GRAALVM_HOME}/bin/gu install native-image
```

Some previous releases of GraalVM included the `native-image` tool by default. This is no longer the case; it must be installed as a second step after GraalVM itself is installed. Note: there is an outstanding issue [using GraalVM with macOS Catalina](#).

4. (Optional) Set the `JAVA_HOME` environment variable to the GraalVM installation directory.

```
export JAVA_HOME=${GRAALVM_HOME}
```

5. (Optional) Add the GraalVM `bin` directory to the path

```
export PATH=${GRAALVM_HOME}/bin:$PATH
```

#### *Issues using GraalVM with macOS Catalina*

GraalVM binaries are not (yet) notarized for macOS Catalina as reported in this [GraalVM issue](#). This means that you may see the following error when using `gu`:



```
“gu” cannot be opened because the developer cannot be verified
```

Use the following command to recursively delete the `com.apple.quarantine` extended attribute on the GraalVM install directory as a workaround:

```
xattr -r -d com.apple.quarantine ${GRAALVM_HOME}/../..
```

## Solution

We recommend that you follow the instructions in the next sections and package the application step by step. However, you can go right to the completed example.

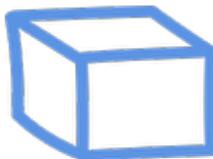
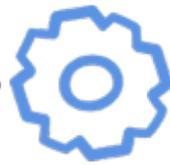
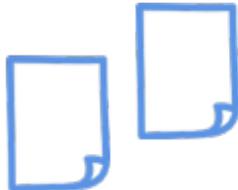
Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `getting-started` directory.

# Producing a native executable

The native executable for our application will contain the application code, required libraries, Java APIs, and a reduced version of a VM. The smaller VM base improves the startup time of the application and produces a minimal disk footprint.

(1) TAKE YOUR APPLICATION



(3) RUN IT!



(4) USE IT!

(2) PRODUCE A NATIVE EXECUTABLE

If you have generated the application from the previous tutorial, you can find in the `pom.xml` the following *profile*:

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



You can provide custom options for the `native-image` command using the `<quarkus.native.additional-build-args>` property. Multiple options may be separated by a comma.

Another possibility is to include the `quarkus.native.additional-build-args` configuration property in your `application.properties`.

You can find more information about how to configure the native image building process in the [Configuring the Native Executable](#) section below.

We use a profile because, you will see very soon, packaging the native executable takes a *few* minutes.

You could just pass `-Dquarkus.package.type=native` as a property on the command line, however it is better to use a profile as this allows native image tests to also be run.

Create a native executable using: `./mvnw package -Pnative`.



#### *Issues with packaging on Windows*

The Microsoft Native Tools for Visual Studio must first be initialized before packaging. You can do this by starting the `x64 Native Tools Command Prompt` that was installed with the Visual Studio Build Tools. At `x64 Native Tools Command Prompt` you can navigate to your project folder and run `mvnw package -Pnative`.

Another solution is to write a script to do this for you:

```
cmd /c 'call "C:\Program Files (x86)\Microsoft Visual Studio\2017\BuildTools\VC\Auxiliary\Build\vcvars64.bat"
&& mvn package -Pnative'
```

In addition to the regular files, the build also produces `target/getting-started-1.0-SNAPSHOT-runner`. You can run it using: `./target/getting-started-1.0-SNAPSHOT-runner`.

## Testing the native executable

Producing a native executable can lead to a few issues, and so it's also a good idea to run some tests against the application running in the native file.

In the `pom.xml` file, the `native` profile contains:

```

<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.final
Name}-runner</native.image.path>

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

```

This instructs the failsafe-maven-plugin to run integration-test and indicates the location of the produced native executable.

Then, `open` the `src/test/java/org/acme/quickstart/NativeGreetingResourceIT.java`. It contains:

```

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 We extend our previous tests, but you can also implement your tests

To see the `NativeGreetingResourceIT` run against the native executable, use `./mvnw verify -Pnative`:

```
./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
...
```



By default, Quarkus waits for 60 seconds for the native image to start before automatically failing the native tests. This duration can be changed using the `quarkus.test.native-image-wait-time` system property. For example, to increase the duration to 300 seconds, use: `./mvnw verify -Pnative -Dquarkus.test.native-image-wait-time=300`.

By default, native tests runs using the `prod` profile. This can be overridden using the `quarkus.test.native-image-profile` property. For example, in your `application.properties` file, add: `quarkus.test.native-image-profile=test`. Alternatively, you can run your tests with: `./mvnw verify -Pnative -Dquarkus.test.native-image-profile=test`. However, don't forget that when the native executable is built the `prod` profile is enabled. So, the profile you enable this way must be compatible with the produced executable.

## Excluding tests when running as a native executable

When running tests this way, the only things that actually run natively are you application endpoints, which you can only test via HTTP calls. Your test code does not actually run natively, so if you are testing code that does not call your HTTP endpoints, it's probably not a good idea to run them as part of native tests.

If you share your test class between JVM and native executions like we advise above, you can mark certain tests with the `@DisabledOnNativeImage` annotation in order to only run them on the JVM.

## Testing an existing native executable

It is also possible to re-run the tests against a native executable that has already been built. To do this run `./mvnw test-compile failsafe:integration-test`. This will discover the existing native image and run the tests against it using failsafe.

If the process cannot find the native image for some reason, or you want to test a native image that is no longer in the target directory you can specify the executable with the `-Dnative.image.path=` system property.

## Creating a Linux executable without GraalVM installed



Before going further, be sure to have a working container runtime (Docker, podman) environment. If you use Docker on Windows you should share your project's drive at Docker Desktop file share settings and restart Docker Desktop.

Quite often one only needs to create a native Linux executable for their Quarkus application (for example in order to run in a containerized environment) and would like to avoid the trouble of installing the proper GraalVM version in order to accomplish this task (for example, in CI environments it's common practice to install as little software as possible).

To this end, Quarkus provides a very convenient way of creating a native Linux executable by

leveraging a container runtime such as Docker or podman. The easiest way of accomplishing this task is to execute:

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



You can also select the container runtime to use with:

```
# Docker
./mvnw package -Pnative -Dquarkus.native.container
-runtime=docker
# Podman
./mvnw package -Pnative -Dquarkus.native.container
-runtime=podman
```

These are normal Quarkus config properties, so if you always want to build in a container it is recommended you add these to your `application.properties` in order to avoid specifying them every time.

Building with Mandrel requires a custom builder image parameter to be passed additionally:



```
./mvnw package -Pnative -Dquarkus.native.container
-build=true -Dquarkus.native.builder
-image=quay.io/quarkus/ubi-quarkus-mandrel:{mandrel
-flavor}
```

Please note that the above command points to a floating tag. It is highly recommended to use the floating tag, so that your builder image remains up-to-date and secure. If you absolutely must, you may hard-code to a specific tag (see [here](#) for available tags), but be aware that you won't get security updates that way and it's unsupported.

## Creating a container

### Using the container-image extensions

By far the easiest way to create a container-image from your Quarkus application is to leverage one of the container-image extensions.

If one of those extensions is present, then creating a container image for the native executable is essentially a matter of executing a single command:

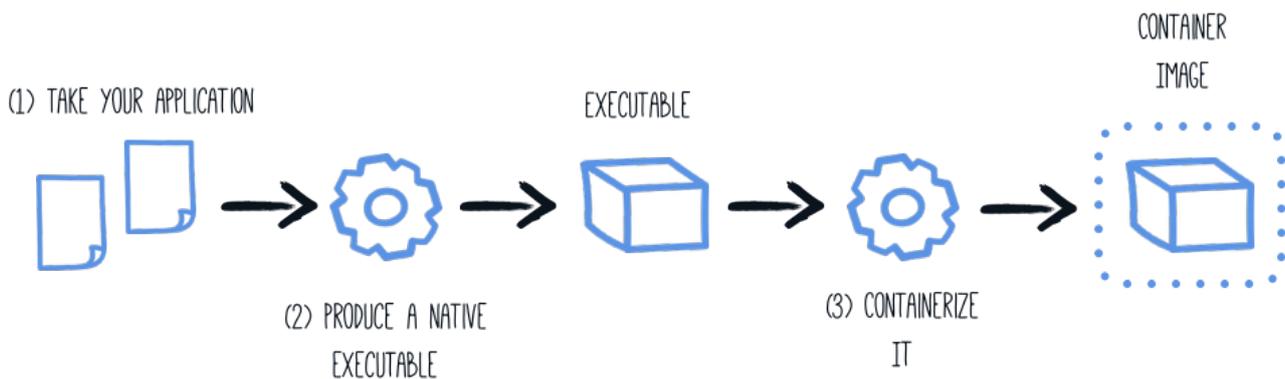
```
./mvnw package -Pnative -Dquarkus.native.container-build=true
-Dquarkus.container-image.build=true
```

- `quarkus.native.container-build=true` allows for creating a Linux executable without GraalVM being installed (and is only necessary if you don't have GraalVM installed locally or your local operating system is not Linux)
- `quarkus.container-image.build=true` instructs Quarkus to create a container-image using the final application artifact (which is the native executable in this case)

See the [Container Image guide](#) for more details.

## Manually

You can run the application in a container using the JAR produced by the Quarkus Maven Plugin. However, in this section we focus on creating a container image using the produced native executable.



When using a local GraalVM installation, the native executable targets your local operating system (Linux, macOS, Windows etc). However, as a container may not use the same *executable* format as the one produced by your operating system, we will instruct the Maven build to produce an executable by leveraging a container runtime (as described in [this section](#)):

The produced executable will be a 64 bit Linux executable, so depending on your operating system it may no longer be runnable. However, it's not an issue as we are going to copy it to a container. The project generation has provided a `Dockerfile.native` in the `src/main/docker` directory with the following content:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY target/*-runner /work/application
RUN chmod 775 /work
EXPOSE 8080
CMD ["/application", "-Dquarkus.http.host=0.0.0.0"]
```

*Ubi?*

The provided **Dockerfiles** use **UBI** (Universal Base Image) as parent image. This base image has been tailored to work perfectly in containers. The **Dockerfiles** use the *minimal version* of the base image to reduce the size of the produced image.



You can read more about UBI on:

- [the UBI image page](#)
- [the \*UBI-minimal\* image page](#)
- [the list of \*UBI-minimal\* tags](#)

Then, if you didn't delete the generated native executable, you can build the docker image with:

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

And finally, run it with:

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



If you are interested in tiny Docker images, check the [distroless](#) version.

## Using a multi-stage Docker build

The previous section showed you how to build a native executable using Maven, but implicitly required that the proper GraalVM version be installed on the building machine (be it your local machine or your CI/CD infrastructure).

In cases where the GraalVM requirement cannot be met, you can use Docker to perform the Maven build by using a multi-stage Docker build. A multi-stage Docker build is like two Dockerfile files combined in one, the first is used to build the artifact used by the second.

In this guide we will use the first stage to generate the native executable using Maven and the second stage to create our runtime image.

```

## Stage 1 : build with maven builder image with native
capabilities
FROM quay.io/quarkus/centos-quarkus-maven:20.2.0-java11 AS build
COPY pom.xml /usr/src/app/
RUN mvn -f /usr/src/app/pom.xml -B de.qaware.maven:go-offline-
maven-plugin:1.2.5:resolve-dependencies
COPY src /usr/src/app/src
USER root
RUN chown -R quarkus /usr/src/app
USER quarkus
RUN mvn -f /usr/src/app/pom.xml -Pnative clean package

## Stage 2 : create the docker final image
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY --from=build /usr/src/app/target/*-runner /work/application

# set up permissions for user `1001`
RUN chmod 775 /work /work/application \
  && chown -R 1001 /work \
  && chmod -R "g+rwX" /work \
  && chown -R 1001:root /work

EXPOSE 8080
USER 1001

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

```

Save this file in `src/main/docker/Dockerfile.multistage` as it is not included in the getting started quickstart.



Before launching our Docker build, we need to update the default `.dockerignore` file as it filters everything except the `target` directory and as we plan to build inside a container we need to be able to copy the `src` directory. So edit your `.dockerignore` and remove or comment its content.

```

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

```

And finally, run it with:

```

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

```



If you need SSL support in your native executable, you can easily include the necessary libraries in your Docker image.

Please see [our Using SSL With Native Executables guide](#) for more information.

## Debugging native executable

Starting with Oracle GraalVM 20.2 or Mandrel 20.1, debug symbols for native executables can be generated for Linux environments (Windows support is still under development). These symbols can be used to debug native executables with tools such as `gdb`.

To generate debug symbols, add `-Dquarkus.native.debug.enabled=true` flag when generating the native executable. You will find the debug symbols for the native executable in a `.debug` file next to the native executable.

Aside from debug symbols, setting `-Dquarkus.native.debug.enabled=true` flag generates a cache of source files for any JDK runtime classes, GraalVM classes and application classes resolved during native executable generation. This source cache is useful for native debugging tools, to establish the link between the symbols and matching source code. It provides a convenient way of making just the necessary sources available to the debugger/IDE when debugging a native executable.

Sources for third party jar dependencies, including Quarkus source code, are not added to the source cache by default. To include those, make sure you invoke `mvn dependency:sources` first. This step is required in order to pull the sources for these dependencies, and get them included in the source cache.

The location of source cache is `target/sources-cache` folder.

## Configuring the Native Executable

There are a lot of different configuration options that can affect how the native executable is generated. These are provided in `application.properties` the same as any other config property.

The properties are shown below:

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.native.additional-build-args</code> Additional arguments to pass to the build process	list of string	

<p> <code>quarkus.native.enable-http-url-handler</code></p> <p>If the HTTP url handler should be enabled, allowing you to do <code>URL.openConnection()</code> for HTTP URLs</p>	boolean	<code>true</code>
<p> <code>quarkus.native.enable-https-url-handler</code></p> <p>If the HTTPS url handler should be enabled, allowing you to do <code>URL.openConnection()</code> for HTTPS URLs</p>	boolean	<code>false</code>
<p> <code>quarkus.native.enable-all-security-services</code></p> <p>If all security services should be added to the native image</p>	boolean	<code>false</code>
<p> <code>quarkus.native.add-all-charsets</code></p> <p>If all character sets should be added to the native image. This increases image size</p>	boolean	<code>false</code>
<p> <code>quarkus.native.graalvm-home</code></p> <p>The location of the Graal distribution</p>	string	<code>\${GRAALVM_HOME:}</code>
<p> <code>quarkus.native.java-home</code></p> <p>The location of the JDK</p>	File	<code>\${java.home}</code>
<p> <code>quarkus.native.native-image-xxmx</code></p> <p>The maximum Java heap to be used during the native image generation</p>	string	
<p> <code>quarkus.native.debug-build-process</code></p> <p>If the native image build should wait for a debugger to be attached before running. This is an advanced option and is generally only intended for those familiar with GraalVM internals</p>	boolean	<code>false</code>
<p> <code>quarkus.native.publish-debug-build-process-port</code></p> <p>If the debug port should be published when building with docker and <code>debug-build-process</code> is true</p>	boolean	<code>true</code>
<p> <code>quarkus.native.cleanup-server</code></p> <p>If the native image server should be restarted</p>	boolean	<code>false</code>

<p> <code>quarkus.native.enable-isolates</code></p> <p>If isolates should be enabled</p>	boolean	<code>true</code>
<p> <code>quarkus.native.enable-fallback-images</code></p> <p>If a JVM based 'fallback image' should be created if native image fails. This is not recommended, as this is functionally the same as just running the application in a JVM</p>	boolean	<code>false</code>
<p> <code>quarkus.native.enable-server</code></p> <p>If the native image server should be used. This can speed up compilation but can result in changes not always being picked up due to cache invalidation not working 100%</p>	boolean	<code>false</code>
<p> <code>quarkus.native.auto-service-loader-registration</code></p> <p>If all META-INF/services entries should be automatically registered</p>	boolean	<code>false</code>
<p> <code>quarkus.native.dump-proxies</code></p> <p>If the bytecode of all proxies should be dumped for inspection</p>	boolean	<code>false</code>
<p> <code>quarkus.native.container-build</code></p> <p>If this build should be done using a container runtime. If this is set docker will be used by default, unless container-runtime is also set.</p>	boolean	<code>false</code>
<p> <code>quarkus.native.builder-image</code></p> <p>The docker image to use to do the image build</p>	string	<code>quay.io/quarkus/ubi-quarkus-native-image:20.2.0-java11</code>
<p> <code>quarkus.native.container-runtime</code></p> <p>The container runtime (e.g. docker) that is used to do an image based build. If this is set then a container build is always done.</p>	string	

<p> <code>quarkus.native.container-runtime-options</code></p> <p>Options to pass to the container runtime</p>	list of string	
<p> <code>quarkus.native.enable-vm-inspection</code></p> <p>If the resulting image should allow VM introspection</p>	boolean	<code>false</code>
<p> <code>quarkus.native.full-stack-traces</code></p> <p>If full stack traces are enabled in the resulting image</p>	boolean	<code>true</code>
<p> <code>quarkus.native.enable-reports</code></p> <p>If the reports on call paths and included packages/classes/methods should be generated</p>	boolean	<code>false</code>
<p> <code>quarkus.native.report-exception-stack-traces</code></p> <p>If exceptions should be reported with a full stack trace</p>	boolean	<code>true</code>
<p> <code>quarkus.native.report-errors-at-runtime</code></p> <p>If errors should be reported at runtime. This is a more relaxed setting, however it is not recommended as it means your application may fail at runtime if an unsupported feature is used by accident.</p>	boolean	<code>false</code>

<p> <code>quarkus.native.resources.includes</code></p> <p>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. By default, no resources are included. Example: Given that you have <code>src/main/resources/ignored.png</code> and <code>src/main/resources/foo/selected.png</code> in your source tree and one of your dependency JARs contains <code>bar/some.txt</code> file, with the following configuration <code>quarkus.native.resources.includes = foo/**,bar/**/*.*</code> the files <code>src/main/resources/foo/selected.png</code> and <code>bar/some.txt</code> will be included in the native image, while <code>src/main/resources/ignored.png</code> will not be included. Supported glob features</p> <p>Feature Description</p> <ul style="list-style-type: none"> <li><code>*</code> Matches a (possibly empty) sequence of characters that does not contain slash (/)</li> <li><code>**</code> Matches a (possibly empty) sequence of characters that may contain slash (/)</li> <li><code>?</code> Matches one character, but not slash</li> <li><code>[abc]</code> Matches one character given in the bracket, but not slash</li> <li><code>[a-z]</code> Matches one character from the range given in the bracket, but not slash</li> <li><code>[!abc]</code> Matches one character not named in the bracket; does not match slash</li> <li><code>[a-z]</code> Matches one character outside the range given in the bracket; does not match slash</li> <li><code>{one,two,three}</code> Matches any of the alternating tokens separated by comma; the tokens may contain wildcards, nested alternations and ranges</li> <li><code>\</code> The escape character</li> </ul> <p>Note that there are three levels of escaping when passing this option via <code>application.properties</code>:  <code>application.properties</code> parser - MicroProfile Config list converter that splits the comma separated list - Glob parser  All three levels use backslash (\) as the escaping character. So you need to use an appropriate number of backslashes depending on which level you want to escape. Note that Quarkus extensions typically include the resources they require by themselves. This option is useful in situations when the built-in functionality is not sufficient.</p>	list of string	
<p> <code>quarkus.native.debug.enabled</code></p> <p>If debug is enabled and debug symbols are generated. The symbols will be generated in a separate <code>.debug</code> file.</p>	boolean	<code>false</code>

## What's next?

This guide covered the creation of a native (binary) executable for your application. It provides an application exhibiting a swift startup time and consuming less memory. However, there is much more.

We recommend continuing the journey with the [deployment to Kubernetes and OpenShift](#).