Quarkus - Container Images

Quarkus provides extensions for building (and pushing) container images. Currently it supports:

- Jib

- Docker

- S2I

Container Image extensions

Jib

The extension `quarkus-container-image-jib` is powered by Jib for performing container image builds. The major benefit of using Jib with Quarkus is that all the dependencies (everything found under `target/lib`) are cached in a different layer than the actual application making rebuilds really fast and small (when it comes to pushing). Another important benefit of using this extension is that it provides the ability to create a container image without having to have any dedicated client side tooling (like Docker) or running daemon processes (like the Docker daemon) when all that is needed is the ability to push to a container image registry.

To use this feature, add the following extension to your project:

```
./mvnw quarkus:add-extension -Dextensions="container-image-jib"
```

> ℹ️ In situations where all that is needed to build a container image and no push to a registry is necessary (essentially by having set `quarkus.container-image.build=true` and left `quarkus.container-image.push` unset - it defaults to `false`), then this extension creates a container image and registers it with the Docker daemon. This means that although Docker isn't used to build the image, it is nevertheless necessary. Also note that using this mode, the built container image **will** show up when executing `docker images`.

Including extra files

There are cases when additional files (other than ones produced by the Quarkus build) need to be added to a container image. To support these cases, Quarkus copies any file under `src/main/jib` into the built container image (which is essentially the same idea that the Jib Maven and Gradle plugins support). For example, the presence of `src/main/jib/foo/bar` would result in `/foo/bar` being added into the container filesystem.

Docker

The extension `quarkus-container-image-docker` is using the Docker binary and the generated Dockerfiles under `src/main/docker` in order to perform Docker builds.

To use this feature, add the following extension to your project.

```
./mvnw quarkus:add-extension -Dextensions="container-image-docker"
```

S2I

The extension `quarkus-container-image-s2i` is using S2I binary builds in order to perform container builds inside the OpenShift cluster. The idea behind the binary build is that you just upload the artifact and its dependencies to the cluster and during the build they will be merged to a builder image (defaults to `fabric8/s2i-java`).

The benefit of this approach, is that it can be combined with OpenShift's `DeploymentConfig` that makes it easy to roll out changes to the cluster.

To use this feature, add the following extension to your project.

```
./mvnw quarkus:add-extension -Dextensions="container-image-s2i"
```

S2I builds require creating a `BuildConfig` and two `ImageStream` resources, one for the builder image and one for the output image. The creation of such objects is being taken care of by the Quarkus Kubernetes extension.

Building

To build a container image for your project, `quarkus.container-image.build=true` needs to be set using any of the ways that Quarkus supports.

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

Pushing

To push a container image for your project, `quarkus.container-image.push=true` needs to be set using any of the ways that Quarkus supports.

```
./mvnw clean package -Dquarkus.container-image.push=true
```

i If no registry is set (using `quarkus.container-image.registry`) then `docker.io` will be used as the default.

Selecting among multiple extensions

It does not make sense to use multiple extension as part of the same build. When multiple container image extensions are present, an error will be raised to inform the user. The user can either remove the uneeded extensions or select one using `application.properties`.

For example, if both `container-image-docker` and `container-image-s2i` are present and the user needs to use `container-image-docker`:

```
quarkus.container-image.builder=docker
```

Customizing

The following properties can be used to customize the container image build process.

Container Image Options

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

Configuration property	Type	Default
🔒 quarkus.container-image.group The group the container image will be part of	string	`${user.name}`
🔒 quarkus.container-image.name The name of the container image. If not set defaults to the application name	string	`${quarkus.application.name:unset}`
🔒 quarkus.container-image.tag The tag of the container image. If not set defaults to the application version	string	`${quarkus.application.version:latest}`

🔒 `quarkus.container-image.additional-tags` Additional tags of the container image.	list of string	
🔒 `quarkus.container-image.registry` The container registry to use	string	
🔒 `quarkus.container-image.username` The username to use to authenticate with the registry where the built image will be pushed	string	
🔒 `quarkus.container-image.password` The password to use to authenticate with the registry where the built image will be pushed	string	
🔒 `quarkus.container-image.insecure` Whether or not insecure registries are allowed	boolean	`false`
🔒 `quarkus.container-image.build` Whether or not a image build will be performed.	boolean	`false`
🔒 `quarkus.container-image.push` Whether or not an image push will be performed.	boolean	`false`
🔒 `quarkus.container-image.builder` The name of the container image extension to use (e.g. docker, jib, s2i). The option will be used in case multiple extensions are present.	string	

Jib Options

In addition to the generic container image options, the `container-image-jib` also provides the following options:

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

Configuration property	Type	Default

🔒 `quarkus.jib.base-jvm-image` The base image to be used when a container image is being produced for the jar build	string	`fabric8/java-alpine-openjdk11-jre`
🔒 `quarkus.jib.base-native-image` The base image to be used when a container image is being produced for the native binary build	string	`registry.access.redhat.com/ubi8/ubi-minimal`
🔒 `quarkus.jib.jvm-arguments` Additional JVM arguments to pass to the JVM when starting the application	list of string	`-Dquarkus.http.host=0.0.0.0,-Djava.util.logging.manager=org.jboss.logmanager.LogManager`
🔒 `quarkus.jib.native-arguments` Additional arguments to pass when starting the native application	list of string	`-Dquarkus.http.host=0.0.0.0`

🔒 `quarkus.jib.jvm-entrypoint` If this is set, then it will be used as the entry point of the container image. There are a few things to be aware of when creating an entry point - A valid entrypoint is jar package specific (see `quarkus.package.type`) - A valid entrypoint depends on the location of both the launching scripts and the application jar file. To that end it's helpful to remember that when `fast-jar` packaging is used, all necessary application jars are added to the `/work` directory and that the same directory is also used as the working directory. When `legacy` or `uber-jar` are used, the application jars are unpacked under the `/app` directory and that directory is used as the working directory. - Even if the `jvmArguments` field is set, it is ignored completely When this is not set, a proper default entrypoint will be constructed. As a final note, a very useful tool for inspecting container image layers that can greatly aid when debugging problems with endpoints is dive	list of string	
🔒 `quarkus.jib.native-entrypoint` If this is set, then it will be used as the entry point of the container image. There are a few things to be aware of when creating an entry point - A valid entrypoint depends on the location of both the launching scripts and the native binary file. To that end it's helpful to remember that the native application is added to the `/work` directory and that and the same directory is also used as the working directory - Even if the `nativeArguments` field is set, it is ignored completely When this is not set, a proper default entrypoint will be constructed. As a final note, a very useful tool for inspecting container image layers that can greatly aid when debugging problems with endpoints is dive	list of string	
🔒 `quarkus.jib.base-registry-username` The username to use to authenticate with the registry used to pull the base JVM image	string	
🔒 `quarkus.jib.base-registry-password` The password to use to authenticate with the registry used to pull the base JVM image	string	
🔒 `quarkus.jib.environment-variables` Environment variables to add to the container image	`Map<String,String>`	required ❗
🔒 `quarkus.jib.labels` Custom labels to add to the generated image	`Map<String,String>`	required ❗

Docker Options

In addition to the generic container image options, the `container-image-docker` also provides the following options:

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

Configuration property	Type	Default
🔒 `quarkus.docker.dockerfile-jvm-path` Path to the the JVM Dockerfile. If not set ${project.root}/src/main/docker/Dockerfile.jvm will be used If set to an absolute path then the absolute path will be used, otherwise the path will be considered relative to the project root	string	
🔒 `quarkus.docker.dockerfile-native-path` Path to the the JVM Dockerfile. If not set ${project.root}/src/main/docker/Dockerfile.native will be used If set to an absolute path then the absolute path will be used, otherwise the path will be considered relative to the project root	string	
🔒 `quarkus.docker.cache-from` Images to consider as cache sources. Values are passed to `docker build` via the `cache-from` option	list of string	
🔒 `quarkus.docker.build-args` Build args passed to docker via `--build-arg`	`Map<String,String>`	required ❗

S2I Options

In addition to the generic container image options, the `container-image-s2i` also provides the following options:

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

Configuration property	Type	Default

🔒 `quarkus.s2i.base-jvm-image` The base image to be used when a container image is being produced for the jar build	string	`registry.access.redhat.com/ubi8/openjdk-11`
🔒 `quarkus.s2i.base-native-image` The base image to be used when a container image is being produced for the native binary build	string	`quay.io/quarkus/ubi-quarkus-native-binary-s2i:1.0`
🔒 `quarkus.s2i.jvm-arguments` Additional JVM arguments to pass to the JVM when starting the application	list of string	`-Dquarkus.http.host=0.0.0.0,-Djava.util.logging.manager=org.jboss.logmanager.LogManager`
🔒 `quarkus.s2i.native-arguments` Additional arguments to pass when starting the native application	list of string	`-Dquarkus.http.host=0.0.0.0`
🔒 `quarkus.s2i.jar-directory` The directory where the jar is added during the assemble phase. This is dependent on the S2I image and should be supplied if a non default image is used.	string	`/deployments/`

🔒 `quarkus.s2i.jar-file-name` The resulting filename of the jar in the S2I image. This option may be used if the selected S2I image uses a fixed name for the jar.	string	
🔒 `quarkus.s2i.native-binary-directory` The directory where the native binary is added during the assemble phase. This is dependent on the S2I image and should be supplied if a non-default image is used.	string	`/home/quarkus/`
🔒 `quarkus.s2i.native-binary-file-name` The resulting filename of the native binary in the S2I image. This option may be used if the selected S2I image uses a fixed name for the native binary.	string	
🔒 `quarkus.s2i.build-timeout` The build timeout.	Duration ❓	`PT5M`

ℹ️ *About the Duration format*

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.