

Quarkus - Deploying on OpenShift

This guide covers generating and deploying OpenShift resources based on sane default and user supplied configuration.

Prerequisites

To complete this guide, you need:

- roughly 5 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- access to an OpenShift cluster (Minishift is a viable option)
- OpenShift CLI (Optional. Only required for manually deploying)

Creating the Maven project

First, we need a new project that contains the OpenShift extension. This can be done using the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.2.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=openshift-quickstart \
  -DclassName="org.acme.rest.GreetingResource" \
  -Dpath="/greeting" \
  -Dextensions="openshift"

cd openshift-quickstart
```

OpenShift

Quarkus offers the ability to automatically generate OpenShift resources based on sane default and user supplied configuration. The OpenShift extension is actually a wrapper extension that brings together the [kubernetes](#) and [container-image-s2i](#) extensions with sensible defaults so that it's easier for the user to get started with Quarkus on OpenShift.

When we added the OpenShift extension to the command line invocation above, the following dependency was added to the `pom.xml`

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-openshift</artifactId>
</dependency>
```

By adding this dependency, we now have the ability to configure the OpenShift resource generation and application using the usual `application.properties` approach that Quarkus provides. The configuration items that are available can be found in: `io.quarkus.kubernetes.deployment.OpenShiftConfig` class. Furthermore, the items provided by `io.quarkus.deployment.ApplicationConfig` affect the OpenShift resources.

Build and Deploy (in separate steps)

If the OpenShift extension was not included during the bootstrapping of the project nor was it added subsequently (check `pom.xml` file for it), then it can be added like this:

```
./mvnw quarkus:add-extension -Dextensions="openshift"
```

Building is handled by the `container-image-s2i` extension. To trigger a build:

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

The build that will be performed is an s2i binary build. The input of the build is the jar that has been built locally and the output of the build is an `ImageStream` that is configured to automatically trigger a deployment.

To deploy the container image created in the above step to OpenShift, follow these commands:

```
oc get is
oc new-app --name=greeting <project>/openshift-quickstart:1.0-SNAPSHOT
oc get svc
oc expose svc/greeting
oc get routes
curl http://<route>/greeting
```

In the above, `oc get is` will list the image stream created. It should be tagged as `<project>/openshift-quickstart:1.0-SNAPSHOT`.

Similarly, `oc get route` will list the route URL for the exposed service "greeting" so that you can use it to test the application.

Non S2i Builds

Out of the box the openshift extension is configured to use `container-image-s2i`. However, it's still possible to use other container image extensions like:

- `container-image-docker`
- `container-image-jib`

When a non-s2i container image extension is used, an `ImageStream` is created that is pointing to an external `dockerImageRepository`. The image is built and pushed to the registry and the `ImageStream` populates the tags that are available in the `dockerImageRepository`.

To select which extension will be used for building the image:

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

or

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

Build and Deploy (in a single step)

If the OpenShift extension was not included during the bootstrapping of the project nor was it added subsequently (check `pom.xml` file for it), then it can be added like this:

```
./mvnw quarkus:add-extension -Dextensions="openshift"
```

To trigger a build and deployment in a single step:

```
./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

The aforementioned command will build a jar file locally, trigger a container image build and then apply the generated OpenShift resources. The generated resources are using OpenShift's `DeploymentConfig` that is configured to automatically trigger a redeployment when a change in the `ImageStream` is noticed. In other words, any container image build after the initial deployment will automatically trigger redeployment, without the need to delete, update or re-apply the generated resources.

To confirm the above command has created an image stream, a service resource and has deployed the application (has a pod running), apply these commands:

```
oc get is
oc get pods
oc get svc
```

To expose the created service to a route and test it:

```
oc expose svc/greeting
oc get routes
curl http://<route>/greeting
```

Customizing

All available customization options are available in the [OpenShift configuration options](#).

Some examples are provided in the sections below:

Exposing Routes

To expose a **Route** for the Quarkus application:

```
quarkus.openshift.expose=true
```

Tip: You don't necessarily need to add this property in the **application.properties**. You can pass it as a command line argument:

```
./mvnw clean package -Dquarkus.openshift.expose=true
```

The same applies to all properties listed below.

Labels

To add a label in the generated resources:

```
quarkus.openshift.labels.foo=bar
```

Annotations

To add an annotation in the generated resources:

```
quarkus.openshift.annotations.foo=bar
```

Environment variables

To add an annotation in the generated resources:

```
quarkus.openshift.env-vars.my-env-var.value=foobar
```

The command above will add `MY_ENV_VAR=foobar` as an environment variable. Please note that the key `my-env-var` will be converted to uppercase and dashes will be replaced by underscores resulting in `MY_ENV_VAR`.

You may also noticed that in contrast to labels, and annotations for environment variables you don't just use a key=value approach. That is because for environment variables there are additional options rather than just value.

Environment variables from Secret

To add all key value pairs of a `Secret` as environment variables:

```
quarkus.openshift.env-vars.my-env-var.secret=my-secret
```

Environment variables from ConfigMap

To add all key value pairs of a `ConfigMap` as environment variables:

```
quarkus.openshift.env-vars.my-env-var.configmap=my-secret
```

Mounting volumes

The OpenShift extension allows the user to configure both volumes and mounts for the application.

Any volume can be mounted with a simple configuration:

```
quarkus.openshift.mounts.my-volume.path=/where/to/mount
```

This will add a mount to my pod for volume `my-volume` to path `/where/to/mount`

The volumes themselves can be configured as shown in the sections below:

Secret volumes

```
quarkus.openshift.secret-volumes.my-volume.secret-name=my-secret
```

ConfigMap volumes

```
quarkus.openshift.config-map-volumes.my-volume.config-map-name=my-secret
```

Knative - OpenShift Serverless

OpenShift also provides the ability to use Knative via the [OpenShift Serverless](#) functionality.

The first order of business is to instruct Quarkus to generate Knative resources by setting:

```
quarkus.kubernetes.deployment-target=knative
```

In order to leverage OpenShift S2I to build the container image on the cluster and use the resulting container image for the Knative application, we need to set a couple of configuration properties:

```
# set the Kubernetes namespace which will be used to run the application
quarkus.container-image.group=geoand
# set the container image registry - this is the standard URL used to refer to the internal OpenShift registry
quarkus.container-image.registry=image-registry.openshift-image-registry.svc:5000
```

The application can then be deployed to OpenShift Serverless by enabling the standard `quarkus.kubernetes.deploy=true` property.