

Quarkus - Kubernetes extension

This guide covers generating and deploying Kubernetes resources based on sane defaults and user supplied configuration. In detail, it supports generating resources for [Kubernetes](#), [OpenShift](#) and [Knative](#). Also it supports automatically [Deploying](#) these resources to the target platform.

Prerequisites

To complete this guide, you need:

- roughly 10 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- access to a Kubernetes or cluster (Minikube is a viable options)

Creating the Maven project

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

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=kubernetes-quickstart \
  -DclassName="org.acme.rest.GreetingResource" \
  -Dpath="/greeting" \
  -Dextensions="kubernetes"
cd kubernetes-quickstart
```

Kubernetes

Quarkus offers the ability to automatically generate Kubernetes resources based on sane defaults and user supplied configuration. The implementation that takes care of generating the actual Kubernetes resources is provided by [dekorate](#). Currently it supports the generation of resources for vanilla Kubernetes and OpenShift.

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

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

By adding this dependency, we enable the generation of Kubernetes manifests each time we perform a build. For example, following the execution of `./mvnw package` you will notice amongst the other files that are created, two files named `kubernetes.json` and `kubernetes.yml` in the `target/kubernetes/` directory.

If you look at either file you will see that it contains both a Kubernetes `Deployment` and a `Service`.

The full source of the `kubernetes.json` file looks something like this:

```
{
  {
    "apiVersion" : "apps/v1",
    "kind" : "Deployment",
    "metadata" : {
      "annotations": {
        "app.quarkus.io/vcs-url" : "<some url>",
        "app.quarkus.io/commit-id" : "<some git SHA>",
      },
      "labels" : {
        "app.kubernetes.io/name" : "test-quarkus-app",
        "app.kubernetes.io/version" : "1.0-SNAPSHOT",
      },
      "name" : "test-quarkus-app"
    },
    "spec" : {
      "replicas" : 1,
      "selector" : {
        "matchLabels" : {
          "app.kubernetes.io/name" : "test-quarkus-app",
          "app.kubernetes.io/version" : "1.0-SNAPSHOT",
        }
      },
      "template" : {
        "metadata" : {
          "labels" : {
            "app.kubernetes.io/name" : "test-quarkus-app",
            "app.kubernetes.io/version" : "1.0-SNAPSHOT"
          }
        },
        "spec" : {
          "containers" : [ {
            "env" : [ {
```

```

        "name" : "KUBERNETES_NAMESPACE",
        "valueFrom" : {
            "fieldRef" : {
                "fieldPath" : "metadata.namespace"
            }
        }
    } ],
    "image" : "yourDockerUsername/test-quarkus-app:1.0-
SNAPSHOT",
    "imagePullPolicy" : "IfNotPresent",
    "name" : "test-quarkus-app"
} ]
}
}
},
{
    "apiVersion" : "v1",
    "kind" : "Service",
    "metadata" : {
        "annotations": {
            "app.quarkus.io/vcs-url" : "<some url>",
            "app.quarkus.io/commit-id" : "<some git SHA>",
        },
        "labels" : {
            "app.kubernetes.io/name" : "test-quarkus-app",
            "app.kubernetes.io/version" : "1.0-SNAPSHOT",
        },
        "name" : "test-quarkus-app"
    },
    "spec" : {
        "ports" : [ {
            "name" : "http",
            "port" : 8080,
            "targetPort" : 8080
        } ],
        "selector" : {
            "app.kubernetes.io/name" : "test-quarkus-app",
            "app.kubernetes.io/version" : "1.0-SNAPSHOT"
        },
        "type" : "ClusterIP"
    }
}
}
}

```

An important thing to note about the `Deployment` is that it uses `yourDockerUsername/test-quarkus-app:1.0-SNAPSHOT` as the Docker image of the `Pod`. The name of the image is controlled by the `container-image` extension and can be customized using the usual

`application.properties`.

For example with a configuration like:

```
quarkus.container-image.group=quarkus #optional, default to the
system user name
quarkus.container-image.name=demo-app #optional, defaults to the
application name
quarkus.container-image.tag=1.0          #optional, defaults to the
application version
```

The image that will be used in the generated manifests will be `quarkus/demo-app:1.0`

Defining a Docker registry

The Docker registry can be specified with the following property:

```
quarkus.container-image.registry=http://my.docker-registry.net
```

By adding this property along with the rest of the container-image properties of the previous section, the generated manifests will use the image `http://my.docker-registry.net/quarkus/demo-app:1.0`.

The image is not the only thing that can be customized in the generated manifests. The kubernetes extension provides a rich set of configuration properties that allow customization.

Labels and Annotations

Labels

The generated manifests use the Kubernetes [recommended labels](#). These labels can be customized using `quarkus.kubernetes.name`, `quarkus.kubernetes.version` and `quarkus.kubernetes.part-of`. For example by adding the following configuration to your `application.properties`:

```
quarkus.kubernetes.part-of=todo-app
quarkus.kubernetes.name=todo-rest
quarkus.kubernetes.version=1.0-rc.1
```

The generated generated resources will have the labels:

```
"labels" : {  
  "app.kubernetes.io/part-of" : "todo-app",  
  "app.kubernetes.io/name" : "todo-rest",  
  "app.kubernetes.io/version" : "1.0-rc.1"  
}
```

Custom Labels

To add additional custom labels, for example `foo=bar`:

```
quarkus.kubernetes.labels.foo=bar
```

Annotations

Out of the box, the generated resources will be annotated with version control related information that can be used either by tooling, or by the user for troubleshooting purposes.

```
"annotations": {  
  "app.quarkus.io/vcs-url" : "<some url>",  
  "app.quarkus.io/commit-id" : "<some git SHA>",  
}
```

Custom Annotations

Custom annotations can be added in a way similar to labels. For example to add the annotation `foo=bar`:

```
quarkus.kubernetes.annotations.foo=bar
```

Environment variables

Kubernetes provides multiple ways of defining environment variables:

- key value pairs
- from Secret
- from ConfigMap
- from fields

To add a key value pair as an environment variable in the generated resources:

```
quarkus.kubernetes.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 have 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 setting a value.

Environment variables from Secret

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

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

Mounting volumes

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

Any volume can be mounted with a simple configuration:

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

ConfigMap volumes

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

Changing the number of replicas:

To change the number of replicas from 1 to 3:

```
quarkus.kubernetes.replicas=3
```

Add readiness and liveness probes

By default the Kubernetes resources do not contain readiness and liveness probes in the generated **Deployment**. Adding them however is just a matter of adding the SmallRye Health extension like so:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

The values of the generated probes will be determined by the configured health properties: `quarkus.smallrye-health.root-path`, `quarkus.smallrye-health.liveness-path` and `quarkus.smallrye-health.readiness-path`. More information about the health extension can be found in the relevant [guide](#).

Customizing the readiness probe:

To set the initial delay of the probe to 20 seconds and the period to 45:

```
quarkus.kubernetes.readiness-probe.initial-delay-seconds=20
quarkus.kubernetes.readiness-probe.period-seconds=45
```

Here you can find a complete reference to all the available configuration options:

Using the Kubernetes client

Applications that are deployed to Kubernetes and need to access the API server will usually make use of the **kubernetes-client** extension:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-kubernetes-client</artifactId>
</dependency>
```

To access the API server from within a Kubernetes cluster, some RBAC related resources are required (e.g. a ServiceAccount, a RoleBinding etc.). So, when the **kubernetes-client** extension is present, the **kubernetes** extension is going to create those resources automatically, so that application will

be granted the **view** role. If more roles are required, they will have to be added manually.

Tuning the generated resources using `application.properties`

The Kubernetes extension allows tuning the generated manifest, using the `application.properties` file. Here are some examples:

Configuration options

The table below describe all the available configuration options.

Table 1. Kubernetes

Property	Type	Description	Default Value
quarkus.kubernetes.name	String		<code>\${quarkus.container-image.name}</code>
quarkus.kubernetes.version	String		<code>\${quarkus.container-image.tag}</code>
quarkus.kubernetes.part-of	String		
quarkus.kubernetes.init-containers	Map<String, Container>		
quarkus.kubernetes.labels	Map		
quarkus.kubernetes.annotations	Map		
quarkus.kubernetes.env-vars	Map<String, Env>		
quarkus.kubernetes.working-dir	String		
quarkus.kubernetes.command	String[]		
quarkus.kubernetes.arguments	String[]		
quarkus.kubernetes.replicas	int		1
quarkus.kubernetes.service-account	String		

quarkus.kubernetes.host	String		
quarkus.kubernetes.ports	Map<String, Port>		
quarkus.kubernetes.service-type	ServiceType		ClusterIP
quarkus.kubernetes.pvc-volumes	Map<String, PersistentVolumeClaimVolume>		
quarkus.kubernetes.secret-volumes	Map<String, SecretVolume>		
quarkus.kubernetes.config-map-volumes	Map<String, ConfigMapVolume>		
quarkus.kubernetes.git-repo-volumes	Map<String, GitRepoVolume>		
quarkus.kubernetes.aws-elastic-block-store-volumes	Map<String, AwsElasticBlockStoreVolume>		
quarkus.kubernetes.azure-disk-volumes	Map<String, AzureDiskVolume>		
quarkus.kubernetes.azure-file-volumes	Map<String, AzureFileVolume>		
quarkus.kubernetes.mounts	Map<String, Mount>		
quarkus.kubernetes.image-pull-policy	ImagePullPolicy		IfNotPresent
quarkus.kubernetes.image-pull-secrets	String[]		
quarkus.kubernetes.liveness-probe	Probe		(see Probe)
quarkus.kubernetes.readiness-probe	Probe		(see Probe)
quarkus.kubernetes.sidecars	Map<String, Container>		
quarkus.kubernetes.expose	boolean		false
quarkus.kubernetes.headless	boolean		false

Properties that use non standard types, can be referenced by expanding the property. For example to define a `kubernetes-readiness-probe` which is of type `Probe`:

```
quarkus.kubernetes.readiness-probe.initial-delay-seconds=20
quarkus.kubernetes.readiness-probe.period-seconds=45
```

In this example `initial-delay` and `period-seconds` are fields of the type `Probe`. Below you will find tables describing all available types.

Basic Types

Table 2. Env

Property	Type	Description	Default Value
value	String		
secret	String		
configmap	String		
field	String		

Table 3. Probe

Property	Type	Description	Default Value
http-action-path	String		
exec-action	String		
tcp-socket-action	String		
initial-delay-seconds	int		0
period-seconds	int		30
timeout-seconds	int		10

Table 4. Port

Property	Type	Description	Default Value
container-port	int		
host-port	int		0
path	String		/
protocol	Protocol		TCP

Table 5. Container

Property	Type	Description	Default Value
----------	------	-------------	---------------

image	String		
env-vars	Env[]		
working-dir	String		
command	String[]		
arguments	String[]		
ports	Port[]		
mounts	Mount[]		
image-pull-policy	ImagePullPolicy		IfNotPresent
liveness-probe	Probe		
readiness-probe	Probe		

Mounts and Volumes

Table 6. Mount

Property	Type	Description	Default Value
path	String		
sub-path	String		
read-only	boolean		false

Table 7. ConfigMapVolume

Property	Type	Description	Default Value
config-map-name	String		
default-mode	int		384
optional	boolean		false

Table 8. SecretVolume

Property	Type	Description	Default Value
secret-name	String		
default-mode	int		384
optional	boolean		false

Table 9. AzureDiskVolume

Property	Type	Description	Default Value
disk-name	String		

disk-uri	String		
kind	String		Managed
caching-mode	String		ReadWrite
fs-type	String		ext4
read-only	boolean		false

Table 10. *AwsElasticBlockStoreVolume*

Property	Type	Description	Default Value
volume-id	String		
partition	int		
fs-type	String		ext4
read-only	boolean		false

Table 11. *GitRepoVolume*

Property	Type	Description	Default Value
repository	String		
directory	String		
revision	String		

Table 12. *PersistentVolumeClaimVolume*

Property	Type	Description	Default Value
claim-name	String		
read-only	boolean		false

Table 13. *AzureFileVolume*

Property	Type	Description	Default Value
share-name	String		
secret-name	String		
read-only	boolean		false

OpenShift

To enable the generation of OpenShift resources, you need to include OpenShift in the target platforms:

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

If you need to generate resources for both platforms (vanilla Kubernetes and OpenShift), then you need to include both (coma separated).

```
quarkus.kubernetes.deployment-target=kubernetes, openshift
```

Note: In latest versions of Quarkus a wrapper extension of [Openshift](#) is provided. If you use it, there is no need to explicitly set this property.

The OpenShift resources can be customized in a similar approach with Kubernetes.

Table 14. Openshift

Property	Type	Description	Default Value
quarkus.openshift.name	String		\${quarkus.container-image.name}
quarkus.openshift.version	String		\${quarkus.container-image.tag}
quarkus.openshift.part-of	String		
quarkus.openshift.init-containers	Map<String, Container>		
quarkus.openshift.labels	Map		
quarkus.openshift.annotations	Map		
quarkus.openshift.env-vars	Map<String, Env>		
quarkus.openshift.working-dir	String		
quarkus.openshift.command	String[]		
quarkus.openshift.arguments	String[]		
quarkus.openshift.replicas	int		1
quarkus.openshift.service-account	String		
quarkus.openshift.host	String		

quarkus.openshift.ports	Map<String, Port>		
quarkus.openshift.service-type	ServiceType		ClusterIP
quarkus.openshift.pvc-volumes	Map<String, PersistentVolumeClaim Volume>		
quarkus.openshift.secret-volumes	Map<String, SecretVolume>		
quarkus.openshift.config-map-volumes	Map<String, ConfigMapVolume>		
quarkus.openshift.git-repo-volumes	Map<String, GitRepoVolume>		
quarkus.openshift.aws-elastic-block-store-volumes	Map<String, AwsElasticBlockStoreVolume>		
quarkus.openshift.azure-disk-volumes	Map<String, AzureDiskVolume>		
quarkus.openshift.azure-file-volumes	Map<String, AzureFileVolume>		
quarkus.openshift.mounts	Map<String, Mount>		
quarkus.openshift.image-pull-policy	ImagePullPolicy		IfNotPresent
quarkus.openshift.image-pull-secrets	String[]		
quarkus.openshift.liveness-probe	Probe		(see Probe)
quarkus.openshift.readiness-probe	Probe		(see Probe)
quarkus.openshift.sidecars	Map<String, Container>		
quarkus.openshift.expose	boolean		false
quarkus.openshift.headless	boolean		false

Knative

To enable the generation of `Quarkus.Knative.resources`, you need to include Knative in the target platforms:

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

Following the execution of `./mvnw package` you will notice amongst the other files that are created, two files named `quarkus.knative.json` and `knative.yml` in the `target/kubernetes/` directory.

If you look at either file you will see that it contains a `Quarkus.Knative.Service`.

The full source of the `quarkus.knative.json` file looks something like this:

```

{
  {
    "apiVersion" : "serving.quarkus.knative.dev/v1alpha1",
    "kind" : "Service",
    "metadata" : {
      "annotations": {
        "app.quarkus.io/vcs-url" : "<some url>",
        "app.quarkus.io/commit-id" : "<some git SHA>"
      },
      "labels" : {
        "app.kubernetes.io/name" : "test-quarkus-app",
        "app.kubernetes.io/version" : "1.0-SNAPSHOT"
      },
      "name" : "quarkus.knative.
    },
    "spec" : {
      "runLatest" : {
        "configuration" : {
          "revisionTemplate" : {
            "spec" : {
              "container" : {
                "image" : "dev.local/yourDockerUsername/test-quark
us-app:1.0-SNAPSHOT",
                "imagePullPolicy" : "IfNotPresent"
              }
            }
          }
        }
      }
    }
  }
}

```

The generated service can be customized using the following properties:

Table 15. Knative

Property	Type	Description	Default Value
quarkus.knative.name	String		\${quarkus.container-image.name}
quarkus.knative.version	String		\${quarkus.container-image.tag}
quarkus.knative.part-of	String		
quarkus.knative.init-containers	Map<String, Container>		

quarkus.knative.labels	Map		
quarkus.knative.annotations	Map		
quarkus.knative.env-vars	Map<String, Env>		
quarkus.knative.working-dir	String		
quarkus.knative.command	String[]		
quarkus.knative.arguments	String[]		
quarkus.knative.replicas	int		1
quarkus.knative.service-account	String		
quarkus.knative.host	String		
quarkus.knative.ports	Map<String, Port>		
quarkus.knative.service-type	ServiceType		ClusterIP
quarkus.knative.pvc-volumes	Map<String, PersistentVolumeClaim Volume>		
quarkus.knative.secret-volumes	Map<String, SecretVolume>		
quarkus.knative.config-map-volumes	Map<String, ConfigMapVolume>		
quarkus.knative.git-repo-volumes	Map<String, GitRepoVolume>		
quarkus.knative.aws-elastic-block-store-volumes	Map<String, AwsElasticBlockStoreVolume>		
quarkus.knative.azure-disk-volumes	Map<String, AzureDiskVolume>		
quarkus.knative.azure-file-volumes	Map<String, AzureFileVolume>		
quarkus.knative.mounts	Map<String, Mount>		
quarkus.knative.image-pull-policy	ImagePullPolicy		IfNotPresent

quarkus.knative.image-pull-secrets	String[]		
quarkus.knative.liveness-probe	Probe		(see Probe)
quarkus.knative.readiness-probe	Probe		(see Probe)
quarkus.knative.sidecars	Map<String, Container>		

Deprecated configuration

The following categories of configuration properties have been deprecated.

Properties without the quarkus prefix

In earlier versions of the extension, the `quarkus.` was missing from those properties. These properties are now deprecated.

Docker and S2i properties

The properties for configuring `docker` and `s2i` are also deprecated in favor of the new container-image extensions.

Config group arrays

Properties referring to config group arrays (e.g. `kubernetes.labels[0]`, `kubernetes.env-vars[0]` etc) have been converted to maps, to align with the rest of the quarkus ecosystem.

The code below demonstrates the change in `labels` config:

```
# Old labels config:
kubernetes.labels[0].name=foo
kubernetes.labels[0].value=bar

# New labels
quarkus.kubernetes.labels.foo=bar
```

The code below demonstrates the change in `env-vars` config:

```
# Old env-vars config:
kubernetes.env-vars[0].name=foo
kubernetes.env-vars[0].configmap=my-configmap

# New env-vars
quarkus.kubernetes.env-vars.foo.configmap=myconfigmap
```

Deployment

To trigger building and deploying a container image you need to enable the `quarkus.container.deploy` flag. This can be easily done from the command line:

```
mvn clean package -Dquarkus.kubernetes.deploy=true
```

Building

Building is possible, using any of the 3 available `container-image` extensions:

- [Docker](#)
- [Jib](#)
- [s2i](#)

Each time deployment is requested, a container build will be implicitly triggered (no additional properties are required).

Deploying

When deployment is enabled, the kubernetes extension will selected the resources specified by `quarkus.kubernetes.deployment.target` and deploy them. This assumes that a `.kube/config` is available in your user directory that points to a real kubernetes cluster. In other words the extension will use whatever cluster `kubectl` uses. The same applies to credentials.

At the moment no additional options are provided for further customization.