Quarkus - Kubernetes extension

Quarkus offers the ability to automatically generate Kubernetes resources based on sane defaults and user-supplied configuration using dekorate. It currently supports generating resources for vanilla Kubernetes, OpenShift and Knative. Furthermore, Quarkus can deploy the application to a target Kubernetes cluster by applying the generated manifests to the target cluster's API Server. Finally, when either one of container image extensions is present (see the container image guide for more details), Quarkus has the ability to create a container image and push it to a registry **before** deploying the application 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 3.6.3
- access to a Kubernetes cluster (Minikube is a viable option)

Kubernetes

Let's create a new project that contains both the Kubernetes and Jib extensions:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=kubernetes-quickstart \
    -DclassName="org.acme.rest.GreetingResource" \
    -Dpath="/greeting" \
    -Dextensions="kubernetes, jib"
cd kubernetes-quickstart
```

This added the following dependencies to the `pom.xml`

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

By adding these dependencies, we enable the generation of Kubernetes manifests each time we perform a build while also enabling the build of a container image using Jib. 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"
          }
```

```json
        },
        "spec" : {
          "containers" : [ {
            "env" : [ {
              "name" : "KUBERNETES_NAMESPACE",
              "valueFrom" : {
                "fieldRef" : {
                  "fieldPath" : "metadata.namespace"
                }
              }
            } ],
            "image" : "yourDockerUsername/test-quarkus-app:1.0-
SNAPSHOT",
            "imagePullPolicy" : "Always",
            "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"
  }
 }
}
```

The generated manifest can be applied to the cluster from the project root using `kubectl`:

```
kubectl apply -f target/kubernetes/kubernetes.json
```

An important thing to note about the `Deployment` is that is uses `yourDockerUsername/test-quarkus-app:1.0-SNAPSHOT` as the container image of the `Pod`. The name of the image is controlled by the Jib 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=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 `my.docker-registry.net/quarkus/demo-app:1.0`. The image is not the only thing that can be customized in the generated manifests, as will become evident in the following sections.

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 labels in generated resources will look like:

```
"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` just apply the following configuration:

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

> ℹ️ When using the `quarkus-container-image-jib` extension to build a container image, then any label added via the aforementioned property will also be added to the generated container image.

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` and `app.quarkus/id=42` just apply the following configuration:

```
quarkus.kubernetes.annotations.foo=bar
quarkus."app.quarkus/id"=42
```

Environment variables

Kubernetes provides multiple ways of defining environment variables:

- key/value pairs

- import all values from a Secret or ConfigMap

- interpolate a single value identified by a given field in a Secret or ConfigMap

- interpolate a value from a field within the same resource

Environment variables from key/value pairs

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

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

Environment variables from Secret

To add all key/value pairs of `Secret` as environment variables just apply the following configuration, separating each `Secret` to be used as source by a comma (`,`):

```
quarkus.kubernetes.env.secrets=my-secret,my-other-secret
```

which would generate the following in the container definition:

```
envFrom:
  - secretRef:
      name: my-secret
      optional: false
  - secretRef:
      name: my-other-secret
      optional: false
```

The following extracts a value identified by the `keyName` field from the `my-secret` Secret into a `foo` environment variable:

```
quarkus.kubernetes.env.mapping.foo.from-secret=my-secret
quarkus.kubernetes.env.mapping.foo.with-key=keyName
```

This would generate the following in the `env` section of your container:

```
- env:
  - name: FOO
    valueFrom:
      secretKeyRef:
        key: keyName
        name: my-secret
        optional: false
```

Environment variables from ConfigMap

To add all key/value pairs from `ConfigMap` as environment variables just apply the following configuration, separating each `ConfigMap` to be used as source by a comma (`,`):

```
quarkus.kubernetes.env.configmaps=my-config-map,another-config-map
```

which would generate the following in the container definition:

```
envFrom:
  - configMapRef:
      name: my-config-map
      optional: false
  - configMapRef:
      name: another-config-map
      optional: false
```

The following extracts a value identified by the `keyName` field from the `my-config-map` ConfigMap into a `foo` environment variable:

```
quarkus.kubernetes.env.mapping.foo.from-configmap=my-configmap
quarkus.kubernetes.env.mapping.foo.with-key=keyName
```

This would generate the following in the `env` section of your container:

```
- env:
  - name: FOO
    valueFrom:
      configMapRefKey:
        key: keyName
        name: my-configmap
        optional: false
```

Environment variables from fields

It's also possible to use the value from another field to add a new environment variable by specifying the path of the field to be used as a source, as follows:

```
quarkus.kubernetes.env.fields.foo=metadata.name
```

Validation

A conflict between two definitions, e.g. mistakenly assigning both a value and specifying that a variable is derived from a field, will result in an error being thrown at build time so that you get the

opportunity to fix the issue before you deploy your application to your cluster where it might be more difficult to diagnose the source of the issue.

Similarly, two redundant definitions, e.g. defining an injection from the same secret twice, will not cause an issue but will indeed report a warning to let you know that you might not have intended to duplicate that definition.

Backwards compatibility

Previous versions of the Kubernetes extension supported a different syntax to add environment variables.The older syntax is still supported but is deprecated and it's advised that you migrate to the new syntax.

Table 1. Old vs. new syntax

	Old	New	
Plain variable	`quarkus.kubernetes.env-vars.my-env-var.value=foobar`	`quarkus.kubernetes.env.vars.my-env-var=foobar`	
From field	`quarkus.kubernetes.env-vars.my-env-var.field=foobar`	`quarkus.kubernetes.env.fields.my-env-var=foobar`	
All from `ConfigMap`	`quarkus.kubernetes.env-vars.xxx.configmap=foobar`	`quarkus.kubernetes.env.configmaps=foobar`	
All from `Secret`	`quarkus.kubernetes.env-vars.xxx.secret=foobar`	`quarkus.kubernetes.env.secrets=foobar`	
From one `Secret` field	`quarkus.kubernetes.env-vars.foo.secret=foobar`	`quarkus.kubernetes.env.mapping.foo.from-secret=foobar`	
	`quarkus.kubernetes.env-vars.foo.value=field`	`quarkus.kubernetes.env.mapping.foo.with-key=field`	
From one `ConfigMap` field	`quarkus.kubernetes.env-vars.foo.configmap=foobar`	`quarkus.kubernetes.env.mapping.foo.from-configmap=foobar`	

	quarkus.kubernetes.env-vars.foo.value=field	quarkus.kubernetes.env.mapping.foo.with-key=field	

> **ℹ** If you redefine the same variable using the new syntax while keeping the old syntax, **ONLY** the new version will be kept and a warning will be issued to alert you of the problem.For example, if you define both `quarkus.kubernetes.env-vars.my-env-var.value=foobar` and `quarkus.kubernetes.env.vars.my-env-var=newValue`, the extension will only generate an environment variable `MY_ENV_VAR=newValue` and issue a warning.

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 the 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=20s
quarkus.kubernetes.readiness-probe.period=45s
```

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.

Deploying to Minikube

Minikube is quite popular when a Kubernetes cluster is needed for development purposes. To make the deployment to Minikube experience as frictionless as possible, Quarkus provides the `quarkus-minikube` extension. This extension can be added to a project like so:

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

The purpose of this extension is to generate Kubernetes manifests (`minikube.yaml` and `minikube.json`) that are tailored to Minikube. This extension assumes a couple things:

- Users won't be using an image registry and will instead make their container image accessible to the Kubernetes cluster by building it directly into Minikube's Docker daemon. To use Minikube's Docker daemon you must first execute:

```
eval $(minikube -p minikube docker-env)
```

- Applications deployed to Kubernetes won't be accessed via a Kubernetes `Ingress`, but rather as a `NodePort Service`. The advantage of doing this is that the URL of an application can be retrieved trivially by executing:

```
minikube service list
```

To control the nodePort that is used in this case, users can set `quarkus.kubernetes.node-port`. Note however that this configuration is entirely optional because Quarkus will automatically use a proper (and non-changing) value if none is set.

> ⚠️ It is highly discouraged to use the manifests generated by the Minikube extension when deploying to production as these manifests are intended for development purposes only. When deploying to production, consider using the vanilla Kubernetes manifests (or the OpenShift ones when targeting OpenShift).

> ℹ️ If the assumptions the Minikube extension makes don't fit your workflow, nothing prevents you from using the regular Kubernetes extension to generate Kubernetes manifests and apply those to your Minikube cluster.

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 2. Kubernetes

Property	Type	Description	Default Value
quarkus.kubernetes.name	String		${quarkus.container-image.name}

quarkus.kubernetes.version	String		${quarkus.container-image.tag}
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, AwsElasticBlockStoreV olume>		
quarkus.kubernetes.azu re-disk-volumes	Map<String, AzureDiskVolume>		
quarkus.kubernetes.azu re-file-volumes	Map<String, AzureFileVolume>		
quarkus.kubernetes.mo unts	Map<String, Mount>		
quarkus.kubernetes.ima ge-pull-policy	ImagePullPolicy		Always
quarkus.kubernetes.ima ge-pull-secrets	String[]		
quarkus.kubernetes.live ness-probe	Probe		(see Probe)
quarkus.kubernetes.rea diness-probe	Probe		(see Probe)
quarkus.kubernetes.side cars	Map<String, Container>		
quarkus.kubernetes.exp ose	boolean		false
quarkus.kubernetes.hea dless	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=20s
quarkus.kubernetes.readiness-probe.period=45s
```

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

Basic Types

ServiceType

Allowed values: `cluster-ip`, `node-port`, `load-balancer`, `external-name`

Table 3. Env

Property	Type	Description	Default Value

value	String		
secret	String		
configmap	String		
field	String		

Table 4. Probe

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

Table 5. Port

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

Table 6. 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		Always
liveness-probe	Probe		
readiness-probe	Probe		

Mounts and Volumes

Table 7. Mount

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

Table 8. ConfigMapVolume

Property	Type	Description	Default Value
config-map-name	String		
default-mode	int		0600
optional	boolean		false

Table 9. SecretVolume

Property	Type	Description	Default Value
secret-name	String		
default-mode	int		0600
optional	boolean		false

Table 10. 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 11. AwsElasticBlockStoreVolume

Property	Type	Description	Default Value
volume-id	String		
partition	int		
fs-type	String		ext4

read-only	boolean		false

Table 12. GitRepoVolume

Property	Type	Description	Default Value
repository	String		
directory	String		
revision	String		

Table 13. PersistentVolumeClaimVolume

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

Table 14. 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 (comma separated).

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

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

These manifests can be deployed as is to a running cluster, using `kubectl`:

```
kubectl apply -f target/kubernetes/openshift.json
```

Openshift users might want to use `oc` instead of `kubectl`:

```
oc apply -f target/kubernetes/openshift.json
```

> **ℹ** Quarkus also provides the OpenShift extension. This extension is basically a wrapper around the Kubernetes extension and relieves OpenShift users of the necessity of setting the `deployment-target` property to `openshift`

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

Table 15. 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, AwsElasticBlockStoreV olume>		
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		Always
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 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 `knative.json` and `knative.yml` in the `target/kubernetes/` directory.

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

The full source of the `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" : "knative.
    },
    "spec" : {
      "runLatest" : {
        "configuration" : {
          "revisionTemplate" : {
            "spec" : {
              "container" : {
                "image" : "dev.local/yourDockerUsername/test-
quarkus-app:1.0-SNAPSHOT",
                "imagePullPolicy" : "Always"
              }
            }
          }
        }
      }
    }
  }
}
```

The generated manifest can be deployed as is to a running cluster, using `kubectl`:

```
kubectl apply -f target/kubernetes/knative.json
```

The generated service can be customized using the following properties:

Table 16. 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		Always
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>		

Deployment targets

Mentioned in the previous sections was the concept of `deployment-target`. This concept allows users to control which Kubernetes manifests will be generated and deployed to a cluster (if `quarkus.kubernetes.deploy` has been set to `true`).

By default, when no `deployment-target` is set, then only vanilla Kubernetes resources are generated and deployed. When multiple values are set (for example `quarkus.kubernetes.deployment-target=kubernetes,openshift`) then the resources for all targets are generated, but only the resources that correspond to the **first** target are applied to the cluster (if deployment is enabled).

In the case of wrapper extensions like OpenShift and Minikube, when these extensions have been explicitly added to the project, the default `deployment-target` is set by those extensions. For example if `quarkus-minikube` has been added to a project, then `minikube` becomes the default deployment target and its resources will be applied to the Kubernetes cluster when deployment via `quarkus.kubernetes.deploy` has been set. Users can still override the deployment-targets manually using `quarkus.kubernetes.deployment-target`.

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
```

`env-vars` properties

`quarkus.kubernetes.env-vars` are deprecated (though still currently supported as of this writing) and the new declaration style should be used instead. See Environment variables and more specifically Backwards compatibility for more details.

Deployment

To trigger building and deploying a container image you need to enable the `quarkus.kubernetes.deploy` flag (the flag is disabled by default - furthermore it has no effect during test runs or dev mode). This can be easily done with the command line:

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

Building a container image

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

- Docker
- Jib
- s2i

Each time deployment is requested, a container image build will be implicitly triggered (no additional properties are required when the Kubernetes deployment has been enabled).

Deploying

When deployment is enabled, the Kubernetes extension will select 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 the target 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.

Using existing resources

Sometimes it's desirable to either provide additional resources (e.g. a ConfigMap, a Secret, a Deployment for a database etc) or provide custom ones that will be used as a `base` for the generation process. Those resources can be added under `src/main/kubernetes` directory and can be named after the target environment (e.g. kubernetes.json, openshift.json, knative.json, or the yml equivalents). Each of these files may contain one or more Kubernetes resources.

Any resource found will be added in the generated manifests. Global modifications (e.g. labels, annotations etc) will also be applied to those resources. If one of the provided resources has the same name as one of the generated ones, then the generated resource will be created on top of the provided resource, respecting existing content when possible (e.g. existing labels, annotations, environment variables, mounts, replicas etc).

The name of the resource is determined by the application name and may be overridden by `quarkus.kubernetes.name`, `quarkus.openshift.name` and `quarkus.knative.name`.

For example, in the `kubernetes-quickstart` application, we can add a `kubernetes.yml` file in the `src/main/kubernetes` that looks like:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubernetes-quickstart
  labels:
    app: quickstart
spec:
  replicas: 3
  selector:
    matchLabels:
      app: quickstart
  template:
    metadata:
      labels:
        app: quickstart
    spec:
      containers:
      - name: kubernetes-quickstart
        image: someimage:latest
        ports:
        - containerPort: 80
        env:
        - name: FOO
          value: BAR
```

The generated `kubernetes.yml` will look like:

```
apiVersion: "apps/v1"
kind: "Deployment"
metadata:
  annotations:
    app.quarkus.io/build-timestamp: "2020-04-10 - 12:54:37 +0000"
  labels:
    app: "quickstart"
  name: "kubernetes-quickstart"
spec:
  replicas: 3 ①
  selector:
    matchLabels:
      app.kubernetes.io/name: "kubernetes-quickstart"
      app.kubernetes.io/version: "1.0-SNAPSHOT"
  template:
    metadata:
      annotations:
        app.quarkus.io/build-timestamp: "2020-04-10 - 12:54:37
+0000"
      labels:
        app: "quickstart" ②
    spec:
      containers:
      - env:
        - name: "FOO" ③
          value: "BAR"
        image: "<<yourDockerUsernmae>>/kubernetes-quickstart:1.0-
SNAPSHOT" ④
        imagePullPolicy: "Always"
        name: "kubernetes-quickstart"
        ports:
        - containerPort: 8080 ⑤
          name: "http"
          protocol: "TCP"
      serviceAccount: "kubernetes-quickstart"
```

The provided replicas <1>, labels <2> and environment variables <3> were retained. However, the image <4> and container port <5> were modified. Moreover, the default annotations have been added.

> ℹ️ If the resource name does not match the application name (or the overridden name) instead of reusing the resource a new one will be added. Same goes for the container. If the name of the container does not match the application name (or the overridden name), container specific configuration will be ignored.