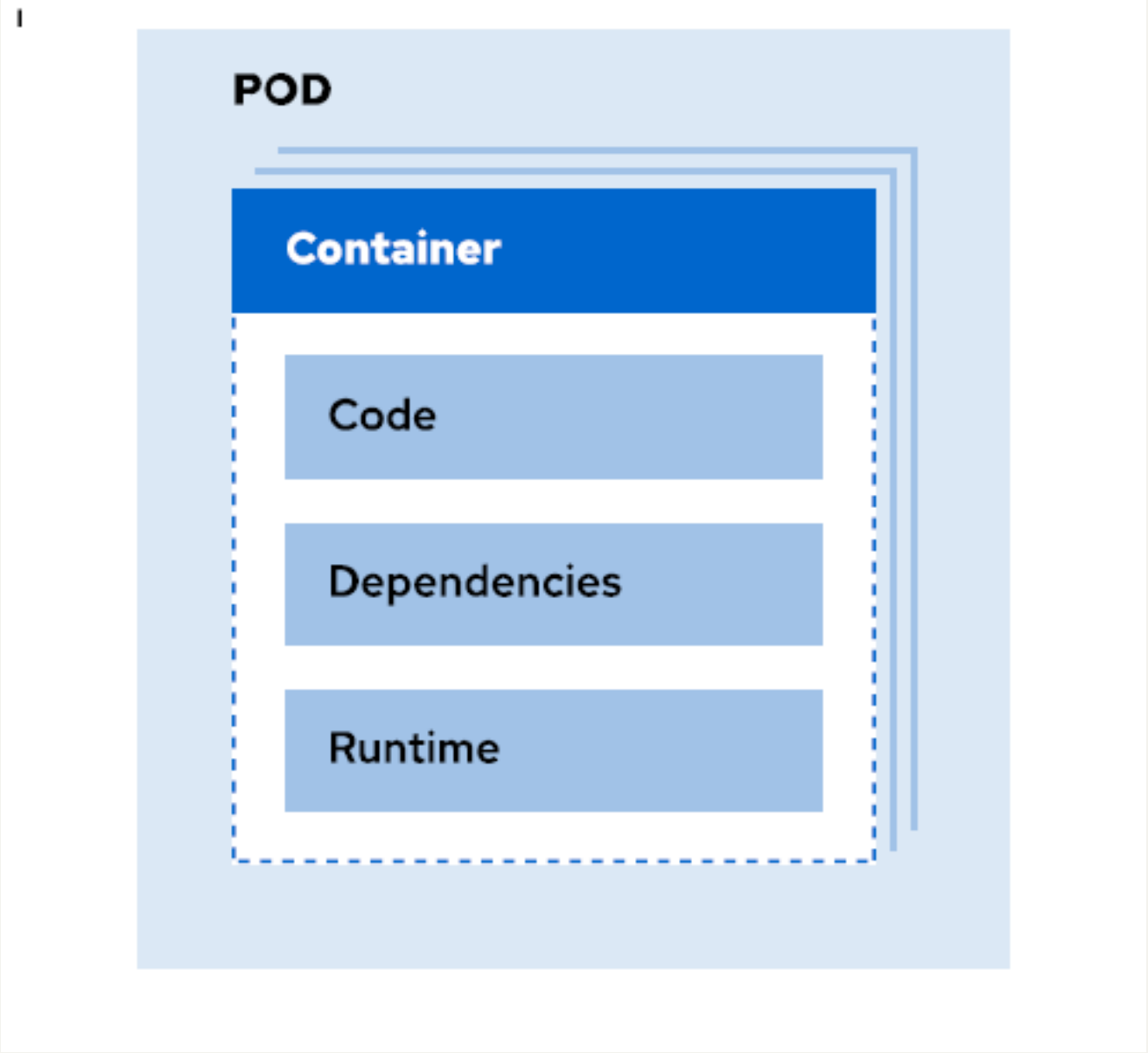

Quarkus en
Kubernetes



Pod

The representation of one or more containers running on the cluster. Pods are usually stateless, temporary resources. Kubernetes creates and destroys them as needed.

Service

An abstraction that defines the network access to a group of pods. Because pods are ephemeral, pod IP addresses often change. Kubernetes might destroy a pod at any time. Use services to provide a permanent IP address for a group of pods.

Route

The mapping between a URL and a service. Routes define how to route traffic from outside the cluster to application pods.

Deployment

A description of how to deploy the application. A `Deployment` resource manages pods and pod configuration, including values such as the pod container images, the number of pods to run, and the runtime configuration to inject.

ConfigMap

A set of keys and values, usually injected into containers as environment variables.

Quarkus Dockerfiles

You can define a new Containerfile, or use any of the following Containerfiles that the Quarkus application template includes in the `scr/main/docker` directory:

`Dockerfile.jvm`
: Defines the container image for a Quarkus application that runs on the JVM.

`Dockerfile.legacy-jar`
: Defines the container image for a Quarkus application that runs on the JVM by using a legacy JAR. Quarkus uses the *Fast-jar* packaging format by default, but you can switch to the legacy format by using the `quarkus.package.type=legacy-jar` property. If this is the case, then use this Containerfile to generate an image from a legacy JAR.

`Dockerfile.native`
: Defines the container image for a Quarkus application compiled as a native executable.

`Dockerfile.native-micro`
: Defines the container image for a Quarkus application compiled as a native executable by using the `quarkus-micro-image` base image. This image is smaller than the image defined in `Dockerfile.native`, but contains less tooling. Red Hat does not support this image in production environments, but you can still use it for development and testing purposes.

Except for the `Dockerfile.native-micro` Containerfile, the rest of the Containerfiles use the Red Hat Universal Base Image (UBI) images as the base image. UBI images are designed to be used as base images of containerized applications. Red Hat maintains these images and updates them regularly.

The `Dockerfile` files provided in the Quarkus project template are functional, but you can adjust them to your needs. For example, if you want modify the logging manager, then you can modify the file as follows:

```
FROM registry.access.redhat.com/ubi8/openjdk-17:1.11

...output omitted...

EXPOSE 8080
USER 185
ENV JAVA_OPTS="-Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager =
 com.acme.MyCustomLogManager"
ENV JAVA_APP_JAR="/deployments/quarkus-run.jar"
```

Important

Red Hat designs UBI images specifically as base images for containerized applications, and keeps them secure and regularly updated. Consequently, Red Hat recommends the use of UBI images as the base images for your containers.

Construyendo la imagen del contenedor

- `mvn package` (-Pnative)

```
[user@host myapp]$ podman build \
-f src/main/docker/Dockerfile.jvm \ 1
-t myapp \ 2
. 3
```

1. The path to the Containerfile. In this case, the command builds the JVM container image.
2. The image name to assign to the generated image.
3. The build context. Normally, this context is the root directory of your Quarkus project.

Ejecutando la imagen del contenedor localmente

```
[user@host myapp]$ podman run \
--name my-application \ 1
--rm \ 2
-p 9090:8080 \ 3
myapp 4
```

1. Container name. If not provided, podman selects a random name.
2. Remove the container after it exits.
3. Route requests from port `9090` on the local machine to port `8080` in the container.
4. Select image for the container. This container uses the `localhost/myapp:latest` image name.

Publicando a Quay.io

To push a local container image to an image registry, such as Quay.io, the image name must follow this format:

```
registry/account/image[:tag]
```

Typically, these are the options to name your image with the correct format:

- Set the correct name upfront, when building the image, by using the `-t` parameter of the `podman build` command.
- Defining a new name for the image with the `podman tag` command.

For example, assume that you want to push the `myapp` local image from the previous example to Quay.io. Because `my_app` does not match the preceding format, you must tag the image with the name that matches the image address in your Quay account, as follows:

```
[user@host myapp]$ podman tag myapp \ (1)
quay.io/your_username/myapp (2)
```

(1) The current name of your local image.

(2) The remote registry address of the image.

Pushing la imagen local a Container Registry

Pushing the Local Image to a Container Registry

To push the image, use the `podman push` command with the following syntax:

```
podman push IMAGE
```

Assuming you want to push the `quay.io/your_username/myapp` image from the previous example, the command would be as follows:

```
[user@host myapp]$ podman push quay.io/your_username/myapp
```

Note that, to push a new image, you must be logged in to the remote registry. You can log in to the registry by using the `podman login` command.

Generando los recursos para K8S

```
[user@host ~]$ oc create deployment myapp \ 1
--image quay.io/example/myapp \ 2
--dry-run=client \ 3
-o yaml 4
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: myapp
  name: myapp
spec:
...output omitted...
```

Generando los recursos para K8S

```
[user@host ~]$ oc create configmap my-config \ 1
--from-literal=key1=value1 \ 2
--from-literal=key2=value2 3
configmap/my-config created
```

Demo K8S