Multi-pod Applications

Objectives

- Deploy a multi-pod application to OpenShift and make it externally available.

Manage Pods with Controllers

Kubernetes encourages developers to use a declarative deployment style, which means that the developer declares the desired application state and Kubernetes reaches the declared state. Kubernetes uses the *controller pattern* to implement the declarative deployment style.

In the controller pattern, a controller continuously watches the current state of a system. If the system state does not match the desired state, then the controller sends signals to the system until the system state matches the desired state. This is called the *control loop*.

Kubernetes uses a number of controllers to deal with constant change. Consider a single-container *bare pod* application, which means an application that is deployed by using the `Pod` object. When a bare pod application fails, for example due to a memory leak, Kubernetes can handle the failure by restarting the containers according to the pod's `restartPolicy`. However, in case of a node failure, Kubernetes does not reschedule a bare pod, which is why developers rarely use bare pods for application deployment.

You can use a number of controller objects provided by Kubernetes, such as `Deployment`, `ReplicaSet`, `StatefulSet`, and others.

Deploy Pods with Deployments

The `Deployment` object is a Kubernetes controller that manages pods. Developers use deployments for the following use cases:

Managing Pod Scaling
You can configure the number of pod replicas for a given deployment. If the actual replica count decreases, for example due to node failure or a network partition, then the deployment schedules new pods until the declared replica count is reached.

Managing Application Changes
Most of the `Pod` object fields are immutable, such as the name or environment variable configuration. To change those fields with bare pods, developers must delete the pod and recreate it with the new configuration.

When you manage pods with a deployment, you declare the new configuration in the deployment object. The deployment controller then deletes the original pods and recreates the new pods that contain the new configuration.

Managing Application Updates
Deployments implement the `RollingUpdate` strategy for gradually updating application pods when you declare a new application image. This ensures zero-downtime application updates.

Additionally, you can *roll back* to the previous application version in case the update does not work correctly.

Create Deployments Declaratively

The following YAML object demonstrates a Kubernetes deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata: (1)
  labels:
    app: deployment-label
  name: example-deployment
spec:
  replicas: 3 (2)
  selector: (3)
    matchLabels:
      app: example-deployment
  strategy: RollingUpdate (4)
  template: (5)
    metadata:
      labels:
        app: example-deployment
    spec: (6)
      containers:
      - image: quay.io/example/awesome-container
        name: awesome-pod
```

1. Deployment object metadata. This deployment uses the `app=deployment-label` label.
2. Number of pod replicas. This deployment maintains `3` identical containers in `3` pods.
3. Selector for pods that the deployment manages. This deployment manages pods that use the `app=example-deployment` label.
4. Strategy for updating pods. The default `RollingUpdate` strategy ensures gradual, zero-downtime pod rollout when you modify the container image.
5. The template configuration for pods that the deployment creates and manages.
6. The `.spec.template.spec` field corresponds to the `.spec` field of the `Pod` object.

Create Deployments Imperatively

You can use the `oc create deployment` command to create a deployment:

```
[user@host ~]$ oc create deployment example-deployment \ (1)
  --image=quay.io/example/awesome-container \ (2)
  --replicas=3 (3)
deployment/example-deployment created
```

1. Set the name to `example-deployment`.

2 Set the image.

3 Create and maintain 3 replica pods.

You can also use the `--dry-run=client` and `-o` options to generate a deployment definition, for example:

```
[user@host ~]$ oc create deployment example-deployment \
  --image=quay.io/example/awesome-container \
  --replicas=3 \
  --dry-run=client -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
...output omitted...
```

Deployment Pod Selection

Controllers create and manage the lifecycle of pods that are specified in the controller configuration. Consequently, a controller must have an ownership relationship with pods, and one pod can be owned by at most one controller.

The deployment controller uses labels to target dependent pods. For example, consider the following deployment configuration:

```
apiVersion: apps/v1
kind: Deployment
...configuration omitted...
spec:
  selector:
    matchLabels: 1
      app: example-deployment
  template:
    metadata:
      labels: 2
        app: example-deployment
    spec:
...configuration omitted...
```

1 The `.spec.selector.matchLabels` field.

2 The `.spec.template.metadata.labels` field.

The `.spec.template.metadata.labels` field determines the set of labels applied to the pods created or managed by this deployment. Consequently, the `.spec.selector.matchLabels` field must be a subset of labels of the `.spec.template.metadata.labels` field.

A deployment that targets pod labels missing from the pod template is considered invalid.

NOTE

The deployment controller targets a `ReplicaSet` object, which in turn targets pods. Additionally, pod ownership uses object references so that multiple pods can use the same labels and be managed by different controllers.

See the references section for complete information about object ownership and the deployment controller.

Expose Applications for External Access

Developers commonly expose applications for access from outside of the Kubernetes cluster. The default `Service` object provides a stable, internal IP address and a DNS record for pods. However, applications outside of the Kubernetes cluster cannot connect to the IP address or resolve the DNS record.

You can use the `Route` object to expose applications. Route is a Red Hat OpenShift Container Platform (RHOCP) object that connects a public-facing IP address and DNS hostname to an internal-facing IP address. Routes use information from the service object to route requests to pods.

RHOCP exposes an HAProxy router pod that listens on a RHOCP node public IP address. The router serves as an ingress entrypoint to the internal RHOCP traffic.

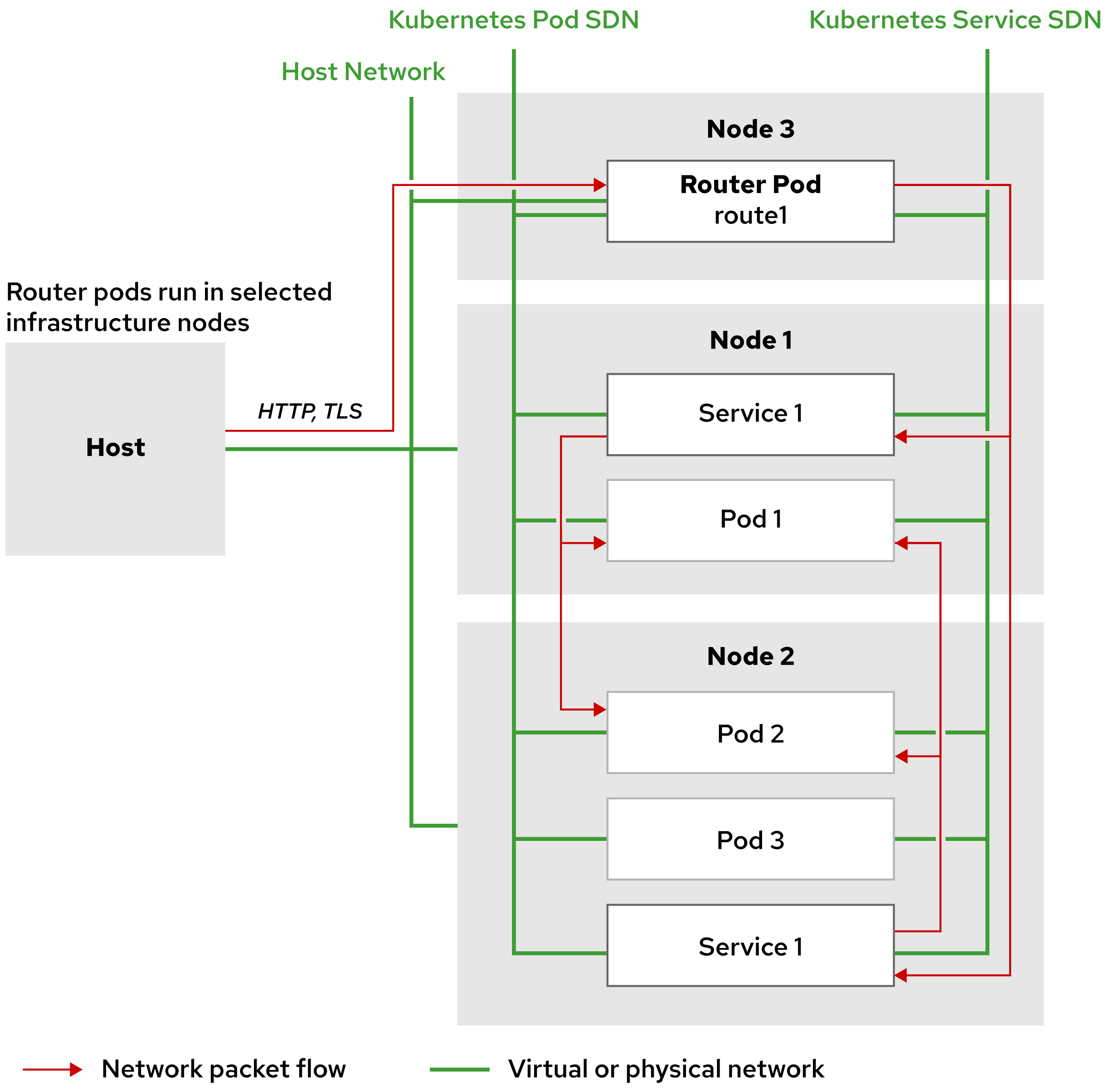

Figure 8.5: RHOCP network architecture

Routes are RHOCP-specific objects. If you require full compatibility with other Kubernetes distributions, you can use the `Ingress` object to create and manage routes, which is out of scope of this course.

Create Routes Declaratively

The following YAML object demonstrates a RHOCP route:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  labels:
    app: app-ui
  name: app-ui
  namespace: awesome-app
spec:
  port:
    targetPort: 8080 (1)
  host: ""
  to: (2)
    kind: "Service"
    name: "app-ui"
```

(1) The target port on pods selected by the service this route points to. If you use a string, then the route uses a named port in the target endpoints port list.

(2) The route target. Currently, only the `Service` target is allowed.

The preceding route routes requests to the `app-ui` service endpoints on port `8080`. Because the `app-ui` route does not specify the hostname in the `.spec.host` field, the hostname is generated in the following format:

```
route-name-project-name.default-domain
```

In the preceding example, the hostname in the classroom RHOCP cluster is `app-ui-awesome-app.apps.ocp4.example.com`. Consequently, RHOCP routes external requests from the `http://app-ui-awesome-app.apps.ocp4.example.com` domain to the `app-ui:8080` internal RHOCP service.

Administrators configure the default domain during RHOCP deployment.

Create Routes Imperatively

You can use the `oc expose service` command to create a route:

```
[user@host ~]$ oc expose service app-ui
route.route.openshift.io/app-ui exposed
```

You can also use the `--dry-run=client` and `-o` options to generate a route definition, for example:

```
[user@host ~]$ oc expose service app-ui \
  --dry-run=client -o yaml
apiVersion: apps/v1
kind: Route
metadata:
  creationTimestamp: null
...output omitted...
```

Note that you can use the `oc expose` imperative command in the following forms:

- `oc expose pod` *`POD_NAME`*: create a service for a specific pod.
- `oc expose deployment` *`DEPLOYMENT_NAME`*: create a service for all pods managed by a controller, in this case a deployment controller.
- `oc expose service` *`SERVICE_NAME`*: create a route that targets the specified service.

REFERENCES

Controllers | Kubernetes

Deployments | Kubernetes

Garbage Collection | Kubernetes

Workloads | Kubernetes

For more information about RHOCP networking, refer to the Red Hat OpenShift Container Platform 4.18 *Networking* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.18/html-single/networking/index