Chapter 4.  Deploy Managed and Networked Applications on Kubernetes

[**Deploy Applications from Images and Templates**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Deploy Applications from Images and Templates**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s02/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Manage Long-lived and Short-lived Applications by Using the Kubernetes Workload API**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s03/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Manage Long-lived and Short-lived Applications by Using the Kubernetes Workload API**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s04/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Kubernetes Pod and Service Networks**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s05/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Kubernetes Pod and Service Networks**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s06/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Scale and Expose Applications to External Access**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s07/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Scale and Expose Applications to External Access**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s08/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Lab: Deploy Managed and Networked Applications on Kubernetes**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s09/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s10/c2358540-87d5-48de-b49e-6f23bdcd629c)

**Abstract**

|  |  |
| --- | --- |
| **Goal** | Deploy applications and expose them to network access from inside and outside a Kubernetes cluster. |
| **Objectives** | * Identify the main resources and settings that Kubernetes uses to manage long-lived applications and demonstrate how OpenShift simplifies common application deployment workflows. * Deploy containerized applications as pods that Kubernetes workload resources manage. * Interconnect applications pods inside the same cluster by using Kubernetes services. * Expose applications to clients outside the cluster by using Kubernetes ingress and OpenShift routes. |
| **Sections** | * Deploy Applications from Images and Templates (and Guided Exercise) * Manage Long-lived and Short-lived Applications by Using the Kubernetes Workload API (and Guided Exercise) * Kubernetes Pod and Service Networks (and Guided Exercise) * Scale and Expose Applications to External Access (and Guided Exercise) |
| **Lab** | * Deploy Managed and Networked Applications on Kubernetes |

Deploy Applications from Images and Templates

Objectives

* Identify the main resources and settings that Kubernetes uses to manage long-lived applications and demonstrate how OpenShift simplifies common application deployment workflows.

Deploying Applications

Microservices and DevOps are growing trends in enterprise software. Containers and Kubernetes gained popularity alongside those trends, but have become categories of their own. Container-based infrastructures support most types of traditional and modern applications.

The term *application* can refer to your software system or to a service within it. Given this ambiguity, it is clearer to refer directly to resources, services, and other components.

Resources and Resource Definitions

Kubernetes manages applications, or services, as a loose collection of resources. *Resources* are configuration pieces for components in the cluster. When you create a resource, the corresponding component is not created immediately. Instead, the cluster is responsible for eventually creating the component.

A *resource type* represents a specific component type, such as a pod. Kubernetes ships with many default resource types, some of which overlap in function. Red Hat OpenShift Container Platform (RHOCP) includes the default Kubernetes resource types, and provides other resource types of its own. To add resource types, you can create or import custom resource definitions (CRDs).

Managing Resources

You can add, view, and edit resources in various formats, including YAML and JSON. Traditionally, YAML is the most common format.

You can delete resources in batch by using label selectors or by deleting the entire project or namespace. For example, the following command deletes only deployments with the app=my-app label.

[user@host ~]$ **oc delete deployment -l app=my-app**

*...output omitted...*

Similar to creation, deleting a resource is not immediate, but is instead a request for eventual deletion.

**Note**

Commands that are executed without specifying a namespace are executed in the user's current namespace.

Common Resource Types and Their Uses

The following resources are standard Kubernetes resources unless otherwise noted.

Templates

Similar to projects, templates are an RHOCP addition to Kubernetes. A *template* is a YAML manifest that contains parameterized definitions of one or more resources. RHOCP provides predefined templates in the openshift namespace.

Process a template into a list of resources by using the oc process command, which replaces values and generates resource definitions. The resulting resource definitions create or update resources in the cluster by supplying them to the oc apply command.

For example, the following command processes a mysql-template.yaml template file and generates four resource definitions.

[user@host ~]$ **oc process -f mysql-template.yaml -o yaml**

apiVersion: v1

items:

- apiVersion: v1

kind: Secret

*...output omitted...*

- apiVersion: v1

kind: Service

*...output omitted...*

- apiVersion: v1

kind: PersistentVolumeClaim

*...output omitted...*

- apiVersion: apps.openshift.io/v1

kind: DeploymentConfig

*...output omitted...*

The --parameters option instead displays the parameters of a template. For example, the following command lists the parameters of the mysql-template.yaml file.

[user@host ~]$ **oc process -f mysql-template.yaml --parameters**

NAME DESCRIPTION *...output omitted...*

*...output omitted...*

MYSQL\_USER Username for MySQL user *...output omitted...*

MYSQL\_PASSWORD Password for the MySQL connection *...output omitted...*

MYSQL\_ROOT\_PASSWORD Password for the MySQL root user. *...output omitted...*

MYSQL\_DATABASE Name of the MySQL database accessed. *...output omitted...*

VOLUME\_CAPACITY Volume space available for data, *...output omitted...*

You can use templates with the new-app command from RHOCP. In the following example, the new-app command uses the mysql-persistent template to create a MySQL application and its supporting resources.

[user@host ~]$ **oc new-app --template mysql-persistent**

--> Deploying template "db-app/mysql-persistent" to project db-app

*...output omitted...*

The following service(s) have been created in your project: mysql.

Username: userQSL

Password: pyf0yElPvFWYQQou

Database Name: sampledb

Connection URL: mysql://mysql:3306/

*...output omitted...*

\* With parameters:

\* Memory Limit=512Mi

\* Namespace=openshift

\* Database Service Name=mysql

\* MySQL Connection Username=userQSL # generated

\* MySQL Connection Password=pyf0yElPvFWYQQou # generated

\* MySQL root user Password=HHbdurqWO5gAog2m # generated

\* MySQL Database Name=sampledb

\* Volume Capacity=1Gi

\* Version of MySQL Image=8.0-el8

--> Creating resources ...

secret "mysql" created

service "mysql" created

persistentvolumeclaim "mysql" created

deploymentconfig.apps.openshift.io "mysql" created

--> Success

*...output omitted...*

|  |  |
| --- | --- |
|  | Notice that several resources are created to meet the requirements of the deployment, including a secret, a service, and a persistent volume claim. |

**Note**

You can specify environment variables to be configured in creating your application.

You can also use templates when creating applications from the web console by using the Developer Catalog. From the **Developer** perspective, navigate to the **+Add** menu and click **All Services** in the **Developer Catalog** section to open the catalog. Then, enter the application name in the filter to search for a template for your application. You can instantiate the template and change the default values, such as the Git repository, the memory limits, the application version, and much more.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

You can add an application based on a template by changing to the **Developer** perspective, and then moving to the **Topology** menu and clicking **Start building application** or clicking the book icon.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Pod

From the RHOCP documentation, a *pod* is defined as "the smallest compute unit that can be defined, deployed, and managed". A pod runs one or more containers that represent a single application. Containers in the pod share resources, such as networking and storage.

The following example shows a definition of a pod:

apiVersion: v1

kind: Pod

metadata:

annotations: { ... }

labels:

deployment: docker-registry-1

deploymentconfig: docker-registry

name: registry

namespace: pod-registries

spec:

containers:

- env:

- name: OPENSHIFT\_CA\_DATA

value: ...

image: openshift/origin-docker-registry:v0.6.2

imagePullPolicy: IfNotPresent

name: registry

ports:

- containerPort: 5000

protocol: TCP

resources: {}

securityContext: { ... }

volumeMounts:

- mountPath: /registry

name: registry-storage

dnsPolicy: ClusterFirst

imagePullSecrets:

- name: default-dockercfg-at06w

restartPolicy: Always

serviceAccount: default

volumes:

- emptyDir: {}

name: registry-storage

status:

conditions: { ... }

|  |  |
| --- | --- |
|  | Resource kind set to Pod. |
|  | Information that describes your application, such as the name, project, attached labels, and annotations. |
|  | Section where the application requirements are specified, such as the container name, the container image, environment variables, volume mounts, network configuration, and volumes. |
|  | Indicates the last condition of the pad, such as the last probe time, the last transition time, the status setting as true or false, and more. |

Deployment

Similar to deployment configurations, *deployments* define the intended state of a replica set. *Replica sets* maintain a configurable number of pods that match a specification.

Replica sets are generally similar to and a successor to replication controllers. This difference in intermediate resources is the primary difference between deployments and deployment configurations.

Deployments and replica sets are a Kubernetes-standard replacement for deployment configurations and replication controllers, respectively. Use a deployment unless you need a feature that is specific to deployment configurations.

The following example shows the definition of a deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-openshift

spec:

replicas: 1

selector:

matchLabels:

app: hello-openshift

template:

metadata:

labels:

app: hello-openshift

spec:

containers:

- name: hello-openshift

image: openshift/hello-openshift:latest

ports:

- containerPort: 80

|  |  |
| --- | --- |
|  | Resource kind set to Deployment. |
|  | Name of the deployment resource. |
|  | Number of running instances. |
|  | Section to define the metadata, labels, and the container information of the deployment resource. |
|  | Name of the container. |
|  | Resource of the image to use for creating the deployment resource. |
|  | Port configuration, such as the port number, name of the port, and the protocol. |

Deployment Configurations

*Deployment configurations* define the specification of a pod. They manage pods by creating *replication controllers*, which manage the number of replicas of a pod. Deployment configurations and replication controllers are an RHOCP addition to Kubernetes.

**Note**

As of OpenShift Container Platform 4.14, deployment configuration objects are deprecated. Deployment configurations objects are still supported, but are not recommended for new installations.

Instead, use Deployment objects or another alternative to provide declarative updates for pods.

Projects

RHOCP adds projects to enhance the function of Kubernetes namespaces. A *project* is a Kubernetes namespace with additional annotations, and is the primary method for managing access to resources for regular users. Projects can be created from templates and must use Role Based Access Control (RBAC) for organization and permission management. Administrators must grant cluster users access to a project. If a cluster user is allowed to create projects, then the user automatically has access to their created projects.

Projects provide logical and organizational isolation to separate your application component resources. Resources in one project can access resources in other projects, but not by default.

The following example shows the definition of a project:

apiVersion: project.openshift.io/v1

kind: Project

metadata:

name: test

spec:

finalizers:

- kubernetes

|  |  |
| --- | --- |
|  | Resource kind set to Project. |
|  | Name of the project. |
|  | A finalizer is a special metadata key that tells Kubernetes to wait until a specific condition is met before it fully deletes a resource. |

Services

You can configure internal pod-to-pod network communication in RHOCP by using the *Service* object. Applications send requests to the service name and port. RHOCP provides a virtual network, which reroutes such requests to the pods that the service targets by using labels.

The following example shows the definition of a service:

apiVersion: v1

kind: Service

metadata:

name: docker-registry

namespace: test

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

|  |  |
| --- | --- |
|  | Resource kind set to Service. |
|  | Name of the service. |
|  | Project name where the service resource exists. |
|  | The label selector identifies all pods with the attached app=MyApp label and adds the pods to the service endpoints. |
|  | Internet protocol set to TCP. |
|  | Port that the service listens on. |
|  | Port on the backing pods, which the service forwards connections to. |

Persistent Volume Claims

RHOCP uses the Kubernetes persistent volume (PV) framework to enable cluster administrators to provision persistent storage for a cluster. Developers can use persistent volume claims (PVCs) to request PV resources without having specific knowledge of the underlying storage infrastructure. After a PV is bound to a PVC, that PV cannot then be bound to additional PVCs. Because PVCs are namespaced objects and PVs are globally scoped objects, this binding effectively scopes a bound PV to a single namespace until the binding PVC is deleted.

The following example shows the definition of a persistent volume claim:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mysql-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

storageClassName: nfs-storage

status:

...

|  |  |
| --- | --- |
|  | Resource kind set to PersistentVolumeClaim. |
|  | Name of the service. |
|  | The access mode, to define the read/write and mount permissions. |
|  | The amount of storage that is available to the PVC. |
|  | Name of the StorageClass that the claim requires. |

Secrets

Secrets provide a mechanism to hold sensitive information, such as passwords, private source repository credentials, sensitive configuration files, and credentials to an external resource, such as an SSH key or OAuth token. You can mount secrets into containers by using a volume plug-in. Kubernetes can also use secrets to perform actions, such as declaring environment variables, on a pod. Secrets can store any type of data. Kubernetes and OpenShift support different types of secrets, such as service account tokens, SSH keys, and TLS certificates.

The following example shows the definition of a secret:

apiVersion: v1

kind: Secret

metadata:

name: example-secret

namespace: my-app

type: Opaque

data:

username: bXl1c2VyCg==

password: bXlQQDU1Cg==

stringData:

hostname: myapp.mydomain.com

secret.properties: |

property1=valueA

property2=valueB

|  |  |
| --- | --- |
|  | Resource kind set to Secret. |
|  | Name of the service. |
|  | Project name where the service resource exists. |
|  | Specifies the type of secret. |
|  | Specifies the encoded string and data. |
|  | Specifies the decoded string and data. |

Managing Resources from the Command Line

Kubernetes and RHOCP have many commands to create and modify cluster resources. Some commands are part of core Kubernetes, whereas others are exclusive additions to RHOCP.

The resource management commands generally fall into one of two categories: declarative or imperative. An *imperative* command instructs what the cluster does. A *declarative* command defines the state that the cluster attempts to match.

Imperative Resource Management

The create command is an imperative way to create resources, and is included in both of the oc and kubectl commands. For example, the following command creates a deployment called my-app that creates pods that are based on the specified image.

[user@host ~]$ **oc create deployment my-app --image example.com/my-image:dev**

deployment.apps/my-app created

Use the set command to define attributes on a resource, such as environment variables. For example, the following command adds the TEAM=red environment variable to the preceding deployment.

[user@host ~]$ **oc set env deployment/my-app TEAM=red**

deployment.apps/my-app updated

Another imperative approach to creating a resource is the run command. In the following example, the run command creates the example-pod pod.

[user@host ~]$ **oc run example-pod \**

**--image=registry.access.redhat.com/ubi8/httpd-24 \**

**--env GREETING='Hello from the awesome container' \**

**--port 8080**

pod/example-pod created

|  |  |
| --- | --- |
|  | The pod .metadata.name definition. |
|  | The image for the single container in this pod. |
|  | The environment variable for the single container in this pod. |
|  | The port metadata definition. |

The imperative commands are a faster way of creating pods, because such commands do not require a pod object definition. However, developers cannot handle versioning or incrementally change the pod definition.

Generally, developers test a deployment by using imperative commands, and then use the imperative commands to generate the pod object definition. Use the --dry-run=client option to avoid creating the object in RHOCP. Additionally, use the -o yaml or -o json option to configure the definition format.

The following command is an example of generating the YAML definition for the example-pod pod:

[user@host ~]$ **oc run example-pod \**

**--image=registry.access.redhat.com/ubi8/httpd-24 \**

**--env GREETING='Hello from the awesome container' \**

**--port 8080 \**

**--dry-run=client -o yaml**

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: null

labels:

run: example-pod

name: example-pod

spec:

containers:

*...output omitted...*

Managing resources in this way is imperative, because you are instructing the cluster what to do rather than declaring the intended outcomes.

Declarative Resource Management

Using the create command does not guarantee that you are creating resources imperatively. For example, providing a manifest declaratively creates the resources that are defined in the YAML file, such as in the following command.

[user@host ~]$ **oc create -f my-app-deployment.yaml**

deployment.apps/my-app created

RHOCP adds the new-app command, which provides another declarative way to create resources. This command uses heuristics to automatically determine which types of resources to create based on the specified parameters. For example, the following command deploys the my-app application by creating several resources, including a deployment resource, from a YAML manifest file.

[user@host ~]$ **oc new-app --file=./example/my-app.yaml**

*...output omitted...*

--> Creating resources ...

imagestream.image.openshift.io "my-app" created

**deployment.apps "my-app" created**

services "my-app" created

*...output omitted...*

In both of the preceding create and new-app examples, you are *declaring* the intended state of the resources, and so they are declarative.

You can also use the new-app command with templates for resource management. A template describes the intended state of resources that must be created for an application to run, such as deployment configurations and services. Supplying a template to the new-app command is a form of declarative resource management.

The new-app command also includes options, such as the --param option, that customize an application deployment declaratively. For example, when the new-app is used with a template, you can include the --param option to override a parameter value in the template.

[user@host ~]$ **oc new-app --template mysql-persistent \**

**--param MYSQL\_USER=operator --param MYSQL\_PASSWORD=myP@55 \**

**--param MYSQL\_DATABASE=mydata \**

**--param DATABASE\_SERVICE\_NAME=db**

--> Deploying template "db-app/mysql-persistent" to project db-app

*...output omitted...*

The following service(s) have been created in your project: db.

Username: operator

Password: myP@55

Database Name: mydata

Connection URL: mysql://db:3306/

*...output omitted...*

\* With parameters:

\* Memory Limit=512Mi

\* Namespace=openshift

\* Database Service Name=db

\* MySQL Connection Username=operator

\* MySQL Connection Password=myP@55

\* MySQL root user Password=tlH8BThuVgnIrCon # generated

\* MySQL Database Name=mydata

\* Volume Capacity=1Gi

\* Version of MySQL Image=8.0-el8

--> Creating resources ...

secret "db" created

service "db" created

persistentvolumeclaim "db" created

deploymentconfig.apps.openshift.io "db" created

--> Success

*...output omitted...*

Similar to the create command, you can use the new-app command imperatively. When using a container image with the new-app, you are instructing the cluster what to do, rather than declaring the intended outcomes. For example, the following command deploys the example.com/my-app:dev image by creating several resources, including a deployment resource.

[user@host ~]$ **oc new-app --image example.com/my-app:dev**

*...output omitted...*

--> Creating resources ...

imagestream.image.openshift.io "my-app" created

**deployment.apps "my-app" created**

*...output omitted...*

You can also supply a Git repository to the new-app command. The following command creates an application named httpd24 by using a Git repository.

[user@host ~]$ **oc new-app https://github.com/apache/httpd.git#2.4.56**

*...output omitted...*

--> Creating resources ...

imagestream.image.openshift.io "httpd24" created

**deployment.apps "httpd24" created**

*...output omitted...*

Retrieving Resource Information

You can supply an all argument to commands, to specify a list of common resource types. However, the all option does not include every available resource type. Instead, all is a shorthand form for a predefined subset of types. When you use this command argument, ensure that all includes any intended types to address.

You can view detailed information about a resource, such as the defined parameters, by using the describe command. For example, RHOCP provides templates in the openshift project to use with the oc new-app command. The following example command displays detailed information about the mysql-ephemeral template:

[user@host ~]$ **oc describe template mysql-ephemeral -n openshift**

Name: mysql-ephemeral

Namespace: openshift

*...output omitted...*

Parameters:

Name: MEMORY\_LIMIT

Display Name: Memory Limit

Description: Maximum amount of memory the container can use.

Required: true

Value: 512Mi

Name: NAMESPACE

Display Name: Namespace

Description: The OpenShift Namespace where the ImageStream resides.

Required: false

Value: openshift

*...output omitted...*

Objects:

Secret ${DATABASE\_SERVICE\_NAME}

Service ${DATABASE\_SERVICE\_NAME}

DeploymentConfig.apps.openshift.io ${DATABASE\_SERVICE\_NAME}

The describe command cannot generate structured output, such as the YAML or JSON formats. Without a structured format, the describe command cannot parse or filter the output with tools such as JSONPath or Go templates. Instead, use the get command to generate and then to parse the structured output of a resource.

Guided Exercise: Deploy Applications from Images and Templates

Deploy a database from a container image and from a template by using the OpenShift command-line interface and compare the resources and attributes generated by each method.

**Outcomes**

In this exercise, you deploy two MySQL database server pods to compare deployment methods and the resources that each creates.

* Deploy a database from a template.
* Deploy a database from a container image.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that resources are available for the exercise.

[student@workstation ~]$ **lab start deploy-newapp**

**Instructions**

1. As the developer user, create a project and verify that it is not empty after creation.
   1. Log in as the developer user with the developer password.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**

*...output omitted...*

* 1. Create a project named deploy-newapp.
  2. [student@workstation ~]$ **oc new-project deploy-newapp**

Now using project "deploy-newapp" on server *...output omitted...*

The new project is automatically selected.

* 1. Observe that resources for the new project are not returned with the oc get all command.
  2. [student@workstation ~]$ **oc get all**

No resources found in deploy-newapp namespace.

**Important**

Commands that use all for the resource type do not include every available resource type. Instead, all is a shorthand form for a predefined subset of types. When you use this command argument, ensure that all includes any types that you intend to address.

* 1. Observe that the new project contains other types of resources.
  2. [student@workstation ~]$ **oc get serviceaccounts**
  3. NAME SECRETS AGE
  4. builder 1 20s
  5. default 1 20s

deployer 1 20s

[student@workstation ~]$ **oc get secrets**

NAME TYPE DATA AGE

builder-dockercfg-sczxg kubernetes.io/dockercfg 1 36m

builder-token-gsnqj kubernetes.io/service-account-token 4 36m

default-dockercfg-6f8nm kubernetes.io/dockercfg 1 36m

*...output omitted...*

1. Create two MySQL instances by using the oc new-app command with different options.
   1. View the mysql-persistent template definition to see the resources that it creates. Specify the project that houses the template by using the -n openshift option.
   2. [student@workstation ~]$ **oc describe template mysql-persistent -n openshift**
   3. Name: mysql-persistent
   4. Namespace: openshift
   5. *...output omitted...*
   6. Objects:
   7. Secret ${DATABASE\_SERVICE\_NAME}
   8. Service ${DATABASE\_SERVICE\_NAME}
   9. PersistentVolumeClaim ${DATABASE\_SERVICE\_NAME}

DeploymentConfig.apps.openshift.io ${DATABASE\_SERVICE\_NAME}

The objects attribute specifies several resource definitions that are applied on using the template. These resources include one of each of the following types: secret, service (svc), persistent volume claim (pvc), and deployment configuration (dc).

* 1. Create an instance by using the mysql-persistent template. Specify a name option and attach a custom team=red label to the created resources.
  2. [student@workstation ~]$ **oc new-app -l team=red --template mysql-persistent \**
  3. **-p MYSQL\_USER=developer \**
  4. **-p MYSQL\_PASSWORD=developer**
  5. *...output omitted...*
  6. --> Creating resources with label team=red ...
  7. **secret** "mysql" created
  8. **service** "mysql" created
  9. **persistentvolumeclaim** "mysql" created
  10. **deploymentconfig.apps.openshift.io** "mysql" created
  11. --> Success

*...output omitted...*

The template creates resources of the types from the preceding step.

* 1. View and wait for the pod to start, which takes a few minutes to complete. You might need to run the command several times before the status changes to Running.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. mysql-1-deploy **0/1** Completed 0 93s

mysql-1-qmxbf 1/1 Running 1 60s

Your pod names might differ from the previous output.

* 1. Create an instance by using a container image. Specify a name option and attach a custom team=blue label to the created resources.
  2. [student@workstation ~]$ **oc new-app --name db-image -l team=blue \**
  3. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1 \**
  4. **-e MYSQL\_USER=developer \**
  5. **-e MYSQL\_PASSWORD=developer \**
  6. **-e MYSQL\_ROOT\_PASSWORD=redhat**
  7. *...output omitted...*
  8. --> Creating resources with label team=blue ...
  9. imagestream.image.openshift.io "db-image" created
  10. deployment.apps "db-image" created
  11. service "db-image" created
  12. --> Success

*...output omitted...*

The command creates predefined resources that are needed to deploy an image. These resource types are image stream (is), deployment, and service (svc). Image streams and services are discussed in more detail elsewhere in the course.

**Note**

It is safe to ignore pod security warnings for exercises in this course. OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security.

* 1. Wait for the pod to start. After a few moments, list all pods that contain team as a label.
  2. [student@workstation ~]$ **oc get pods -L team**
  3. NAME READY STATUS RESTARTS AGE TEAM
  4. db-image-8d4b97594-6jb85 1/1 Running 0 55s blue
  5. mysql-1-deploy 0/1 Completed 0 2m

mysql-1-hn64v 1/1 Running 0 1m30s

Your pod name might differ from the previous output. Without a readinessProbe, this pod shows as ready before the MySQL service is ready for requests. Readiness probes are discussed in more detail elsewhere in the course.

Notice that only the db-image pod has a label that contains the word team. Pods that the mysql-persistent template creates do not have the team=red label, because the template does not define this label in its pod specification template.

1. Compare the resources that each image and template method creates.
   1. View the template-created pod and observe that it contains a readiness probe.
   2. [student@workstation ~]$ **oc get pods -l deploymentconfig=mysql \**
   3. **-o jsonpath='{.items[0].spec.containers[0].readinessProbe}' | jq**
   4. {
   5. "exec": {
   6. "command": [
   7. "/bin/sh",
   8. "-i",
   9. "-c",
   10. "MYSQL\_PWD=\"$MYSQL\_PASSWORD\" mysqladmin -u $MYSQL\_USER ping"
   11. ]
   12. },
   13. "failureThreshold": 3,
   14. "initialDelaySeconds": 5,
   15. "periodSeconds": 10,
   16. "successThreshold": 1,
   17. "timeoutSeconds": 1

}

**Note**

The results of the preceding oc command are passed to the jq command, which formats the JSON output.

* 1. Observe that the image-based pod does not contain a readiness probe.
  2. [student@workstation ~]$ **oc get pods -l deployment=db-image \**
  3. **-o jsonpath='{.items[0].spec.containers[0].readinessProbe}' | jq**

***no output expected***

* 1. Observe that the template-based pod has a memory resource limit, which restricts allocated memory to the resulting pods. Resource limits are discussed in more detail elsewhere in the course.
  2. [student@workstation ~]$ **oc get pods -l deploymentconfig=mysql \**
  3. **-o jsonpath='{.items[0].spec.containers[0].resources.limits}' | jq**
  4. {
  5. "memory": "512Mi"

}

* 1. Observe that the image-based pod has no resource limits.
  2. [student@workstation ~]$ **oc get pods -l deployment=db-image \**
  3. **-o jsonpath='{.items[0].spec.containers[0].resources}' | jq**

{}

* 1. Retrieve secrets in the project. Notice that the template produced a secret, whereas the pod that was created with only an image did not.
  2. [student@workstation ~]$ **oc get secrets**
  3. NAME TYPE DATA AGE
  4. *...output omitted...*

mysql Opaque 4 3m

1. Explore filtering resources via labels.
   1. Observe that not supplying a label shows all services.
   2. [student@workstation ~]$ **oc get services**
   3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
   4. db-image ClusterIP 172.30.38.113 <none> 3306/TCP 1m30s

mysql ClusterIP 172.30.95.52 <none> 3306/TCP 2m30s

* 1. Observe that supplying a label shows only the services with the label.
  2. [student@workstation ~]$ **oc get services -l team=red**
  3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

mysql ClusterIP 172.30.95.52 <none> 3306/TCP 2m43s

* 1. Observe that not all resources include the label, such as the pods that are created with the template.
  2. [student@workstation ~]$ **oc get pods -l team=red**

No resources found in deploy-newapp namespace.

1. Use labels to delete only the resources that are associated with the image-based deployment.
   1. Delete only the resources that use the team=red label by using it with the oc delete command. List the resource types from the template to ensure that all relevant resources are deleted.
   2. [student@workstation ~]$ **oc delete all -l team=red**
   3. replicationcontroller "mysql-1" deleted
   4. service "mysql" deleted
   5. *...output omitted...*

deploymentconfig.apps.openshift.io "mysql" deleted

[studen@workstation ~]$ **oc delete secret,pvc -l team=red**

secret "mysql" deleted

persistentvolumeclaim "mysql" deleted

**Note**

By using the oc delete all -l team=red command, some resources are deleted, but the persistent volume claim and secret remain.

* 1. Observe that the resources that the template created are deleted.
  2. [student@workstation ~]$ **oc get secret,svc,pvc,dc -l team=red**
  3. *...output omitted...*

No resources found in deploy-newapp namespace.

* 1. Observe that the image-based resources remain unchanged.
  2. [student@workstation ~]$ **oc get is,deployment,svc**
  3. NAME IMAGE REPOSITORY *...output omitted...*
  4. imagestream.image.openshift.io/**db-image** image-registry.openshift*...output omitted...*
  5. NAME READY UP-TO-DATE AVAILABLE AGE
  6. deployment.apps/**db-image** 1/1 1 1 46m
  7. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/**db-image** ClusterIP 172.30.71.0 <none> 3306/TCP 46m

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish deploy-newapp**

Manage Long-lived and Short-lived Applications by Using the Kubernetes Workload API

Objectives

* Deploy containerized applications as pods that Kubernetes workload resources manage.

Kubernetes Workload Resources

The Kubernetes Workloads API consists of resources that represent various types of cluster tasks, jobs, and workloads. This API is composed of various Kubernetes APIs and extensions that simplify deploying and managing applications. These resource types of the Workloads API are most often used:

* Jobs
* Deployments
* Stateful sets

Jobs

A *job* resource represents a one-time task to perform on the cluster. As with most cluster tasks, jobs are executed via pods. If a job's pod fails, then the cluster retries a number of times that the job specifies. The job does not run again after a specified number of successful completions.

Jobs differ from using the kubectl run and oc run commands; both of the latter create only a pod.

Common uses for jobs include the following tasks:

* Initializing or migrating a database
* Calculating one-off metrics from information within the cluster
* Creating or restoring from a data backup

The following example command creates a job that logs the date and time:

[user@host ~]$ **oc create job \**

**date-job \**

**--image registry.access.redhat.com/ubi8/ubi \**

**-- /bin/bash -c "date"**

|  |  |
| --- | --- |
|  | Creates a job resource. |
|  | Specifies a job name of date-job. |
|  | Sets registry.access.redhat.com/ubi8/ubi as the container image for the job pods. |
|  | Specifies the command to run within the pods. |

You can also create a job from the web console by clicking the **Workloads** → **Jobs** menu. Click **Create Job** and customize the YAML manifest.

Cron Jobs

A *cron job* resource builds on a regular job resource by enabling you to specify how often the job should run. Cron jobs are useful for creating periodic and recurring tasks, such as backups or report generation. Cron jobs can also schedule individual tasks for a specific time, such as to schedule a job for a low activity period. Similar to the crontab (cron table) file on a Linux system, the CronJob resource uses the Cron format for scheduling. A CronJob resource creates a job resource based on the configured time zone on the control plane node that runs the cron job controller.

The following example command creates a cron job named date that prints the system date and time every minute:

[user@host ~]$ **oc create cronjob date-cronjob \**

**--image registry.access.redhat.com/ubi8/ubi \**

**--schedule "\*/1 \* \* \* \*" \**

**-- date**

|  |  |
| --- | --- |
|  | Creates a cron job resource with a name of date-cronjob. |
|  | Sets the registry.access.redhat.com/ubi8/ubi as the container image for the job pods. |
|  | Specifies the schedule for the job in Cron format. |
|  | The command to execute within the pods. |

You can also create a cronjob from the web console by clicking the **Workloads** → **CronJobs** menu. Click **Create CronJob** and customize the YAML manifest.

Deployments

A *deployment* creates a replica set to maintain pods. A *replica set* maintains a specified number of replicas of a pod. Replica sets use selectors, such as a label, to identify pods that are part of the set. Pods are created or removed until the replicas reach the number that the deployment specifies. Replica sets are not managed directly. Instead, deployments indirectly manage replica sets.

Unlike a job, a deployment's pods are re-created after crashing or deletion. The reason is that deployments use replica sets.

The following example command creates a deployment:

[user@host ~]$ **oc create deployment \**

**my-deployment \**

**--image registry.access.redhat.com/ubi8/ubi \**

**--replicas 3**

|  |  |
| --- | --- |
|  | Creates a deployment resource. |
|  | Specifies my-deployment as the deployment name. |
|  | Sets registry.access.redhat.com/ubi8/ubi as the container image for the pods. |
|  | Sets the deployment to maintain three instances of the pod. |

You can also create a deployment from the web console by clicking the **Deployments** tab in the **Workloads** menu.

|  |
| --- |
|  |

Click **Create Deployment** and customize the form or the YAML manifest.

|  |
| --- |
| A screenshot of a chat  Description automatically generated |

Pods in a replica set are identical and match the pod template in the replica set definition. If the number of replicas is not met, then a new pod is created by using the template. For example, if a pod crashes or is otherwise deleted, then a new one is created to replace it.

*Labels* are a type of resource metadata that are represented as string key-value pairs. A label indicates a common trait for resources with that label. For example, you might attach a layer=frontend label to resources that relate to an application's UI components.

Many oc and kubectl commands accept a label to filter affected resources. For example, the following command deletes all pods with the environment=testing label:

[user@host ~]$ **oc delete pod -l environment=testing**

By liberally applying labels to resources, you can cross-reference resources and craft precise selectors. A *selector* is a query object that describes the attributes of matching resources.

Certain resources use selectors to find other resources. In the following YAML excerpt, an example replica set uses a selector to match its pods.

apiVersion: apps/v1

kind: ReplicaSet

*...output omitted...*

spec:

replicas: 1

**selector:**

**matchLabels:**

**app: httpd**

**pod-template-hash: 7c84fbdb57**

*...output omitted...*

Stateful Sets

Like deployments, *stateful sets* manage a set of pods based on a container specification. However, each pod that a stateful set creates is unique. Pod uniqueness is useful when, for example, a pod needs a unique network identifier or persistent storage.

As their name implies, stateful sets are for pods that require state within the cluster. Deployments are used for stateless pods.

Stateful sets are covered further in a later chapter.

## Guided Exercise: Manage Long-lived and Short-lived Applications by Using the Kubernetes Workload API

Deploy a batch application managed by a job resource and a database server that a deployment resource manages by using the Kubernetes command-line interface.

**Outcomes**

In this exercise, you deploy a database server and a batch application that are both managed by workload resources.

* Create deployments.
* Update environment variables on a pod template.
* Create and run job resources.
* Retrieve the logs and termination status of a job.
* View the pod template of a job resource.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that resources are available for the exercise.

[student@workstation ~]$ **lab start deploy-workloads**

**Instructions**

1. As the developer user, create a MySQL deployment in a new project.
   1. Log in as the developer user with the developer password.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**

*...output omitted...*

* 1. Create a project named deploy-workloads.
  2. [student@workstation ~]$ **oc new-project deploy-workloads**
  3. Now using project "deploy-workloads" on server "https://api.ocp4.example.com:6443".

*...output omitted...*

* 1. Create a deployment that runs an ephemeral MySQL server.
  2. [student@workstation ~]$ **oc create deployment my-db \**
  3. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1**
  4. Warning: would violate PodSecurity "restricted:v1.24"
  5. *...output omitted...*

deployment.apps/my-db created

**Note**

It is safe to ignore pod security warnings for exercises in this course. OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security.

* 1. Retrieve the status of the deployment.
  2. [student@workstation ~]$ **oc get deployments**
  3. NAME READY UP-TO-DATE AVAILABLE AGE

my-db **0/1** 1 0 67s

The deployment never has a ready instance.

* 1. Retrieve the status of the created pod. Your pod name might differ from the output.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

my-db-8567b478dd-d28f7 0/1 **CrashLoopBackOff** 4 (60s ago) 2m35s

The pod fails to start and repeatedly crashes.

* 1. Review the logs for the pod to determine why it fails to start.
  2. [student@workstation ~]$ **oc logs deploy/my-db**
  3. *...output omitted...*
  4. **You must either specify the following environment variables:**
  5. MYSQL\_USER (regex: '^$')
  6. MYSQL\_PASSWORD (regex: '^[a-zA-Z0-9\_~!@#$%^&\*()-=<>,.?;:|]$')
  7. MYSQL\_DATABASE (regex: '^$')
  8. Or the following environment variable:
  9. MYSQL\_ROOT\_PASSWORD (regex: '^[a-zA-Z0-9\_~!@#$%^&\*()-=<>,.?;:|]$')

*...output omitted...*

Note that the container fails to start due to missing environment variables.

1. Fix the database deployment and verify that the server is running.
   1. Set the MYSQL\_USER, MYSQL\_PASSWORD, and MYSQL\_DATABASE environment variables.
   2. [student@workstation ~]$ **oc set env deployment/my-db \**
   3. **MYSQL\_USER=developer \**
   4. **MYSQL\_PASSWORD=developer \**
   5. **MYSQL\_DATABASE=sampledb**

deployment.apps/my-db updated

* 1. Retrieve the list of deployments and observe that the my-db deployment has a running pod.
  2. [student@workstation ~]$ **oc get deployments**
  3. NAME READY UP-TO-DATE AVAILABLE AGE

my-db **1/1** 1 1 4m50s

* 1. Retrieve the internal IP address of the MySQL pod within the list of all pods.
  2. [student@workstation ~]$ **oc get pods -o wide**
  3. NAME READY STATUS RESTARTS AGE IP ...

my-db-748c97d478-g8xc9 1/1 Running 0 64s 10.8.0.91 ...

The -o wide option enables additional output, such as IP addresses. Your IP address value might differ from the previous output.

* 1. Verify that the database server is running, by running a query. Replace the IP address with the one that you retrieved in the preceding step.
  2. [student@workstation ~]$ **oc run -it db-test --restart=Never \**
  3. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1 \**
  4. **-- mysql sampledb -h *10.8.0.91* -u developer --password=developer \**
  5. **-e "select 1;"**
  6. *...output omitted...*
  7. ---
  8. | 1 |
  9. ---
  10. | 1 |

---

1. Delete the database server pod and observe that the deployment causes the pod to be re-created.
   1. Delete the existing MySQL pod by using the label that is associated with the deployment.
   2. [student@workstation ~]$ **oc delete pod -l app=my-db**

pod "my-db-84c8995d5-2sssl" deleted

* 1. Retrieve the information for the MySQL pod and observe that it is newly created. Your pod name might differ in your output.
  2. [student@workstation ~]$ **oc get pod -l app=my-db**
  3. NAME READY STATUS RESTARTS AGE

my-db-fbccb9447-p99jd 1/1 Running 0 **6s**

1. Create and apply a job resource that prints the time and date repeatedly.
   1. Create a job resource called date-loop that runs a script. Ignore the warning.
   2. [student@workstation ~]$ **oc create job date-loop \**
   3. **--image registry.ocp4.example.com:8443/ubi9/ubi \**
   4. **-- /bin/bash -c "for i in {1..30}; do date; done"**

job.batch/date-loop created

* 1. Retrieve the job resource to review the pod specification.
  2. [student@workstation ~]$ **oc get job date-loop -o yaml**
  3. *...output omitted...*
  4. spec:
  5. containers:
  6. - command:
  7. - /bin/bash
  8. - -c
  9. - for i in {1..30}; do date; done
  10. image: registry.ocp4.example.com:8443/ubi9/ubi
  11. imagePullPolicy: Always
  12. name: date-loop
  13. resources: {}
  14. terminationMessagePath: /dev/termination-log
  15. terminationMessagePolicy: File
  16. dnsPolicy: ClusterFirst
  17. restartPolicy: Never
  18. schedulerName: default-scheduler
  19. securityContext: {}
  20. terminationGracePeriodSeconds: 30

*...output omitted...*

|  |  |
| --- | --- |
|  | The command object, which specifies the defined script to execute within the pod. |
|  | Sets the container image for the pod. |
|  | Defines the restart policy for the pod. Kubernetes does not restart the job pod after the pod exits. |

* 1. List the jobs to see that the date-loop job completed successfully.
  2. [student@workstation ~]$ **oc get jobs**
  3. NAME COMPLETIONS DURATION AGE

date-loop **1/1** 7s 8s

You might need to wait for the script to finish and run the command again.

* 1. Retrieve the logs for the associated pod. The log values might differ in your output.
  2. [student@workstation ~]$ **oc logs job/date-loop**
  3. Fri Nov 18 14:50:56 UTC 2022
  4. Fri Nov 18 14:50:59 UTC 2022

*...output omitted...*

1. Delete the pod for the date-loop job and observe that the pod is not created again.
   1. Delete the associated pod.
   2. [student@workstation ~]$ **oc delete pod -l job-name=date-loop**

pod "date-loop-wvn2q" deleted

* 1. View the list of pods and observe that the pod is not re-created for the job.
  2. [student@workstation ~]$ **oc get pod -l job-name=date-loop**

No resources found in deploy-workloads namespace.

* 1. Verify that the job status is still listed as successfully completed.
  2. [student@workstation ~]$ **oc get job -l job-name=date-loop**
  3. NAME COMPLETIONS DURATION AGE

date-loop **1/1** 7s 7m36s

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish deploy-workloads**

Kubernetes Pod and Service Networks

Objectives

* Interconnect applications pods inside the same cluster by using Kubernetes services.

The Software-defined Network

Kubernetes implements software-defined networking (SDN) to manage the network infrastructure of the cluster and user applications. The SDN is a virtual network that encompasses all cluster nodes. The virtual network enables communication between any container or pod inside the cluster. Cluster node processes that Kubernetes pods manage can access the SDN. However, the SDN is not accessible from outside the cluster, nor to regular processes on cluster nodes. With the software-defined networking model, you can manage network services through the abstraction of several networking layers.

With the SDN, you can manage the network traffic and network resources programmatically, so that the organization teams can decide how to expose their applications. The SDN implementation creates a model that is compatible with traditional networking practices. It makes pods akin to virtual machines in terms of port allocation, IP address leasing, and reservation.

With the SDN design, you do not need to change how application components communicate with each other, which helps to containerize legacy applications. If your application is composed of many services that communicate over the TCP/UDP stack, then this approach still works, because containers in a pod use the same network stack.

The following diagram shows how all pods are connected to a shared network:

|  |
| --- |
|  |

Figure 4.5: How the Kubernetes SDN manages the network

Among the many features of SDN, with open standards, vendors can propose their solutions for centralized management, dynamic routing, and tenant isolation.

Kubernetes Networking

Networking in Kubernetes provides a scalable means of communication between containers.

Kubernetes networking provides the following capabilities:

* Highly coupled container-to-container communications
* Pod-to-pod communications
* Pod-to-service communications
* External-to-service communication: covered in [the section called “ Scale and Expose Applications to External Access ”](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s07/c2358540-87d5-48de-b49e-6f23bdcd629c)

Kubernetes automatically assigns an IP address to every pod. However, pod IP addresses are unstable, because pods are ephemeral. Pods are constantly created and destroyed across the nodes in the cluster. For example, when you deploy a new version of your application, Kubernetes destroys the existing pods and then deploys new ones.

All containers within a pod share networking resources. The IP address and MAC address that are assigned to the pod are shared among all containers in the pod. Thus, all containers within a pod can reach each other's ports through the loopback address, localhost. Ports that are bound to localhost are available to all containers that run within the pod, but never to containers outside it.

By default, the pods can communicate with each other even if they run on different cluster nodes or belong to different Kubernetes namespaces. Every pod is assigned an IP address in a flat shared networking namespace that has full communication with other physical computers and containers across the network. All pods are assigned a unique IP address from a Classless Inter-Domain Routing (CIDR) range of host addresses. The shared address range places all pods in the same subnet.

Because all the pods are on the same subnet, pods on all nodes can communicate with pods on any other node without the aid of Network Address Translation (NAT). Kubernetes also provides a service subnet, which links the stable IP address of a service resource to a set of specified pods. The traffic is forwarded in a transparent way to the pods; an agent (depending on the network mode that you use) manages routing rules to route traffic to the pods that match the service resource selectors. Thus, pods can be treated much like Virtual Machines (VMs) or physical hosts from the perspective of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration. Kubernetes implements this infrastructure by managing the SDN.

The following illustration gives further insight into how the infrastructure components work along with the pod and service subnets to enable network access between pods inside an OpenShift instance.

Figure 4.6: Network access between pods in a cluster

The shared networking namespace of pods enables a straightforward communication model. However, the dynamic nature of pods presents a problem. Pods can be added on the fly to handle increased traffic. Likewise, pods can be dynamically scaled down. If a pod fails, then Kubernetes automatically replaces the pod with a new one. These events change pod IP addresses.

Figure 4.7: Problem with direct access to pods

In the diagram, the Before side shows the Front-end container that is running in a pod with a 10.8.0.1 IP address. The container also refers to a Back-end container that is running in a pod with a 10.8.0.2 IP address. In this example, an event occurs that causes the Back-end container to fail. A pod can fail for many reasons. In response to the failure, Kubernetes creates a pod for the Back-end container that uses a new IP address of 10.8.0.4. From the After side of the diagram, the Front-end container now has an invalid reference to the Back-end container because of the IP address change. Kubernetes resolves this problem with service resources.

Using Services

Containers inside Kubernetes pods must not connect directly to each other's dynamic IP address. Instead, Kubernetes assigns a stable IP address to a service resource that is linked to a set of specified pods. The service then acts as a virtual network load balancer for the pods that are linked to the service.

If the pods are restarted, replicated, or rescheduled to different nodes, then the service endpoints are updated, thus providing scalability and fault tolerance for your applications. Unlike the IP addresses of pods, the IP addresses of services do not change.

Figure 4.8: Services resolve pod failure issues

In the diagram, the Before side shows that the Front-end container now holds a reference to the stable IP address of the Back-end service, instead of to the IP address of the pod that is running the Back-end container. When the Back-end container fails, Kubernetes creates a pod with the New back-end container to replace the failed pod. In response to the change, Kubernetes removes the failed pod from the service's host list, or service endpoints, and then adds the IP address of the New back-end container pod to the service endpoints. With the addition of the service, requests from the Front-end container to the Back-end container continue to work, because the service is dynamically updated with the IP address change. A service provides a permanent, static IP address for a group of pods that belong to the same deployment or replica set for an application. Until you delete the service, the assigned IP address does not change, and the cluster does not reuse it.

Most real-world applications do not run as a single pod. Applications need to scale horizontally. Multiple pods run the same containers to meet a growing user demand. A *Deployment* resource manages multiple pods that execute the same container. A service provides a single IP address for the whole set, and provides load-balancing for client requests among the member pods.

With services, containers in one pod can open network connections to containers in another pod. The pods, which the service tracks, are not required to exist on the same compute node or in the same namespace or project. Because a service provides a stable IP address for other pods to use, a pod also does not need to discover the new IP address of another pod after a restart. The service provides a stable IP address to use, no matter which compute node runs the pod after each restart.

Figure 4.9: Service with pods on many nodes

The *SERVICE* object provides a stable IP address for the *CLIENT* container on *NODE X* to send a request to any one of the *API* containers.

Kubernetes uses labels on the pods to select the pods that are associated with a service. To include a pod in a service, the pod labels must include each of the selector fields of the service.

Figure 4.10: Service selector match to pod labels

In this example, the selector has a key-value pair of app: myapp. Thus, pods with a matching label of app: myapp are included in the set that is associated with the service. The *selector* attribute of a service is used to identify the set of pods that form the endpoints for the service. Each pod in the set is a an endpoint for the service.

To create a service for a deployment, use the oc expose command:

[user@host ~]$ **oc expose deployment/<deployment-name> [--selector <selector>]**

**[--port <port>][--target-port <target port>][--protocol <protocol>][--name <name>]**

The oc expose command can use the --selector option to specify the label selectors to use. When the command is used without the --selector option, the command applies a selector to match the replication controller or replica set.

The --port option of the oc expose command specifies the port that the service listens on. This port is available only to pods within the cluster. If a port value is not provided, then the port is copied from the deployment configuration.

The --target-port option of the oc expose command specifies the name or number of the container port that the service uses to communicate with the pods.

The --protocol option determines the network protocol for the service. TCP is used by default.

The --name option of the oc expose command can explicitly name the service. If not specified, the service uses the same name that is provided for the deployment.

To view the selector that a service uses, use the -o wide option with the oc get command.

[user@host ~]$ **oc get service db-pod -o wide**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE SELECTOR

db-pod ClusterIP 172.30.108.92 <none> 3306/TCP 108s **app=db-pod**

In this example, db-pod is the name of the service. Pods must use the app=db-pod label to be included in the host list for the db-pod service. To see the endpoints that a service uses, use the oc get endpoints command.

[user@host ~]$ **oc get endpoints**

NAME ENDPOINTS AGE

db-pod **10.8.0.86:3306,10.8.0.88:3306** 27s

This example illustrates a service with two pods in the host list. The oc get endpoints command returns the service endpoints in the current selected project. Add the name of the service to the command to show only the endpoints of a single service. Use the --namepace option to view the endpoints in a different namespace.

Use the oc describe deployment <deployment name> command to view the deployment selector.

[user@host ~]$ **oc describe deployment db-pod**

Name: db-pod

Namespace: deploy-services

CreationTimestamp: Wed, 18 Jan 2023 17:46:03 -0500

Labels: app=db-pod

Annotations: deployment.kubernetes.io/revision: 2

**Selector: app=db-pod**

*...output omitted...*

You can view or parse the selector from the YAML or JSON output for the deployment resource from the spec.selector.matchLabels object. In this example, the -o yaml option of the oc get command returns the selector label that the deployment uses.

[user@host ~]$ **oc get deployment/<deployment\_name> -o yaml**

...***output ommitted***...

selector:

matchLabels:

**app: db-pod**

...***output ommitted***...

Kubernetes DNS for Service Discovery

Kubernetes uses an internal Domain Name System (DNS) server that the DNS operator deploys. The DNS operator creates a default cluster DNS name, and assigns DNS names to services that you define. The DNS operator implements the DNS API from the operator.openshift.io API group. The operator deploys CoreDNS; creates a service resource for the CoreDNS; and then configures the kubelet to instruct pods to use the CoreDNS service IP for name resolution. When a service does not have a cluster IP address, the DNS operator assigns to the service a DNS record that resolves to the set of IP addresses of the pods behind the service.

The DNS server discovers a service from a pod by using the internal DNS server, which is visible only to pods. Each service is dynamically assigned a *Fully Qualified Domain Name* (FQDN) that uses the following format:

***SVC-NAME***.***PROJECT-NAME***.svc.***CLUSTER-DOMAIN***

When a pod is created, Kubernetes provides the container with a /etc/resolv.conf file with similar contents to the following items:

[user@host ~]$ **cat /etc/resolv.conf**

search deploy-services.svc.cluster.local svc.cluster.local ...

nameserver 172.30.0.10

options ndots:5

In this example, deploy-services is the project name for the pod, and cluster.local is the cluster domain.

The nameserver directive provides the IP address of the Kubernetes internal DNS server. The options ndots directive specifies the number of dots that must appear in a name to qualify for an initial absolute query. Alternative hostname values are derived by appending values from the Search directive to the name that is sent to the DNS server.

In the search directive in this example, the svc.cluster.local entry enables any pod to communicate with another pod in the same cluster by using the service name and project name:

***SVC-NAME***.***PROJECT-NAME***

The first entry in the search directive enables a pod to use the service name to specify another pod in the same project. In RHOCP, a project is also the namespace for the pod. The service name alone is sufficient for pods in the same RHOCP project:

***SVC-NAME***

Kubernetes Networking Drivers

Container Network Interface (CNI) plug-ins provide a common interface between the network provider and the container runtime. CNI defines the specifications for plug-ins that configure network interfaces inside containers. Plug-ins that are written to the specification enable different network providers to control the RHOCP cluster network.

Red Hat provides the following CNI plug-ins for a RHOCP cluster:

* OVN-Kubernetes: The default plug-in for first-time installations of RHOCP, starting with RHOCP 4.10.
* OpenShift SDN: An earlier plug-in from RHOCP 3.x; it is incompatible with some later features of RHOCP 4.x.
* Kuryr: A plug-in for integration and performance on OpenStack deployments.

Certified CNI-plugins from other vendors are also compatible with an RHOCP cluster.

The SDN uses CNI plug-ins to create Linux namespaces to partition the usage of resources and processes on physical and virtual hosts. With this implementation, containers inside pods can share network resources, such as devices, IP stacks, firewall rules, and routing tables. The SDN allocates a unique routable IP to each pod, so that you can access the pod from any other service in the same network.

In OpenShift 4.14, OVN-Kubernetes is the default network provider.

OVN-Kubernetes uses Open Virtual Network (OVN) to manage the cluster network. A cluster that uses the OVN-Kubernetes plug-in also runs Open vSwitch (OVS) on each node. OVN configures OVS on each node to implement the declared network configuration.

The OpenShift Cluster Network Operator

RHOCP provides a *Cluster Network Operator* (CNO) that configures OpenShift cluster networking. The CNO is a OpenShift cluster operator that loads and configures Container Network Interface (CNI) plug-ins. As a cluster administrator, execute the following command to observe the status of the CNO:

[user@host ~]$ **oc get -n openshift-network-operator deployment/network-operator**

NAME READY UP-TO-DATE AVAILABLE AGE

network-operator 1/1 1 1 41d

An administrator configures the cluster network operator at installation time. To see the configuration, use the following command:

[user@host ~]$ **oc describe network.config/cluster**

Name: cluster

*...output omitted...*

Spec:

Cluster Network:

Cidr: 10.8.0.0/14

Host Prefix: 23

External IP:

Policy:

Network Type: OVNKubernetes

Service Network:

172.30.0.0/16

*...output omitted...*

|  |  |
| --- | --- |
|  | The Cluster Network CIDR defines the range of IPs for all pods in the cluster. |
|  | The Service Network CIDR defines the range of IPs for all services in the cluster. |

ubernetes Pod and Service Networks

Objectives

* Interconnect applications pods inside the same cluster by using Kubernetes services.

The Software-defined Network

Kubernetes implements software-defined networking (SDN) to manage the network infrastructure of the cluster and user applications. The SDN is a virtual network that encompasses all cluster nodes. The virtual network enables communication between any container or pod inside the cluster. Cluster node processes that Kubernetes pods manage can access the SDN. However, the SDN is not accessible from outside the cluster, nor to regular processes on cluster nodes. With the software-defined networking model, you can manage network services through the abstraction of several networking layers.

With the SDN, you can manage the network traffic and network resources programmatically, so that the organization teams can decide how to expose their applications. The SDN implementation creates a model that is compatible with traditional networking practices. It makes pods akin to virtual machines in terms of port allocation, IP address leasing, and reservation.

With the SDN design, you do not need to change how application components communicate with each other, which helps to containerize legacy applications. If your application is composed of many services that communicate over the TCP/UDP stack, then this approach still works, because containers in a pod use the same network stack.

The following diagram shows how all pods are connected to a shared network:

|  |
| --- |
|  |

Figure 4.5: How the Kubernetes SDN manages the network

Among the many features of SDN, with open standards, vendors can propose their solutions for centralized management, dynamic routing, and tenant isolation.

Kubernetes Networking

Networking in Kubernetes provides a scalable means of communication between containers.

Kubernetes networking provides the following capabilities:

* Highly coupled container-to-container communications
* Pod-to-pod communications
* Pod-to-service communications
* External-to-service communication: covered in [the section called “ Scale and Expose Applications to External Access ”](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch04s07/c2358540-87d5-48de-b49e-6f23bdcd629c)

Kubernetes automatically assigns an IP address to every pod. However, pod IP addresses are unstable, because pods are ephemeral. Pods are constantly created and destroyed across the nodes in the cluster. For example, when you deploy a new version of your application, Kubernetes destroys the existing pods and then deploys new ones.

All containers within a pod share networking resources. The IP address and MAC address that are assigned to the pod are shared among all containers in the pod. Thus, all containers within a pod can reach each other's ports through the loopback address, localhost. Ports that are bound to localhost are available to all containers that run within the pod, but never to containers outside it.

By default, the pods can communicate with each other even if they run on different cluster nodes or belong to different Kubernetes namespaces. Every pod is assigned an IP address in a flat shared networking namespace that has full communication with other physical computers and containers across the network. All pods are assigned a unique IP address from a Classless Inter-Domain Routing (CIDR) range of host addresses. The shared address range places all pods in the same subnet.

Because all the pods are on the same subnet, pods on all nodes can communicate with pods on any other node without the aid of Network Address Translation (NAT). Kubernetes also provides a service subnet, which links the stable IP address of a service resource to a set of specified pods. The traffic is forwarded in a transparent way to the pods; an agent (depending on the network mode that you use) manages routing rules to route traffic to the pods that match the service resource selectors. Thus, pods can be treated much like Virtual Machines (VMs) or physical hosts from the perspective of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration. Kubernetes implements this infrastructure by managing the SDN.

The following illustration gives further insight into how the infrastructure components work along with the pod and service subnets to enable network access between pods inside an OpenShift instance.

Figure 4.6: Network access between pods in a cluster

The shared networking namespace of pods enables a straightforward communication model. However, the dynamic nature of pods presents a problem. Pods can be added on the fly to handle increased traffic. Likewise, pods can be dynamically scaled down. If a pod fails, then Kubernetes automatically replaces the pod with a new one. These events change pod IP addresses.

Figure 4.7: Problem with direct access to pods

In the diagram, the Before side shows the Front-end container that is running in a pod with a 10.8.0.1 IP address. The container also refers to a Back-end container that is running in a pod with a 10.8.0.2 IP address. In this example, an event occurs that causes the Back-end container to fail. A pod can fail for many reasons. In response to the failure, Kubernetes creates a pod for the Back-end container that uses a new IP address of 10.8.0.4. From the After side of the diagram, the Front-end container now has an invalid reference to the Back-end container because of the IP address change. Kubernetes resolves this problem with service resources.

Using Services

Containers inside Kubernetes pods must not connect directly to each other's dynamic IP address. Instead, Kubernetes assigns a stable IP address to a service resource that is linked to a set of specified pods. The service then acts as a virtual network load balancer for the pods that are linked to the service.

If the pods are restarted, replicated, or rescheduled to different nodes, then the service endpoints are updated, thus providing scalability and fault tolerance for your applications. Unlike the IP addresses of pods, the IP addresses of services do not change.

Figure 4.8: Services resolve pod failure issues

In the diagram, the Before side shows that the Front-end container now holds a reference to the stable IP address of the Back-end service, instead of to the IP address of the pod that is running the Back-end container. When the Back-end container fails, Kubernetes creates a pod with the New back-end container to replace the failed pod. In response to the change, Kubernetes removes the failed pod from the service's host list, or service endpoints, and then adds the IP address of the New back-end container pod to the service endpoints. With the addition of the service, requests from the Front-end container to the Back-end container continue to work, because the service is dynamically updated with the IP address change. A service provides a permanent, static IP address for a group of pods that belong to the same deployment or replica set for an application. Until you delete the service, the assigned IP address does not change, and the cluster does not reuse it.

Most real-world applications do not run as a single pod. Applications need to scale horizontally. Multiple pods run the same containers to meet a growing user demand. A *Deployment* resource manages multiple pods that execute the same container. A service provides a single IP address for the whole set, and provides load-balancing for client requests among the member pods.

With services, containers in one pod can open network connections to containers in another pod. The pods, which the service tracks, are not required to exist on the same compute node or in the same namespace or project. Because a service provides a stable IP address for other pods to use, a pod also does not need to discover the new IP address of another pod after a restart. The service provides a stable IP address to use, no matter which compute node runs the pod after each restart.

Figure 4.9: Service with pods on many nodes

The *SERVICE* object provides a stable IP address for the *CLIENT* container on *NODE X* to send a request to any one of the *API* containers.

Kubernetes uses labels on the pods to select the pods that are associated with a service. To include a pod in a service, the pod labels must include each of the selector fields of the service.

Figure 4.10: Service selector match to pod labels

In this example, the selector has a key-value pair of app: myapp. Thus, pods with a matching label of app: myapp are included in the set that is associated with the service. The *selector* attribute of a service is used to identify the set of pods that form the endpoints for the service. Each pod in the set is a an endpoint for the service.

To create a service for a deployment, use the oc expose command:

[user@host ~]$ **oc expose deployment/<deployment-name> [--selector <selector>]**

**[--port <port>][--target-port <target port>][--protocol <protocol>][--name <name>]**

The oc expose command can use the --selector option to specify the label selectors to use. When the command is used without the --selector option, the command applies a selector to match the replication controller or replica set.

The --port option of the oc expose command specifies the port that the service listens on. This port is available only to pods within the cluster. If a port value is not provided, then the port is copied from the deployment configuration.

The --target-port option of the oc expose command specifies the name or number of the container port that the service uses to communicate with the pods.

The --protocol option determines the network protocol for the service. TCP is used by default.

The --name option of the oc expose command can explicitly name the service. If not specified, the service uses the same name that is provided for the deployment.

To view the selector that a service uses, use the -o wide option with the oc get command.

[user@host ~]$ **oc get service db-pod -o wide**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE SELECTOR

db-pod ClusterIP 172.30.108.92 <none> 3306/TCP 108s **app=db-pod**

In this example, db-pod is the name of the service. Pods must use the app=db-pod label to be included in the host list for the db-pod service. To see the endpoints that a service uses, use the oc get endpoints command.

[user@host ~]$ **oc get endpoints**

NAME ENDPOINTS AGE

db-pod **10.8.0.86:3306,10.8.0.88:3306** 27s

This example illustrates a service with two pods in the host list. The oc get endpoints command returns the service endpoints in the current selected project. Add the name of the service to the command to show only the endpoints of a single service. Use the --namepace option to view the endpoints in a different namespace.

Use the oc describe deployment <deployment name> command to view the deployment selector.

[user@host ~]$ **oc describe deployment db-pod**

Name: db-pod

Namespace: deploy-services

CreationTimestamp: Wed, 18 Jan 2023 17:46:03 -0500

Labels: app=db-pod

Annotations: deployment.kubernetes.io/revision: 2

**Selector: app=db-pod**

*...output omitted...*

You can view or parse the selector from the YAML or JSON output for the deployment resource from the spec.selector.matchLabels object. In this example, the -o yaml option of the oc get command returns the selector label that the deployment uses.

[user@host ~]$ **oc get deployment/<deployment\_name> -o yaml**

...***output ommitted***...

selector:

matchLabels:

**app: db-pod**

...***output ommitted***...

Kubernetes DNS for Service Discovery

Kubernetes uses an internal Domain Name System (DNS) server that the DNS operator deploys. The DNS operator creates a default cluster DNS name, and assigns DNS names to services that you define. The DNS operator implements the DNS API from the operator.openshift.io API group. The operator deploys CoreDNS; creates a service resource for the CoreDNS; and then configures the kubelet to instruct pods to use the CoreDNS service IP for name resolution. When a service does not have a cluster IP address, the DNS operator assigns to the service a DNS record that resolves to the set of IP addresses of the pods behind the service.

The DNS server discovers a service from a pod by using the internal DNS server, which is visible only to pods. Each service is dynamically assigned a *Fully Qualified Domain Name* (FQDN) that uses the following format:

***SVC-NAME***.***PROJECT-NAME***.svc.***CLUSTER-DOMAIN***

When a pod is created, Kubernetes provides the container with a /etc/resolv.conf file with similar contents to the following items:

[user@host ~]$ **cat /etc/resolv.conf**

search deploy-services.svc.cluster.local svc.cluster.local ...

nameserver 172.30.0.10

options ndots:5

In this example, deploy-services is the project name for the pod, and cluster.local is the cluster domain.

The nameserver directive provides the IP address of the Kubernetes internal DNS server. The options ndots directive specifies the number of dots that must appear in a name to qualify for an initial absolute query. Alternative hostname values are derived by appending values from the Search directive to the name that is sent to the DNS server.

In the search directive in this example, the svc.cluster.local entry enables any pod to communicate with another pod in the same cluster by using the service name and project name:

***SVC-NAME***.***PROJECT-NAME***

The first entry in the search directive enables a pod to use the service name to specify another pod in the same project. In RHOCP, a project is also the namespace for the pod. The service name alone is sufficient for pods in the same RHOCP project:

***SVC-NAME***

Kubernetes Networking Drivers

Container Network Interface (CNI) plug-ins provide a common interface between the network provider and the container runtime. CNI defines the specifications for plug-ins that configure network interfaces inside containers. Plug-ins that are written to the specification enable different network providers to control the RHOCP cluster network.

Red Hat provides the following CNI plug-ins for a RHOCP cluster:

* OVN-Kubernetes: The default plug-in for first-time installations of RHOCP, starting with RHOCP 4.10.
* OpenShift SDN: An earlier plug-in from RHOCP 3.x; it is incompatible with some later features of RHOCP 4.x.
* Kuryr: A plug-in for integration and performance on OpenStack deployments.

Certified CNI-plugins from other vendors are also compatible with an RHOCP cluster.

The SDN uses CNI plug-ins to create Linux namespaces to partition the usage of resources and processes on physical and virtual hosts. With this implementation, containers inside pods can share network resources, such as devices, IP stacks, firewall rules, and routing tables. The SDN allocates a unique routable IP to each pod, so that you can access the pod from any other service in the same network.

In OpenShift 4.14, OVN-Kubernetes is the default network provider.

OVN-Kubernetes uses Open Virtual Network (OVN) to manage the cluster network. A cluster that uses the OVN-Kubernetes plug-in also runs Open vSwitch (OVS) on each node. OVN configures OVS on each node to implement the declared network configuration.

The OpenShift Cluster Network Operator

RHOCP provides a *Cluster Network Operator* (CNO) that configures OpenShift cluster networking. The CNO is a OpenShift cluster operator that loads and configures Container Network Interface (CNI) plug-ins. As a cluster administrator, execute the following command to observe the status of the CNO:

[user@host ~]$ **oc get -n openshift-network-operator deployment/network-operator**

NAME READY UP-TO-DATE AVAILABLE AGE

network-operator 1/1 1 1 41d

An administrator configures the cluster network operator at installation time. To see the configuration, use the following command:

[user@host ~]$ **oc describe network.config/cluster**

Name: cluster

*...output omitted...*

Spec:

Cluster Network:

Cidr: 10.8.0.0/14

Host Prefix: 23

External IP:

Policy:

Network Type: OVNKubernetes

Service Network:

172.30.0.0/16

*...output omitted...*

|  |  |
| --- | --- |
|  | The Cluster Network CIDR defines the range of IPs for all pods in the cluster. |
|  | The Service Network CIDR defines the range of IPs for all services in the cluster. |

Guided Exercise: Kubernetes Pod and Service Networks

Deploy a database server and access it through a Kubernetes service.

**Outcomes**

You should be able to deploy a database server, and access it indirectly through a Kubernetes service, and also directly pod-to-pod for troubleshooting.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that all resources are available for this exercise. It also creates the deploy-services project and the /home/student/DO180/labs/deploy-services/resources.txt file. The resources.txt file contains some commands that you use during the exercise. You can use the file to copy and paste these commands.

[student@workstation ~]$ **lab start deploy-services**

**Note**

It is safe to ignore pod security warnings for exercises in this course. OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security.

**Instructions**

1. Log in to the OpenShift cluster as the developer user with the developer password. Use the deploy-services project.
   1. Log in to the OpenShift cluster.
   2. [student@workstation ~]$ **oc login -u developer -p developer** \
   3. **https://api.ocp4.example.com:6443**
   4. Login successful.

*...output omitted...*

* 1. Set the deploy-services project as the active project.
  2. [student@workstation ~]$ **oc project deploy-services**

*...output omitted...*

1. Use the registry.ocp4.example.com:8443/rhel8/mysql-80 container image to create a MySQL deployment named db-pod. Add the missing environment variables for the pod to run.
   1. Create the db-pod deployment.
   2. [student@workstation ~]$ **oc create deployment db-pod --port 3306 \**
   3. **--image registry.ocp4.example.com:8443/rhel8/mysql-80**

deployment.apps/db-pod created

* 1. Add the environment variables.
  2. [student@workstation ~]$ **oc set env deployment/db-pod \**
  3. **MYSQL\_USER=user1 \**
  4. **MYSQL\_PASSWORD=mypa55w0rd \**
  5. **MYSQL\_DATABASE=items**

deployment.apps/db-pod updated

* 1. Confirm that the pod is running.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

db-pod-6ccc485cfc-vrc4r 1/1 Running 0 2m30s

Your pod name might differ from the previous output.

1. Expose the db-pod deployment to create a ClusterIP service.
   1. View the deployment for the pod.
   2. [student@workstation ~]$ **oc get deployment**
   3. NAME READY UP-TO-DATE AVAILABLE AGE

db-pod 1/1 1 1 3m36s

* 1. Expose the db-pod deployment to create a service.
  2. [student@workstation ~]$ **oc expose deployment/db-pod**

service/db-pod exposed

1. Validate the service. Confirm that the service selector matches the label on the pod. Then, confirm that the db-pod service endpoint matches the IP of the pod.
   1. Identify the selector for the db-pod service. Use the oc get service command with the -o wide option to retrieve the selector that the service uses.
   2. [student@workstation ~]$ **oc get service db-pod -o wide**
   3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE SELECTOR

db-pod ClusterIP 172.30.108.92 <none> 3306/TCP 108s **app=db-pod**

The selector shows an app=db-pod key:value pair.

* 1. Capture the name of the pod in a variable.
  2. [student@workstation ~]$ **PODNAME=$(oc get pods** \

**-o jsonpath='{.items[0].metadata.name}')**

* 1. Query the label on the pod.
  2. [student@workstation ~]$ **oc get pod $PODNAME --show-labels**
  3. NAME READY STATUS RESTARTS AGE LABELS

db-pod-6ccc485cfc-vrc4r 1/1 Running 0 6m50s **app=db-pod** ...

Notice that the label list includes the app=db-pod key-value pair, which is the selector for the db-pod service.

* 1. Retrieve the endpoints for the db-pod service.
  2. [student@workstation ~]$ **oc get endpoints**
  3. NAME ENDPOINTS AGE

db-pod **10.8.0.85:3306** 4m38s

Your endpoints values might differ from the previous output.

* 1. Verify that the service endpoint matches the db-pod IP address. Use the oc get pods command with the -o wide option to view the pod IP address.
  2. [student@workstation ~]$ **oc get pods -o wide**
  3. NAME READY STATUS RESTARTS AGE IP ...

db-pod-6ccc485cfc-vrc4r 1/1 Running 0 54m **10.8.0.85** ...

The service endpoint resolves to the IP address that is assigned to the pod.

1. Delete and then re-create the db-pod deployment. Confirm that the db-pod service endpoint automatically resolves to the IP address of the new pod.
   1. Delete the db-pod deployment.
   2. [student@workstation ~]$ **oc delete deployment.apps/db-pod**

deployment.apps "db-pod" deleted

* 1. Verify that the service still exists without the deployment.
  2. [student@workstation ~]$ **oc get service**
  3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

db-pod ClusterIP 172.30.108.92 <none> 3306/TCP 9m53s

* 1. The list of endpoints for the service is now empty.
  2. [student@workstation ~]$ **oc get endpoints**
  3. NAME ENDPOINTS AGE

db-pod <none> 12m

* 1. Re-create the db-pod deployment.
  2. [student@workstation ~]$ **oc create deployment db-pod --port 3306 \**
  3. **--image registry.ocp4.example.com:8443/rhel8/mysql-80**

deployment.apps/db-pod created

* 1. Add the environment variables.
  2. [student@workstation ~]$ **oc set env deployment/db-pod \**
  3. **MYSQL\_USER=user1 \**
  4. **MYSQL\_PASSWORD=mypa55w0rd \**
  5. **MYSQL\_DATABASE=items**

deployment.apps/db-pod updated

* 1. Confirm that the newly created pod has the app=db-pod selector.
  2. [student@workstation ~]$ **oc get pods --selector app=db-pod -o wide**
  3. NAME READY STATUS RESTARTS AGE IP ...

**db-pod-6ccc485cfc-l2x** 1/1 Running 0 32s 10.8.0.85 ...

Notice the change in the pod name. The pod IP address might also change. Your pod name and IP address might differ from the previous output.

* 1. Confirm that the endpoints for the db-pod service include the newly created pod.
  2. [student@workstation ~]$ **oc get endpoints**
  3. NAME ENDPOINTS AGE

db-pod **10.8.0.85**:3306 16m

1. Create a pod to identify the available DNS name assignments for the service.
   1. Create a pod named shell to use for troubleshooting. Use the oc run command and the registry.ocp4.example.com:8443/openshift4/network-tools-rhel8 container image.
   2. [student@workstation ~]$ **oc run shell -it \**
   3. **--image registry.ocp4.example.com:8443/openshift4/network-tools-rhel8**
   4. If you don't see a command prompt, try pressing enter.

bash-4.4$

* 1. From the prompt inside the shell pod, view the /etc/resolv.conf file to identify the cluster-domain name.
  2. bash-4.4$ **cat /etc/resolv.conf**
  3. search deploy-services.svc.**cluster.local** svc.**cluster.local** ...
  4. nameserver 172.30.0.10

options ndots:5

The container uses the values from the search directive as suffix values on DNS searches. The container appends these values to a DNS query, in the written order, to resolve the search. The cluster-domain name is the last few components of these values that start after svc.

* 1. Use the nc and echo commands to test the available DNS names for the service.

nc -z ***<service>\_***<server>\_ ***<port>***

The long version of the DNS name is required when accessing the service from a different project. When the pod is in the same project, you can use a shorter version of the DNS name.

bash-4.4$ **nc -z db-pod.deploy-services 3306 && \**

**echo "Connection success to db-pod.deploy-services:3306" || \**

**echo "Connection failed"**

Connection success to db-pod.deploy-services:3306

* 1. Exit the interactive session.
  2. bash-4.4$ **exit**

Session ended, resume using 'oc attach shell -c shell -i -t' command when the pod is running

* 1. Delete the pod for the shell.
  2. [student@workstation ~]$ **oc delete pod shell**

pod "shell" deleted

1. Use a new project to test pod communications across namespaces.
   1. Create a second namespace with the oc new-project command.
   2. [student@workstation ~]$ **oc new-project deploy-services-2**
   3. Now using project "deploy-services-2" on server "https://api.ocp4.example.com:6443".

*...output omitted...*

* 1. Execute the nc and echo commands from a pod to test the DNS name access to another namespace.
  2. [student@workstation ~]$ **oc run shell -it --rm \**
  3. **--image registry.ocp4.example.com:8443/openshift4/network-tools-rhel8 \**
  4. **--restart Never -- nc -z db-pod.deploy-services.svc.cluster.local 3306 && \**
  5. **echo "Connection success to db-pod.deploy-services.svc.cluster.local:3306" \**
  6. **|| echo "Connection failed"**
  7. pod "shell" deleted

Connection success to db-pod.deploy-services.svc.cluster.local:3306

* 1. Return to the deploy-services project.
  2. [student@workstation ~]$ **oc project deploy-services**

Now using project "deploy-services" on server "https://api.ocp4.example.com:6443".

1. Use a Kubernetes job to add initialization data to the database.
   1. Create a job named mysql-init that uses the registry.ocp4.example.com:8443/redhattraining/do180-dbinit:v1 container image. This image uses the mysql-80 container image as a base image, and it includes a script that adds a few initial records to the database.
   2. [student@workstation ~]$ **oc create job mysql-init \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/do180-dbinit:v1 \**
   4. **-- /bin/bash -c "mysql -uuser1 -pmypa55w0rd --protocol tcp \**
   5. **-h db-pod -P3306 items </tmp/db-init.sql"**

job.batch/mysql-init created

The -h option of the mysql command directs the command to communicate with the DNS short name of the db-pod service. The db-pod short name can be used here, because the pod for the job is created in the same namespace as the service.

The double dash -- before /bin/bash separates the oc command arguments from the command in the pod. The -c option of /bin/bash directs the command interpreter in the container to execute the command string. The /tmp/db-init.sql file is redirected as input for the command. The db-init.sql file is included in the image, and contains the following script.

DROP TABLE IF EXISTS `Item`;

CREATE TABLE `Item` (`id` BIGINT not null auto\_increment primary key, `description` VARCHAR(100), `done` BIT);

INSERT INTO `Item` (`id`,`description`,`done`) VALUES (1,'Pick up newspaper', 0);

INSERT INTO `Item` (`id`,`description`,`done`) VALUES (2,'Buy groceries', 1);

* 1. Confirm the status of the mysql-init job. Wait for the job to complete.
  2. [student@workstation ~]$ **oc get job**
  3. NAME COMPLETIONS DURATION AGE

mysql-init **1/1** 22m

* 1. Retrieve the status of the mysql-init job pod, to confirm that the pod has a Completed status.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. db-pod-6ccc485cfc-2lklx 1/1 Running 0 4h24m

mysql-init-ln9cg 0/1 **Completed** 0 23m

* 1. Delete the mysql-init job, because it is no longer needed.
  2. [student@workstation ~]$ **oc delete job mysql-init**

job.batch "mysql-init" deleted

* 1. Verify that the corresponding mysql-init pod is also deleted.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

db-pod-6ccc485cfc-2lklx 1/1 Running 0 4h2

1. Create a query-db pod by using the oc run command and the registry.ocp4.example.com:8443/redhattraining/do180-dbinit:v1 container image. Use the pod to execute a query against the database service.
   1. Create the query-db pod. Configure the pod to use the MySQL client to execute a query against the db-pod service. You can use the db-pod service short name, which provides a stable reference.
   2. [student@workstation ~]$ **oc run query-db -it --rm \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/do180-dbinit:v1 \**
   4. **--restart Never \**
   5. **-- mysql -uuser1 -pmypa55w0rd --protocol tcp \**
   6. **-h db-pod -P3306 items -e 'select \* from Item;'**
   7. mysql: [Warning] Using a password on the command line interface can be insecure.
   8. +----+-------------------+------------+
   9. | id | description | done |
   10. +----+-------------------+------------+
   11. | 1 | Pick up newspaper | 0x00 |
   12. | 2 | Buy groceries | 0x01 |
   13. +----+-------------------+------------+

pod "query-db" deleted

1. It might be necessary to use pod-to-pod communications for troubleshooting. Use the oc run command to create a pod that executes a network test against the IP address of the database pod.
   1. Confirm the IP address of the MySQL database pod. Your pod IP address might differ from the output.
   2. [student@workstation ~]$ **oc get pods -o wide**
   3. NAME READY STATUS RESTARTS AGE IP ...

db-pod-6ccc485cfc-2lklx 1/1 Running 0 4h5 10.8.0.69 ...

* 1. Capture the IP address in an environment variable.
  2. [student@workstation ~]$ **POD\_IP=$(oc get pod -l app=db-pod \**

**-o jsonpath='{.items[0].status.podIP}')**

* 1. Create a test pod named shell with the oc run command. Execute the nc command to test against the $POD\_IP environment variable and the 3306 port for the database.
  2. [student@workstation ~]$ **oc run shell --env POD\_IP=$POD\_IP -it --rm \**
  3. **--image registry.ocp4.example.com:8443/openshift4/network-tools-rhel8 \**
  4. **--restart Never \**
  5. **-- nc -z $POD\_IP 3306 && echo "Connection success to $POD\_IP:3306" \**
  6. **|| echo "Connection failed"**
  7. pod "shell" deleted

Connection success to 10.8.0.69:3306

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish deploy-services**

Scale and Expose Applications to External Access

Objectives

* Expose applications to clients outside the cluster by using Kubernetes ingress and OpenShift routes.

IP Addresses for Pods and Services

Most real-world applications do not run as a single pod. Because applications need to scale horizontally, many pods run the same containers from the same pod resource definition, to meet growing user demand. A service defines a single IP/port combination, and provides a single IP address to a pool of pods, and a load-balancing client request among member pods.

By default, services connect clients to pods in a round-robin fashion, and each service is assigned a unique IP address for clients to connect to. This IP address comes from an internal OpenShift virtual network, which although distinct from the pods' internal network, is visible only to pods. Each pod that matches the selector is added to the service resource as an endpoint.

Containers inside Kubernetes pods must not connect to each other's dynamic IP address directly. Services resolve this problem by linking more stable IP addresses from the SDN to the pods. If pods are restarted, replicated, or rescheduled to different nodes, then services are updated, to provide scalability and fault tolerance.

Service Types

You can choose between several service types depending on your application needs, cluster infrastructure, and security requirements.

**ClusterIP**

This type is the default, unless you explicitly specify a type for a service. The ClusterIP type exposes the service on a cluster-internal IP address. If you choose this value, then the service is reachable only from within the cluster.

The ClusterIP service type is used for pod-to-pod routing within the RHOCP cluster, and enables pods to communicate with and to access each other. IP addresses for the ClusterIP services are assigned from a dedicated service network that is accessible only from inside the cluster. Most applications should use this service type, for which Kubernetes automates the management.

**Load balancer**

This resource instructs RHOCP to activate a load balancer in a cloud environment. A load balancer instructs Kubernetes to interact with the cloud provider that the cluster is running in, to provision a load balancer. The load balancer then provides an externally accessible IP address to the application.

Take all necessary precautions before deploying this service type. Load balancers are typically too expensive to assign one for each application in a cluster. Furthermore, applications that use this service type become accessible from networks outside the cluster. Additional security configuration is required to prevent unintended access.

**NodePort**

With this method, Kubernetes exposes a service on a port on the node IP address. The port is exposed on all cluster nodes, and each node redirects traffic to the endpoints (pods) of the service.

A NodePort service requires allowing direct network connections to a cluster node, which is a security risk.

**ExternalName**

This service tells Kubernetes that the DNS name in the externalName field is the location of the resource that backs the service. When a DNS request is made against the Kubernetes DNS server, it returns the externalName in a *Canonical Name (CNAME)* record, and directs the client to look up the returned name to get the IP address.

Using Routes for External Connectivity

RHOCP provides resources to expose your applications to external networks outside the cluster. You can expose HTTP and HTTPS traffic, TCP applications, and also non-TCP traffic. However, you should expose only HTTP and TLS-based applications to external access. Applications that use other protocols, such as databases, are usually not exposed to external access (from outside a cluster). Routes and ingress are the main resources for handling ingress traffic.

RHOCP provides the route resource to expose your applications to external networks. With routes, you can access your application with a unique hostname that is publicly accessible. Routes rely on a Kubernetes ingress controller to redirect the traffic from the public IP address to pods. By default, Kubernetes provides an ingress controller, starting from the 1.24 release. For RHOCP clusters, the OpenShift ingress operator provides the ingress controller. RHOCP clusters can also use various third-party ingress controllers that can be deployed in parallel with the OpenShift ingress controller.

Routes provide ingress traffic to services in the cluster. Routes were created before Kubernetes ingress objects, and provide more features. Routes provide advanced features that Kubernetes ingress controllers might not support through a standard interface, such as TLS re-encryption, TLS passthrough, and split traffic for blue-green deployments.

To create a route (secure or insecure) with the oc CLI, use the oc expose service *service-name* command. Include the --hostname option to provide a custom hostname for the route.

[user@host ~]$ **oc expose service api-frontend \**

**--hostname api.apps.acme.com**

If you omit the hostname, then RHOCP automatically generates a hostname with the following structure: <route-name>-<project-name>.<default-domain>. For example, if you create a frontend route in an api project, in a cluster that uses apps.example.com as the wildcard domain, then the route hostname is as follows:

**frontend-api.apps.example.com**

**Important**

The DNS server that hosts the wildcard domain is unaware of any route hostnames; it resolves any name only to the configured IPs. Only the RHOCP router knows about route hostnames, and treats each one as an HTTP virtual host.

Invalid wildcard domain hostnames, or hostnames that do not correspond to any route, are blocked by the RHOCP router and result in an HTTP 503 error.

Consider the following settings when creating a route:

* The name of a service. The route uses the service to determine the pods to direct the traffic to.
* A hostname for the route. A route is always a subdomain of your cluster wildcard domain. For example, if you are using a wildcard domain of apps.dev-cluster.acme.com, and need to expose a frontend service through a route, then the route name is as follows:

**frontend.apps.dev-cluster.acme.com.**

RHOCP can also automatically generate a hostname for the route.

* An optional path, for path-based routes.
* A target port that the application listens to. The target port corresponds to the port that you define in the targetPort key of the service.
* An encryption strategy, depending on whether you need a secure or an insecure route.

The following listing shows a minimal definition for a route:

kind: Route

apiVersion: route.openshift.io/v1

metadata:

name: a-simple-route

labels:

app: API

name: api-frontend

spec:

host: api.apps.acme.com

to:

kind: Service

name: api-frontend

port: 8080

targetPort: 8443

|  |  |
| --- | --- |
|  | The name of the route. This name must be unique. |
|  | A set of labels that you can use as selectors. |
|  | The hostname of the route. This hostname must be a subdomain of your wildcard domain, because RHOCP routes the wildcard domain to the routers. |
|  | The service to redirect the traffic to. Although you use a service name, the route uses this information only to determine the list of pods that receive the traffic. |
|  | Port mapping from a router to an endpoint in the service endpoints. The target port on pods that are selected by the service that this route points to. |

**Note**

Some ecosystem components have an integration with ingress resources, but not with route resources. For this case, RHOCP automatically creates managed route objects when an ingress object is created. These route objects are deleted when the corresponding ingress objects are deleted.

You can delete a route by using the oc delete route *route-name* command.

[user@host ~]$ **oc delete route myapp-route**

You can also expose a service from the web console by clicking the **Networking** → **Routes** menu. Click **Create Route** and customize the name, the hostname, the path, and the service to route to by using the form view or the YAML manifest.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Using Ingress Objects for External Connectivity

An ingress is a Kubernetes resource that provides some of the same features as routes (which are an RHOCP resource). Ingress objects accept external requests and transfer the requests based on the route. You can enable only certain types of traffic: HTTP, HTTPS and server name identification (SNI), and TLS with SNI. Standard Kubernetes ingress resources are typically minimal. Many common features that applications rely on, such as TLS termination, path redirecting, and sticky sessions, depend on the ingress controller. Kubernetes does not define the configuration syntax. In RHOCP, routes are generated to meet the conditions that the ingress object specifies.

**Note**

The ingress resource is commonly used for Kubernetes. However, the route resource is the preferred method for external connectivity in RHOCP.

To create an ingress object, use the oc create ingress ingress-name --rule=URL\_route=service-name:port-number command. Use the --rule option to provide a custom rule in the host/path=service:port[,tls=secretname] format. If the TLS option is omitted, then an insecure route is created.

[user@host ~]$ **oc create ingress ingr-sakila \**

**--rule="ingr-sakila.apps.ocp4.example.com/\*=sakila-service:8080"**

The following listing shows a minimal definition for an ingress object:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: frontend

spec:

rules:

- host: "www.example.com

http:

paths:

- backend:

service:

name: frontend

port:

number: 80

pathType: Prefix

path: /

tls:

- hosts:

- www.example.com

secretName: example-com-tls-certificate

|  |  |
| --- | --- |
|  | The name of the ingress object. This name must be unique. |
|  | The HTTP or HTTPS rule for the ingress object. |
|  | The host for the ingress object. Applies the HTTP rule to the inbound HTTP traffic of the specified host. |
|  | The backend to redirect traffic to. Defines the service name, port number, and port names for the ingress object. To connect to the back end, incoming requests must match the host and path of the rule. |
|  | The configuration of TLS for the ingress object; it is required for secured paths. The host in the TLS object must match the host in the rules object. |

You can delete an ingress object by using the oc delete ingress *ingress-name* command.

[user@host ~]$ **oc delete ingress example-ingress**

Sticky Sessions

Sticky sessions enable stateful application traffic by ensuring that all requests reach the same endpoint. RHOCP uses cookies to configure session persistence for ingress and route resources. The ingress controller selects an endpoint to handle any user requests, and creates a cookie for the session. The cookie is passed back in response to the request, and the user sends back the cookie with the next request in the session. The cookie tells the ingress controller which endpoint is handling the session, to ensure that client requests use the cookie so that they are routed to the same pod.

RHOCP auto-generates the cookie name for ingress and route resources. You can overwrite the default cookie name by using the annotate command with either the kubectl or the oc commands. With this annotation, the application that receives route traffic knows the cookie name.

The following example configures a cookie for an ingress object:

[user@host ~]$ **oc annotate ingress ingr-example \**

**ingress.kubernetes.io/affinity=cookie**

The following example configures a cookie named myapp for a route object:

[user@host ~]$ **oc annotate route route-example \**

**router.openshift.io/cookie\_name=myapp**

After you annotate the route, capture the route hostname in a variable:

[user@host ~]$ **ROUTE\_NAME=$(oc get route <route\_name> \**

**-o jsonpath='{.spec.host}')**

Then, use the curl command to save the cookie and access the route:

[user@host ~]$ **curl $ROUTE\_NAME -k -c /tmp/cookie\_jar**

The cookie is passed back in response to the request, and is saved to the /tmp/cookie\_jar directory. Use the curl command and the cookie that was saved from the previous command to connect to the route:

[user@host ~]$ **curl $ROUTE\_NAME -k -b /tmp/cookie\_jar**

By using the saved cookie, the request is sent to the same pod as the previous request.

Load Balance and Scale Applications

Developers and administrators can choose to manually scale the number of replica pods in a deployment. More pods might be needed for an anticipated surge in traffic, or the pod count might be reduced to reclaim resources that the cluster can use elsewhere.

You can change the number of replicas in a deployment resource manually by using the oc scale command.

[user@host ~]$ **oc scale --replicas 5 deployment/scale**

The deployment resource propagates the change to the replica set. The replica set reacts to the change by creating pods (replicas) or by deleting existing ones, depending on whether the new intended replica count is less than or greater than the existing count.

Although you can manipulate a replica set resource directly, the recommended practice is to manipulate the deployment resource instead. A new deployment creates either a replica set or a replication controller, and direct changes to a previous replica set or replication controller are ignored.

Load Balance Pods

A Kubernetes service serves as an internal load balancer. Standard services act as a load balancer or a proxy, and give access to the workload object by using the service name. A service identifies a set of replicated pods to transfer the connections that it receives.

A router uses the service selector to find the service and the endpoints, or pods, that back the service. When both a router and a service provide load balancing, RHOCP uses the router to load-balance traffic to pods. A router detects relevant changes in the IP addresses of its services, and adapts its configuration accordingly. Custom routers can thereby communicate modifications of API objects to an external routing solution.

RHOCP routers map external hostnames, and load-balance service endpoints over protocols that pass distinguishing information directly to the router. The hostname must exist in the protocol for the router to determine where to send it.

Guided Exercise: Scale and Expose Applications to External Access

Deploy one web server and access it through a Kubernetes ingress; and deploy another web server and access it through an OpenShift route.

**Outcomes**

In this exercise, you deploy two web applications to access them through an ingress object and a route, and scale them to verify the load-balance between the pods.

* Deploy two web applications.
* Create a route and an ingress object to access the web applications.
* Enable the sticky sessions for the web applications.
* Scale the web applications to load-balance the service.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise. This command ensures that the cluster is accessible.

[student@workstation ~]$ **lab start deploy-routes**

**Instructions**

1. Create two web application deployments, named satir-app and sakila-app. Use the registry.ocp4.example.com:8443/httpd-app:v1 container image for both deployments.
   1. Log in to the OpenShift cluster as the developer user with the developer password.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**
   4. Login successful

*...output omitted...*

* 1. Change to the web-applications project.
  2. [student@workstation ~]$ **oc project web-applications**
  3. Now using project "web-applications" on server "https://api.ocp4.example.com:6443".

*...output omitted...*

* 1. Create the satir-app web application deployment by using the registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1 container image. Ignore the warning message.
  2. [student@workstation ~]$ **oc create deployment satir-app \**
  3. **--image registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1**

deployment.apps/satir-app created

* 1. After a few moments, verify that the deployment is successful.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS ...

satir-app-787b7d7858-5dfsh 1/1 Running 0 ...

[student@workstation ~]$ **oc status**

*...output omitted...*

deployment/satir-app deploys registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1

deployment #1 running for 20 seconds - 1 pod

*...output omitted...*

* 1. Create the sakila-app web application deployment by using the registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1 image. Ignore the warning message.
  2. [student@workstation ~]$ **oc create deployment sakila-app \**
  3. **--image registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1**

deployment.apps/sakila-app created

* 1. Wait a few moments and then verify that the deployment is successful.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS ...
  4. sakila-app-6694...5kpd 1/1 Running 0 ...

satir-app-787b7...dfsh 1/1 Running 0 ...

[student@workstation ~]$ **oc status**

*...output omitted...*

deployment/satir-app deploys registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1

deployment #1 running for 5 minutes - 1 pod

deployment/sakila-app deploys registry.ocp4.example.com:8443/redhattraining/do180-httpd-app:v1

deployment #1 running for 2 minutes - 1 pod

*...output omitted...*

1. Create services for the web application deployments. Then, use the services to create a route for the satir-app application and an ingress object for the sakila-app application.
   1. Expose the satir-app deployment. Name the service satir-svc, and specify port 8080 as the port and target port.
   2. [student@workstation ~]$ **oc expose deployment satir-app --name satir-svc \**
   3. **--port 8080 --target-port 8080**

service/satir-svc exposed

* 1. Expose the sakila-app deployment to create the sakila-svc service.
  2. [student@workstation ~]$ **oc expose deployment sakila-app --name sakila-svc \**
  3. **--port 8080 --target-port 8080**

service/sakila-svc exposed

* 1. Verify the status of the services.
  2. [student@workstation ~]$ **oc get services**
  3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) ...
  4. sakila-svc ClusterIP 172.30.230.41 <none> 8080/TCP ...

satir-svc ClusterIP 172.30.143.15 <none> 8080/TCP ...

[student@workstation ~]$ **oc get endpoints**

NAME ENDPOINTS ...

sakila-svc 10.8.0.66:8080 ...

satir-svc 10.8.0.65:8080 ...

[student@workstation ~]$ **oc get pods -o wide**

NAME READY STATUS RESTARTS AGE IP ...

sakila-app-6694...5kpd 1/1 Running 0 92s 10.8.0.66 ...

satir-app-787b7...dfsh 1/1 Running 0 2m49s 10.8.0.65 ...

* 1. Create a route named satir for the satir-app web application by exposing the satir-svc service.
  2. [student@workstation ~]$ **oc expose service satir-svc --name satir**

route.route.openshift.io/satir exposed

[student@workstation ~]$ **oc get routes**

NAME HOST/PORT ... SERVICES PORT ...

satir satir-web-applications.apps.ocp4.example.com ... satir-svc 8080 ...

* 1. Create an ingress object named ingr-sakila for the sakila-svc service. Configure the --rule option with the following values:

| **Field** | **Value** |
| --- | --- |
| Host | ingr-sakila.apps.ocp4.example.com |
| Service name | sakila-svc |
| Port number | 8080 |

* 1. [student@workstation ~]$ **oc create ingress ingr-sakila \**
  2. **--rule "ingr-sakila.apps.ocp4.example.com/\*=sakila-svc:8080"**
  3. ingress.networking.k8s.io/ingr-sakila created
  4. [student@workstation ~]$ **oc get ingress**
  5. NAME ... HOSTS ADDRESS PORTS ...
  6. ingr-sakila ... ingr-sakila.apps.ocp4.example.com router...com 80 ...
  7. Confirm that a route exists for the ingr-sakila ingress object.
  8. [student@workstation ~]$ **oc get routes**
  9. NAME HOST/PORT ... SERVICES PORT
  10. ingr-sakila... ingr-sakila.apps.ocp4.example.com ... sakila-svc <all>

satir satir-web-applications.apps.ocp4.example.com ... satir-svc 8080

A specific port is not assigned to routes that ingress objects created. By contrast, a route that an exposed service created is assigned the same ports as the service.

* 1. Use the curl command to access the ingr-sakila ingress object and the satir route. The output states the name of the pod that is servicing the request.
  2. [student@workstation ~]$ **curl ingr-sakila.apps.ocp4.example.com**

Welcome to Red Hat Training, from sakila-app-66947cdd78-x5kpd

[student@workstation ~]$ **curl satir-web-applications.apps.ocp4.example.com**

Welcome to Red Hat Training, from satir-app-787b7d7858-bdfsh

1. Scale the web application deployments to load-balance their services. Scale the sakila-app to two replicas, and the satir-app to three replicas.
   1. Scale the sakila-app deployment with two replicas.
   2. [student@workstation ~]$ **oc scale deployment sakila-app --replicas 2**

deployment.apps/sakila-app scaled

* 1. Wait a few moments and then verify the status of the replica pods.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS ...
  4. sakila-app-6694...5kpd 1/1 Running 0 ...
  5. sakila-app-6694...rfzg 1/1 Running 0 ...

satir-app-787b...dfsh 1/1 Running 0 ...

* 1. Scale the satir-app deployment with three replicas.
  2. [student@workstation ~]$ **oc scale deployment satir-app --replicas 3**

deployment.apps/satir-app scaled

* 1. Wait a few moments and then verify the status of the replica pods.
  2. [student@workstation ~]$ **oc get pods -o wide**
  3. NAME READY STATUS RESTARTS ... IP ...
  4. sakila-app-6694...5kpd 1/1 Running 0 ... 10.8.0.66 ...
  5. sakila-app-6694...rfzg 1/1 Running 0 ... 10.8.0.67 ...
  6. satir-app-787b...dfsh 1/1 Running 0 ... 10.8.0.65 ...
  7. satir-app-787b...z8xm 1/1 Running 0 ... 10.8.0.69 ...

satir-app-787b...7bhj 1/1 Running 0 ... 10.8.0.70 ...

* 1. Retrieve the service endpoints to confirm that the services are load-balanced between the additional replica pods.
  2. [student@workstation ~]$ **oc get endpoints**
  3. NAME ENDPOINTS ...
  4. sakila-svc 10.8.0.66:8080,10.8.0.67:8080 ...

satir-svc 10.8.0.65:8080,10.8.0.69:8080,10.8.0.70:8080 ...

1. Enable the sticky sessions for the sakila-app web application. Then, use the curl command to confirm that the sticky sessions are working for the ingr-sakila object.
   1. Configure a cookie for the ingr-sakila ingress object.
   2. [student@workstation ~]$ **oc annotate ingress ingr-sakila \**
   3. **ingress.kubernetes.io/affinity=cookie**

ingress.networking.k8s.io/ingr-sakila annotated

* 1. Use the curl command to access the ingr-sakila ingress object. The output states the name of the pod that is servicing the request. Notice that the connection is load-balanced between the replicas.
  2. [student@workstation ~]$ **for i in {1..3}; do \**
  3. **curl ingr-sakila.apps.ocp4.example.com ; done**
  4. Welcome to Red Hat Training, from sakila-app-66947cdd78-x5kpd
  5. Welcome to Red Hat Training, from sakila-app-66947cdd78-xrfzg

Welcome to Red Hat Training, from sakila-app-66947cdd78-x5kpd

* 1. Use the curl command to save the ingr-sakila ingress object cookie to the /tmp/cookie\_jar file. Confirm that the cookie exists in the /tmp/cookie\_jar file.
  2. [student@workstation ~]$ **curl ingr-sakila.apps.ocp4.example.com \**
  3. **-c /tmp/cookie\_jar**

Welcome to Red Hat Training, from sakila-app-66947cdd78-xrfzg

[student@workstation ~]$ **cat /tmp/cookie\_jar**

*...output omitted...*

#HttpOnly\_ingr-sakila.apps.ocp4.example.com FALSE / FALSE 0 b9b484110526b4b1b3159860d3aebe04 921e139c5145950d00424bf3b0a46d22

* 1. The cookie provides session stickiness for connections to the ingr-sakila route. Use the curl command and the cookie in the /tmp/cookie\_jar file to connect to the ingr-sakila route again. Confirm that you are connected to the same pod that handled the request in the previous step.
  2. [student@workstation ~]$ **for i in {1..3}; do \**
  3. **curl ingr-sakila.apps.ocp4.example.com -b /tmp/cookie\_jar; done**
  4. Welcome to Red Hat Training, from sakila-app-66947cdd78-xrfzg
  5. Welcome to Red Hat Training, from sakila-app-66947cdd78-xrfzg

Welcome to Red Hat Training, from sakila-app-66947cdd78-xrfzg

* 1. Use the curl command to connect to the ingr-sakila route without the cookie. Observe that session stickiness occurs only with the cookie.
  2. [student@workstation ~]$ **for i in {1..3}; do \**
  3. **curl ingr-sakila.apps.ocp4.example.com ; done**
  4. Welcome to Red Hat Training, from sakila-app-66947cdd78-x5kpd
  5. Welcome to Red Hat Training, from sakila-app-66947cdd78-xrfzg

Welcome to Red Hat Training, from sakila-app-66947cdd78-x5kpd

1. Enable the sticky sessions for the satir-app web application. Then, use the curl command to confirm that sticky sessions are active for the satir route.
   1. Configure a cookie with a hello value for the satir route.
   2. [student@workstation ~]$ **oc annotate route satir \**
   3. **router.openshift.io/cookie\_name="hello"**

route.route.openshift.io/satir annotated

* 1. Use the curl command to access the satir route. The output states the name of the pod that is servicing the request. Notice that the connection is load-balanced between the three replica pods.
  2. [student@workstation ~]$ **for i in {1..3}; do \**
  3. **curl satir-web-applications.apps.ocp4.example.com; done**
  4. Welcome to Red Hat Training, from satir-app-787b7d7858-bdfsh
  5. Welcome to Red Hat Training, from satir-app-787b7d7858-gz8xm

Welcome to Red Hat Training, from satir-app-787b7d7858-q7bhj

* 1. Use the curl command to save the hello cookie to the /tmp/cookie\_jar file. Afterward, confirm that the hello cookie exists in the /tmp/cookie\_jar file.
  2. [student@workstation ~]$ **curl satir-web-applications.apps.ocp4.example.com \**
  3. **-c /tmp/cookie\_jar**

Welcome to Red Hat Training, from satir-app-787b7d7858-q7bhj

[student@workstation ~]$ **cat /tmp/cookie\_jar**

*...output omitted...*

#HttpOnly\_satir-web-applications.apps.ocp4.example.com FALSE / FALSE 0 hello b7dd73d32003e513a072e25a32b6c881

* 1. The hello cookie provides session stickiness for connections to the satir route. Use the curl command and the hello cookie in the /tmp/cookie\_jar file to connect to the satir route again. Confirm that you are connected to the same pod that handled the request in the previous step.
  2. [student@workstation ~]$ **for i in {1..3}; do \**
  3. **curl satir-web-applications.apps.ocp4.example.com -b /tmp/cookie\_jar; done**
  4. Welcome to Red Hat Training, from satir-app-787b7d7858-q7bhj
  5. Welcome to Red Hat Training, from satir-app-787b7d7858-q7bhj

Welcome to Red Hat Training, from satir-app-787b7d7858-q7bhj

* 1. Use the curl command to connect to the satir route without the hello cookie. Observe that session stickiness occurs only with the cookie.
  2. [student@workstation ~]$ **for i in {1..3}; do \**
  3. **curl satir-web-applications.apps.ocp4.example.com; done**
  4. Welcome to Red Hat Training, from satir-app-787b7d7858-gz8xm
  5. Welcome to Red Hat Training, from satir-app-787b7d7858-q7bhj

Welcome to Red Hat Training, from satir-app-787b7d7858-bdfsh

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish deploy-routes**

Lab: Deploy Managed and Networked Applications on Kubernetes

Deploy a database server and a web application that connects to that database and expose the web application to external access.

**Outcomes**

* Deploy a MySQL database from a container image.
* Deploy a web application from a container image.
* Configure environment variables for a deployment.
* Expose the web application for external access.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that the cluster is accessible and that all exercise resources are available. It also creates the database-applications project.

[student@workstation ~]$ **lab start deploy-review**

**Instructions**

The API URL of your OpenShift cluster is https://api.ocp4.example.com:6443, and the oc command is already installed on your workstation machine.

Log in to the OpenShift cluster as the developer user with the developer password.

Use the database-applications project for your work.

1. Log in to the OpenShift cluster and change to the database-applications project.
   1. Log in to the OpenShift cluster.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**
   4. Login successful

*...output omitted...*

* 1. Change to the database-applications project.
  2. [student@workstation ~]$ **oc project database-applications**
  3. Now using project "database-applications" on server "https://api.ocp4.example.com:6443".

*...output omitted...*

1. Hide Solution
2. Create a MySQL database deployment named mysql-app by using the registry.ocp4.example.com:8443/redhattraining/mysql-app:v1 image, and identify the root cause of the failure.
   1. Create the MySQL database deployment. Ignore the warning message.
   2. [student@workstation ~]$ **oc create deployment mysql-app \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/mysql-app:v1**

deployment.apps/mysql-app created

* 1. Verify the deployment status. The pod name might differ in your output.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. mysql-app-75dfd58f99-5xfqc 0/1 Error ...
  5. [student@workstation ~]$ **oc status**
  6. *...output omitted...*
  7. Errors:
  8. pod/mysql-app-75dfd58f99-5xfqc is crash-looping

1 error, 1 info identified, use 'oc status --suggest' to see details.

* 1. Identify the root cause of the deployment failure.
  2. [student@workstation ~]$ **oc logs mysql-app-*75dfd58f99-5xfqc***
  3. *...output omitted...*
  4. You must either specify the following environment variables:
  5. MYSQL\_USER
  6. MYSQL\_PASSWORD
  7. MYSQL\_DATABASE
  8. Or the following environment variable:
  9. MYSQL\_ROOT\_PASSWORD (regex: '^[a-zA-Z0-9\_~!@#$%^&\*()-=<>,.?;:|]+$')

*...output omitted...*

1. Hide Solution
2. Configure the environment variables for the mysql-app deployment by using the following information:

| **Field** | **Value** |
| --- | --- |
| MYSQL\_USER | redhat |
| MYSQL\_PASSWORD | redhat123 |
| MYSQL\_DATABASE | world\_x |

1. Then, execute the following command in the mysql-app deployment pod to load the world\_x database:
2. /bin/bash -c "mysql -uredhat -predhat123 </tmp/world\_x.sql"
   1. Update the environment variables for the mysql-app deployment.
   2. [student@workstation ~]$ **oc set env deployment/mysql-app \**
   3. **MYSQL\_USER=redhat MYSQL\_PASSWORD=redhat123 MYSQL\_DATABASE=world\_x**

deployment.apps/mysql-app updated

* 1. Verify that the mysql-app application pod is in the RUNNING state. The pod name might differ in your output.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...

mysql-app-57c44f646-5qt2k 1/1 Running ...

* 1. Load the world\_x database.
  2. [student@workstation ~]$ **oc exec -it mysql-app-*57c44f646-5qt2k* \**
  3. **-- /bin/bash -c "mysql -uredhat -predhat123 </tmp/world\_x.sql"**
  4. ...output omitted..

[student@workstation ~]$

* 1. Confirm that you can access the MySQL database.

[student@workstation ~]$ **oc rsh mysql-app-*57c44f646-5qt2k***

sh-4.4$ **mysql -uredhat -predhat123 world\_x**

*...output omitted...*

mysql>

* 1. Exit the MySQL database, and then exit the container.
  2. mysql> **exit**
  3. Bye

sh-4.4$ **exit**

1. Hide Solution
2. Create a service for the mysql-app deployment by using the following information:

| **Field** | **Value** |
| --- | --- |
| Name | mysql-service |
| Port | 3306 |
| Target port | 3306 |

* 1. Expose the mysql-app deployment.
  2. [student@workstation ~]$ **oc expose deployment mysql-app --name mysql-service \**
  3. **--port 3306 --target-port 3306**

service/mysql-service created

* 1. Verify the service configuration. The endpoint IP address might differ in your output.
  2. [student@workstation ~]$ **oc get services**
  3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
  4. mysql-service ClusterIP 172.30.146.213 <none> 3306/TCP 10s
  5. [student@workstation ~]$ **oc get endpoints**
  6. NAME ENDPOINTS AGE

mysql-service 10.8.0.102:3306 19s

1. Hide Solution
2. Create a web application deployment named php-app by using the registry.ocp4.example.com:8443/redhattraining/php-webapp:v1 image.
   1. Create the web application deployment. Ignore the warning message.
   2. [student@workstation ~]$ **oc create deployment php-app \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/php-webapp:v1**

deployment.apps/php-app created

* 1. Verify the deployment status. Verify that the php-app application pod is in the RUNNING state.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. php-app-725... 1/1 Running ...

mysql-app-57c... 1/1 Running ...

[student@workstation ~]$ **oc status**

*...output omitted...*

deployment/php-app deploys registry.ocp4.example.com:8443/redhattraining/php-webapp:v1

deployment #1 running for about a minute - 1 pod

*...output omitted...*

1. Hide Solution
2. Create a service for the php-app deployment by using the following information:

| **Field** | **Value** |
| --- | --- |
| Name | php-svc |
| Port | 8080 |
| Target port | 8080 |

1. Then, create a route named phpapp to expose the web application to external access.
   1. Expose the php-app deployment.
   2. [student@workstation ~]$ **oc expose deployment php-app --name php-svc \**
   3. **--port 8080 --target-port 8080**

service/php-svc exposed

* 1. Verify the service configuration. The endpoint IP address might differ in your output.
  2. [student@workstation ~]$ **oc get services**
  3. NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
  4. mysql-service ClusterIP 172.30.146.213 <none> 3306/TCP 7m47s
  5. **php-svc** ClusterIP 172.30.228.80 <none> 8080/TCP 4m34s
  6. [student@workstation ~]$ **oc get endpoints**
  7. NAME ENDPOINTS AGE
  8. mysql-service 10.8.0.102:3306 7m50s

**php-svc** 10.8.0.107:8080 4m37s

* 1. Expose the php-svc service.
  2. [student@workstation ~]$ **oc expose service/php-svc --name phpapp**

route.route.openshift.io/phpapp exposed

[student@workstation ~]$ **oc get routes**

NAME HOST/PORT ...

phpapp phpapp-database-applications.apps.ocp4.example.com ...

1. Hide Solution
2. Test the connectivity between the web application and the MySQL database. In a web browser, navigate to the phpapp-database-applications.apps.ocp4.example.com route, and verify that the application retrieves data from the MySQL database.
   1. Navigate to the phpapp-database-applications.apps.ocp4.example.com route in the web browser.

|  |
| --- |
| A table with text on it  Description automatically generated |

1. Hide Solution

**Evaluation**

As the student user on the workstation machine, use the lab command to grade your work. Correct any reported failures and rerun the command until successful.

[student@workstation ~]$ **lab grade deploy-review**

**Finish**

As the student user on the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish deploy-review**

## **Summary**

* Many resources in Kubernetes and RHOCP create or affect pods.
* Resources are created imperatively or declaratively. The imperative strategy instructs the cluster what to do. The declarative strategy defines the state that the cluster matches.
* The oc new-app command creates resources that are determined via heuristics.
* The main way to deploy an application is by creating a deployment.
* The workload API includes several resources to create pods. The choice between resources depends on for how long and how often the pod needs to run.
* A job resource executes a one-time task on the cluster via a pod. The cluster retries the job until it succeeds, or it retries a specified number of attempts.
* Resources are organized into projects and are selected via labels.
* A route connects a public-facing IP address and a DNS hostname to an internal-facing service IP address. Services provide network access between pods, whereas routes provide network access to pods from users and applications outside the RHOCP cluster.