Externalize the Configuration of Applications

Objectives

* Configure applications by using Kubernetes secrets and configuration maps to initialize environment variables and to provide text and binary configuration files.

Configuring Kubernetes Applications

When an application is run in Kubernetes with a pre-existing image, the application uses the default configuration. This action is valid for testing purposes. However, for production environments, you might need to customize your applications before deploying them.

With Kubernetes, you can use manifests in JSON and YAML formats to specify the intended configuration for each application. You can define the name of the application, labels, the image source, storage, environment variables, and more.

The following snippet shows an example of a YAML manifest file of a deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: hello-deployment

spec:

replicas: 1

selector:

matchLabels:

app: hello-deployment

template:

metadata:

labels:

app: hello-deployment

spec:

containers:

- env:

- name: ENV\_VARIABLE\_1

valueFrom:

secretKeyRef:

key: hello

name: world

image: quay.io/hello-image:latest

|  |  |
| --- | --- |
|  | API version of the resource. |
|  | Deployment resource type. |
|  | In this section, you specify the metadata of your application, such as the name. |
|  | You can define the general configuration of the resource that is applied to the deployment, such as the number of replicas (pods), the selector label, and the template data. |
|  | In this section, you specify the configuration for your application, such as the image name, the container name, ports, environment variables, and more. |
|  | You can define the environment variables to configure your application needs. |

Sometimes your application requires configuring a combination of files. For example, at the time of creation, a database deployment must have preloaded databases and data. You most commonly configure applications by using environment variables, external files, or command-line arguments. This process of configuration externalization ensures that the application is portable across environments when the container image, external files, and environment variables are available in the environment where the application runs.

Kubernetes provides a mechanism to externalize the configuration of your applications by using configuration maps and secrets.

You can use configuration maps to inject containers with configuration data. The *ConfigMap* (configuration map) namespaced objects provide ways to inject configuration data into containers, which helps to maintain platform independence of the containers. These objects can store fine-grained information, such as individual properties, or coarse-grained information, such as entire configuration files or JSON blobs (JSON sections). The information in configuration maps does not require protection.

The following listing shows an example of a configuration map:

apiVersion: v1

kind: ConfigMap

metadata:

name: example-configmap

namespace: my-app

data:

example.property.1: hello

example.property.2: world

example.property.file: |-

property.1=value-1

property.2=value-2

property.3=value-3

binaryData:

bar: L3Jvb3QvMTAw

|  |  |
| --- | --- |
|  | ConfigMap resource type. |
|  | Contains the configuration data. |
|  | Points to an encoded file in base64 that contains non-UTF-8 data, for example, a binary Java keystore file. Place a key followed by the encoded file. |

Applications often require access to sensitive information. For example, a back-end web application requires access to database credentials to query a database. Kubernetes and OpenShift use secrets to hold sensitive information. For example, you can use secrets to store the following types of sensitive information:

* Passwords
* Sensitive configuration files
* Credentials to an external resource, such as an SSH key or OAuth token

The following listing shows an example 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

|  |  |
| --- | --- |
|  | Specifies the type of secret. |
|  | Specifies the encoded string and data. |
|  | Specifies the decoded string and data. |

A secret is a namespaced object and it can store any type of data. Data in a secret is Base64-encoded, and is not stored in plain text. Secret data is not encrypted; you can decode the secret from Base64 format to access the original data. The following example shows the decoded values for the username and password objects from the example-secret secret:

[user@host] **echo bXl1c2VyCg== | base64 --decode**

myuser

[user@host] **echo bXlQQDU1Cg== | base64 --decode**

myP@55

Kubernetes and OpenShift support the following types of secrets:

* Opaque secrets: An opaque secret store key and value pairs that contain arbitrary values, and are not validated to conform to any convention for key names or values.
* Service account tokens: Store a token credential for applications that authenticate to the Kubernetes API.
* Basic authentication secrets: Store the needed credentials for basic authentication. The data parameter of the secret object must contain the user and the password keys that are encoded in the Base64 format.
* SSH keys: Store data that is used for SSH authentication.
* TLS certificates: Store a certificate and a key that are used for TLS.
* Docker configuration secrets: Store the credentials for accessing a container image registry.

When you store information in a specific secret resource type, Kubernetes validates that the data conforms to the type of secret.

**Note**

By default, configuration maps and secrets are not encrypted. To encrypt your secret data at rest, you must encrypt the Etcd database. When enabled, Etcd encrypts the following resources: secrets, configuration maps, routes, OAuth access tokens, and OAuth authorization tokens. Encrypting the Etcd database is outside the scope of the course.

For more information, refer to the *Encrypting Etcd Data* chapter in the Red Hat OpenShift Container Platform 4.14 *Security and Compliance* documentation at <https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/security_and_compliance/index#encrypting-etcd>

Creating Secrets

If a pod requires access to sensitive information, then create a secret for the information before you deploy the pod. Both the oc and kubectl command-line tools provide the create secret command. Use one of the following commands to create a secret:

* Create a generic secret that contains key-value pairs from literal values that are typed on the command line:
* [user@host ~]$ **oc create secret generic secret\_name \**
* **--from-literal key1=secret1 \**

**--from-literal key2=secret2**

* Create a generic secret by using key names that are specified on the command line and values from files:
* [user@host ~]$ **kubectl create secret generic ssh-keys \**
* **--from-file id\_rsa=/path-to/id\_rsa \**

**--from-file id\_rsa.pub=/path-to/id\_rsa.pub**

* Create a TLS secret that specifies a certificate and the associated key:
* [user@host ~]$ **oc create secret tls secret-tls \**

**--cert /path-to-certificate --key /path-to-key**

To create an opaque secret from the web console, click the **Workloads** → **Secrets** menu. Click **Create** and select **Key/value secret**. Complete the form with the key name, and specify the value by writing it in the following section, or by extracting it from a file.

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

To create a secret from the web console that stores the credentials for accessing a container image registry, click the **Workloads** → **Secrets** menu. Click **Create** and select **Image pull secret**. Complete the form or upload a configuration file with the secret name, select the authentication type, and add the registry server address, the username, password, and email credentials.

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

Creating Configuration Maps

The syntax for creating a configuration map and for creating a secret closely match. You can enter key-value pairs on the command line, or use the content of a file as the value of a specified key. You can use either the oc or kubectl command-line tools to create a configuration map. The following command shows how to create a configuration map:

[user@host ~]$ **kubectl create configmap my-config \**

**--from-literal key1=config1 --from-literal key2=config2**

You can also use the cm shortname to create a configuration map.

[user@host ~]$ **oc create cm my-config \**

**--from-literal key1=config1 --from-literal key2=config2**

To create a configuration map from the web console, click the **Workloads** → **ConfigMaps** menu. Click **Create ConfigMap** and complete the configuration map by using the form view or the YAML view.

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

You can use files on each key that you add by clicking **Browse** beside the **Value** field. The **Key** field must be the name of the added file in the **Value** field.

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

**Note**

Use a binary data key instead of a data key if the file uses the binary format, such as a PNG file.

Using Configuration Maps and Secrets to Initialize Environment Variables

You can use configuration maps to populate individual environment variables that configure your application. Unlike secrets, the information in configuration maps does not require protection. The following listing shows an initialization example of environment variables:

apiVersion: v1

kind: ConfigMap

metadata:

name: config-map-example

namespace: example-app

data:

database.name: sakila

database.user: redhat

|  |  |
| --- | --- |
|  | The project where the configuration map resides. ConfigMap objects can be referenced only by pods in the same project. |
|  | Initializes the database.name variable to the sakila value. |
|  | Initializes the database.user variable to the redhat value. |

You can then use the configuration map to populate environment variables for your application. The following example shows a pod resource that populates specific environment variables by using a configuration map.

apiVersion: v1

kind: Pod

metadata:

name: config-map-example-pod

namespace: example-app

spec:

containers:

- name: example-container

image: registry.example.com/mysql-80:1-237

command: [ "/bin/sh", "-c", "env" ]

env:

- name: MYSQL\_DATABASE

valueFrom:

configMapKeyRef:

name: config-map-example

key: database.name

- name: MYSQL\_USER

valueFrom:

configMapKeyRef:

name: config-map-example

key: database.user

optional: true

|  |  |
| --- | --- |
|  | The attribute to specify environment variables for the pod. |
|  | The name of a pod environment variable where you are populating a key's value. |
|  | Name of the ConfigMap object to pull the environment variables from. |
|  | The environment variable to pull from the ConfigMap object. |
|  | Sets the environment variable as optional. The pod is started even if the specified ConfigMap object and keys do not exist. |

The following example shows a pod resource that injects all environment variables from a configuration map:

apiVersion: v1

kind: Pod

metadata:

name: config-map-example-pod2

namespace: example-app

spec:

containers:

- name: example-container

image: registry.example.com/mysql-80:1-237

command: [ "/bin/sh", "-c", "env" ]

envFrom:

- configMapRef:

name: config-map-example

restartPolicy: Never

|  |  |
| --- | --- |
|  | The attribute to pull all environment variables from a ConfigMap object. |
|  | The name of the ConfigMap object to pull environment variables from. |

You can use secrets with other Kubernetes resources such as pods, deployments, builds, and more. You can specify secret keys or volumes with a mount path to store your secrets. The following snippet shows an example of a pod that populates environment variables with data from the test-secret Kubernetes secret:

apiVersion: v1

kind: Pod

metadata:

name: secret-example-pod

spec:

containers:

- name: secret-test-container

image: busybox

command: [ "/bin/sh", "-c", "export" ]

env:

- name: TEST\_SECRET\_USERNAME\_ENV\_VAR

valueFrom:

secretKeyRef:

name: test-secret

key: username

|  |  |
| --- | --- |
|  | Specifies the environment variables for the pod. |
|  | Indicates the source of the environment variables. |
|  | The secretKeyRef source object of the environment variables. |
|  | Name of the secret, which must exist. |
|  | The key that is extracted from the secret is the username for authentication. |

In contrast with configuration maps, the values in secrets are always encoded (not encrypted), and their access is restricted to fewer authorized users.

Using Secrets and Configuration Maps as Volumes

To expose a secret to a pod, you must first create the secret in the same namespace, or project, as the pod. In the secret, assign each piece of sensitive data to a key. After creation, the secret contains key-value pairs.

The following command creates a generic secret that contains key-value pairs from literal values that are typed on the command line: user with the demo-user value, and root\_password with the zT1kTgk value.

[user@host ~]$ **oc create secret generic demo-secret \**

**--from-literal user=demo-user \**

**--from-literal root\_password=zT1KTgk**

You can also create a generic secret by specifying key names on the command line and values from files:

[user@host ~]$ **oc create secret generic demo-secret \**

**--from-file user=/tmp/demo/user \**

**--from-file root\_password=/tmp/demo/root\_password**

You can mount a secret to a directory within a pod. Kubernetes creates a file for each key in the secret that uses the name of the key. The content of each file is the decoded value of the secret. The following command shows how to mount secrets in a pod:

[user@host ~]$ **oc set volume deployment/demo \**

**--add --type secret \**

**--secret-name demo-secret \**

**--mount-path /app-secrets**

|  |  |
| --- | --- |
|  | Modify the volume configuration in the demo deployment. |
|  | Add a new volume from a secret. |
|  | Use the demo-secret secret. |
|  | Make the secret data available in the /app-secrets directory in the pod. The content of the /app-secrets/user file is demo-user. The content of the /app/secrets/root\_password file is zT1KTgk. |

To assign a secret as a volume to a deployment from the web console, list the available secrets from the **Workloads** → **Secrets** menu.

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

Select a secret and click **Add Secret to workload**.

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

Select the workload, choose the **Volume** option, and define the mount path for the secret.

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

Similar to secrets, you must first create a configuration map before a pod can consume it. The configuration map must exist in the same namespace, or project, as the pod. The following command shows how to create a configuration map from an external configuration file:

[user@host ~]$ **oc create configmap demo-map \**

**--from-file=config-files/httpd.conf**

You can similarly add a configuration map as a volume by using the following command:

[user@host ~]$ **oc set volume deployment/demo \**

**--add --type configmap \**

**--configmap-name demo-map \**

**--mount-path /app-secrets**

To confirm that the volume is attached to the deployment, use the following command:

[user@host ~]$ **oc set volume deployment/demo**

demo

configMap/demo-map as volume-du9in

mounted at /app-secrets

You can also use the oc set env command to set application environment variables from either secrets or configuration maps. In some cases, you can modify the names of the keys to match the names of environment variables by using the --prefix option. In the following example, the user key from the demo-secret secret sets the MYSQL\_USER environment variable, and the root\_password key from the demo-secret secret sets the MYSQL\_ROOT\_PASSWORD environment variable. If the key name from the secret is lowercase, then the corresponding environment variable is converted to uppercase to match the pattern that the --prefix option defines.

[user@host ~]$ **oc set env deployment/demo \**

**--from secret/demo-secret --prefix MYSQL\_**

**Note**

You cannot assign configuration maps by using the web console.

Updating Secrets and Configuration Maps

Secrets and configuration maps occasionally require updates. OpenShift provides the oc extract command to ensure that you have the latest data. You can save the data to a specific directory by using the --to option. Each key in the secret or configuration map creates a file with the same name as the key. The content of each file is the value of the associated key. If you run the oc extract command more than one time, then you must use the --confirm option to overwrite the existing files. You can also use the --confirm option to create the target directory for the extracted content.

[user@host ~]$ **oc extract secret/demo-secrets -n demo \**

**--to /tmp/demo --confirm**

[user@host ~]$ ls /tmp/demo/

user root\_password

[user@host ~]$ cat /tmp/demo/root\_password

zT1KTgk

[user@host ~]$ echo k8qhcw3m0 > /tmp/demo/root\_password

After updating the locally saved files, use the oc set data command to update the secret or configuration map. For each key that requires an update, specify the name of a key and the associated value. If a file contains the value, then use the --from-file option.

[user@host ~]$ **oc set data secret/demo-secrets -n demo \**

**--from-file /tmp/demo/root\_password**

You must restart pods that use environment variables for the pods to read the updated secret or configuration map. Pods that use a volume mount to reference secrets or configuration maps receive the updates without a restart by using an eventually consistent approach. By default, the kubelet agent watches for changes to the keys and values that are used in volumes for pods on the node. The kubelet agent detects changes and propagates the changes to the pods to keep volume data consistent. Despite the automatic updates that Kubernetes provides, a restart of the pod is still required if the software reads configuration data only at startup time.

Deleting Secrets and Configuration Maps

Similar to other Kubernetes resources, you can use the delete command to delete secrets and configuration maps that are no longer needed or in use.

[user@host ~]$ **kubectl delete secret/demo-secrets -n demo**

[user@host ~]$ **oc delete configmap/demo-map -n demo**

Guided Exercise: Externalize the Configuration of Applications

Deploy a web server taking configuration files from a configuration map.

**Outcomes**

In this exercise, you deploy a web application to mount the missing files from a configuration map.

* Create a web application deployment.
* Expose the web application deployment to external access.
* Create a configuration map from two files.
* Mount the configuration map in the web application deployment.

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 storage-configs**

**Instructions**

1. Create a web application deployment named webconfig from the web console. Use the registry.ocp4.example.com:8443/rhscl/httpd-24-rhel7:latest container image.
   1. Log in to the OpenShift cluster as the developer user with the developer password by using the OpenShift web console https://console-openshift-console.apps.ocp4.example.com URL.
   2. Change to the **Administrator** perspective, and change to the storage-configs project. Select the **Workloads** → **Deployments** option and click the **Create Deployments** button. Add the webconfig deployment name and the image name, and leave the default for the other values.

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

* 1. Select the **Create** button. Verify the successful deployment of three pods.

1. Expose the web application to external access from the web console. Use the following information to create a service and a route for the web application, and leave the hostname and path fields blank.

| **Service field** | **Service value** |
| --- | --- |
| Service name | webconfig-svc |
| App selector | webconfig |
| Port number | 8080 |
| Target port | 8080 |
| **Route field** | **Route value** |
| Route name | webconfig-rt |
| Service name | webconfig-svc |
| Target port | 8080 |

* 1. Select the **Networking** → **Services** option and click the **Create Service** button to create the webconfig-svc service that exposes the webconfig deployment. Verify the status of the service.

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

* 1. Select the **Networking** → **Routes** option and click the **Create Route** button to create the webconfig-rt service that exposes the webconfig-svc service.

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

* 1. Use a web browser to navigate to the http://webconfig-rt-storage-configs.apps.ocp4.example.com route. A testing page is displayed by default because of the missing files.

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

1. Use the missing files to create a configuration map from the web console.
   1. Select the **Workloads** → **ConfigMaps** option and click the **Create ConfigMap** button to create the webfiles configuration map by using the redhatlogo.png file and the index.html file in the /home/DO180/labs/storage-configs directory, and verify the creation of the configuration map. Add a **Data** key, define the index.html name as the **Key** value, and open the /home/DO180/labs/storage-configs/index.html file.

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

* 1. Add a **Binary Data** key, define the redhatlogo.png name as the **Key** value, and open the /home/DO180/labs/storage-configs/redhatlogo.png file.

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

1. Log in to the OpenShift cluster from the command line, and mount the webfiles configuration map as a volume in the webconfig deployment from the command line.
   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**

..output omitted...

* 1. Mount the webfiles configuration map as a volume.
  2. [student@workstation ~]$ **oc set volume deployment/webconfig \**
  3. **--add --type configmap --configmap-name webfiles \**
  4. **--name webfiles-vol --mount-path /var/www/html/**

deployment.apps/webconfig volume updated

* 1. Verify the deployment status. Verify that a new pod was created.
  2. [student@workstation ~]$ **oc status**
  3. *...output omitted...*
  4. http://webconfig-rt-storage-configs.apps.ocp4.example.com to pod port 8080 (svc/webconfig-svc)
  5. deployment/webconfig deploys registry.ocp4.example.com:8443/redhattraining/httpd-noimage:v2
  6. deployment #2 running for 2 minutes - 1 pod
  7. deployment #1 deployed 17 minutes ago

*...output omitted...*

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

NAME READY STATUS ...

webconfig-654bcf6cf-wcnnk 1/1 Running ...

*...output omitted...*

* 1. Return to the web browser, and navigate to the webconfig-rt-storage-configs.apps.ocp4.example.com route. Click the **Click me!** link to open the file.

|  |
| --- |
| A black and white rectangular sign with a red heart  Description automatically generated |

* 1. The configuration map successfully added the missing files to the web application.

**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 storage-configs**

Provision Persistent Data Volumes

Objectives

* Provide applications with persistent storage volumes for block and file-based data.

Kubernetes Persistent Storage

Containers have ephemeral storage by default. The lifetime of this ephemeral storage does not extend beyond the life of the individual pod, and this ephemeral storage cannot be shared across pods. When a container is deleted, all the files and data inside it are also deleted. To preserve the files, containers use persistent storage volumes.

Because OpenShift Container Platform uses the Kubernetes persistent volume (PV) framework, cluster administrators can provision persistent storage for a cluster. Developers can use persistent volume claims (PVCs) to request PV resources without specific knowledge of the underlying storage infrastructure.

Two ways exist to provision storage for the cluster: static and dynamic. Static provisioning requires the cluster administrator to create persistent volumes manually. Dynamic provisioning uses storage classes to create the persistent volumes on demand.

Administrators can use storage classes to provide persistent storage. Storage classes describe types of storage for the cluster. Cluster administrators create storage classes to manage storage services or storage tiers of a service. Rather than specifying provisioned storage, PVCs instead refer to a storage class.

Developers use PVCs to add persistent volumes to their applications. Developers need not know details of the storage infrastructure. With static provisioning, developers use previously created PVs, or ask a cluster administrator to manually create persistent volumes for their applications. With dynamic provisioning, developers declare the storage requirements of the application, and the cluster creates a PV to fill the request.

Persistent Volumes

Not all storage is equal. Storage types vary in cost, performance, and reliability. Multiple storage types are usually available for each Kubernetes cluster.

The following list of commonly used storage volume types and their use cases is not exhaustive.

**configMap**

The configMap volume externalizes the application configuration data. This use of the configMap resource ensures that the application configuration is portable across environments and can be version-controlled.

**emptyDir**

An emptyDir volume provides a per-pod directory for scratch data. The directory is usually empty after provisioning. emptyDir volumes are often required for ephemeral storage.

**hostPath**

A hostPath volume mounts a file or directory from the host node into your pod. To use a hostPath volume, the cluster administrator must configure pods to run as privileged. This configuration grants access to other pods in the same node.

Red Hat does not recommend the use of hostPath volumes in production. Instead, Red Hat supports hostPath mounting for development and testing on a single-node cluster. Although most pods do not need a hostPath volume, it does offer a quick option for testing if an application requires it.

**iSCSI**

Internet Small Computer System Interface (iSCSI) is an IP-based standard that provides block-level access to storage devices. With iSCSI volumes, Kubernetes workloads can consume persistent storage from iSCSI targets.

**local**

You can use Local persistent volumes to access local storage devices, such as a disk or partition, by using the standard PVC interface. Local volumes are subject to the availability of the underlying node, and are not suitable for all applications.

**NFS**

An NFS (Network File System) volume can be accessed from multiple pods at the same time, and thus provides shared data between pods. The NFS volume type is commonly used because of its ability to share data safely. Red Hat recommends to use NFS only for non-production systems.

Volume Access Mode

Persistent volume providers vary in capabilities. A volume uses access modes to specify the modes that it supports. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. OpenShift defines the following access modes, as summarized in the following table:

**Table 5.1. Volume Access Modes**

| **Access mode** | **Abbreviation** | **Description** |
| --- | --- | --- |
| ReadWriteOnce | RWO | A single node mounts the volume as read/write. |
| ReadOnlyMany | ROX | Many nodes mount the volume as read-only. |
| ReadWriteMany | RWX | Many nodes mount the volume as read/write. |

Developers must select a volume type that supports the required access level by the application. The following table shows some example supported access modes:

**Table 5.2. Access Mode Support**

| **Volume type** | **RWO** | **ROX** | **RWX** |
| --- | --- | --- | --- |
| configMap | Yes | No | No |
| emptyDir | Yes | No | No |
| hostPath | Yes | No | No |
| iSCSI | Yes | Yes | No |
| local | Yes | No | No |
| NFS | Yes | Yes | Yes |

Volume Modes

Kubernetes supports two volume modes for persistent volumes: Filesystem and Block. If the volume mode is not defined for a volume, then Kubernetes assigns the default volume mode, Filesystem, to the volume.

OpenShift Container Platform can provision raw block volumes. These volumes do not have a file system, and can provide performance benefits for applications that either write to the disk directly or that implement their own storage service. Raw block volumes are provisioned by specifying volumeMode: Block in the PV and PVC specification.

The following table provides examples of storage options with block volume support:

**Table 5.3. Block Volume Support**

| **Volume plug-in** | **Manually provisioned** | **Dynamically provisioned** |
| --- | --- | --- |
| AWS EBS | Yes | Yes |
| Azure disk | Yes | Yes |
| Cinder | Yes | Yes |
| Fibre channel | Yes | No |
| GCP | Yes | Yes |
| iSCSI | Yes | No |
| local | Yes | No |
| Red Hat OpenShift Data Foundation | Yes | Yes |
| VMware vSphere | Yes | Yes |

Manually Creating a PV

Use a PersistentVolume manifest file to manually create a persistent volume. The following example creates a persistent volume from a fiber channel storage device that uses block mode.

apiVersion: v1

kind: PersistentVolume

metadata:

name: block-pv

spec:

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

volumeMode: Block

persistentVolumeReclaimPolicy: Retain

fc:

targetWWNs: ["50060e801049cfd1"]

lun: 0

readOnly: false

|  |  |
| --- | --- |
|  | PersistentVolume is the resource type for PVs. |
|  | Provide a name for the PV, which subsequent claims use to access the PV. |
|  | Specify the amount of storage that is allocated to this volume. |
|  | The storage device must support the access mode that the PV specifies. |
|  | The volumeMode attribute is optional for Filesystem volumes, but is required for Block volumes. |
|  | The persistentVolumeReclaimPolicy determines how the cluster handles the PV when the PVC is deleted. Valid options are Retain or Delete. |
|  | The remaining attributes are specific to the storage type. In this example, the fc object specifies the Fiber Channel storage type attributes. |

If the previous manifest is in a file named my-fc-volume.yaml, then the following command can create the PV resource on RHOCP:

[user@host]$ **oc create -f my-fc-volume.yaml**

To create a persistent volume from the web console, click the **Storage** → **PersistentVolumes** menu.

Persistent Volume Claims

A persistent volume claim (PVC) resource represents a request from an application for storage. A PVC specifies the minimal storage characteristics, such as capacity and access mode. A PVC does not specify a storage technology, such as iSCSI or NFS.

The lifecycle of a PVC is not tied to a pod, but to a namespace. Multiple pods from the same namespace but with potentially different workload controllers can connect to the same PVC. You can also sequentially connect storage to and detach storage from different application pods, to initialize, convert, migrate, or back up data.

Kubernetes matches each PVC to a persistent volume (PV) resource that can satisfy the requirements of the claim. It is not an exact match. A PVC might be bound to a PV with a larger disk size than is requested. A PVC that specifies single access might be bound to a PV that is shareable for multiple concurrent accesses. Rather than enforcing policy, PVCs declare what an application needs, which Kubernetes provides on a best-effort basis.

Creating a PVC

A PVC belongs to a specific project. To create a PVC, you must specify the access mode and size, among other options. A PVC cannot be shared between projects. Developers use a PVC to access a persistent volume (PV). Persistent volumes are not exclusive to projects, and are accessible across the entire OpenShift cluster. When a PV binds to a PVC, the PV cannot be bound to another PVC.

To add a volume to an application deployment, create a PersistentVolumeClaim resource, and add it to the application as a volume. Create the PVC by using either a Kubernetes manifest or the oc set volumes command. In addition to either creating a PVC or using an existing PVC, the oc set volumes command can modify a deployment to mount the PVC as a volume within the pod.

To add a volume to an application deployment, use the oc set volumes command:

[user@host ~]$ **oc set volumes deployment/example-application \**

**--add \**

**--name example-pv-storage \**

**--type persistentVolumeClaim \**

**--claim-mode rwo \**

**--claim-size 15Gi \**

**--mount-path /var/lib/example-app \**

**--claim-name example-pv-claim**

|  |  |
| --- | --- |
|  | Specify the name of the deployment that requires the PVC resource. |
|  | Setting the add option to true adds volumes and volume mounts for containers. |
|  | The name option specifies a volume name. If not specified, a name is autogenerated. |
|  | The supported types, for the add operation, include emptyDir, hostPath, secret, configMap, and persistentVolumeClaim. |
|  | The claim-mode option defaults to ReadWriteOnce. The valid values are ReadWriteOnce (RWO), ReadWriteMany (RWX), and ReadOnlyMany (ROX). |
|  | Create a claim with the given size in bytes, if specified along with the persistent volume type. The size must use SI notation, for example, 15, 15 G, or 15 Gi. |
|  | The mount-path option specifies the mount path inside the container. |
|  | The claim-name option provides the name for the PVC, and is required for the persistentVolumeClaim type. |

The command creates a PVC resource and adds it to the application as a volume within the pod.

The command updates the deployment for the application with volumeMounts and volumes specifications.

apiVersion: apps/v1

kind: Deployment

metadata:

*...output omitted...*

namespace: storage-volumes

*...output omitted...*

spec:

*...output omitted...*

template:

*...output omitted...*

spec:

*...output omitted...*

**volumeMounts:**

- mountPath: /var/lib/example-app

name: example-pv-storage

*...output omitted...*

**volumes:**

- name: example-pv-storage

persistentVolumeClaim:

claimName: example-pv-claim

*...output omitted...*

|  |  |
| --- | --- |
|  | The deployment, which must be in the same namespace as the PVC. |
|  | The mount path in the container. |
|  | The volume name, which is used to specify the volume that is associated with the mount. |
|  | The claim name that is bound to the volume. |

The following example specifies a PVC by using a YAML manifest to create a PersistentVolumeClaim API object:

---

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: example-pv-claim

labels:

app: example-application

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 15Gi

|  |  |
| --- | --- |
|  | PersistentVolumeClaim is the resource type for a PVC. |
|  | Use this name in the claimName field of the persistentVolumeClaim element in the volumes section of a deployment manifest. |
|  | Specify the access mode that this PVC requests. The storage class provisioner must provide this access mode. If persistent volumes are created statically, then an eligible persistent volume must provide this access mode. |
|  | The storage class creates a persistent volume that matches this size request. If persistent volumes are created statically, then an eligible persistent volume must be at least the requested size. |

Use the oc create command to create the PVC from the manifest file.

[user@host ~]$ **oc create -f *pvc\_file\_name.yaml***

Use oc get pvc to view the available PVCs in the current namespace.

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

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS ...

db-pod-pvc Bound pvc-13...ca45 1Gi RWO nfs-storage ...

To create a persistent volume claim from the web console, click the **Storage** → **PersistentVolumesClaims** menu.

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

Click **Create PersistentVolumeClaim** and complete the form by adding the name, the storage class, the size, the access mode, and the volume mode.

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

Kubernetes Dynamic Provisioning

PVs are defined by a PersistentVolume API object, which is from existing storage in the cluster. The cluster administrator must statically provision some storage types. Alternatively, the Kubernetes persistent volume framework can use a StorageClass object to dynamically provision PVs.

When you create a PVC, you specify a storage amount, the required access mode, and a storage class to describe and classify the storage. The control loop in the RHOCP control node watches for new PVCs, and binds the new PVC to an appropriate PV. If an appropriate PV does not exist, then a provisioner for the storage class creates one.

Claims remain unbound indefinitely if a matching volume does not exist or if a volume cannot be created with any available provisioner that services a storage class. Claims are bound when matching volumes become available. For example, a cluster with many manually provisioned 50 Gi volumes would not match a PVC that requests 100 Gi. The PVC can be bound when a 100 Gi PV is added to the cluster.

Use oc get storageclass to view the storage classes that the cluster provides.

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

NAME PROVISIONER ...

nfs-storage (default) k8s-sigs.io/nfs-subdir-external-provisioner ...

lvms-vg1 topolvm.io ...

In the example, the nfs-storage storage class is marked as the default storage class. When a default storage class is configured, the PVC must explicitly name any other storage class to use, or can set the storageClassName annotation to "", to be bound to a PV without a storage class.

The following oc set volume command example uses the claim-class option to specify a dynamically provisioned PV.

[user@host ~]$ **oc set volumes deployment/example-application \**

**--add --name example-pv-storage --type pvc --claim-class nfs-storage \**

**--claim-mode rwo --claim-size 15Gi --mount-path /var/lib/example-app \**

**--claim-name example-pv-claim**

**Note**

Because a cluster administrator can change the default storage class, Red Hat recommends that you always specify the storage class when you create a PVC.

PV and PVC Lifecycles

When you create a PVC, you request a specific amount of storage, access mode, and storage class. Kubernetes binds the PVC to an appropriate PV. If an appropriate PV does not exist, then a provisioner for the storage class creates one.

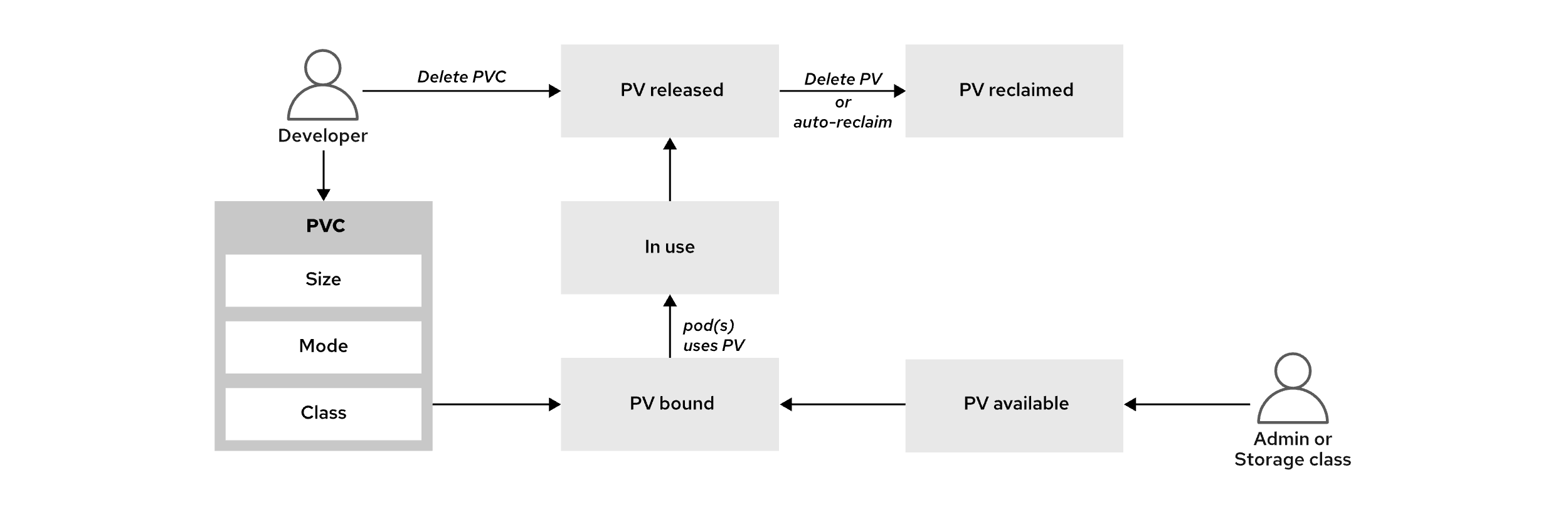


Figure 5.17: PV lifecycle

PVs follow a lifecycle based on their relationship to the PVC.

**Available**

After a PV is created, it becomes *available* for any PVC to use in the cluster in any namespace.

**Bound**

A PV that is *bound* to a PVC is also bound to the same namespace as the PVC, and no other PVC can use it.

**In Use**

You can delete a PVC if no pods actively use it. The *Storage Object in Use Protection* feature prevents the removal of bound PVs and PVCs that pods are actively using. Such removal can result in data loss. Storage Object in Use Protection is enabled by default.

If a user deletes a PVC that a pod actively uses, then the PVC is not removed immediately. PVC removal is postponed until no pods actively use the PVC. Also, if a cluster administrator deletes a PV that is bound to a PVC, then the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

**Released**

After the developer deletes the PVC that is bound to a PV, the PV is *released*, and the storage that the PV used can be reclaimed.

**Reclaimed**

The reclaim policy of a persistent volume tells the cluster what to do with the volume after it is released. A volume's reclaim policy can be Retain or Delete.

**Table 5.4. Volume Reclaim Policy**

| **Policy** | **Description** |
| --- | --- |
| Retain | Enables manual reclamation of the resource for those volume plug-ins that support it. |
| Delete | Deletes both the PersistentVolume object from OpenShift Container Platform and the associated storage asset in external infrastructure. |

Deleting a Persistent Volume Claim

To delete a volume, use the oc delete command to delete the PVC. The storage class reclaims the volume after removing the PVC.

[user@host ~]$ **oc delete pvc/example-pvc-storage**

Guided Exercise: Provision Persistent Data Volumes

Deploy a MySQL database with persistent storage from a PVC and identify the PV and storage provisioner that backs the application.

**Outcomes**

You should be able to do the following tasks:

* Deploy a MySQL database with persistent storage from a PVC.
* Identify the PV that backs the application.
* Identify the storage provisioner that created the PV.

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.

[student@workstation ~]$ **lab start storage-volumes**

**Instructions**

1. Log in to the OpenShift cluster as the developer user with the developer password. Select the storage-volumes project.
   1. Use a web browser to navigate to the OpenShift web console at https://console-openshift-console.apps.ocp4.example.com.
   2. Log in by using Red Hat Identity Management with the developer username and the developer password.
   3. Select the **Administrator** view to access the administrative menu.
   4. On the **Home** → **Projects** page, select the storage-volumes project.
2. Identify the default storage class for the cluster.
   1. Select the **Storage** → **StorageClasses** menu option.

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

* 1. Figure 5.18: Storage classes
  2. The nfs-storage storage class has the Default label.

1. Use the registry.ocp4.example.com:8443/rhel8/mysql-80 container image to create a MySQL deployment named db-pod. Use the storage-volumes project. Add a service for the database.
   1. Select the **Workloads** → **Deployments** menu option.
   2. Verify that the storage-volumes project is active and select the **Create Deployment** button.
   3. Use db-pod for the deployment name and change the image name to registry.ocp4.example.com:8443/rhel8/mysql-80.
   4. Add the environment variables.

**Table 5.5. MYSQL Environment Variables**

| **Name** | **Value** |
| --- | --- |
| MYSQL\_USER | user1 |
| MYSQL\_PASSWORD | mypa55w0rd |
| MYSQL\_DATABASE | items |

* 1. Select the **Advanced options** → **Scaling** link and set the replicas to one.
  2. Click **Create**. Wait for the blue circle to indicate that a single pod is running.
  3. Click **Networking** → **Services** → **Create Service**.
  4. Add the service values.

**Table 5.6. Service Fields**

| **Field name** | **Value** |
| --- | --- |
| Name | db-pod |
| selector | app=db-pod |
| Port | 3306 |
| Target port | 3306 |

|  |
| --- |
| A screen shot of a computer program  Description automatically generated |

Figure 5.19: Add Service

* 1. Click the **Create** button.

1. Add a 1 Gi, RWO PVC named db-pod-pvc to the deployment. Set the /var/lib/mysql directory as the mount path.
   1. Select the **Workloads** → **Deployments** menu item.
   2. Click the three vertical dots in the row with the db-pod deployment, and select the **Add storage** menu option.
   3. In the Add Storage form, click **Create new claim**.
   4. Add the following field values to the form.

**Table 5.7. Add PVC Storage Fields**

| **Field name** | **Value** |
| --- | --- |
| PersistentVolumeClaim name | db-pod-pvc |
| Access mode | RWO |
| Size | 1 GiB |
| Volume mode | Filesystem |
| Mount path | /var/lib/mysql |

* 1. Click **Save**.
  2. Scroll down in the deployment details to the Volumes section.
  3. Select the db-pod-pvc link to see the PVC details.

|  |
| --- |
| A close-up of a computer screen  Description automatically generated |

* 1. Figure 5.20: PVC Link

1. Observe how the volume mount changed the deployment.
   1. Select the **Workloads** → **Deployments** → **dp-pod** → **YAML** tab.
   2. Observe the volumes and volumeMounts additions to the deployment.
   3. apiVersion: apps/v1
   4. kind: Deployment
   5. *...output omitted...*
   6. volumes:
   7. - name: nfs-volume-storage
   8. persistentVolumeClaim:
   9. claimName: db-pod-pvc
   10. *...output omitted...*
   11. volumeMounts:
   12. - mountPath: /var/lib/mysql
   13. name: nfs-volume-storage

*...output omitted...*

1. Use a configuration map resource to add initialization data to the database.
   1. Observe the contents of the init-db.sql script that initializes the database.
   2. [student@workstation ~]$ **cat \**

**~/DO180/labs/storage-volumes/configmap/init-db.sql**

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. 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 storage-volumes project as the active project.
  2. [student@workstation ~]$ **oc project storage-volumes**

*...output omitted...*

* 1. Use the contents of the init-db.sql file to create a configuration map named init-db-cm.
  2. [student@workstation ~]$ **oc create configmap init-db-cm \**
  3. **--from-file=/home/student/DO180/labs/storage-volumes/configmap/init-db.sql**

configmap/init-db-cm created

* 1. Add the init-db-cm configuration map resource as a volume named init-db-volume to the deployment. Specify the volume type as configmap, and set the /var/db/config directory as the mount path.
  2. [student@workstation ~]$ **oc set volumes deployment/db-pod \**
  3. **--add --name init-db-volume --type configmap --configmap-name init-db-cm \**
  4. **--mount-path /var/db/config**

deployment.apps/db-pod volume updated

* 1. Start a remote shell session inside the container.
  2. [student@workstation ~]$ **oc rsh deployment/db-pod**

sh-4.4$

* 1. Use the mysql client to execute the database script in the /var/db/config/init-db volume.
  2. sh-4.4$ **mysql -uuser1 -pmypa55w0rd items </var/db/config/init-db.sql**
  3. mysql: [Warning] Using a password on the command line interface can be insecure.

sh-4.4$

* 1. Execute a query to verify the database contents.
  2. sh-4.4$ **mysql -uuser1 -pmypa55w0rd items -e 'select \* from Item;'**
  3. mysql: [Warning] Using a password on the command line interface can be insecure.
  4. +----+-------------------+------------+
  5. | id | description | done |
  6. +----+-------------------+------------+
  7. | 1 | Pick up newspaper | 0x00 |
  8. | 2 | Buy groceries | 0x01 |
  9. +----+-------------------+------------+

sh-4.4$

* 1. Exit the shell session.

sh-4.4$ **exit**

1. Delete and then re-create the db-pod deployment.
   1. Delete the db-pod deployment.
   2. [student@workstation ~]$ **oc delete deployment/db-pod**

deployment.apps "db-pod" deleted

* 1. Verify that the PVC still exists without the deployment.
  2. [student@workstation ~]$ **oc get pvc**
  3. NAME STATUS VOLUME CAPACITY ...

db-pod-pvc Bound pvc-ab5b38c7-359a-4e99-b81c-f7d11ef91cc9 1Gi ...

* 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. Use the oc set volume command to attach the existing PVC to the deployment.
2. [student@workstation ~]$ **oc set volumes deployment/db-pod \**
3. **--add --type pvc \**
4. **--mount-path /var/lib/mysql \**
5. **--name db-pod-vol \**
6. **--claim-name db-pod-pvc**

deployment.apps/db-pod volume updated

1. Create a query-db pod by using the oc run command and the registry.ocp4.example.com:8443/redhattraining/do180-dbinit 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.
   2. [student@workstation ~]$ **oc run query-db -it --rm \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/do180-dbinit \**
   4. **--restart Never \**
   5. **-- /bin/bash -c "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. Delete the db-pod deployment and the db-pod-pvc PVC.
   1. Delete the db-pod deployment.
   2. [student@workstation ~]$ **oc delete deployment/db-pod**

deployment.apps "db-pod" deleted

* 1. Delete the db-pod-pvc PVC.
  2. [student@workstation ~]$ **oc delete pvc/db-pod-pvc**

persistentvolumeclaim "db-pod-pvc" deleted

**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 storage-volumes**

Select a Storage Class for an Application

Objectives

* Match applications with storage classes that provide storage services to satisfy application requirements.

Storage Class Selection

Storage classes are a way to describe types of storage for the cluster and to provision dynamic storage on demand. The cluster administrator determines the meaning of a storage class, which is also called a profile in other storage systems. For example, an administrator can create one storage class for development use, and another for production use.

Kubernetes supports multiple storage back ends. The storage options differ in cost, performance, reliability, and function. An administrator can create different storage classes for these options. As a result, developers can select the storage solution that fits the needs of the application. Developers do not need to know the storage infrastructure details.

Recall that an administrator selects the default storage class for dynamic provisioning. A default storage class enables Kubernetes to automatically provision a persistent volume claim (PVC) that does not specify a storage class. Because an administrator can change the default storage class, a developer should explicitly set the storage class for an application.

Reclaim Policy

Outside the application function, the developer must also consider the impact of the reclaim policy on storage requirements. A reclaim policy determines what happens to the data on a PVC after the PVC is deleted. When you are finished with a volume, you can delete the PVC object from the API, which enables reclamation of the resource. Kubernetes releases the volume when the PVC is deleted, but the volume is not yet available for another claim. The previous claimant's data remains on the volume and must be handled according to the policy. To keep your data, choose a storage class with a retain reclaim policy.

By using the retain reclaim policy, when you delete a PVC, only the PVC object is deleted from the cluster. The Persistent Volume (PV) that backed the PVC, the physical storage device that the PV used, and your data still exist. To reclaim the storage and use it in your cluster again, the cluster administrator must take manual steps.

To manually reclaim a PV as a cluster administrator, follow these steps:

1. Delete the PV.

[user@host ~]$ **oc delete pv <pv-name>**

The associated asset in the external storage infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume, still exists after the PV is deleted.

1. At this point, the cluster administrator can create another PV by using the same storage and data from the previous PV. A developer could then mount the new PV and access the data from the previous PV.
2. Alternatively, the cluster administrator can remove the data on the storage asset, and then delete the storage asset.

To automatically delete the PV, the data, and the physical storage for a deleted PVC, you must choose a storage class that uses the delete reclaim policy. This reclaim policy automatically reclaims your storage volume when the PVC is deleted. The delete reclaim policy is the default setting for all storage provisioners that adhere to the Kubernetes Container Storage Interface (CSI) standards. If you use a storage class that does not specify a reclaim policy, then the delete reclaim policy is used.

For more information about the Kubernetes Container Storage Interface standards, refer to the *Kubernetes CSI Developer Documentation* website at <https://kubernetes-csi.github.io/docs/>.

Kubernetes and Application Responsibilities

Kubernetes does not change how an application relates to storage. The application is responsible for working with its storage devices and for ensuring data integrity and consistency. Kubernetes storage does not prevent an application from doing dangerous things, such as sharing a data volume between two databases that require exclusive access to data.

Because a PVC is a storage device that your Linux host mounts, an improperly configured application could behave unexpectedly. For example, you could have an iSCSI LUN, which is expressed as an RWO PVC that is not supposed to be shared, and then mount that same PVC on two pods of the same host. Whether this situation is problematic depends on the applications.

Usually, it is fine for two processes on the same host to share a disk. After all, many applications on your personal machine share a local disk. However, nothing prevents one text editor from overwriting and losing all edits from another text editor. The use of Kubernetes storage must come with the same caution.

Single-node access (RWO) and shared access (RWX) do not ensure that files can be shared safely and reliably. RWO means that only one cluster node can read and write to the PVC. Alternatively, with RWX, Kubernetes provides a storage volume that any pod can access for reading or writing.

Use Cases for Storage Classes

The administrator creates storage classes that serve the needs of the developers. A storageClass object defines each storage class, and the object contains information about the storage provider and the capabilities of the storage medium. The provider creates PVs to match the specifications of the storage class. Administrators can create storage classes with various functional levels, based on many factors.

**Storage volume modes**

A storage class with block volume mode support can increase performance for applications that can use raw block devices. Consider using a storage class with Filesystem volume mode support for applications that share files or that provide file access.

**Quality of Service (QoS) levels**

A Solid State Drive (SSD) provides excellent speed and support for frequently accessed files. Use a lower cost and a slower hard drive (HDD) for files that are accessed less often.

**Administrative support tier**

A production-tier storage class can include volumes that are backed up often. In contrast, a development-tier storage class might include volumes that are not configured with a backup schedule.

Storage classes can use a combination of these factors and others to best fit the needs of the developers.

Kubernetes matches PVCs with the best available PV that is not bound to another PVC. The PV must provide the access mode that is specified in the PVC, and the volume must be at least as large as the requested size in the PVC. The supported access modes depend on the capabilities of the storage provider. A PVC can specify additional criteria, such as the name of a storage class. If a PVC cannot find a PV that matches all criteria, then the PVC enters a pending state and waits until an appropriate PV becomes available.

PVCs can request a specific storage class by specifying the storageClassName attribute. This method of selecting a specific storage class ensures that the storage medium is a good fit for the application requirements. Only PVs of the requested storage class can be bound to the PVC. The cluster administrator can configure dynamic provisioners to service storage classes. The cluster administrator can also create a PV on demand that matches the specifications in the PVC.

Create a Storage Class

The following YAML excerpt describes the basic definition for a StorageClass object. A cluster administrator or a storage-admin user creates globally scoped StorageClass objects. The following resource shows the parameters for configuring a storage class. This example uses the AWS ElasticBlockStore (EBS) object definition.

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: io1-gold-storage

annotations:

storageclass.kubernetes.io/is-default-class: 'false'

description:'Provides RWO and RWOP Filesystem & Block volumes'

...

parameters:

type: io1

iopsPerGB: "10"

...

provisioner: kubernetes.io/aws-ebs

reclaimPolicy: Delete

volumeBindingMode: Immediate

allowVolumeExpansion: true

|  |  |
| --- | --- |
|  | A required item that specifies the current API version. |
|  | A required item that specifies the API object type. |
|  | A required item that specifies the name of the storage class. |
|  | An optional item that specifies annotations for the storage class. |
|  | An optional item that specifies the required parameters for the specific provisioner; this object differs between plug-ins. |
|  | A required item that specifies the type of provisioner that is associated with this storage class. |
|  | An optional item that specifies the selected reclaim policy for the storage class. |
|  | An optional item that specifies the selected volume binding mode for the storage class. |
|  | An optional item that specifies the volume expansion setting. |

Several attributes, such as the API version, API object type, and annotations, are common for Kubernetes objects, whereas other attributes are specific to storage class objects.

**Parameters**

Parameters can configure file types, change storage types, enable encryption, enable replication, and so on. Each provisioner has different parameter options. Accepted parameters depend on the storage provisioner. For example, the io1 value for the type parameter, and the iopsPerGB parameter, are specific to EBS. When a parameter is omitted, the storage provisioner uses the default value.

**Provisioners**

The provisioner attribute identifies the source of the storage medium plug-in. Provisioners with names that begin with a kubernetes.io value are available by default in a Kubernetes cluster.

**ReclaimPolicy**

The default reclaim policy, Delete, automatically reclaims the storage volume when the PVC is deleted. Reclaiming storage in this way can reduce the storage costs. The Retain reclaim policy does not delete the storage volume, so that data is not lost if the wrong PVC is deleted. This reclaim policy can result in higher storage costs if space is not manually reclaimed.

**VolumeBindingMode**

The volumeBindingMode attribute determines how volume attachments are handled for a requesting PVC. Using the default Immediate volume binding mode creates a PV to match the PVC when the PVC is created. This setting does not wait for the pod to use the PVC, and thus can be inefficient. The Immediate binding mode can also cause problems for storage back ends that are topology-constrained or are not globally accessible from all nodes in the cluster. PVs are also bound without the knowledge of a pod's scheduling requirements, which might result in unschedulable pods.

By using the WaitForFirstConsumer mode, the volume is created after the pod that uses the PVC is in use. With this mode, Kubernetes creates PVs that conform to the pod's scheduling constraints, such as resource requirements and selectors.

**AllowVolumeExpansion**

When set to a true value, the storage class specifies that the underlying storage volume can be expanded if more storage is required. Users can resize the volume by editing the corresponding PVC object. This feature can be used only to grow a volume, not to shrink it.

The cluster administrator can use the create command to create a storage class from a YAML manifest file. The resulting storage class is non-namespaced, and thus is available to all projects in the cluster.

[user@host ~]$ **oc create -f <storage-class-filename.yaml>**

To create a storage class from the web console, click the **Storage** → **StorageClasses** menu. Click **Create StorageClass** and complete the form or the YAML manifest.

Cluster Storage Classes

Use the oc get storageclass command to view the storage class options that are available in a cluster.

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

A regular cluster user can view the attributes of a storage class by using the describe command. The following example queries the attributes of the storage class with the name lvms-vg1.

[user@host ~]$ **oc describe storageclass lvms-vg1**

IsDefaultClass: No

Annotations: description=Provides RWO and RWOP Filesystem & Block volumes

Provisioner: topolvm.io

Parameters: csi.storage.k8s.io/fstype=xfs,topolvm.io/device-class=vg1

AllowVolumeExpansion: True

MountOptions: <none>

ReclaimPolicy: Delete

VolumeBindingMode: WaitForFirstConsumer

Events: <none>

The describe command can help a developer to decide whether the storage class is a good fit for an application. If none of the storage classes in the cluster are appropriate for the application, then the developer can request the cluster administrator to create a PV with the required features.

Storage Class Usage

Recall that the oc set volume command can add a PVC and an associated PV to a deployment. A YAML manifest file can declare the parameters of a PVC independently from the deployment. This method is the preferred option to support repeatability, configuration management, and version control. Use the storageClassName attribute to specify the storage class for the PVC.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-block-pvc

spec:

accessModes:

- RWO

volumeMode: Block

**storageClassName: <storage-class-name>**

resources:

requests:

storage: 10Gi

Use the create command to create the resource from the YAML manifest file.

[user@host ~]$ **oc create -f <my-pvc-filename.yaml>**

Use the --claim-name option with the set volume command to add the pre-existing PVC to a deployment.

[user@host ~]$ oc set volume deployment/<deployment-name> \

--add --name <my-volume-name> \

**--claim-name my-block-pvc** \

--mount-path /var/tmp

Manage Non-shared Storage with Stateful Sets

Objectives

* Deploy applications that scale without sharing storage.

Application Clustering

Clustering applications, such as MySQL and Cassandra, typically require persistent storage to maintain the integrity of the data and files that the application uses. When many applications require persistent storage at the same time, multi-disk provisioning might not be possible due to the limited amount of available resources.

Shared storage solves this problem by allocating the same resources from a single device to multiple services.

Storage Services

File storage solutions provide the directory structure that is found in many environments. Using file storage is ideal when applications generate or consume reasonable volumes of organized data. Applications that use file-based implementations are prevalent, easy to manage, and provide an affordable storage solution.

File-based solutions are a good fit for data backup and archiving, due to their reliability, as are also file sharing and collaboration services. Most data centers provide file storage solutions, such as a network-attached storage (NAS) cluster, for these scenarios.

*Network-attached storage (NAS)* is a file-based storage architecture that makes stored data accessible to networked devices. NAS gives networks a single access point for storage with built-in security, management, and fault-tolerant capabilities. Out of the multiple data transfer protocols that networks can run, two protocols are fundamental to most networks: *internet protocol (IP)* and *transmission control protocol (TCP)*.

The files that are transferred across these protocols can be formatted as one of the following protocols:

* *Network File Systems (NFS)*: This protocol enables remote hosts to mount file systems over a network and to interact with those file systems as though they are mounted locally.
* *Server Message Blocks (SMB)*: This protocol implements an application-layer network protocol that is used to access resources on a server, such as file shares and shared printers.

NAS solutions can provide file-based storage to applications within the same data center. This approach is common to many application architectures, including the following architectures:

* Web server content
* File share services
* FTP storage
* Backup archives

These applications take advantage of data reliability and the ease of file sharing that is available by using file storage. Additionally, for file storage data, the OS and file system handle the locking and caching of the files.

Although familiar and prevalent, file storage solutions are not ideal for all application scenarios. One particular pitfall of file storage is poor handling of large data sets or unstructured data.

Block storage solutions, such as Storage Area Network (SAN) and iSCSI technologies, provide access to raw block devices for application storage. These block devices function as independent storage volumes, such as the physical drives in servers, and typically require formatting and mounting for application access.

Using block storage is ideal when applications require faster access for optimizing computationally heavy data workloads. Applications that use block-level storage implementations gain efficiencies by communicating at the raw device level, instead of relying on operating system layer access.

Block-level approaches enable data distribution on blocks across the storage volume. Blocks also use basic metadata, including a unique identification number for each block of data, for quick retrieval and reassembly of blocks for reading.

SAN and iSCSI technologies provide applications with block-level volumes from network-based storage pools. Using block-level access to storage volumes is common for application architectures, including the following architectures:

* SQL Databases (single node access).
* Virtual Machines (multinode access).
* High-performance data access.
* Server-side processing applications.
* Multiple block device RAID configurations.

Application storage that uses several block devices in a RAID configuration benefits from the data integrity and performance that the various arrays provide.

With Red Hat OpenShift Container Platform (RHOCP), you can create customized storage classes for your applications. With the NAS and the SAN storage technologies, RHOCP applications can use either the NFS protocol for file-based storage, or the block-level protocol for block storage.

Introduction to Stateful Sets

A stateful application is characterized by acting according to past states or transactions, which affect the current state and future ones of the application. Using a stateful application simplifies recovery from failures by starting from a certain point in time.

A stateful set is the representation of a set of pods with consistent identities. These identities are defined as a network with a single stable DNS, hostname, and storage from as many volume claims as the stateful set specifies. A stateful set guarantees that a given network identity maps to the same storage identity.

Deployments represent a set of containers within a pod. Each deployment can have many active replicas, depending on the user specification. These replicas can be scaled up or down, as needed. A replica set is a native Kubernetes API object that ensures that the specified number of pod replicas are running. Deployments are used for stateless applications by default, and they can be used for stateful application by attaching a persistent volume. All pods in a deployment share a volume and PVC.

In contrast with deployments, stateful set pods do not share a persistent volume. Instead, stateful set pods each have their own unique persistent volumes. Pods are created without a replica set, and each replica records its own transactions. Each replica has its own identifier, which is maintained in any rescheduling. You must configure application-level clustering so that stateful set pods have the same data.

Stateful sets are the best option for applications, such as databases, that require consistent identities and non-shared persistent storage.

Working with Stateful Sets

With Kubernetes, you can use manifest files to specify the intended configuration of a stateful set. You can define the name of the application, labels, the image source, storage, environment variables, and more.

The following snippet shows an example of a YAML manifest file for a stateful set:

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: dbserver

spec:

selector:

matchLabels:

app: database

replicas: 3

template:

metadata:

labels:

app: database

spec:

containers:

- env:

- name: MYSQL\_USER

valueFrom:

secretKeyRef:

key: user

name: sakila-cred

image: registry.ocp4.example.com:8443/redhattraining/mysql-app:v1

name: database

ports:

- containerPort: 3306

name: database

volumeMounts:

- mountPath: /var/lib/mysql

name: data

terminationGracePeriodSeconds: 10

volumeClaimTemplates:

- metadata:

name: data

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "lvms-vg1"

resources:

requests:

storage: 1Gi

|  |  |
| --- | --- |
|  | Name of the stateful set. |
|  | Application labels. |
|  | Number of replicas. |
|  | Environment variables, which can be explicitly defined, or by using a secret object. |
|  | Image source. |
|  | Container name. |
|  | Container ports. |
|  | Mount path information for the persistent volumes for each replica. Each persistent volume has the same configuration. |
|  | The access mode of the persistent volume. You can choose between the ReadWriteOnce, ReadWriteMany, and ReadOnlyMany options. |
|  | The storage class that the persistent volume uses. |
|  | Size of the persistent volume. |

**Note**

Stateful sets can be created only by using manifest files. The oc and kubectl CLI do not have commands to create stateful sets imperatively.

Create the stateful set by using the create command:

[user@host ~]$ **oc create -f statefulset-dbserver.yml**

Verify the creation of the stateful set named dbserver:

[user@host ~]$ **kubectl get statefulset**

NAME READY AGE

dbserver 3/3 6s

Verify the status of the instances:

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

NAME READY STATUS RESTARTS AGE

dbserver-0 1/1 Running 0 85s

dbserver-1 1/1 Running 0 82s

dbserver-2 1/1 Running 0 79s

Verify the status of the persistent volumes:

[user@host ~]$ **kubectl get pvc**

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS ...

data-dbserver-0 Bound pvc-c28f61ee-... 1Gi RWO nfs-storage ...

data-dbserver-1 Bound pvc-ddbe6af1-... 1Gi RWO nfs-storage ...

data-dbserver-2 Bound pvc-8302924a-... 1Gi RWO nfs-storage ...

Notice that three PVCs were created. Confirm that persistent volumes are attached to each instance:

[user@host ~]$ **oc describe pod dbserver-0**

*...output omitted...*

Volumes:

data:

Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)

ClaimName: **data-dbserver-0**

*...output omitted...*

[user@host ~]$ **oc describe pod dbserver-1**

*...output omitted...*

Volumes:

data:

Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)

ClaimName: **data-dbserver-1**

*...output omitted...*

[user@host ~]$ **oc describe pod dbserver-2**

*...output omitted...*

Volumes:

data:

Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)

ClaimName: **data-dbserver-2**

*...output omitted...*

**Note**

You must configure application-level clustering for stateful set pods to have the same data.

You can update the number of replicas of the stateful set by using the scale command:

[user@host ~]$ **oc scale statefulset/dbserver --replicas 1**

NAME READY STATUS RESTARTS ...

dbserver-0 1/1 Running 0 ...

To delete the stateful set, use the delete statefulset command:

[user@host ~]$ **kubectl delete statefulset dbserver**

statefulset.apps "dbserver" deleted

Notice that the PVCs are not deleted after the execution of the oc delete statefulset command:

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

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS ...

data-dbserver-0 Bound pvc-c28f61ee-... 1Gi RWO nfs-storage ...

data-dbserver-1 Bound pvc-ddbe6af1-... 1Gi RWO nfs-storage ...

data-dbserver-2 Bound pvc-8302924a-... 1Gi RWO nfs-storage ...

You can create a stateful set from the web console by clicking the **Workloads** → **StatefulSets** menu. Click **Create StatefulSet** and customize the YAML manifest.

Guided Exercise: Manage Non-shared Storage with Stateful Sets

Deploy a replicated web server by using a deployment and verify that all web server pods share a PV; and deploy a replicated MySQL database by using a stateful set and verify that each database instance gets a dedicated PV.

**Outcomes**

In this exercise, you deploy a web server with a shared persistent volume between the replicas, and a database server from a stateful set with dedicated persistent volumes for each instance.

* Deploy a web server with persistent storage.
* Add data to the persistent storage.
* Scale the web server deployment and observe the data that is shared with the replicas.
* Create a database server with a stateful set by using a YAML manifest file.
* Verify that each instance from the stateful set has a persistent volume claim.

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.

[student@workstation ~]$ **lab start storage-statefulsets**

**Instructions**

1. Create a web server deployment named web-server. Use the registry.ocp4.example.com:8443/redhattraining/hello-world-nginx:latest container image.
   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**

*...output omitted...*

* 1. Change to the storage-statefulsets project.
  2. [student@workstation]$ **oc project storage-statefulsets**

Now using project "storage-statefulsets" on server *...output omitted...*

* 1. Create the web-server deployment.
  2. [student@workstation ~]$ **oc create deployment web-server \**
  3. **--image registry.ocp4.example.com:8443/redhattraining/hello-world-nginx:latest**

deployment.apps/web-server created

* 1. Verify the deployment status.
  2. [student@workstation ~]$ **oc get pods -l app=web-server**
  3. NAME READY STATUS RESTARTS AGE

web-server-7d7cb4cdc7-t7hx8 1/1 Running 0 4s

1. Add the web-pv persistent volume to the web-server deployment. Use the default storage class and the following information to create the persistent volume:

| **Field** | **Value** |
| --- | --- |
| Name | web-pv |
| Type | persistentVolumeClaim |
| Claim mode | rwo |
| Claim size | 5Gi |
| Mount path | /usr/share/nginx/html |
| Claim name | web-pv-claim |

* 1. Add the web-pv persistent volume to the web-server deployment.
  2. [student@workstation ~]$ **oc set volumes deployment/web-server \**
  3. **--add --name web-pv --type persistentVolumeClaim --claim-mode rwo \**
  4. **--claim-size 5Gi --mount-path /usr/share/nginx/html --claim-name web-pv-claim**

deployment.apps/web-server volume updated

Because a storage class was not specified with the --claim-class option, the command uses the default storage class to create the persistent volume.

* 1. Verify the deployment status. Notice that a new pod is created.
  2. [student@workstation ~]$ **oc get pods -l app=web-server**
  3. NAME READY STATUS RESTARTS AGE

web-server-64689877c6-mdr6f 1/1 Running 0 5s

* 1. Verify the persistent volume status.
  2. [student@workstation ~]$ **oc get pvc**
  3. NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

web-pv-claim Bound pvc-42...63ab 5Gi RWO nfs-storage 29s

The default storage class, nfs-storage, provisioned the persistent volume.

1. Add data to the PV by using the exec command.
   1. List pods to retrieve the web-server pod name.
   2. [student@workstation ~]$ **oc get pods**
   3. NAME READY STATUS RESTARTS AGE

web-server-64689877c6-mdr6f 1/1 Running 0 17m

The pod name might differ in your output.

* 1. Use the exec command to add the pod name that you retrieved from the previous step to the /usr/share/nginx/html/index.html file on the pod. Then, retrieve the contents of the /var/www/hmtl/index.html file to confirm that the pod name is in the file.
  2. [student@workstation ~]$ **oc exec -it pod/web-server-*64689877c6-mdr6f* \**
  3. **-- /bin/bash -c \**

**'echo "Hello, World from ${HOSTNAME}" > /usr/share/nginx/html/index.html'**

[student@workstation ~]$ **oc exec -it pod/web-server-*64689877c6-mdr6f* \**

**-- cat /usr/share/nginx/html/index.html**

Hello, World from web-server-64689877c6-mdr6f

1. Scale the web-server deployment to two replicas and confirm that an additional pod is created.
   1. Scale the web-server deployment to two replicas.
   2. [student@workstation ~]$ **oc scale deployment web-server --replicas 2**

deployment.apps/web-server scaled

* 1. Verify the replica status and retrieve the pod names.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. web-server-64689877c6-mbj6g 1/1 Running 0 2s

web-server-64689877c6-mdr6f 1/1 Running 0 17m

The pod names might differ from your output.

1. Retrieve the content of the /usr/share/nginx/html/index.html file on the web-server pods by using the oc exec command to verify that the file is the same in both pods.
   1. Verify that the /usr/share/nginx/html/index.html file is the same in both pods.
   2. [student@workstation ~]$ **oc exec -it pod/web-server-*64689877c6-mbj6g* \**
   3. **-- cat /usr/share/nginx/html/index.html**

Hello, World from web-server-64689877c6-mdr6f

[student@workstation ~]$ **oc exec -it pod/web-server-*64689877c6-mdr6f* \**

**-- cat /usr/share/nginx/html/index.html**

Hello, World from web-server-64689877c6-mdr6f

Notice that both files show the name of the first instance, because they share the persistent volume.

1. Create a database server with a stateful set by using the statefulset-db.yml file in the /home/student/DO180/labs/storage-statefulsets directory. Update the file with the following information:

| **Field** | **Value** |
| --- | --- |
| metadata.name | dbserver |
| spec.selector.matchLabels.app | database |
| spec.template.metadata.labels.app | database |
| spec.template.spec.containers.name | dbserver |
| spec.template.spec.containers.volumeMounts.name | data |
| spec.template.spec.containers.volumeMounts.mountPath | /var/lib/mysql |
| spec.volumeClaimTemplates.metadata.name | data |
| spec.volumeClaimTemplates.spec.storageClassName | lvms-vg1 |

* 1. Open the /home/student/DO180/labs/storage-statefulsets/statefulset-db.yml file in an editor. Replace the <CHANGE\_ME> objects with values from the previous table:
  2. apiVersion: apps/v1
  3. kind: StatefulSet
  4. metadata:
  5. name: **dbserver**
  6. spec:
  7. selector:
  8. matchLabels:
  9. app: **database**
  10. replicas: 2
  11. template:
  12. metadata:
  13. labels:
  14. app: **database**
  15. spec:
  16. terminationGracePeriodSeconds: 10
  17. containers:
  18. - name: **dbserver**
  19. image: registry.ocp4.example.com:8443/redhattraining/mysql-app:v1
  20. ports:
  21. - name: database
  22. containerPort: 3306
  23. env:
  24. - name: MYSQL\_USER
  25. value: "redhat"
  26. - name: MYSQL\_PASSWORD
  27. value: "redhat123"
  28. - name: MYSQL\_DATABASE
  29. value: "sakila"
  30. volumeMounts:
  31. - name: **data**
  32. mountPath: **/var/lib/mysql**
  33. volumeClaimTemplates:
  34. - metadata:
  35. name: **data**
  36. spec:
  37. accessModes: [ "ReadWriteOnce" ]
  38. storageClassName: **"lvms-vg1"**
  39. resources:
  40. requests:

storage: 1Gi

* 1. Create the database server by using the oc create -f /home/student/DO180/labs/storage-statefulsets/statefulset-db.yml command.
  2. [student@workstation ~]$ **oc create -f \**
  3. **/home/student/DO180/labs/storage-statefulsets/statefulset-db.yml**

statefulset.apps/bdserver created

* 1. Wait a few moments and then verify the status of the stateful set and its instances.
  2. [student@workstation ~]$ **oc get statefulset**
  3. NAME READY AGE

dbserver 2/2 10s

[student@workstation ~]$ **oc get pods -l app=database**

NAME READY STATUS ...

dbserver-0 1/1 Running ...

dbserver-1 1/1 Running ...

* 1. Use the exec command to add data to each of the stateful set pods.
  2. [student@workstation ~]$ **oc exec -it pod/dbserver-0 -- /bin/bash -c \**
  3. **"mysql -uredhat -predhat123 sakila -e 'create table items (count INT);'"**

mysql: [Warning] Using a password on the command line interface can be insecure.

[student@workstation ~]$ **oc exec -it pod/dbserver-1 -- /bin/bash -c \**

**"mysql -uredhat -predhat123 sakila -e 'create table inventory (count INT);'"**

mysql: [Warning] Using a password on the command line interface can be insecure.

1. Confirm that each instance from the dbserver stateful set has a persistent volume claim. Then, verify that each persistent volume claim contains unique data.
   1. Confirm that the persistent volume claims have a Bound status.
   2. [student@workstation ~]$ **oc get pvc -l app=database**
   3. NAME STATUS ... CAPACITY ACCESS MODE ...
   4. data-dbserver-0 Bound ... 1Gi RWO ...

data-dbserver-1 Bound ... 1Gi RWO ...

* 1. Verify that each instance from the dbserver stateful set has its own persistent volume claim by using the oc get pod *pod-name* -o json | jq .spec.volumes[0].persistentVolumeClaim.claimName command.
  2. [student@workstation ~]$ **oc get pod dbserver-0 -o json | \**
  3. **jq .spec.volumes[0].persistentVolumeClaim.claimName**

"data-dbserver-0"

[student@workstation ~]$ **oc get pod dbserver-1 -o json | \**

**jq .spec.volumes[0].persistentVolumeClaim.claimName**

"data-dbserver-1"

* 1. Application-level clustering is not enabled for the dbserver stateful set. Verify that each instance of the dbserver stateful set has unique data.
  2. [student@workstation ~]$ **oc exec -it pod/dbserver-0 -- /bin/bash -c \**
  3. **"mysql -uredhat -predhat123 sakila -e 'show tables;'"**
  4. mysql: [Warning] Using a password on the command line interface can be insecure.
  5. ------------------
  6. | Tables\_in\_sakila |
  7. ------------------
  8. | items |

------------------

[student@workstation ~]$ **oc exec -it pod/dbserver-1 -- /bin/bash -c \**

**"mysql -uredhat -predhat123 sakila -e 'show tables;'"**

mysql: [Warning] Using a password on the command line interface can be insecure.

------------------

| Tables\_in\_sakila |

------------------

| inventory |

------------------

1. Delete a pod in the dbserver stateful set. Confirm that a new pod is created and that the pod uses the PVC from the previous pod. Verify that the previously added table exists in the sakila database.
   1. Delete the dbserver-0 pod in the dbserver stateful set. Confirm that a new pod is generated for the stateful set. Then, confirm that the data-dbserver-0 PVC still exists.
   2. [student@workstation ~]$ **oc delete pod dbserver-0**

pod "dbserver-0" deleted

[student@workstation ~]$ **oc get pods -l app=database**

NAME READY STATUS RESTARTS AGE

dbserver-0 1/1 Running 0 4s

dbserver-1 1/1 Running 0 5m

[student@workstation ~]$ **oc get pvc -l app=database**

NAME STATUS ... CAPACITY ACCESS MODE ...

data-dbserver-0 Bound ... 1Gi RWO ...

data-dbserver-1 Bound ... 1Gi RWO ...

* 1. Use the exec command to verify that the new dbserver-0 pod has the items table in the sakila database.
  2. [student@workstation ~]$ **oc exec -it pod/dbserver-0 -- /bin/bash -c \**
  3. **"mysql -uredhat -predhat123 sakila -e 'show tables;'"**
  4. mysql: [Warning] Using a password on the command line interface can be insecure.
  5. ------------------
  6. | Tables\_in\_sakila |
  7. ------------------
  8. | items |

------------------

Lab: Manage Storage for Application Configuration and Data

Deploy a web application and its database that share database credentials from a secret. The database should use the default storage for the cluster. Also deploy a file-sharing application that runs with multiple replicas and shares its storage volume with a file uploader application. The file sharing and file uploader applications take configuration files from a config map and should use NFS file storage for shareability. The database should use local storage for increased and performance.

**Outcomes**

* Deploy a database server.
* Deploy a web application.
* Create a secret that contains the database server credentials.
* Create a configuration map that contains an SQL file.
* Add and remove a volume on the database server and the web application.
* Expose the database server and the web application.
* Scale up the web application.
* Mount the configuration map as a volume.

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 storage-review project, and it creates files that this lab uses, in the /home/student/DO180/labs/storage-review directory.

[student@workstation ~]$ **lab start storage-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 storage-review project for your work.

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

*...output omitted...*

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

*...output omitted...*

1. Hide Solution
2. Create a secret named world-cred that contains the following data:

| **Field** | **Value** |
| --- | --- |
| user | redhat |
| password | redhat123 |
| database | world\_x |

* 1. Create a secret that contains the database credentials.
  2. [student@workstation]$ **oc create secret generic world-cred \**
  3. **--from-literal user=redhat \**
  4. **--from-literal password=redhat123 \**
  5. **--from-literal database=world\_x**

secret/world-cred created

* 1. Confirm the creation of the secret.
  2. [student@workstation ~]$ **oc get secrets world-cred**
  3. NAME TYPE DATA AGE

world-cred Opaque 3 2m34s

1. Hide Solution
2. Create a configuration map named dbfiles by using the ~/DO180/labs/storage-review/insertdata.sql file.
   1. Create a configuration map named dbfiles by using the insertdata.sql file in the ﻿~/DO180/labs/storage-review directory.
   2. [student@workstation ~]$ **oc create configmap dbfiles \**
   3. **--from-file ~/DO180/labs/storage-review/insertdata.sql**

configmap/dbfiles created

* 1. Verify the creation of the configuration map.
  2. [student@workstation]$ **oc get configmaps**
  3. NAME DATA AGE
  4. dbfiles 1 11s

*...output omitted...*

1. Hide Solution
2. Create a database server deployment named dbserver by using the registry.ocp4.example.com:8443/redhattraining/mysql-app:v1 container image. Then, set the missing environment variables by using the world-cred secret.
   1. Create the database server deployment.
   2. [student@workstation ~]$ **oc create deployment dbserver \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/mysql-app:v1**

deployment.apps/dbserver created

* 1. Set the missing environment variables.
  2. [student@workstation ~]$ **oc set env deployment/dbserver --from secret/world-cred \**
  3. **--prefix MYSQL\_**

deployment.apps/dbserver updated

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

dbserver-***6d5bf5d86c-ptrb2*** 1/1 Running ...

1. Hide Solution
2. Add a volume to the dbserver deployment by using the following information:

| **Field** | **Value** |
| --- | --- |
| name | dbserver-lvm |
| type | persistentVolumeClaim |
| claim mode | rwo |
| claim size | 1Gi |
| mount path | /var/lib/mysql |
| claim class | lvms-vg1 |
| claim name | dbserver-lvm-pvc |

* 1. Add a volume to the dbserver deployment.
  2. [student@workstation ~]$ **oc set volume deployment/dbserver \**
  3. **--add --name dbserver-lvm --type persistentVolumeClaim \**
  4. **--claim-mode rwo --claim-size 1Gi --mount-path /var/lib/mysql \**
  5. **--claim-class lvms-vg1 --claim-name dbserver-lvm-pvc**

deployment.apps/dbserver volume updated

* 1. Verify the deployment status.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...

dbserver-***5bc6bd5d7b-7z7lv*** 1/1 Running ...

* 1. Verify the volume status.
  2. [student@workstation ~]$ **oc get pvc**
  3. NAME **STATUS** VOLUME CAPACITY *...output omitted...*

dbserver-lvm-pvc **Bound** pvc-2cb85025-*... 1Gi ...output omitted...*

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

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

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

service/mysql-service 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) ...

mysql-service ClusterIP 172.30.240.100 <none> 3306/TCP ...

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

NAME ENDPOINTS ...

mysql-service 10.8.1.36:3306 ...

1. Hide Solution
2. Create a web application deployment named file-sharing by using the registry.ocp4.example.com:8443/redhattraining/php-webapp-mysql:v1 container image. Scale the deployment to two replicas. Then, expose the deployment by using the following information:

| **Field** | **Value** |
| --- | --- |
| Name | file-sharing |
| Port | 8080 |
| Target port | 8080 |

1. Create a route named file-sharing to expose the file-sharing web application to external access. Access the file-sharing route in a web browser to test the connection between the web application and the database server.
   1. Create a web application deployment.
   2. [student@workstation ~]$ **oc create deployment file-sharing \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/php-webapp-mysql:v1**

deployment.apps/file-sharing created

* 1. Verify the deployment status. Verify that the file-sharing application pod is in the RUNNING state. The pod names might differ on your system.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. dbserver-***5bc6bd5d7b-7z7lv*** 1/1 Running ...

file-sharing-***789c5948c8-gdrlz*** 1/1 Running ...

* 1. Scale the deployment to two replicas.
  2. [student@workstation ~]$ **oc scale deployment file-sharing --replicas 2**

deployment.apps/file-sharing scaled

* 1. Verify the replica status and retrieve the pod name. The pod names might differ on your system.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. dbserver-***5bc6bd5d7b-7z7lv*** 1/1 Running ...
  5. file-sharing-***789c5948c8-62j9s*** 1/1 Running ...

file-sharing-***789c5948c8-gdrlz*** 1/1 Running ...

* 1. Expose the file-sharing deployment.
  2. [student@workstation ~]$ **oc expose deployment file-sharing --name file-sharing \**
  3. **--port 8080 --target-port 8080**

service/file-sharing 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) ...
  4. file-sharing ClusterIP 172.30.139.210 <none> 8080/TCP ...

mysql-service ClusterIP 172.30.240.100 <none> 3306/TCP ...

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

NAME ENDPOINTS

file-sharing 10.8.1.37:8080,10.8.1.38:8080 ...

mysql-service 10.8.1.36:3306 ...

* 1. Expose the file-sharing service.
  2. [student@workstation ~]$ **oc expose service/file-sharing**

route.route.openshift.io/file-sharing exposed

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

NAME HOST/PORT ... SERVICES ...

file-sharing file-sharing-storage-review.apps.ocp4.example.com ... file-sharing ...

* 1. Test the connectivity between the web application and the database server. In a web browser, navigate to http://file-sharing-storage-review.apps.ocp4.example.com, and verify that a Connected successfully message is displayed.

Hide Solution

1. Mount the dbfiles configuration map to the file-sharing deployment as a volume named config-map-pvc. Set the mount path to the /home/database-files directory. Then, verify the content of the insertdata.sql file.
   1. Mount the dbfiles configuration map to the file-sharing deployment.
   2. [student@workstation ~]$ **oc set volume deployment/file-sharing \**
   3. **--add --name config-map-pvc --type configmap \**
   4. **--configmap-name dbfiles \**
   5. **--mount-path /home/database-files**

deployment.apps/file-sharing volume updated

* 1. Verify the deployment status.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. dbserver-***5bc6bd5d7b-7z7lv*** 1/1 Running ...
  5. file-sharing-***7f77855b7f-949lg*** 1/1 Running ...

file-sharing-***7f77855b7f-9zvwq*** 1/1 Running ...

* 1. Verify the content of the /home/database-files/insertdata.sql file.
  2. [student@workstation ~]$ **oc exec -it pod/file-sharing-*7f77855b7f-949lg* -- \**
  3. **head /home/database-files/insertdata.sql**
  4. -- MySQL dump 10.13 Distrib 8.0.19, for osx10.14 (x86\_64)
  5. --
  6. -- Host: 127.0.0.1 Database: world\_x
  7. -- ------------------------------------------------------
  8. -- Server version 8.0.19-debug

*...output omitted...*

1. Hide Solution
2. Add a shared volume to the file-sharing deployment. Use the following information to create the volume:

| **Field** | **Value** |
| --- | --- |
| Name | shared-volume |
| Type | persistentVolumeClaim |
| Claim mode | rwo |
| Claim size | 1Gi |
| Mount path | /home/sharedfiles |
| Claim class | nfs-storage |
| Claim name | shared-pvc |

1. Next, connect to a file-sharing deployment pod and then use the cp command to copy the /home/database-files/insertdata.sql file to the /home/sharedfiles directory. Then, remove the config-map-pvc volume from the file-sharing deployment.
   1. Add the shared-volume volume to the file-sharing deployment.
   2. [student@workstation ~]$ **oc set volume deployment/file-sharing \**
   3. **--add --name shared-volume --type persistentVolumeClaim \**
   4. **--claim-mode rwo --claim-size 1Gi --mount-path /home/sharedfiles \**
   5. **--claim-class nfs-storage --claim-name shared-pvc**

deployment.apps/file-sharing volume updated

* 1. Verify the deployment status. Your pod names might differ on your system.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. dbserver-***5bc6bd5d7b-7z7lv*** 1/1 Running ...
  5. file-sharing-***65884f75bb-92fxf*** 1/1 Running ...

file-sharing-***65884f75bb-gsghk*** 1/1 Running ...

* 1. Verify the volume status.
  2. [student@workstation ~]$ **oc get pvc**
  3. NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS
  4. dbserver-lvm-pvc Bound pvc-2cb... 1Gi RWO lvms-vg1 ...

shared-pvc Bound pvc-cf2... 1Gi RWO nfs-storage ...

* 1. Copy the /home/database-files/insertdata.sql file to the /home/sharedfiles path.
  2. [student@workstation ~]$ **oc exec -it pod/file-sharing-*65884f75bb-92fxf* -- \**

**cp /home/database-files/insertdata.sql /home/sharedfiles/**

[student@workstation ~]$ **oc exec -it pod/file-sharing-*65884f75bb-92fxf* -- \**

**ls /home/sharedfiles/**

insertdata.sql

* 1. Remove the config-map-pvc volume from the file-sharing deployment.
  2. [student@workstation ~]$ **oc set volume deployment/file-sharing \**
  3. **--remove --name=config-map-pvc**

deployment.apps/file-sharing volume updated

1. Hide Solution
2. Add the shared-volume PVC to the dbserver deployment. Then, connect to a dbserver deployment pod and verify the content of the /home/sharedfiles/insertdata.sql file.
   1. Add the shared-volume volume to the dbserver deployment.
   2. [student@workstation ~]$ **oc set volume deployment/dbserver \**
   3. **--add --name shared-volume \**
   4. **--claim-name shared-pvc \**
   5. **--mount-path /home/sharedfiles**

deployment.apps/dbserver volume updated

* 1. Verify the deployment status. The pod names might differ on your system.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS ...
  4. dbserver-***6676fbf5fc-n9hpk*** 1/1 Running ...
  5. file-sharing-***5fdb44cf57-2hhwj*** 1/1 Running ...

file-sharing-***5fdb44cf57-z4n7g*** 1/1 Running ...

* 1. Verify the content of the /home/sharedfiles/insertdata.sql file.
  2. [student@workstation ~]$ **oc exec -it pod/dbserver-*6676fbf5fc-n9hpk* -- \**
  3. **head /home/sharedfiles/insertdata.sql**
  4. -- MySQL dump 10.13 Distrib 8.0.19, for osx10.14 (x86\_64)
  5. --
  6. -- Host: 127.0.0.1 Database: world\_x
  7. -- ------------------------------------------------------
  8. -- Server version 8.0.19-debug

*...output omitted...*

1. Hide Solution
2. Connect to the database server and execute the /home/sharedfiles/insertdata.sql file to add data to the world\_x database. You can execute the file by using the following command:

**mysql -u$MYSQL\_USER -p$MYSQL\_PASSWORD world\_x </home/sharedfiles/insertdata.sql**

Then, confirm connectivity between the web application and database server by accessing the file-sharing route in a web browser.

* 1. Connect to the database server and execute the /home/sharedfiles/insertdata.sql file. Then, exit the database server.

[student@workstation ~]$ **oc rsh dbserver-*6676fbf5fc-n9hpk***

sh-4.4$ **mysql -u$MYSQL\_USER -p$MYSQL\_PASSWORD world\_x </home/sharedfiles/insertdata.sql**

mysql: [Warning] Using a password on the command line interface can be insecure.

sh-4.4$ **exit**

exit

* 1. Test the connectivity between the web application and the database server. In a web browser, navigate to http://file-sharing-storage-review.apps.ocp4.example.com, and verify that the application retrieves data from the world\_x database.

|  |
| --- |
| A group of black text  Description automatically generated |

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 storage-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 storage-review**

Summary

* Configuration maps are objects that provide mechanisms to inject configuration data into containers.
* The values in secrets are always encoded (not encrypted).
* A persistent volume claim (PVC) resource represents a request from an application for storage, and specifies the minimal storage characteristics, such as the capacity and access mode.
* Kubernetes supports two volume modes for persistent volumes: Filesystem and Block.
* Storage classes are a way to describe types of storage for the cluster and to provision dynamic storage on demand.
* A reclaim policy determines what happens to the data on a PVC after the PVC is deleted.
* A storage class with block volume mode support can improve performance for applications that can use raw block devices.
* A stateful set is the representation of a set of pods with consistent identities.
* Stateful set pods are assigned individual persistent volumes.