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. Multiple pods on that single node can access the volume simultaneously.
`ReadWriteOncePod`	RWOP	A single pod in the cluster 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	RWOP	ROX	RWX
configMap	Yes	Yes	No	No
emptyDir	Yes	Yes	No	No
hostPath	Yes	Yes	No	No
iSCSI	Yes	Yes	Yes	No
local	Yes	Yes	No	No
NFS	Yes	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

Volume plug-in	Manually provisioned	Dynamically provisioned
Red Hat OpenShift Data Foundation	Yes	Yes
VMware vSphere	Yes	Yes

Manually Creating a PV

Administrators use `PersistentVolume` manifest files to manually create persistent volumes. The following example creates a persistent volume from a fiber channel storage device that uses block mode.

```
apiVersion: v1
kind: PersistentVolume (1)
metadata:
  name: block-pv (2)
spec:
  capacity:
    storage: 10Gi (3)
  accessModes:
    - ReadWriteOnce (4)
  volumeMode: Block (5)
  persistentVolumeReclaimPolicy: Retain (6)
  fc: (7)
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

1. `PersistentVolume` is the resource type for PVs.
2. Provide a name for the PV, which subsequent claims use to access the PV.
3. Specify the amount of storage that is allocated to this volume.
4. The storage device must support the access mode that the PV specifies.
5. The `volumeMode` attribute is optional for `Filesystem` volumes, but is required for `Block` volumes.
6. The `persistentVolumeReclaimPolicy` determines how the cluster handles the PV when the PVC is deleted. Valid options are `Retain` or `Delete`.
7. 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 log in as an administrator and run the following command to create the PV resource:

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

To create a persistent volume from the web console, log in as an administrator and go to **Storage** → **PersistentVolumes**.

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, developers 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 your 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, developers can use the `oc set volumes` command:

```
[user@host ~]$ oc set volumes deployment/example-application \ 1
--add \ 2
--name example-pv-storage \ 3
--type persistentVolumeClaim \ 4
--claim-mode rwo \ 5
--claim-size 15Gi \ 6
--mount-path /var/lib/example-app \ 7
--claim-name mypvc 8
```

1. Specify the name of the deployment that requires the PVC resource.
2. Setting the `add` option to `true` adds volumes and volume mounts for containers.
3. The `name` option specifies a volume name. If not specified, a name is autogenerated.
4. The supported types, for the `add` operation, include `emptyDir`, `hostPath`, `secret`, `configMap`, and `persistentVolumeClaim`.
5. The `claim-mode` option defaults to `ReadWriteOnce`. The valid values are `ReadWriteOnce` (RWO), `ReadWriteOncePod` (RWOP), `ReadWriteMany` (RWX), and `ReadOnlyMany` (ROX).
6. 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.
7. The `mount-path` option specifies the mount path inside the container.
8. 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 1
  ...output omitted...
spec:
  ...output omitted...
  template:
  ...output omitted...
    spec:
  ...output omitted...
      volumeMounts:
      - mountPath: /var/lib/example-app 2
        name: example-pv-storage 3
  ...output omitted...
    volumes:
    - name: example-pv-storage 4
      persistentVolumeClaim:
        claimName: mypvc 5
  ...output omitted...
```

1. The deployment, which must be in the same namespace as the PVC.
2. The mount path in the container.
3. 4 The volume name, which is used to specify the volume that is associated with the mount.
5. 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 (1)
metadata:
  name: mypvc (2)
  labels:
    app: example-application
spec:
  accessModes:
    - ReadWriteOnce (3)
  resources:
    requests:
      storage: 15Gi (4)
```

1. `PersistentVolumeClaim` is the resource type for a PVC.
2. Use this name in the `claimName` field of the `persistentVolumeClaim` element in the `volumes` section of a deployment manifest.
3. 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.
4. 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.

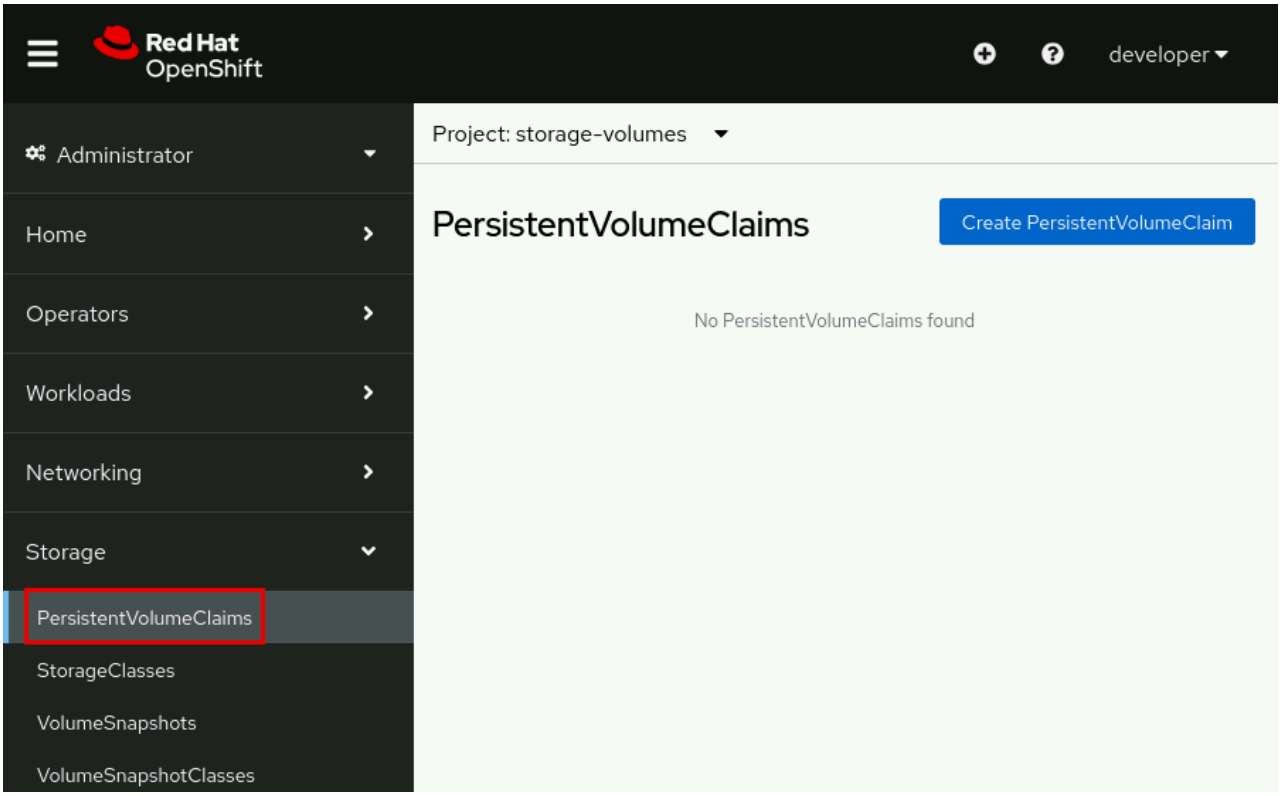
Developers can use the `oc create` command to create the PVC from the manifest file.

```
[user@host ~]$ oc create -f pvc_file_name.yaml
```

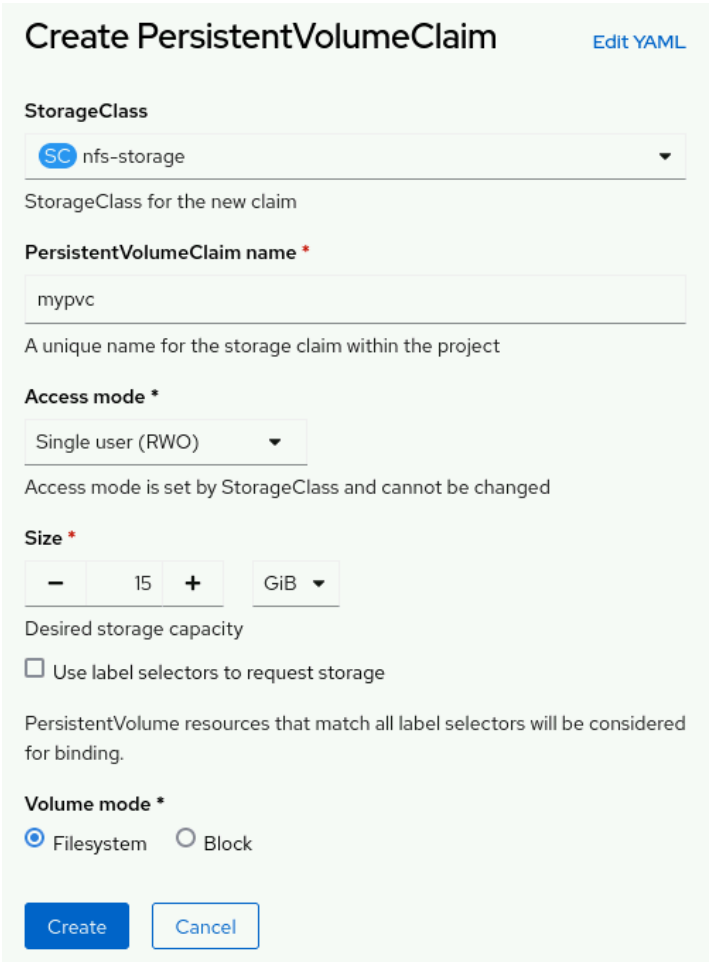
Use `oc get pvc` to view the available PVCs in the current namespace.

```
[user@host ~]$ oc get pvc
NAME   STATUS  VOLUME    CAPACITY  ACCESS MODES  STORAGECLASS ...
mypvc  Bound   pvc-13... 15Gi      RWO           nfs-storage  ...
```

As a developer, to create a persistent volume claim from the web console, go to **Storage** → **PersistentVolumesClaims**.



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



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.

To create PVCs, developers specify the 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 GiB volumes would not match a PVC that requests 100 GiB. The PVC can be bound when a 100 GiB PV is added to the cluster.

Developers can run the `oc get storageclass` command to view the storage classes that the cluster provides.

```
[user@host ~]$ oc get storageclass
NAME                 PROVISIONER ...
lvms-vg1             topolvm.io  ...
nfs-storage (default) k8s-sigs.io/nfs-subdir-external-provisioner
```

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 mypvc
```

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

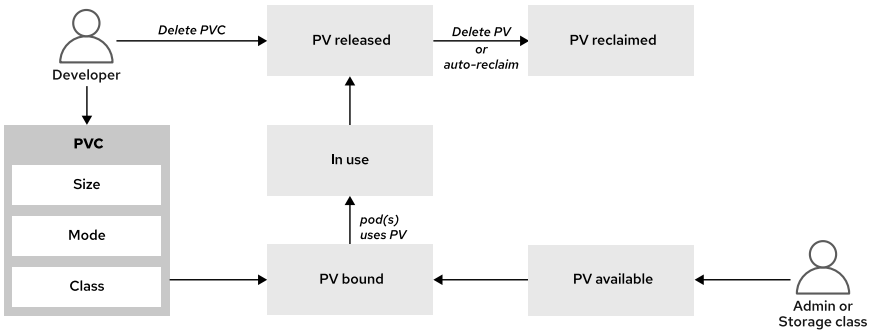


Figure 5.18: PV lifecycle

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, developers can use the `oc delete` command to delete the PVC. The storage class reclaims the volume after removing the PVC.

```
[user@host ~]$ oc delete pvc/mypvc
```

REFERENCES

For more information about ephemeral storage, refer to the *Understanding Ephemeral Storage* chapter in the Red Hat OpenShift Container Platform 4.18 *Storage* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/html-single/storage/index#understanding-ephemeral-storage

For more information about persistent storage, refer to the *Understanding Persistent Storage* chapter in the Red Hat OpenShift Container Platform 4.18 *Storage* documentation at https://docs.redhat.com/en/documentation/openshift_container_platform/4.14/html-single/storage/index#understanding-persistent-storage

How Kubernetes Storage Improves Developer Agility

A Developer's Guide to Kubernetes Storage Concepts

Kubernetes Persistent Volume: Examples and Best Practices