



OpenShift Container Platform 4.4

Storage

Configuring and managing storage in OpenShift Container Platform

OpenShift Container Platform 4.4 Storage

Configuring and managing storage in OpenShift Container Platform

Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides instructions for configuring persistent volumes from various storage back ends and managing dynamic allocation from Pods.

Table of Contents

CHAPTER 1. UNDERSTANDING EPHEMERAL STORAGE	5
1.1. OVERVIEW	5
1.2. TYPES OF EPHEMERAL STORAGE	5
Root	5
Runtime	5
1.3. EPHEMERAL STORAGE MANAGEMENT	5
1.4. MONITORING EPHEMERAL STORAGE	5
CHAPTER 2. UNDERSTANDING PERSISTENT STORAGE	7
2.1. PERSISTENT STORAGE OVERVIEW	7
2.2. LIFECYCLE OF A VOLUME AND CLAIM	7
2.2.1. Provision storage	7
2.2.2. Bind claims	7
2.2.3. Use pods and claimed PVs	8
2.2.4. Storage Object in Use Protection	8
2.2.5. Release a persistent volume	8
2.2.6. Reclaim policy for persistent volumes	8
2.2.7. Reclaiming a persistent volume manually	9
2.2.8. Changing the reclaim policy of a persistent volume	9
2.3. PERSISTENT VOLUMES	10
2.3.1. Types of PVs	11
2.3.2. Capacity	11
2.3.3. Access modes	11
2.3.4. Phase	13
2.3.4.1. Mount options	14
2.4. PERSISTENT VOLUME CLAIMS	15
2.4.1. Storage classes	15
2.4.2. Access modes	16
2.4.3. Resources	16
2.4.4. Claims as volumes	16
2.5. BLOCK VOLUME SUPPORT	16
2.5.1. Block volume examples	17
CHAPTER 3. CONFIGURING PERSISTENT STORAGE	20
3.1. PERSISTENT STORAGE USING AWS ELASTIC FILE SYSTEM	20
3.1.1. Prerequisites	20
3.1.2. Store the EFS variables in a config map	20
3.1.3. Configuring authorization for EFS volumes	21
3.1.4. Create the EFS storage class	23
3.1.5. Create the EFS provisioner	23
3.1.6. Create the EFS persistent volume claim	25
3.2. PERSISTENT STORAGE USING AWS ELASTIC BLOCK STORE	26
3.2.1. Creating the EBS storage class	26
3.2.2. Creating the persistent volume claim	27
3.2.3. Volume format	27
3.2.4. Maximum number of EBS volumes on a node	27
3.3. PERSISTENT STORAGE USING AZURE	27
3.3.1. Creating the Azure storage class	28
3.3.2. Creating the persistent volume claim	29
3.3.3. Volume format	29
3.4. PERSISTENT STORAGE USING AZURE FILE	29

3.4.1. Create the Azure File share persistent volume claim	30
3.4.2. Mount the Azure File share in a pod	31
3.5. PERSISTENT STORAGE USING CINDER	32
3.5.1. Manual provisioning with Cinder	32
3.5.1.1. Creating the persistent volume	32
3.5.1.2. Persistent volume formatting	33
3.5.1.3. Cinder volume security	33
3.6. PERSISTENT STORAGE USING FIBRE CHANNEL	34
3.6.1. Provisioning	35
3.6.1.1. Enforcing disk quotas	35
3.6.1.2. Fibre Channel volume security	36
3.7. PERSISTENT STORAGE USING FLEXVOLUME	36
3.7.1. About FlexVolume drivers	36
3.7.2. FlexVolume driver example	36
3.7.3. Installing FlexVolume drivers	37
3.7.4. Consuming storage using FlexVolume drivers	38
3.8. PERSISTENT STORAGE USING GCE PERSISTENT DISK	39
3.8.1. Creating the GCE storage class	40
3.8.2. Creating the persistent volume claim	40
3.8.3. Volume format	41
3.9. PERSISTENT STORAGE USING HOSTPATH	41
3.9.1. Overview	41
3.9.2. Statically provisioning hostPath volumes	41
3.9.3. Mounting the hostPath share in a privileged pod	42
3.10. PERSISTENT STORAGE USING ISCSI	43
3.10.1. Provisioning	44
3.10.2. Enforcing disk quotas	44
3.10.3. iSCSI volume security	44
3.10.3.1. Challenge Handshake Authentication Protocol (CHAP) configuration	44
3.10.4. iSCSI multipathing	45
3.10.5. iSCSI custom initiator IQN	45
3.11. PERSISTENT STORAGE USING LOCAL VOLUMES	46
3.11.1. Installing the Local Storage Operator	46
3.11.2. Provisioning the local volumes	48
3.11.3. Creating the local volume persistent volume claim	51
3.11.4. Attach the local claim	52
3.11.5. Using tolerations with Local Storage Operator pods	53
3.11.6. Deleting the Local Storage Operator's resources	54
3.11.6.1. Removing a local volume	54
3.11.6.2. Uninstalling the Local Storage Operator	55
3.12. PERSISTENT STORAGE USING NFS	56
3.12.1. Provisioning	56
3.12.2. Enforcing disk quotas	58
3.12.3. NFS volume security	58
3.12.3.1. Group IDs	58
3.12.3.2. User IDs	59
3.12.3.3. SELinux	60
3.12.3.4. Export settings	60
3.12.4. Reclaiming resources	61
3.12.5. Additional configuration and troubleshooting	62
3.13. RED HAT OPENSIFT CONTAINER STORAGE	62
3.14. PERSISTENT STORAGE USING VMWARE VSPHERE VOLUMES	63
3.14.1. Dynamically provisioning VMware vSphere volumes	63

3.14.2. Prerequisites	63
3.14.2.1. Dynamically provisioning VMware vSphere volumes using the UI	64
3.14.2.2. Dynamically provisioning VMware vSphere volumes using the CLI	64
3.14.3. Statically provisioning VMware vSphere volumes	65
3.14.3.1. Formatting VMware vSphere volumes	67
3.14.4. Backing up VMware vSphere volumes	67
CHAPTER 4. USING CONTAINER STORAGE INTERFACE (CSI)	68
4.1. CONFIGURING CSI VOLUMES	68
4.1.1. CSI Architecture	68
4.1.1.1. External CSI controllers	68
4.1.1.2. CSI driver daemon set	69
4.1.2. Dynamic provisioning	69
4.1.3. Example using the CSI driver	70
4.2. CSI VOLUME SNAPSHOTS	70
4.2.1. Overview of CSI volume snapshots	71
4.2.2. CSI snapshot controller and sidecar	71
4.2.2.1. External controller	72
4.2.2.2. External sidecar	72
4.2.3. About the CSI Snapshot Controller Operator	72
4.2.3.1. Volume snapshot CRDs	72
4.2.4. Volume snapshot provisioning	73
4.2.4.1. Dynamic provisioning	73
4.2.4.2. Manual provisioning	73
4.2.5. Creating a volume snapshot	73
4.2.6. Deleting a volume snapshot	76
4.2.7. Restoring a volume snapshot	76
4.3. CSI VOLUME CLONING	77
4.3.1. Overview of CSI volume cloning	77
4.3.1.1. Support limitations	78
4.3.2. Provisioning a CSI volume clone	78
CHAPTER 5. EXPANDING PERSISTENT VOLUMES	81
5.1. ENABLING VOLUME EXPANSION SUPPORT	81
5.2. EXPANDING CSI VOLUMES	81
5.3. EXPANDING FLEXVOLUME WITH A SUPPORTED DRIVER	81
5.4. EXPANDING PERSISTENT VOLUME CLAIMS (PVCS) WITH A FILE SYSTEM	82
5.5. RECOVERING FROM FAILURE WHEN EXPANDING VOLUMES	83
CHAPTER 6. DYNAMIC PROVISIONING	84
6.1. ABOUT DYNAMIC PROVISIONING	84
6.2. AVAILABLE DYNAMIC PROVISIONING PLUG-INS	84
6.3. DEFINING A STORAGE CLASS	85
6.3.1. Basic StorageClass object definition	85
6.3.2. Storage class annotations	86
6.3.3. RHOSP Cinder object definition	87
6.3.4. AWS Elastic Block Store (EBS) object definition	87
6.3.5. Azure Disk object definition	88
6.3.6. Azure File object definition	89
6.3.6.1. Considerations when using Azure File	90
6.3.7. GCE PersistentDisk (gcePD) object definition	91
6.3.8. VMware vSphere object definition	91
6.3.9. Red Hat OpenShift Container Storage object definition	91
6.4. CHANGING THE DEFAULT STORAGE CLASS	91

CHAPTER 1. UNDERSTANDING EPHEMERAL STORAGE

1.1. OVERVIEW

In addition to persistent storage, pods and containers can require ephemeral or transient local storage for their operation. 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.

Pods use ephemeral local storage for scratch space, caching, and logs. Issues related to the lack of local storage accounting and isolation include the following:

- Pods do not know how much local storage is available to them.
- Pods cannot request guaranteed local storage.
- Local storage is a best effort resource.
- Pods can be evicted due to other pods filling the local storage, after which new pods are not admitted until sufficient storage has been reclaimed.

Unlike persistent volumes, ephemeral storage is unstructured and the space is shared between all pods running on a node, in addition to other uses by the system, the container runtime, and OpenShift Container Platform. The ephemeral storage framework allows pods to specify their transient local storage needs. It also allows OpenShift Container Platform to schedule pods where appropriate, and to protect the node against excessive use of local storage.

While the ephemeral storage framework allows administrators and developers to better manage this local storage, it does not provide any promises related to I/O throughput and latency.

1.2. TYPES OF EPHEMERAL STORAGE

Ephemeral local storage is always made available in the primary partition. There are two basic ways of creating the primary partition: `root` and `runtime`.

Root

This partition holds the kubelet root directory, `/var/lib/kubelet/` by default, and `/var/log/` directory. This partition can be shared between user pods, the OS, and Kubernetes system daemons. This partition can be consumed by pods through **EmptyDir** volumes, container logs, image layers, and container-writable layers. Kubelet manages shared access and isolation of this partition. This partition is ephemeral, and applications cannot expect any performance SLAs, such as disk IOPS, from this partition.

Runtime

This is an optional partition that runtimes can use for overlay file systems. OpenShift Container Platform attempts to identify and provide shared access along with isolation to this partition. Container image layers and writable layers are stored here. If the runtime partition exists, the **root** partition does not hold any image layer or other writable storage.

1.3. EPHEMERAL STORAGE MANAGEMENT

Cluster administrators can manage ephemeral storage within a project by setting quotas that define the limit ranges and number of requests for ephemeral storage across all pods in a non-terminal state. Developers can also set requests and limits on this compute resource at the pod and container level.

1.4. MONITORING EPHEMERAL STORAGE

You can use **/bin/df** as a tool to monitor ephemeral storage usage on the volume where ephemeral container data is located, which is **/var/lib/kubelet** and **/var/lib/containers**. The available space for only **/var/lib/kubelet** is shown when you use the **df** command if **/var/lib/containers** is placed on a separate disk by the cluster administrator.

To show the human-readable values of used and available space in **/var/lib**, enter the following command:

```
$ df -h /var/lib
```

The output shows the ephemeral storage usage in **/var/lib**:

Example output

```
Filesystem Size Used Avail Use% Mounted on
/dev/sda1 69G 32G 34G 49% /
```

CHAPTER 2. UNDERSTANDING PERSISTENT STORAGE

2.1. PERSISTENT STORAGE OVERVIEW

Managing storage is a distinct problem from managing compute resources. OpenShift Container Platform uses the Kubernetes persistent volume (PV) framework to allow cluster administrators to provision persistent storage for a cluster. Developers can use persistent volume claims (PVCs) to request PV resources without having specific knowledge of the underlying storage infrastructure.

PVCs are specific to a project, and are created and used by developers as a means to use a PV. PV resources on their own are not scoped to any single project; they can be shared across the entire OpenShift Container Platform cluster and claimed from any project. After a PV is bound to a PVC, that PV can not then be bound to additional PVCs. This has the effect of scoping a bound PV to a single namespace, that of the binding project.

PVs are defined by a **PersistentVolume** API object, which represents a piece of existing storage in the cluster that was either statically provisioned by the cluster administrator or dynamically provisioned using a **StorageClass** object. It is a resource in the cluster just like a node is a cluster resource.

PVs are volume plug-ins like **Volumes** but have a lifecycle that is independent of any individual pod that uses the PV. PV objects capture the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.



IMPORTANT

High availability of storage in the infrastructure is left to the underlying storage provider.

PVCs are defined by a **PersistentVolumeClaim** API object, which represents a request for storage by a developer. It is similar to a pod in that pods consume node resources and PVCs consume PV resources. For example, pods can request specific levels of resources, such as CPU and memory, while PVCs can request specific storage capacity and access modes. For example, they can be mounted once read-write or many times read-only.

2.2. LIFECYCLE OF A VOLUME AND CLAIM

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs have the following lifecycle.

2.2.1. Provision storage

In response to requests from a developer defined in a PVC, a cluster administrator configures one or more dynamic provisioners that provision storage and a matching PV.

Alternatively, a cluster administrator can create a number of PVs in advance that carry the details of the real storage that is available for use. PVs exist in the API and are available for use.

2.2.2. Bind claims

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

The size of all PVs might exceed your PVC size. This is especially true with manually provisioned PVs. To minimize the excess, OpenShift Container Platform binds to the smallest PV that matches all other criteria.

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

2.2.3. Use pods and claimed PVs

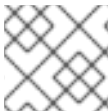
Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a pod. For those volumes that support multiple access modes, you must specify which mode applies when you use the claim as a volume in a pod.

Once you have a claim and that claim is bound, the bound PV belongs to you for as long as you need it. You can schedule pods and access claimed PVs by including **persistentVolumeClaim** in the pod's volumes block.

2.2.4. Storage Object in Use Protection

The Storage Object in Use Protection feature ensures that PVCs in active use by a pod and PVs that are bound to PVCs are not removed from the system, as this can result in data loss.

Storage Object in Use Protection is enabled by default.



NOTE

A PVC is in active use by a pod when a **Pod** object exists that uses the PVC.

If a user deletes a PVC that is in active use by a pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any pods. Also, if a cluster admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

2.2.5. Release a persistent volume

When you are finished with a volume, you can delete the PVC object from the API, which allows reclamation of the resource. The volume is considered released when the claim is deleted, but it is not yet available for another claim. The previous claimant's data remains on the volume and must be handled according to policy.

2.2.6. Reclaim policy for persistent volumes

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**, **Recycle**, or **Delete**.

- **Retain** reclaim policy allows manual reclamation of the resource for those volume plug-ins that support it.
- **Recycle** reclaim policy recycles the volume back into the pool of unbound persistent volumes once it is released from its claim.

**IMPORTANT**

The **Recycle** reclaim policy is deprecated in OpenShift Container Platform 4. Dynamic provisioning is recommended for equivalent and better functionality.

- **Delete** reclaim policy deletes both the **PersistentVolume** object from OpenShift Container Platform and the associated storage asset in external infrastructure, such as AWS EBS or VMware vSphere.

**NOTE**

Dynamically provisioned volumes are always deleted.

2.2.7. Reclaiming a persistent volume manually

When a persistent volume claim (PVC) is deleted, the persistent volume (PV) still exists and is considered "released". However, the PV is not yet available for another claim because the data of the previous claimant remains on the volume.

Procedure

To manually reclaim the PV as a cluster administrator:

1. Delete the PV.

```
$ oc delete <pv-name>
```

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

2. Clean up the data on the associated storage asset.
3. Delete the associated storage asset. Alternately, to reuse the same storage asset, create a new PV with the storage asset definition.

The reclaimed PV is now available for use by another PVC.

2.2.8. Changing the reclaim policy of a persistent volume

To change the reclaim policy of a persistent volume:

1. List the persistent volumes in your cluster:

```
$ oc get pv
```

Example output

```
NAME                                     CAPACITY ACCESSMODES RECLAIMPOLICY STATUS
CLAIM STORAGECLASS REASON AGE
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94 4Gi RWO Delete Bound
default/claim1 manual 10s
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94 4Gi RWO Delete Bound
```

```

default/claim2 manual          6s
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94 4Gi RWO Delete Bound
default/claim3 manual          3s

```

- Choose one of your persistent volumes and change its reclaim policy:

```
$ oc patch pv <your-pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

- Verify that your chosen persistent volume has the right policy:

```
$ oc get pv
```

Example output

```

NAME                                CAPACITY ACCESSMODES RECLAIMPOLICY STATUS
CLAIM STORAGECLASS REASON AGE
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94 4Gi RWO Delete Bound
default/claim1 manual          10s
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94 4Gi RWO Delete Bound
default/claim2 manual          6s
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94 4Gi RWO Retain Bound
default/claim3 manual          3s

```

In the preceding output, the volume bound to claim **default/claim3** now has a **Retain** reclaim policy. The volume will not be automatically deleted when a user deletes claim **default/claim3**.

2.3. PERSISTENT VOLUMES

Each PV contains a **spec** and **status**, which is the specification and status of the volume, for example:

PersistentVolume object definition example

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001 ❶
spec:
  capacity:
    storage: 5Gi ❷
  accessModes:
    - ReadWriteOnce ❸
  persistentVolumeReclaimPolicy: Retain ❹
  ...
status:
  ...

```

- Name of the persistent volume.
- The amount of storage available to the volume.
- The access mode, defining the read-write and mount permissions.
- The reclaim policy, indicating how the resource should be handled once it is released.

2.3.1. Types of PVs

OpenShift Container Platform supports the following persistent volume plug-ins:

- AWS Elastic Block Store (EBS)
- Azure Disk
- Azure File
- Cinder
- Fibre Channel
- GCE Persistent Disk
- HostPath
- iSCSI
- Local volume
- NFS
- OpenStack Manila
- Red Hat OpenShift Container Storage
- VMware vSphere

2.3.2. Capacity

Generally, a persistent volume (PV) has a specific storage capacity. This is set by using the **capacity** attribute of the PV.

Currently, storage capacity is the only resource that can be set or requested. Future attributes may include IOPS, throughput, and so on.

2.3.3. Access modes

A persistent volume can be mounted on a host in any way supported by the resource provider. Providers have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read-write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

Claims are matched to volumes with similar access modes. The only two matching criteria are access modes and size. A claim's access modes represent a request. Therefore, you might be granted more, but never less. For example, if a claim requests RWO, but the only volume available is an NFS PV (RWO+ROX+RWX), the claim would then match NFS because it supports RWO.

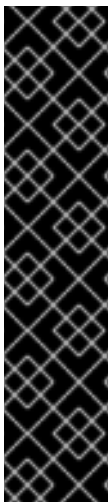
Direct matches are always attempted first. The volume's modes must match or contain more modes than you requested. The size must be greater than or equal to what is expected. If two types of volumes, such as NFS and iSCSI, have the same set of access modes, either of them can match a claim with those modes. There is no ordering between types of volumes and no way to choose one type over another.

All volumes with the same modes are grouped, and then sorted by size, smallest to largest. The binder gets the group with matching modes and iterates over each, in size order, until one size matches.

The following table lists the access modes:

Table 2.1. Access modes

Access Mode	CLI abbreviation	Description
ReadWriteOnce	RWO	The volume can be mounted as read-write by a single node.
ReadOnlyMany	ROX	The volume can be mounted as read-only by many nodes.
ReadWriteMany	RWX	The volume can be mounted as read-write by many nodes.



IMPORTANT

Volume access modes are descriptors of volume capabilities. They are not enforced constraints. The storage provider is responsible for runtime errors resulting from invalid use of the resource.

For example, NFS offers **ReadWriteOnce** access mode. You must mark the claims as **read-only** if you want to use the volume's ROX capability. Errors in the provider show up at runtime as mount errors.

iSCSI and Fibre Channel volumes do not currently have any fencing mechanisms. You must ensure the volumes are only used by one node at a time. In certain situations, such as draining a node, the volumes can be used simultaneously by two nodes. Before draining the node, first ensure the pods that use these volumes are deleted.

Table 2.2. Supported access modes for PVs

Volume plug-in	ReadWriteOnce [1]	ReadOnlyMany	ReadWriteMany
AWS EBS [2]	■	-	-
Azure File	■	■	■
Azure Disk	■	-	-
Cinder	■	-	-
Fibre Channel	■	■	-
GCE Persistent Disk	■	-	-

Volume plug-in	ReadWriteOnce [!]	ReadOnlyMany	ReadWriteMany
HostPath	■	-	-
iSCSI	■	■	-
Local volume	■	-	-
NFS	■	■	■
OpenStack Manila	-	-	■
Red Hat OpenShift Container Storage	■	-	■
VMware vSphere	■	-	-

1. ReadWriteOnce (RWO) volumes cannot be mounted on multiple nodes. If a node fails, the system does not allow the attached RWO volume to be mounted on a new node because it is already assigned to the failed node. If you encounter a multi-attach error message as a result, you can either recover or delete the failed node to make the volume available to other nodes.
2. Use a recreate deployment strategy for pods that rely on AWS EBS.

2.3.4. Phase

Volumes can be found in one of the following phases:

Table 2.3. Volume phases

Phase	Description
Available	A free resource not yet bound to a claim.
Bound	The volume is bound to a claim.
Released	The claim was deleted, but the resource is not yet reclaimed by the cluster.
Failed	The volume has failed its automatic reclamation.

You can view the name of the PVC bound to the PV by running:

```
$ oc get pv <pv-claim>
```

2.3.4.1. Mount options

You can specify mount options while mounting a PV by using the annotation **volume.beta.kubernetes.io/mount-options**.

For example:

Mount options example

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
  annotations:
    volume.beta.kubernetes.io/mount-options: rw,nfsvers=4,noexec 1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /tmp
    server: 172.17.0.2
  persistentVolumeReclaimPolicy: Retain
  claimRef:
    name: claim1
    namespace: default
```

1 Specified mount options are used while mounting the PV to the disk.

The following PV types support mount options:

- AWS Elastic Block Store (EBS)
- Azure Disk
- Azure File
- Cinder
- GCE Persistent Disk
- iSCSI
- Local volume
- NFS
- Red Hat OpenShift Container Storage (Ceph RBD only)
- VMware vSphere



NOTE

Fibre Channel and HostPath PVs do not support mount options.

2.4. PERSISTENT VOLUME CLAIMS

Each **PersistentVolumeClaim** object contains a **spec** and **status**, which is the specification and status of the persistent volume claim (PVC), for example:

PersistentVolumeClaim object definition example

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim 1
spec:
  accessModes:
    - ReadWriteOnce 2
  resources:
    requests:
      storage: 8Gi 3
  storageClassName: gold 4
status:
  ...
```

- 1 Name of the PVC
- 2 The access mode, defining the read-write and mount permissions
- 3 The amount of storage available to the PVC
- 4 Name of the **StorageClass** required by the claim

2.4.1. Storage classes

Claims can optionally request a specific storage class by specifying the storage class's name in the **storageClassName** attribute. Only PVs of the requested class, ones with the same **storageClassName** as the PVC, can be bound to the PVC. The cluster administrator can configure dynamic provisioners to service one or more storage classes. The cluster administrator can create a PV on demand that matches the specifications in the PVC.



IMPORTANT

The Cluster Storage Operator might install a default storage class depending on the platform in use. This storage class is owned and controlled by the operator. It cannot be deleted or modified beyond defining annotations and labels. If different behavior is desired, you must define a custom storage class.

The cluster administrator can also set a default storage class for all PVCs. When a default storage class is configured, the PVC must explicitly ask for **StorageClass** or **storageClassName** annotations set to "" to be bound to a PV without a storage class.



NOTE

If more than one storage class is marked as default, a PVC can only be created if the **storageClassName** is explicitly specified. Therefore, only one storage class should be set as the default.

2.4.2. Access modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

2.4.3. Resources

Claims, such as pods, can request specific quantities of a resource. In this case, the request is for storage. The same resource model applies to volumes and claims.

2.4.4. Claims as volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the pod by using the claim. The cluster finds the claim in the pod's namespace and uses it to get the **PersistentVolume** backing the claim. The volume is mounted to the host and into the pod, for example:

Mount volume to the host and into the pod example

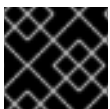
```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: myfrontend
    image: dockerfile/nginx
    volumeMounts:
    - mountPath: "/var/www/html" 1
      name: mypd 2
  volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim 3
```

- 1 Path to mount the volume inside the pod
- 2 Name of the volume to mount
- 3 Name of the PVC, that exists in the same namespace, to use

2.5. BLOCK VOLUME SUPPORT

OpenShift Container Platform can statically 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 implement their own storage service.

Raw block volumes are provisioned by specifying **volumeMode: Block** in the PV and PVC specification.



IMPORTANT

Pods using raw block volumes must be configured to allow privileged containers.

The following table displays which volume plug-ins support block volumes.

Table 2.4. Block volume support

Volume Plug-in	Manually provisioned	Dynamically provisioned	Fully supported
AWS EBS	■	■	■
Azure Disk	■	■	■
Azure File			
Cinder	■	■	
Fibre Channel	■		
GCP	■	■	■
HostPath			
iSCSI	■		■
Local volume	■		■
NFS			
Red Hat OpenShift Container Storage	■	■	■
VMware vSphere	■	■	■

**NOTE**

Any of the block volumes that can be provisioned manually, but are not provided as fully supported, are included as a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see <https://access.redhat.com/support/offerings/techpreview/>.

2.5.1. Block volume examples**PV example**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
```

```

capacity:
  storage: 10Gi
accessModes:
  - ReadWriteOnce
volumeMode: Block 1
persistentVolumeReclaimPolicy: Retain
fc:
  targetWWNs: ["50060e801049cfd1"]
  lun: 0
  readOnly: false

```

- 1** **volumeMode** must be set to **Block** to indicate that this PV is a raw block volume.

PVC example

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block 1
  resources:
    requests:
      storage: 10Gi

```

- 1** **volumeMode** must be set to **Block** to indicate that a raw block PVC is requested.

Pod specification example

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]
      volumeDevices: 1
        - name: data
          devicePath: /dev/xvda 2
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc 3

```

- 1** **volumeDevices**, instead of **volumeMounts**, is used for block devices. Only **PersistentVolumeClaim** sources can be used with raw block volumes.

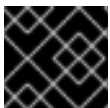
- 2 **devicePath**, instead of **mountPath**, represents the path to the physical device where the raw block is mapped to the system.
- 3 The volume source must be of type **persistentVolumeClaim** and must match the name of the PVC as expected.

Table 2.5. Accepted values for **volumeMode**

Value	Default
Filesystem	Yes
Block	No

Table 2.6. Binding scenarios for block volumes

PV volumeMode	PVC volumeMode	Binding result
Filesystem	Filesystem	Bind
Unspecified	Unspecified	Bind
Filesystem	Unspecified	Bind
Unspecified	Filesystem	Bind
Block	Block	Bind
Unspecified	Block	No Bind
Block	Unspecified	No Bind
Filesystem	Block	No Bind
Block	Filesystem	No Bind

**IMPORTANT**

Unspecified values result in the default value of **Filesystem**.

CHAPTER 3. CONFIGURING PERSISTENT STORAGE

3.1. PERSISTENT STORAGE USING AWS ELASTIC FILE SYSTEM

OpenShift Container Platform allows use of Amazon Web Services (AWS) Elastic File System volumes (EFS). You can provision your OpenShift Container Platform cluster with persistent storage using AWS EC2. Some familiarity with Kubernetes and AWS is assumed.



IMPORTANT

Elastic File System is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see <https://access.redhat.com/support/offerings/techpreview/>.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure. AWS Elastic File System volumes can be provisioned dynamically. Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.

3.1.1. Prerequisites

- Configure the AWS security groups to allow inbound NFS traffic from the EFS volume's security group.
- Configure the AWS EFS volume to allow incoming SSH traffic from any host.

Additional resources

- [Amazon EFS](#)
- [Amazon security groups for EFS](#)

3.1.2. Store the EFS variables in a config map

It is recommended to use a config map to contain all the environment variables that are required for the EFS provisioner.

Procedure

1. Define an OpenShift Container Platform **ConfigMap** object that contains the environment variables by creating a **configmap.yaml** file that contains following contents:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: efs-provisioner
```



```

data:
  file.system.id: <file-system-id> ❶
  aws.region: <aws-region> ❷
  provisioner.name: openshift.org/aws-efs ❸
  dns.name: "" ❹

```

- ❶ Defines the Amazon Web Services (AWS) EFS file system ID.
- ❷ The AWS region of the EFS file system, such as **us-east-1**.
- ❸ The name of the provisioner for the associated storage class.
- ❹ An optional argument that specifies the new DNS name where the EFS volume is located. If no DNS name is provided, the provisioner will search for the EFS volume at **<file-system-id>.efs.<aws-region>.amazonaws.com**.

2. After the file has been configured, create it in your cluster by running the following command:

```
$ oc create -f configmap.yaml -n <namespace>
```

3.1.3. Configuring authorization for EFS volumes

The EFS provisioner must be authorized to communicate to the AWS endpoints, along with observing and updating OpenShift Container Platform storage resources. The following instructions create the necessary permissions for the EFS provisioner.

Procedure

1. Create an **efs-provisioner** service account:

```
$ oc create serviceaccount efs-provisioner
```

2. Create a file, **clusterrole.yaml** that defines the necessary permissions:

```

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: efs-provisioner-runner
rules:
- apiGroups: [""]
  resources: ["persistentvolumes"]
  verbs: ["get", "list", "watch", "create", "delete"]
- apiGroups: [""]
  resources: ["persistentvolumeclaims"]
  verbs: ["get", "list", "watch", "update"]
- apiGroups: ["storage.k8s.io"]
  resources: ["storageclasses"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["events"]
  verbs: ["create", "update", "patch"]
- apiGroups: ["security.openshift.io"]

```

```
resources: ["securitycontextconstraints"]
verbs: ["use"]
resourceNames: ["hostmount-anyuid"]
```

3. Create a file, **clusterrolebinding.yaml**, that defines a cluster role binding that assigns the defined role to the service account:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-efs-provisioner
subjects:
  - kind: ServiceAccount
    name: efs-provisioner
    namespace: default 1
roleRef:
  kind: ClusterRole
  name: efs-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
```

- 1** The namespace where the EFS provisioner pod will run. If the EFS provisioner is running in a namespace other than **default**, this value must be updated.

4. Create a file, **role.yaml**, that defines a role with the necessary permissions:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-efs-provisioner
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
```

5. Create a file, **rolebinding.yaml**, that defines a role binding that assigns this role to the service account:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-efs-provisioner
subjects:
  - kind: ServiceAccount
    name: efs-provisioner
    namespace: default 1
roleRef:
  kind: Role
  name: leader-locking-efs-provisioner
  apiGroup: rbac.authorization.k8s.io
```

- 1** The namespace where the EFS provisioner pod will run. If the EFS provisioner is running in a namespace other than **default**, this value must be updated.

6. Create the resources inside the OpenShift Container Platform cluster:

```
$ oc create -f clusterrole.yaml,clusterrolebinding.yaml,role.yaml,rolebinding.yaml
```

3.1.4. Create the EFS storage class

Before persistent volume claims can be created, a storage class must exist in the OpenShift Container Platform cluster. The following instructions create the storage class for the EFS provisioner.

Procedure

1. Define an OpenShift Container Platform config map that contains the environment variables by creating a **storageclass.yaml** with the following contents:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: aws-efs
provisioner: openshift.org/aws-efs
parameters:
  gidMin: "2048" 1
  gidMax: "2147483647" 2
  gidAllocate: "true" 3
```

- 1 An optional argument that defines the minimum group ID (GID) for volume assignments. The default value is **2048**.
- 2 An optional argument that defines the maximum GID for volume assignments. The default value is **2147483647**.
- 3 An optional argument that determines if GIDs are assigned to volumes. If **false**, dynamically provisioned volumes are not allocated GIDs, allowing all users to read and write to the created volumes. The default value is **true**.

2. After the file has been configured, create it in your cluster by running the following command:

```
$ oc create -f storageclass.yaml
```

3.1.5. Create the EFS provisioner

The EFS provisioner is an OpenShift Container Platform pod that mounts the EFS volume as an NFS share.

Prerequisites

- Create A config map that defines the EFS environment variables.
- Create a service account that contains the necessary cluster and role permissions.
- Create a storage class for provisioning volumes.
- Configure the Amazon Web Services (AWS) security groups to allow incoming NFS traffic on all OpenShift Container Platform nodes.

- Configure the AWS EFS volume security groups to allow incoming SSH traffic from all sources.

Procedure

1. Define the EFS provisioner by creating a **provisioner.yaml** with the following contents:

```
kind: Pod
apiVersion: v1
metadata:
  name: efs-provisioner
spec:
  serviceAccount: efs-provisioner
  containers:
  - name: efs-provisioner
    image: quay.io/external_storage/efs-provisioner:latest
    env:
    - name: PROVISIONER_NAME
      valueFrom:
        configMapKeyRef:
          name: efs-provisioner
          key: provisioner.name
    - name: FILE_SYSTEM_ID
      valueFrom:
        configMapKeyRef:
          name: efs-provisioner
          key: file.system.id
    - name: AWS_REGION
      valueFrom:
        configMapKeyRef:
          name: efs-provisioner
          key: aws.region
    - name: DNS_NAME
      valueFrom:
        configMapKeyRef:
          name: efs-provisioner
          key: dns.name
          optional: true
    volumeMounts:
    - name: pv-volume
      mountPath: /persistentvolumes
  volumes:
  - name: pv-volume
    nfs:
      server: <file-system-id>.efs.<region>.amazonaws.com 1
      path: / 2
```

1 Contains the DNS name of the EFS volume. This field must be updated for the pod to discover the EFS volume.

2 The mount path of the EFS volume. Each persistent volume is created as a separate subdirectory on the EFS volume. If this EFS volume is used for other projects outside of OpenShift Container Platform, then it is recommended to create a unique subdirectory OpenShift Container Platform manually on EFS for the cluster to prevent projects from accessing another project's data. Specifying a directory that does not exist results in an error.

2. After the file has been configured, create it in your cluster by running the following command:

```
$ oc create -f provisioner.yaml
```

3.1.6. Create the EFS persistent volume claim

EFS persistent volume claims are created to allow pods to mount the underlying EFS storage.

Prerequisites

- Create the EFS provisioner pod.

Procedure (UI)

1. In the OpenShift Container Platform console, click **Storage** → **Persistent Volume Claims**
2. In the persistent volume claims overview, click **Create Persistent Volume Claim**
3. Define the required options on the resulting page.
 - a. Select the storage class that you created from the list.
 - b. Enter a unique name for the storage claim.
 - c. Select the access mode to determine the read and write access for the created storage claim.
 - d. Define the size of the storage claim.



NOTE

Although you must enter a size, every pod that access the EFS volume has unlimited storage. Define a value, such as **1Mi**, that will remind you that the storage size is unlimited.

4. Click **Create** to create the persistent volume claim and generate a persistent volume.

Procedure (CLI)

1. Alternately, you can define EFS persistent volume claims by creating a file, **pvc.yaml**, with the following contents:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: efs-claim 1
  namespace: test-efs
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: openshift.org/aws-efs
  finalizers:
    - kubernetes.io/pvc-protection
spec:
  accessModes:
    - ReadWriteOnce 2
```

```
resources:
  requests:
    storage: 5Gi 3
  storageClassName: aws-efs 4
  volumeMode: Filesystem
```

- 1** A unique name for the PVC.
- 2** The access mode to determine the read and write access for the created PVC.
- 3** Defines the size of the PVC.
- 4** Name of the storage class for the EFS provisioner.

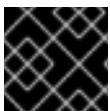
2. After the file has been configured, create it in your cluster by running the following command:

```
$ oc create -f pvc.yaml
```

3.2. PERSISTENT STORAGE USING AWS ELASTIC BLOCK STORE

OpenShift Container Platform supports AWS Elastic Block Store volumes (EBS). You can provision your OpenShift Container Platform cluster with persistent storage using AWS EC2. Some familiarity with Kubernetes and AWS is assumed.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure. AWS Elastic Block Store volumes can be provisioned dynamically. Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.



IMPORTANT

High-availability of storage in the infrastructure is left to the underlying storage provider.

Additional resources

- [Amazon EC2](#)

3.2.1. Creating the EBS storage class

Storage classes are used to differentiate and delineate storage levels and usages. By defining a storage class, users can obtain dynamically provisioned persistent volumes.

Procedure

1. In the OpenShift Container Platform console, click **Storage → Storage Classes**.
2. In the storage class overview, click **Create Storage Class**.
3. Define the desired options on the page that appears.
 - a. Enter a name to reference the storage class.

- b. Enter an optional description.
 - c. Select the reclaim policy.
 - d. Select **kubernetes.io/aws-ebs** from the drop down list.
 - e. Enter additional parameters for the storage class as desired.
4. Click **Create** to create the storage class.

3.2.2. Creating the persistent volume claim

Prerequisites

Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Procedure

1. In the OpenShift Container Platform console, click **Storage → Persistent Volume Claims**
2. In the persistent volume claims overview, click **Create Persistent Volume Claim**
3. Define the desired options on the page that appears.
 - a. Select the storage class created previously from the drop-down menu.
 - b. Enter a unique name for the storage claim.
 - c. Select the access mode. This determines the read and write access for the created storage claim.
 - d. Define the size of the storage claim.
4. Click **Create** to create the persistent volume claim and generate a persistent volume.

3.2.3. Volume format

Before OpenShift Container Platform mounts the volume and passes it to a container, it checks that it contains a file system as specified by the **fsType** parameter in the persistent volume definition. If the device is not formatted with the file system, all data from the device is erased and the device is automatically formatted with the given file system.

This allows using unformatted AWS volumes as persistent volumes, because OpenShift Container Platform formats them before the first use.

3.2.4. Maximum number of EBS volumes on a node

By default, OpenShift Container Platform supports a maximum of 39 EBS volumes attached to one node. This limit is consistent with the [AWS volume limits](#). The volume limit depends on the instance type.

3.3. PERSISTENT STORAGE USING AZURE

OpenShift Container Platform supports Microsoft Azure Disk volumes. You can provision your OpenShift Container Platform cluster with persistent storage using Azure. Some familiarity with

Kubernetes and Azure is assumed. The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure. Azure Disk volumes can be provisioned dynamically. Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.



IMPORTANT

High availability of storage in the infrastructure is left to the underlying storage provider.

Additional resources

- [Microsoft Azure Disk](#)

3.3.1. Creating the Azure storage class

Storage classes are used to differentiate and delineate storage levels and usages. By defining a storage class, users can obtain dynamically provisioned persistent volumes.

Procedure

1. In the OpenShift Container Platform console, click **Storage** → **Storage Classes**.
2. In the storage class overview, click **Create Storage Class**
3. Define the desired options on the page that appears.
 - a. Enter a name to reference the storage class.
 - b. Enter an optional description.
 - c. Select the reclaim policy.
 - d. Select **kubernetes.io/azure-disk** from the drop down list.
 - i. Enter the storage account type. This corresponds to your Azure storage account SKU tier. Valid options are **Premium_LRS**, **Standard_LRS**, **StandardSSD_LRS**, and **UltraSSD_LRS**.
 - ii. Enter the kind of account. Valid options are **shared**, **dedicated**, and **managed**.



IMPORTANT

Red Hat only supports the use of **kind: Managed** in the storage class.

With **Shared** and **Dedicated**, Azure creates unmanaged disks, while OpenShift Container Platform creates a managed disk for machine OS (root) disks. But because Azure Disk does not allow the use of both managed and unmanaged disks on a node, unmanaged disks created with **Shared** or **Dedicated** cannot be attached to OpenShift Container Platform nodes.

- e. Enter additional parameters for the storage class as desired.

4. Click **Create** to create the storage class.

Additional resources

- [Azure Disk Storage Class](#)

3.3.2. Creating the persistent volume claim

Prerequisites

Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Procedure

1. In the OpenShift Container Platform console, click **Storage → Persistent Volume Claims**
2. In the persistent volume claims overview, click **Create Persistent Volume Claim**
3. Define the desired options on the page that appears.
 - a. Select the storage class created previously from the drop-down menu.
 - b. Enter a unique name for the storage claim.
 - c. Select the access mode. This determines the read and write access for the created storage claim.
 - d. Define the size of the storage claim.
4. Click **Create** to create the persistent volume claim and generate a persistent volume.

3.3.3. Volume format

Before OpenShift Container Platform mounts the volume and passes it to a container, it checks that it contains a file system as specified by the **fsType** parameter in the persistent volume definition. If the device is not formatted with the file system, all data from the device is erased and the device is automatically formatted with the given file system.

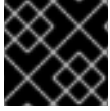
This allows using unformatted Azure volumes as persistent volumes, because OpenShift Container Platform formats them before the first use.

3.4. PERSISTENT STORAGE USING AZURE FILE

OpenShift Container Platform supports Microsoft Azure File volumes. You can provision your OpenShift Container Platform cluster with persistent storage using Azure. Some familiarity with Kubernetes and Azure is assumed.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure. Azure File volumes can be provisioned dynamically.

Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users for use in applications.



IMPORTANT

High availability of storage in the infrastructure is left to the underlying storage provider.

Additional resources

- [Azure Files](#)

3.4.1. Create the Azure File share persistent volume claim

To create the persistent volume claim, you must first define a **Secret** object that contains the Azure account and key. This secret is used in the **PersistentVolume** definition, and will be referenced by the persistent volume claim for use in applications.

Prerequisites

- An Azure File share exists.
- The credentials to access this share, specifically the storage account and key, are available.

Procedure

1. Create a **Secret** object that contains the Azure File credentials:

```
$ oc create secret generic <secret-name> --from-literal=azurestorageaccountname=
<storage-account> \ 1
--from-literal=azurestorageaccountkey=<storage-account-key> 2
```

- 1** The Azure File storage account name.
- 2** The Azure File storage account key.

2. Create a **PersistentVolume** object that references the **Secret** object you created:

```
apiVersion: "v1"
kind: "PersistentVolume"
metadata:
  name: "pv0001" 1
spec:
  capacity:
    storage: "5Gi" 2
  accessModes:
    - "ReadWriteOnce"
  storageClassName: azure-file-sc
  azureFile:
    secretName: <secret-name> 3
    shareName: share-1 4
    readOnly: false
```

- 1** The name of the persistent volume.
- 2** The size of this persistent volume.

- 3 The name of the secret that contains the Azure File share credentials.
- 4 The name of the Azure File share.

3. Create a **PersistentVolumeClaim** object that maps to the persistent volume you created:

```

apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1" 1
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "5Gi" 2
  storageClassName: azure-file-sc 3
  volumeName: "pv0001" 4

```

- 1 The name of the persistent volume claim.
- 2 The size of this persistent volume claim.
- 3 The name of the storage class that is used to provision the persistent volume. Specify the storage class used in the **PersistentVolume** definition.
- 4 The name of the existing **PersistentVolume** object that references the Azure File share.

3.4.2. Mount the Azure File share in a pod

After the persistent volume claim has been created, it can be used inside by an application. The following example demonstrates mounting this share inside of a pod.

Prerequisites

- A persistent volume claim exists that is mapped to the underlying Azure File share.

Procedure

- Create a pod that mounts the existing persistent volume claim:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-name 1
spec:
  containers:
    ...
  volumeMounts:
    - mountPath: "/data" 2
      name: azure-file-share
  volumes:

```

```
- name: azure-file-share
  persistentVolumeClaim:
    claimName: claim1 3
```

- 1** The name of the pod.
- 2** The path to mount the Azure File share inside the pod.
- 3** The name of the **PersistentVolumeClaim** object that has been previously created.

3.5. PERSISTENT STORAGE USING CINDER

OpenShift Container Platform supports OpenStack Cinder. Some familiarity with Kubernetes and OpenStack is assumed.

Cinder volumes can be provisioned dynamically. Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.

Additional resources

- For more information about how OpenStack Block Storage provides persistent block storage management for virtual hard drives, see [OpenStack Cinder](#).

3.5.1. Manual provisioning with Cinder

Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Prerequisites

- OpenShift Container Platform configured for Red Hat OpenStack Platform (RHOSP)
- Cinder volume ID

3.5.1.1. Creating the persistent volume

You must define your persistent volume (PV) in an object definition before creating it in OpenShift Container Platform:

Procedure

1. Save your object definition to a file.

cinder-persistentvolume.yaml

```
apiVersion: "v1"
kind: "PersistentVolume"
metadata:
  name: "pv0001" 1
spec:
  capacity:
    storage: "5Gi" 2
```

```

accessModes:
  - "ReadWriteOnce"
cinder: ❸
fsType: "ext3" ❹
volumeID: "f37a03aa-6212-4c62-a805-9ce139fab180" ❺

```

- ❶ The name of the volume that is used by persistent volume claims or pods.
- ❷ The amount of storage allocated to this volume.
- ❸ Indicates **cinder** for Red Hat OpenStack Platform (RHOSP) Cinder volumes.
- ❹ The file system that is created when the volume is mounted for the first time.
- ❺ The Cinder volume to use.



IMPORTANT

Do not change the **fstype** parameter value after the volume is formatted and provisioned. Changing this value can result in data loss and pod failure.

2. Create the object definition file you saved in the previous step.

```
$ oc create -f cinder-persistentvolume.yaml
```

3.5.1.2. Persistent volume formatting

You can use unformatted Cinder volumes as PVs because OpenShift Container Platform formats them before the first use.

Before OpenShift Container Platform mounts the volume and passes it to a container, the system checks that it contains a file system as specified by the **fsType** parameter in the PV definition. If the device is not formatted with the file system, all data from the device is erased and the device is automatically formatted with the given file system.

3.5.1.3. Cinder volume security

If you use Cinder PVs in your application, configure security for their deployment configurations.

Prerequisite

- An SCC must be created that uses the appropriate **fsGroup** strategy.

Procedure

1. Create a service account and add it to the SCC:

```

$ oc create serviceaccount <service_account>
$ oc adm policy add-scc-to-user <new_scc> -z <service_account> -n <project>

```

2. In your application's deployment configuration, provide the service account name and **securityContext**:

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1 1
  selector: 2
    name: frontend
  template: 3
    metadata:
      labels: 4
        name: frontend 5
    spec:
      containers:
        - image: openshift/hello-openshift
          name: helloworld
          ports:
            - containerPort: 8080
              protocol: TCP
          restartPolicy: Always
          serviceAccountName: <service_account> 6
          securityContext:
            fsGroup: 7777 7

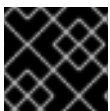
```

- 1** The number of copies of the pod to run.
- 2** The label selector of the pod to run.
- 3** A template for the pod that the controller creates.
- 4** The labels on the pod. They must include labels from the label selector.
- 5** The maximum name length after expanding any parameters is 63 characters.
- 6** Specifies the service account you created.
- 7** Specifies an **fsGroup** for the pods.

3.6. PERSISTENT STORAGE USING FIBRE CHANNEL

OpenShift Container Platform supports Fibre Channel, allowing you to provision your OpenShift Container Platform cluster with persistent storage using Fibre channel volumes. Some familiarity with Kubernetes and Fibre Channel is assumed.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure. Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.



IMPORTANT

High availability of storage in the infrastructure is left to the underlying storage provider.

Additional resources

- [Using Fibre Channel devices](#)

3.6.1. Provisioning

To provision Fibre Channel volumes using the **PersistentVolume** API the following must be available:

- The **targetWWNs** (array of Fibre Channel target's World Wide Names).
- A valid LUN number.
- The filesystem type.

A persistent volume and a LUN have a one-to-one mapping between them.

Prerequisites

- Fibre Channel LUNs must exist in the underlying infrastructure.

PersistentVolume object definition

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  fc:
    targetWWNs: ['500a0981891b8dc5', '500a0981991b8dc5'] 1
    lun: 2
    fsType: ext4
```

- 1 Fibre Channel WWNs are identified as `/dev/disk/by-path/pci-<IDENTIFIER>-fc-0x<WWN>-lun-<LUN#>`, but you do not need to provide any part of the path leading up to the **WWN**, including the **0x**, and anything after, including the **-** (hyphen).



IMPORTANT

Changing the value of the **fstype** parameter after the volume has been formatted and provisioned can result in data loss and pod failure.

3.6.1.1. Enforcing disk quotas

Use LUN partitions to enforce disk quotas and size constraints. Each LUN is mapped to a single persistent volume, and unique names must be used for persistent volumes.

Enforcing quotas in this way allows the end user to request persistent storage by a specific amount, such as 10Gi, and be matched with a corresponding volume of equal or greater capacity.

3.6.1.2. Fibre Channel volume security

Users request storage with a persistent volume claim. This claim only lives in the user's namespace, and can only be referenced by a pod within that same namespace. Any attempt to access a persistent volume across a namespace causes the pod to fail.

Each Fibre Channel LUN must be accessible by all nodes in the cluster.

3.7. PERSISTENT STORAGE USING FLEXVOLUME

OpenShift Container Platform supports FlexVolume, an out-of-tree plug-in that uses an executable model to interface with drivers.

To use storage from a back-end that does not have a built-in plug-in, you can extend OpenShift Container Platform through FlexVolume drivers and provide persistent storage to applications.

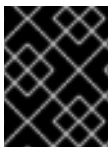
Pods interact with FlexVolume drivers through the **flexvolume** in-tree plugin.

Additional resources

- [Expanding persistent volumes](#)

3.7.1. About FlexVolume drivers

A FlexVolume driver is an executable file that resides in a well-defined directory on all nodes in the cluster. OpenShift Container Platform calls the FlexVolume driver whenever it needs to mount or unmount a volume represented by a **PersistentVolume** object with **flexVolume** as the source.



IMPORTANT

Attach and detach operations are not supported in OpenShift Container Platform for FlexVolume.

3.7.2. FlexVolume driver example

The first command-line argument of the FlexVolume driver is always an operation name. Other parameters are specific to each operation. Most of the operations take a JavaScript Object Notation (JSON) string as a parameter. This parameter is a complete JSON string, and not the name of a file with the JSON data.

The FlexVolume driver contains:

- All **flexVolume.options**.
- Some options from **flexVolume** prefixed by **kubernetes.io/**, such as **fsType** and **readwrite**.
- The content of the referenced secret, if specified, prefixed by **kubernetes.io/secret/**.

FlexVolume driver JSON input example

```
{
  "fooServer": "192.168.0.1:1234", 1
  "fooVolumeName": "bar",
  "kubernetes.io/fsType": "ext4", 2
}
```



```
"kubernetes.io/readwrite": "ro", 3
"kubernetes.io/secret/<key name>": "<key value>", 4
"kubernetes.io/secret/<another key name>": "<another key value>",
}
```

- 1** All options from **flexVolume.options**.
- 2** The value of **flexVolume.fsType**.
- 3** **ro/rw** based on **flexVolume.readOnly**.
- 4** All keys and their values from the secret referenced by **flexVolume.secretRef**.

OpenShift Container Platform expects JSON data on standard output of the driver. When not specified, the output describes the result of the operation.

FlexVolume driver default output example

```
{
  "status": "<Success/Failure/Not supported>",
  "message": "<Reason for success/failure>"
}
```

Exit code of the driver should be **0** for success and **1** for error.

Operations should be idempotent, which means that the mounting of an already mounted volume should result in a successful operation.

3.7.3. Installing FlexVolume drivers

FlexVolume drivers that are used to extend OpenShift Container Platform are executed only on the node. To implement FlexVolumes, a list of operations to call and the installation path are all that is required.

Prerequisites

- FlexVolume drivers must implement these operations:

init

Initializes the driver. It is called during initialization of all nodes.

- Arguments: none
- Executed on: node
- Expected output: default JSON

mount

Mounts a volume to directory. This can include anything that is necessary to mount the volume, including finding the device and then mounting the device.

- Arguments: **<mount-dir>** **<json>**
- Executed on: node

- Expected output: default JSON

unmount

Unmounts a volume from a directory. This can include anything that is necessary to clean up the volume after unmounting.

- Arguments: **<mount-dir>**
- Executed on: node
- Expected output: default JSON

mountdevice

Mounts a volume's device to a directory where individual pods can then bind mount.

This call-out does not pass "secrets" specified in the FlexVolume spec. If your driver requires secrets, do not implement this call-out.

- Arguments: **<mount-dir> <json>**
- Executed on: node
- Expected output: default JSON

unmountdevice

Unmounts a volume's device from a directory.

- Arguments: **<mount-dir>**
- Executed on: node
- Expected output: default JSON
 - All other operations should return JSON with **{"status": "Not supported"}** and exit code **1**.

Procedure

To install the FlexVolume driver:

1. Ensure that the executable file exists on all nodes in the cluster.
2. Place the executable file at the volume plug-in path: **/etc/kubernetes/kubelet-plugins/volume/exec/<vendor>~<driver>/<driver>**.

For example, to install the FlexVolume driver for the storage **foo**, place the executable file at: **/etc/kubernetes/kubelet-plugins/volume/exec/openshift.com~foo/foo**.

3.7.4. Consuming storage using FlexVolume drivers

Each **PersistentVolume** object in OpenShift Container Platform represents one storage asset in the storage back-end, such as a volume.

Procedure

- Use the **PersistentVolume** object to reference the installed storage.

Persistent volume object definition using FlexVolume drivers example

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001 ❶
spec:
  capacity:
    storage: 1Gi ❷
  accessModes:
    - ReadWriteOnce
  flexVolume:
    driver: openshift.com/foo ❸
    fsType: "ext4" ❹
    secretRef: foo-secret ❺
    readOnly: true ❻
    options: ❼
      fooServer: 192.168.0.1:1234
      fooVolumeName: bar

```

- ❶ The name of the volume. This is how it is identified through persistent volume claims or from pods. This name can be different from the name of the volume on back-end storage.
- ❷ The amount of storage allocated to this volume.
- ❸ The name of the driver. This field is mandatory.
- ❹ The file system that is present on the volume. This field is optional.
- ❺ The reference to a secret. Keys and values from this secret are provided to the FlexVolume driver on invocation. This field is optional.
- ❻ The read-only flag. This field is optional.
- ❼ The additional options for the FlexVolume driver. In addition to the flags specified by the user in the **options** field, the following flags are also passed to the executable:

```

"fsType": "<FS type>",
"readwrite": "<rw>",
"secret/key1": "<secret1>"
...
"secret/keyN": "<secretN>"

```



NOTE

Secrets are passed only to mount or unmount call-outs.

3.8. PERSISTENT STORAGE USING GCE PERSISTENT DISK

OpenShift Container Platform supports GCE Persistent Disk volumes (gcePD). You can provision your OpenShift Container Platform cluster with persistent storage using GCE. Some familiarity with Kubernetes and GCE is assumed.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure.

GCE Persistent Disk volumes can be provisioned dynamically.

Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.



IMPORTANT

High availability of storage in the infrastructure is left to the underlying storage provider.

Additional resources

- [GCE Persistent Disk](#)

3.8.1. Creating the GCE storage class

Storage classes are used to differentiate and delineate storage levels and usages. By defining a storage class, users can obtain dynamically provisioned persistent volumes.

Procedure

1. In the OpenShift Container Platform console, click **Storage → Storage Classes**.
2. In the storage class overview, click **Create Storage Class**.
3. Define the desired options on the page that appears.
 - a. Enter a name to reference the storage class.
 - b. Enter an optional description.
 - c. Select the reclaim policy.
 - d. Select **kubernetes.io/gce-pd** from the drop down list.
 - e. Enter additional parameters for the storage class as desired.
4. Click **Create** to create the storage class.

3.8.2. Creating the persistent volume claim

Prerequisites

Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Procedure

1. In the OpenShift Container Platform console, click **Storage → Persistent Volume Claims**.
2. In the persistent volume claims overview, click **Create Persistent Volume Claim**.

3. Define the desired options on the page that appears.
 - a. Select the storage class created previously from the drop-down menu.
 - b. Enter a unique name for the storage claim.
 - c. Select the access mode. This determines the read and write access for the created storage claim.
 - d. Define the size of the storage claim.
4. Click **Create** to create the persistent volume claim and generate a persistent volume.

3.8.3. Volume format

Before OpenShift Container Platform mounts the volume and passes it to a container, it checks that it contains a file system as specified by the **fsType** parameter in the persistent volume definition. If the device is not formatted with the file system, all data from the device is erased and the device is automatically formatted with the given file system.

This allows using unformatted GCE volumes as persistent volumes, because OpenShift Container Platform formats them before the first use.

3.9. PERSISTENT STORAGE USING HOSTPATH

A `hostPath` volume in an OpenShift Container Platform cluster mounts a file or directory from the host node's filesystem into your pod. Most pods will not need a `hostPath` volume, but it does offer a quick option for testing should an application require it.



IMPORTANT

The cluster administrator must configure pods to run as privileged. This grants access to pods in the same node.

3.9.1. Overview

OpenShift Container Platform supports `hostPath` mounting for development and testing on a single-node cluster.

In a production cluster, you would not use `hostPath`. Instead, a cluster administrator would provision a network resource, such as a GCE Persistent Disk volume, an NFS share, or an Amazon EBS volume. Network resources support the use of storage classes to set up dynamic provisioning.

A `hostPath` volume must be provisioned statically.

3.9.2. Statically provisioning `hostPath` volumes

A pod that uses a `hostPath` volume must be referenced by manual (static) provisioning.

Procedure

1. Define the persistent volume (PV). Create a file, **pv.yaml**, with the **PersistentVolume** object definition:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume 1
  labels:
    type: local
spec:
  storageClassName: manual 2
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce 3
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/mnt/data" 4

```

- 1** The name of the volume. This name is how it is identified by persistent volume claims or pods.
- 2** Used to bind persistent volume claim requests to this persistent volume.
- 3** The volume can be mounted as **read-write** by a single node.
- 4** The configuration file specifies that the volume is at **/mnt/data** on the cluster's node.

2. Create the PV from the file:

```
$ oc create -f pv.yaml
```

3. Define the persistent volume claim (PVC). Create a file, **pvc.yaml**, with the **PersistentVolumeClaim** object definition:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pvc-volume
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: manual

```

4. Create the PVC from the file:

```
$ oc create -f pvc.yaml
```

3.9.3. Mounting the hostPath share in a privileged pod

After the persistent volume claim has been created, it can be used inside by an application. The following example demonstrates mounting this share inside of a pod.

Prerequisites

- A persistent volume claim exists that is mapped to the underlying hostPath share.

Procedure

- Create a privileged pod that mounts the existing persistent volume claim:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-name ❶
spec:
  containers:
    ...
    securityContext:
      privileged: true ❷
  volumeMounts:
    - mountPath: /data ❸
      name: hostpath-privileged
    ...
  securityContext: {}
  volumes:
    - name: hostpath-privileged
      persistentVolumeClaim:
        claimName: task-pvc-volume ❹

```

- ❶ The name of the pod.
- ❷ The pod must run as privileged to access the node's storage.
- ❸ The path to mount the hostPath share inside the privileged pod.
- ❹ The name of the **PersistentVolumeClaim** object that has been previously created.

3.10. PERSISTENT STORAGE USING ISCSI

You can provision your OpenShift Container Platform cluster with persistent storage using [iSCSI](#). Some familiarity with Kubernetes and iSCSI is assumed.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure.



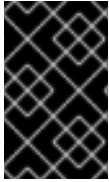
IMPORTANT

High-availability of storage in the infrastructure is left to the underlying storage provider.



IMPORTANT

When you use iSCSI on Amazon Web Services, you must update the default security policy to include TCP traffic between nodes on the iSCSI ports. By default, they are ports **860** and **3260**.



IMPORTANT

OpenShift assumes that all nodes in the cluster have already configured iSCSI initiator, i.e. have installed **iscsi-initiator-utils** package and configured their initiator name in **/etc/iscsi/initiatorname.iscsi**. See Storage Administration Guide linked above.

3.10.1. Provisioning

Verify that the storage exists in the underlying infrastructure before mounting it as a volume in OpenShift Container Platform. All that is required for the iSCSI is the iSCSI target portal, a valid iSCSI Qualified Name (IQN), a valid LUN number, the filesystem type, and the **PersistentVolume** API.

PersistentVolume object definition

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: iscsi-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  iscsi:
    targetPortal: 10.16.154.81:3260
    iqn: iqn.2014-12.example.server:storage.target00
    lun: 0
    fsType: 'ext4'
```

3.10.2. Enforcing disk quotas

Use LUN partitions to enforce disk quotas and size constraints. Each LUN is one persistent volume. Kubernetes enforces unique names for persistent volumes.

Enforcing quotas in this way allows the end user to request persistent storage by a specific amount (e.g, 10Gi) and be matched with a corresponding volume of equal or greater capacity.

3.10.3. iSCSI volume security

Users request storage with a **PersistentVolumeClaim** object. This claim only lives in the user's namespace and can only be referenced by a pod within that same namespace. Any attempt to access a persistent volume claim across a namespace causes the pod to fail.

Each iSCSI LUN must be accessible by all nodes in the cluster.

3.10.3.1. Challenge Handshake Authentication Protocol (CHAP) configuration

Optionally, OpenShift can use CHAP to authenticate itself to iSCSI targets:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: iscsi-pv
spec:
```



```

capacity:
  storage: 1Gi
accessModes:
  - ReadWriteOnce
iscsi:
  targetPortal: 10.0.0.1:3260
  iqn: iqn.2016-04.test.com:storage.target00
  lun: 0
  fsType: ext4
  chapAuthDiscovery: true 1
  chapAuthSession: true 2
  secretRef:
    name: chap-secret 3

```

- 1** Enable CHAP authentication of iSCSI discovery.
- 2** Enable CHAP authentication of iSCSI session.
- 3** Specify name of Secrets object with user name + password. This **Secret** object must be available in all namespaces that can use the referenced volume.

3.10.4. iSCSI multipathing

For iSCSI-based storage, you can configure multiple paths by using the same IQN for more than one target portal IP address. Multipathing ensures access to the persistent volume when one or more of the components in a path fail.

To specify multi-paths in the pod specification use the **portals** field. For example:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: iscsi-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  iscsi:
    targetPortal: 10.0.0.1:3260
    portals: ['10.0.2.16:3260', '10.0.2.17:3260', '10.0.2.18:3260'] 1
    iqn: iqn.2016-04.test.com:storage.target00
    lun: 0
    fsType: ext4
    readOnly: false

```

- 1** Add additional target portals using the **portals** field.

3.10.5. iSCSI custom initiator IQN

Configure the custom initiator iSCSI Qualified Name (IQN) if the iSCSI targets are restricted to certain IQNs, but the nodes that the iSCSI PVs are attached to are not guaranteed to have these IQNs.

To specify a custom initiator IQN, use **initiatorName** field.

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: iscsi-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  iscsi:
    targetPortal: 10.0.0.1:3260
    portals: ['10.0.2.16:3260', '10.0.2.17:3260', '10.0.2.18:3260']
    iqn: iqn.2016-04.test.com:storage.target00
    lun: 0
    initiatorName: iqn.2016-04.test.com:custom.iqn 1
    fsType: ext4
    readOnly: false

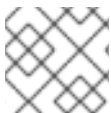
```

1 Specify the name of the initiator.

3.11. PERSISTENT STORAGE USING LOCAL VOLUMES

OpenShift Container Platform can be provisioned with persistent storage by using local volumes. Local persistent volumes allow you to access local storage devices, such as a disk or partition, by using the standard PVC interface.

Local volumes can be used without manually scheduling Pods to nodes, because the system is aware of the volume node's constraints. However, local volumes are still subject to the availability of the underlying node and are not suitable for all applications.



NOTE

Local volumes can only be used as a statically created Persistent Volume.

3.11.1. Installing the Local Storage Operator

The Local Storage Operator is not installed in OpenShift Container Platform by default. Use the following procedure to install and configure this Operator to enable local volumes in your cluster.

Prerequisites

- Access to the OpenShift Container Platform web console or command-line interface (CLI).

Procedure

1. Create the **local-storage** project:

```
$ oc new-project local-storage
```

2. Optional: Allow local storage creation on infrastructure nodes.

You might want to use the Local Storage Operator to create volumes on infrastructure nodes in support of components such as logging and monitoring.

You must adjust the default node selector so that the Local Storage Operator includes the infrastructure nodes, and not just worker nodes.

To block the Local Storage Operator from inheriting the cluster-wide default selector, enter the following command:

```
$ oc annotate project local-storage openshift.io/node-selector=""
```

From the UI

To install the Local Storage Operator from the web console, follow these steps:

1. Log in to the OpenShift Container Platform web console.
2. Navigate to **Operators** → **OperatorHub**.
3. Type **Local Storage** into the filter box to locate the Local Storage Operator.
4. Click **Install**.
5. On the **Create Operator Subscription** page, select **A specific namespace on the cluster**. Select **local-storage** from the drop-down menu.
6. Adjust the values for **Update Channel** and **Approval Strategy** to the values that you want.
7. Click **Subscribe**.

Once finished, the Local Storage Operator will be listed in the **Installed Operators** section of the web console.

From the CLI

1. Install the Local Storage Operator from the CLI.
 - a. Create an object YAML file to define an Operator group and subscription for the Local Storage Operator, such as **local-storage.yaml**:

Example local-storage

```
apiVersion: operators.coreos.com/v1alpha2
kind: OperatorGroup
metadata:
  name: local-operator-group
  namespace: local-storage
spec:
  targetNamespaces:
  - local-storage
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: local-storage-operator
  namespace: local-storage
```

```
spec:
  channel: <channel_version> 1
  installPlanApproval: Automatic 2
  name: local-storage-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

- 1** This field can be edited to match your release selection of OpenShift Container Platform.
- 2** The user approval policy for an install plan.

2. Create the Local Storage Operator object by entering the following command:

```
$ oc apply -f local-storage.yaml
```

At this point, the Operator Lifecycle Manager (OLM) is now aware of the Local Storage Operator. A ClusterServiceVersion (CSV) for the Operator should appear in the target namespace, and APIs provided by the Operator should be available for creation.

3. Verify local storage installation by checking that all pods and the Local Storage Operator have been created:
 - a. Check that all the required pods have been created:

```
$ oc -n local-storage get pods
NAME                                READY STATUS RESTARTS AGE
local-storage-operator-746bf599c9-vlt5t 1/1   Running 0      19m
```

- b. Check the ClusterServiceVersion (CSV) YAML manifest to see that the Local Storage Operator is available in the **local-storage** project:

```
$ oc get csvs -n local-storage
NAME                                DISPLAY          VERSION          REPLACES          PHASE
local-storage-operator.4.2.26-202003230335 Local Storage    4.2.26-202003230335
Succeeded
```

After all checks have passed, the Local Storage Operator is installed successfully.

3.11.2. Provisioning the local volumes

Local volumes cannot be created by dynamic provisioning. Instead, persistent volumes must be created by the Local Storage Operator. This provisioner will look for any devices, both file system and block volumes, at the paths specified in defined resource.

Prerequisites

- The Local Storage Operator is installed.
- Local disks are attached to the OpenShift Container Platform nodes.

Procedure

1. Create the local volume resource. This must define the nodes and paths to the local volumes.



NOTE

Do not use different storage class names for the same device. Doing so will create multiple persistent volumes (PV)s.

Example: Filesystem

```

apiVersion: "local.storage.openshift.io/v1"
kind: "LocalVolume"
metadata:
  name: "local-disks"
  namespace: "local-storage" 1
spec:
  nodeSelector: 2
  nodeSelectorTerms:
    - matchExpressions:
      - key: kubernetes.io/hostname
        operator: In
        values:
          - ip-10-0-140-183
          - ip-10-0-158-139
          - ip-10-0-164-33
  storageClassDevices:
    - storageClassName: "local-sc"
      volumeMode: Filesystem 3
      fsType: xfs 4
      devicePaths: 5
        - /path/to/device 6

```

- 1 The namespace where the Local Storage Operator is installed.
- 2 Optional: A node selector containing a list of nodes where the local storage volumes are attached. This example uses the node host names, obtained from **oc get node**. If a value is not defined, then the Local Storage Operator will attempt to find matching disks on all available nodes.
- 3 The volume mode, either **Filesystem** or **Block**, defining the type of the local volumes.
- 4 The file system that is created when the local volume is mounted for the first time.
- 5 The path containing a list of local storage devices to choose from.
- 6 Replace this value with your actual local disks filepath to the LocalVolume resource, such as **/dev/xvdd**. PVs are created for these local disks when the provisioner is deployed successfully.

Example: Block

```

apiVersion: "local.storage.openshift.io/v1"
kind: "LocalVolume"
metadata:

```

```

name: "local-disks"
namespace: "local-storage" ❶
spec:
  nodeSelector: ❷
  nodeSelectorTerms:
  - matchExpressions:
    - key: kubernetes.io/hostname
      operator: In
      values:
      - ip-10-0-136-143
      - ip-10-0-140-255
      - ip-10-0-144-180
  storageClassDevices:
  - storageClassName: "localblock-sc"
    volumeMode: Block ❸
    devicePaths: ❹
    - /path/to/device ❺

```

- ❶ The namespace where the Local Storage Operator is installed.
- ❷ Optional: A node selector containing a list of nodes where the local storage volumes are attached. This example uses the node host names, obtained from **oc get node**. If a value is not defined, then the Local Storage Operator will attempt to find matching disks on all available nodes.
- ❸ The volume mode, either **Filesystem** or **Block**, defining the type of the local volumes.
- ❹ The path containing a list of local storage devices to choose from.
- ❺ Replace this value with your actual local disks filepath to the LocalVolume resource, such as **/dev/xvdg**. PVs are created for these local disks when the provisioner is deployed successfully.

2. Create the local volume resource in your OpenShift Container Platform cluster, specifying the file you just created:

```
$ oc create -f <local-volume>.yaml
```

3. Verify that the provisioner was created, and that the corresponding daemon sets were created:

```
$ oc get all -n local-storage
```

NAME	READY	STATUS	RESTARTS	AGE
pod/local-disks-local-provisioner-h97hj	1/1	Running	0	46m
pod/local-disks-local-provisioner-j4mnn	1/1	Running	0	46m
pod/local-disks-local-provisioner-kbdnx	1/1	Running	0	46m
pod/local-disks-local-diskmaker-ldldw	1/1	Running	0	46m
pod/local-disks-local-diskmaker-lvrv4	1/1	Running	0	46m
pod/local-disks-local-diskmaker-phxdq	1/1	Running	0	46m
pod/local-storage-operator-54564d9988-vxvhx	1/1	Running	0	47m

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/local-storage-operator	ClusterIP	172.30.49.90	<none>	60000/TCP	47m

```

NAME                                DESIRED CURRENT READY UP-TO-DATE
AVAILABLE NODE SELECTOR AGE
daemonset.apps/local-disks-local-provisioner 3    3    3    3    3    <none>
46m
daemonset.apps/local-disks-local-diskmaker 3    3    3    3    3    <none>
46m

NAME                                READY UP-TO-DATE AVAILABLE AGE
deployment.apps/local-storage-operator 1/1 1    1    47m

NAME                                DESIRED CURRENT READY AGE
replicaset.apps/local-storage-operator-54564d9988 1    1    1    47m

```

Note the desired and current number of daemon set processes. If the desired count is **0**, it indicates that the label selectors were invalid.

- Verify that the persistent volumes were created:

```

$ oc get pv

NAME                CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM
STORAGECLASS REASON AGE
local-pv-1cec77cf 100Gi RWO Delete Available local-sc 88m
local-pv-2ef7cd2a 100Gi RWO Delete Available local-sc
82m
local-pv-3fa1c73 100Gi RWO Delete Available local-sc 48m

```



IMPORTANT

Editing the **LocalVolume** object does not change the **fsType** or **volumeMode** of existing persistent volumes because doing so might result in a destructive operation.

3.11.3. Creating the local volume persistent volume claim

Local volumes must be statically created as a persistent volume claim (PVC) to be accessed by the pod.

Prerequisite

- Persistent volumes have been created using the local volume provisioner.

Procedure

- Create the PVC using the corresponding storage class:

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-pvc-name 1
spec:
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem 2
resources:

```

```

requests:
  storage: 100Gi 3
storageClassName: local-sc 4

```

- 1 Name of the PVC.
- 2 The type of the PVC. Defaults to **Filesystem**.
- 3 The amount of storage available to the PVC.
- 4 Name of the storage class required by the claim.

2. Create the PVC in the OpenShift Container Platform cluster, specifying the file you just created:

```
$ oc create -f <local-pvc>.yaml
```

3.11.4. Attach the local claim

After a local volume has been mapped to a PersistentVolumeClaim (PVC) it can be specified inside of a resource.

Prerequisites

- A PVC exists in the same namespace.

Procedure

1. Include the defined claim in the resource's Spec. The following example declares the PVC inside a Pod:

```

apiVersion: v1
kind: Pod
spec:
  ...
  containers:
    volumeMounts:
      - name: localpvc 1
        mountPath: "/data" 2
  volumes:
    - name: localpvc
      persistentVolumeClaim:
        claimName: localpvc 3

```

- 1 Name of the volume to mount.
- 2 Path inside the Pod where the volume is mounted.
- 3 Name of the existing PVC to use.

2. Create the resource in the OpenShift Container Platform cluster, specifying the file you just created:

-


```
$ oc create -f <local-pod>.yaml
```

3.11.5. Using tolerations with Local Storage Operator pods

Taints can be applied to nodes to prevent them from running general workloads. To allow the Local Storage Operator to use tainted nodes, you must add tolerations to the **Pod** or **DaemonSet** definition. This allows the created resources to run on these tainted nodes.

You apply tolerations to the Local Storage Operator pod through the **LocalVolume** resource and apply taints to a node through the node specification. A taint on a node instructs the node to repel all pods that do not tolerate the taint. Using a specific taint that is not on other pods ensures that the Local Storage Operator pod can also run on that node.



IMPORTANT

Taints and tolerations consist of a key, value, and effect. As an argument, it is expressed as **key=value:effect**. An operator allows you to leave one of these parameters empty.

Prerequisites

- The Local Storage Operator is installed.
- Local disks are attached to OpenShift Container Platform nodes with a taint.
- Tainted nodes are expected to provision local storage.

Procedure

To configure local volumes for scheduling on tainted nodes:

1. Modify the YAML file that defines the **Pod** and add the **LocalVolume** spec, as shown in the following example:

```
apiVersion: "local.storage.openshift.io/v1"
kind: "LocalVolume"
metadata:
  name: "local-disks"
  namespace: "local-storage"
spec:
  tolerations:
    - key: localstorage 1
      operator: Equal 2
      value: "localstorage" 3
  storageClassDevices:
    - storageClassName: "localblock-sc"
      volumeMode: Block 4
      devicePaths: 5
        - /dev/xvdg
```

- 1** Specify the key that you added to the node.
- 2** Specify the **Equal** operator to require the **key/value** parameters to match. If operator is **Exists**, the system checks that the key exists and ignores the value. If operator is **Equal**, then the key and value must match.

- 3 Specify the value **local** of the tainted node.
- 4 The volume mode, either **Filesystem** or **Block**, defining the type of the local volumes.
- 5 The path containing a list of local storage devices to choose from.

The defined tolerations will be passed to the resulting daemon sets, allowing the diskmaker and provisioner pods to be created for nodes that contain the specified taints.

3.11.6. Deleting the Local Storage Operator's resources

3.11.6.1. Removing a local volume

Occasionally, local volumes must be deleted. While removing the entry in the LocalVolume resource and deleting the PersistentVolume is typically enough, if you want to re-use the same device path or have it managed by a different storage class, then additional steps are needed.



WARNING

The following procedure involves accessing a node as the root user. Modifying the state of the node beyond the steps in this procedure could result in cluster instability.

Prerequisite

- The persistent volume must be in a **Released** or **Available** state.



WARNING

Deleting a persistent volume that is still in use can result in data loss or corruption.

Procedure

1. Edit the previously created local volume to remove any unwanted disks.
 - a. Edit the cluster resource:


```
$ oc edit localvolume <name> -n local-storage
```
 - b. Navigate to the lines under **devicePaths**, and delete any representing unwanted disks.
2. Delete any persistent volumes created.

```
$ oc delete pv <pv-name>
```

3. Delete any symlinks on the node.

a. Create a debug pod on the node:

```
$ oc debug node/<node-name>
```

b. Change your root directory to the host:

```
$ chroot /host
```

c. Navigate to the directory containing the local volume symlinks.

```
$ cd /mnt/local-storage/<sc-name> 1
```

1 The name of the storage class used to create the local volumes.

d. Delete the symlink belonging to the removed device.

```
$ rm <symlink>
```

3.11.6.2. Uninstalling the Local Storage Operator

To uninstall the Local Storage Operator, you must remove the Operator and all created resources in the **local-storage** project.



WARNING

Uninstalling the Local Storage Operator while local storage PVs are still in use is not recommended. While the PVs will remain after the Operator's removal, there might be indeterminate behavior if the Operator is uninstalled and reinstalled without removing the PVs and local storage resources.

Prerequisites

- Access to the OpenShift Container Platform web console.


Procedure

1. Delete any local volume resources in the project:

```
$ oc delete localvolume --all --all-namespaces
```

2. Uninstall the Local Storage Operator from the web console.

a. Log in to the OpenShift Container Platform web console.

- b. Navigate to **Operators** → **Installed Operators**.
 - c. Type **Local Storage** into the filter box to locate the Local Storage Operator.
 - d. Click the Options menu  at the end of the Local Storage Operator.
 - e. Click **Uninstall Operator**.
 - f. Click **Remove** in the window that appears.
3. The PVs created by the Local Storage Operator will remain in the cluster until deleted. Once these volumes are no longer in use, delete them by running the following command:

```
$ oc delete pv <pv-name>
```

4. Delete the **local-storage** project:

```
$ oc delete project local-storage
```

3.12. PERSISTENT STORAGE USING NFS

OpenShift Container Platform clusters can be provisioned with persistent storage using NFS. Persistent volumes (PVs) and persistent volume claims (PVCs) provide a convenient method for sharing a volume across a project. While the NFS-specific information contained in a PV definition could also be defined directly in a **Pod** definition, doing so does not create the volume as a distinct cluster resource, making the volume more susceptible to conflicts.

Additional resources

- [Network File System \(NFS\)](#)

3.12.1. Provisioning

Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform. To provision NFS volumes, a list of NFS servers and export paths are all that is required.

Procedure

1. Create an object definition for the PV:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001 ①
spec:
  capacity:
    storage: 5Gi ②
  accessModes:
    - ReadWriteOnce ③
  nfs: ④
```

```

path: /tmp 5
server: 172.17.0.2 6
persistentVolumeReclaimPolicy: Retain 7

```

- 1** The name of the volume. This is the PV identity in various **oc <command> pod** commands.
- 2** The amount of storage allocated to this volume.
- 3** Though this appears to be related to controlling access to the volume, it is actually used similarly to labels and used to match a PVC to a PV. Currently, no access rules are enforced based on the **accessModes**.
- 4** The volume type being used, in this case the **nfs** plug-in.
- 5** The path that is exported by the NFS server.
- 6** The host name or IP address of the NFS server.
- 7** The reclaim policy for the PV. This defines what happens to a volume when released.

**NOTE**

Each NFS volume must be mountable by all schedulable nodes in the cluster.

2. Verify that the PV was created:

```

$ oc get pv
NAME LABELS CAPACITY ACCESSMODES STATUS CLAIM REASON AGE
pv0001 <none> 5Gi RWO Available 31s

```

3. Create a persistent volume claim that binds to the new PV:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-claim1
spec:
  accessModes:
    - ReadWriteOnce 1
  resources:
    requests:
      storage: 5Gi 2

```

- 1** As mentioned above for PVs, the **accessModes** do not enforce security, but rather act as labels to match a PV to a PVC.
- 2** This claim looks for PVs offering **5Gi** or greater capacity.

4. Verify that the persistent volume claim was created:

```
$ oc get pvc
NAME          STATUS  VOLUME  CAPACITY  ACCESS MODES  STORAGECLASS  AGE
nfs-claim1    Bound  pv0001  5Gi       RWO           gp2           2m
```

3.12.2. Enforcing disk quotas

You can use disk partitions to enforce disk quotas and size constraints. Each partition can be its own export. Each export is one PV. OpenShift Container Platform enforces unique names for PVs, but the uniqueness of the NFS volume's server and path is up to the administrator.

Enforcing quotas in this way allows the developer to request persistent storage by a specific amount, such as 10Gi, and be matched with a corresponding volume of equal or greater capacity.

3.12.3. NFS volume security

This section covers NFS volume security, including matching permissions and SELinux considerations. The user is expected to understand the basics of POSIX permissions, process UIDs, supplemental groups, and SELinux.

Developers request NFS storage by referencing either a PVC by name or the NFS volume plug-in directly in the **volumes** section of their **Pod** definition.

The **/etc/exports** file on the NFS server contains the accessible NFS directories. The target NFS directory has POSIX owner and group IDs. The OpenShift Container Platform NFS plug-in mounts the container's NFS directory with the same POSIX ownership and permissions found on the exported NFS directory. However, the container is not run with its effective UID equal to the owner of the NFS mount, which is the desired behavior.

As an example, if the target NFS directory appears on the NFS server as:

```
$ ls -lZ /opt/nfs -d
drwxrws---. nfsnobody 5555 unconfined_u:object_r:usr_t:s0 /opt/nfs

$ id nfsnobody
uid=65534(nfsnobody) gid=65534(nfsnobody) groups=65534(nfsnobody)
```

Then the container must match SELinux labels, and either run with a UID of **65534**, the **nfsnobody** owner, or with **5555** in its supplemental groups in order to access the directory.

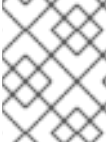


NOTE

The owner ID of **65534** is used as an example. Even though NFS's **root_squash** maps **root**, uid **0**, to **nfsnobody**, uid **65534**, NFS exports can have arbitrary owner IDs. Owner **65534** is not required for NFS exports.

3.12.3.1. Group IDs

The recommended way to handle NFS access, assuming it is not an option to change permissions on the NFS export, is to use supplemental groups. Supplemental groups in OpenShift Container Platform are used for shared storage, of which NFS is an example. In contrast, block storage such as iSCSI uses the **fsGroup** SCC strategy and the **fsGroup** value in the **securityContext** of the pod.

**NOTE**

To gain access to persistent storage, it is generally preferable to use supplemental group IDs versus user IDs.

Because the group ID on the example target NFS directory is **5555**, the Pod can define that group ID using **supplementalGroups** under the **securityContext** definition of the pod. For example:

```
spec:
  containers:
    - name:
      ...
      securityContext: ❶
      supplementalGroups: [5555] ❷
```

- ❶ **securityContext** must be defined at the pod level, not under a specific container.
- ❷ An array of GIDs defined for the pod. In this case, there is one element in the array. Additional GIDs would be comma-separated.

Assuming there are no custom SCCs that might satisfy the pod requirements, the pod likely matches the **restricted** SCC. This SCC has the **supplementalGroups** strategy set to **RunAsAny**, meaning that any supplied group ID is accepted without range checking.

As a result, the above pod passes admissions and is launched. However, if group ID range checking is desired, a custom SCC is the preferred solution. A custom SCC can be created such that minimum and maximum group IDs are defined, group ID range checking is enforced, and a group ID of **5555** is allowed.

**NOTE**

To use a custom SCC, you must first add it to the appropriate service account. For example, use the **default** service account in the given project unless another has been specified on the **Pod** specification.

3.12.3.2. User IDs

User IDs can be defined in the container image or in the **Pod** definition.

**NOTE**

It is generally preferable to use supplemental group IDs to gain access to persistent storage versus using user IDs.

In the example target NFS directory shown above, the container needs its UID set to **65534**, ignoring group IDs for the moment, so the following can be added to the **Pod** definition:

```
spec:
  containers: ❶
    - name:
      ...
      securityContext:
        runAsUser: 65534 ❷
```

- 1 Pods contain a **securityContext** definition specific to each container and a pod's **securityContext** which applies to all containers defined in the pod.
- 2 **65534** is the **nfsnobody** user.

Assuming that the project is **default** and the SCC is **restricted**, the user ID of **65534** as requested by the pod is not allowed. Therefore, the pod fails for the following reasons:

- It requests **65534** as its user ID.
- All SCCs available to the Pod are examined to see which SCC allows a user ID of **65534**. While all policies of the SCCs are checked, the focus here is on user ID.
- Because all available SCCs use **MustRunAsRange** for their **runAsUser** strategy, UID range checking is required.
- **65534** is not included in the SCC or project's user ID range.

It is generally considered a good practice not to modify the predefined SCCs. The preferred way to fix this situation is to create a custom SCC. A custom SCC can be created such that minimum and maximum user IDs are defined, UID range checking is still enforced, and the UID of **65534** is allowed.



NOTE

To use a custom SCC, you must first add it to the appropriate service account. For example, use the **default** service account in the given project unless another has been specified on the **Pod** specification.

3.12.3.3. SELinux

Red Hat Enterprise Linux (RHEL) and Red Hat Enterprise Linux CoreOS (RHCOS) systems are configured to use SELinux on remote NFS servers by default.

For non-RHEL and non-RHCOS systems, SELinux does not allow writing from a pod to a remote NFS server. The NFS volume mounts correctly but it is read-only. You will need to enable the correct SELinux permissions by using the following procedure.

Prerequisites

- The **container-selinux** package must be installed. This package provides the **virt_use_nfs** SELinux boolean.

Procedure

- Enable the **virt_use_nfs** boolean using the following command. The **-P** option makes this boolean persistent across reboots.

```
# setsebool -P virt_use_nfs 1
```

3.12.3.4. Export settings

In order to enable arbitrary container users to read and write the volume, each exported volume on the NFS server should conform to the following conditions:

- Every export must be exported using the following format:

```
/<example_fs> *(rw,root_squash)
```

- The firewall must be configured to allow traffic to the mount point.
 - For NFSv4, configure the default port **2049** (**nfs**).

NFSv4

```
# iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
```

- For NFSv3, there are three ports to configure: **2049** (**nfs**), **20048** (**mountd**), and **111** (**portmapper**).

NFSv3

```
# iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
# iptables -I INPUT 1 -p tcp --dport 20048 -j ACCEPT
# iptables -I INPUT 1 -p tcp --dport 111 -j ACCEPT
```

- The NFS export and directory must be set up so that they are accessible by the target pods. Either set the export to be owned by the container's primary UID, or supply the pod group access using **supplementalGroups**, as shown in group IDs above.

3.12.4. Reclaiming resources

NFS implements the OpenShift Container Platform **Recyclable** plug-in interface. Automatic processes handle reclamation tasks based on policies set on each persistent volume.

By default, PVs are set to **Retain**.

Once claim to a PVC is deleted, and the PV is released, the PV object should not be reused. Instead, a new PV should be created with the same basic volume details as the original.

For example, the administrator creates a PV named **nfs1**:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs1
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

The user creates **PVC1**, which binds to **nfs1**. The user then deletes **PVC1**, releasing claim to **nfs1**. This results in **nfs1** being **Released**. If the administrator wants to make the same NFS share available, they should create a new PV with the same NFS server details, but a different PV name:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs2
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"

```

Deleting the original PV and re-creating it with the same name is discouraged. Attempting to manually change the status of a PV from **Released** to **Available** causes errors and potential data loss.

3.12.5. Additional configuration and troubleshooting

Depending on what version of NFS is being used and how it is configured, there may be additional configuration steps needed for proper export and security mapping. The following are some that may apply:

<p>NFSv4 mount incorrectly shows all files with ownership of nobody:nobody</p>	<ul style="list-style-type: none"> • Could be attributed to the ID mapping settings, found in /etc/idmapd.conf on your NFS. • See this Red Hat Solution.
<p>Disabling ID mapping on NFSv4</p>	<ul style="list-style-type: none"> • On both the NFS client and server, run: <pre># echo 'Y' > /sys/module/nfsd/parameters/nfs4_disable_idmapping</pre>

3.13. RED HAT OPENSIFT CONTAINER STORAGE

Red Hat OpenShift Container Storage is a provider of agnostic persistent storage for OpenShift Container Platform supporting file, block, and object storage, either in-house or in hybrid clouds. As a Red Hat storage solution, Red Hat OpenShift Container Storage is completely integrated with OpenShift Container Platform for deployment, management, and monitoring.

Red Hat OpenShift Container Storage provides its own documentation library. The complete set of Red Hat OpenShift Container Storage documentation identified below is available at https://access.redhat.com/documentation/en-us/red_hat_openshift_container_storage/4.4/

<p>If you are looking for Red Hat OpenShift Container Storage information about...</p>	<p>See the following Red Hat OpenShift Container Storage documentation:</p>
<p>What's new, known issues, notable bug fixes, and Technology Previews</p>	<p>Red Hat OpenShift Container Storage 4.4 Release Notes</p>

If you are looking for Red Hat OpenShift Container Storage information about...	See the following Red Hat OpenShift Container Storage documentation:
Supported workloads, layouts, hardware and software requirements, sizing and scaling recommendations	Planning your Red Hat OpenShift Container Storage 4.4 deployment
Deploying Red Hat OpenShift Container Storage 4.4 on an existing OpenShift Container Platform cluster	Deploying Red Hat OpenShift Container Storage 4.4
Managing a Red Hat OpenShift Container Storage 4.4 cluster	Managing Red Hat OpenShift Container Storage 4.4
Monitoring a Red Hat OpenShift Container Storage 4.4 cluster	Monitoring Red Hat OpenShift Container Storage 4.4
Migrating your OpenShift Container Platform cluster from version 3 to version 4	Migration

3.14. PERSISTENT STORAGE USING VMWARE VSPHERE VOLUMES

OpenShift Container Platform allows use of VMware vSphere’s Virtual Machine Disk (VMDK) volumes. You can provision your OpenShift Container Platform cluster with persistent storage using VMware vSphere. Some familiarity with Kubernetes and VMware vSphere is assumed.

VMware vSphere volumes can be provisioned dynamically. OpenShift Container Platform creates the disk in vSphere and attaches this disk to the correct image.

The Kubernetes persistent volume framework allows administrators to provision a cluster with persistent storage and gives users a way to request those resources without having any knowledge of the underlying infrastructure.

Persistent volumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. Persistent volume claims are specific to a project or namespace and can be requested by users.

Additional resources

- [VMware vSphere](#)

3.14.1. Dynamically provisioning VMware vSphere volumes

Dynamically provisioning VMware vSphere volumes is the recommended method.

3.14.2. Prerequisites

- An OpenShift Container Platform cluster installed on a VMware vSphere version that meets the requirements for the components that you use. See [Installing a cluster on vSphere](#) for information about vSphere version support.

You can use either of the following procedures to dynamically provision these volumes using the default storage class.

3.14.2.1. Dynamically provisioning VMware vSphere volumes using the UI

OpenShift Container Platform installs a default storage class, named **thin**, that uses the **thin** disk format for provisioning volumes.

Prerequisites

- Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Procedure

1. In the OpenShift Container Platform console, click **Storage → Persistent Volume Claims**
2. In the persistent volume claims overview, click **Create Persistent Volume Claim**
3. Define the required options on the resulting page.
 - a. Select the **thin** storage class.
 - b. Enter a unique name for the storage claim.
 - c. Select the access mode to determine the read and write access for the created storage claim.
 - d. Define the size of the storage claim.
4. Click **Create** to create the persistent volume claim and generate a persistent volume.

3.14.2.2. Dynamically provisioning VMware vSphere volumes using the CLI

OpenShift Container Platform installs a default StorageClass, named **thin**, that uses the **thin** disk format for provisioning volumes.

Prerequisites

- Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Procedure (CLI)

1. You can define a VMware vSphere PersistentVolumeClaim by creating a file, **pvc.yaml**, with the following contents:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc 1
spec:
  accessModes:
    - ReadWriteOnce 2
```

```
resources:
  requests:
    storage: 1Gi 3
```

- 1** A unique name that represents the persistent volume claim.
- 2** The access mode of the persistent volume claim. With **ReadWriteOnce**, the volume can be mounted with read and write permissions by a single node.
- 3** The size of the persistent volume claim.

2. Create the **PersistentVolumeClaim** object from the file:

```
$ oc create -f pvc.yaml
```

3.14.3. Statically provisioning VMware vSphere volumes

To statically provision VMware vSphere volumes you must create the virtual machine disks for reference by the persistent volume framework.

Prerequisites

- Storage must exist in the underlying infrastructure before it can be mounted as a volume in OpenShift Container Platform.

Procedure

1. Create the virtual machine disks. Virtual machine disks (VMDKs) must be created manually before statically provisioning VMware vSphere volumes. Use either of the following methods:

- Create using **vmkfstools**. Access ESX through Secure Shell (SSH) and then use following command to create a VMDK volume:

```
$ vmkfstools -c <size> /vmfs/volumes/<datastore-name>/volumes/<disk-name>.vmdk
```

- Create using **vmware-diskmanager**:

```
$ shell vmware-vdiskmanager -c -t 0 -s <size> -a lsilogic <disk-name>.vmdk
```

2. Create a persistent volume that references the VMDKs. Create a file, **pv1.yaml**, with the **PersistentVolume** object definition:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1 1
spec:
  capacity:
    storage: 1Gi 2
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
```

```
vsphereVolume: 3
  volumePath: "[datastore1] volumes/myDisk" 4
  fsType: ext4 5
```

- 1 The name of the volume. This name is how it is identified by persistent volume claims or pods.
- 2 The amount of storage allocated to this volume.
- 3 The volume type used, with **vsphereVolume** for vSphere volumes. The label is used to mount a vSphere VMDK volume into pods. The contents of a volume are preserved when it is unmounted. The volume type supports VMFS and VSAN datastore.
- 4 The existing VMDK volume to use. If you used **vmkfstools**, you must enclose the datastore name in square brackets, [], in the volume definition, as shown previously.
- 5 The file system type to mount. For example, ext4, xfs, or other file systems.



IMPORTANT

Changing the value of the `fsType` parameter after the volume is formatted and provisioned can result in data loss and pod failure.

3. Create the **PersistentVolume** object from the file:

```
$ oc create -f pv1.yaml
```

4. Create a persistent volume claim that maps to the persistent volume you created in the previous step. Create a file, **pvc1.yaml**, with the **PersistentVolumeClaim** object definition:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1 1
spec:
  accessModes:
    - ReadWriteOnce 2
  resources:
    requests:
      storage: "1Gi" 3
  volumeName: pv1 4
```

- 1 A unique name that represents the persistent volume claim.
- 2 The access mode of the persistent volume claim. With `ReadWriteOnce`, the volume can be mounted with read and write permissions by a single node.
- 3 The size of the persistent volume claim.
- 4 The name of the existing persistent volume.

5. Create the **PersistentVolumeClaim** object from the file:

```
$ oc create -f pvc1.yaml
```

3.14.3.1. Formatting VMware vSphere volumes

Before OpenShift Container Platform mounts the volume and passes it to a container, it checks that the volume contains a file system that is specified by the **fsType** parameter value in the **PersistentVolume** (PV) definition. If the device is not formatted with the file system, all data from the device is erased, and the device is automatically formatted with the specified file system.

Because OpenShift Container Platform formats them before the first use, you can use unformatted vSphere volumes as PVs.

3.14.4. Backing up VMware vSphere volumes

OpenShift Container Platform provisions new volumes as independent persistent disks to freely attach and detach the volume on any node in the cluster. As a consequence, it is not possible to back up volumes that use snapshots, or to restore volumes from snapshots. See [Snapshot Limitations](#) for more information.

Procedure

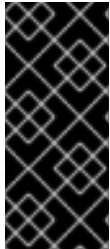
To create a backup of persistent volumes:

1. Stop the application that is using the persistent volume.
2. Clone the persistent volume.
3. Restart the application.
4. Create a backup of the cloned volume.
5. Delete the cloned volume.

CHAPTER 4. USING CONTAINER STORAGE INTERFACE (CSI)

4.1. CONFIGURING CSI VOLUMES

The Container Storage Interface (CSI) allows OpenShift Container Platform to consume storage from storage back ends that implement the [CSI interface](#) as persistent storage.



IMPORTANT

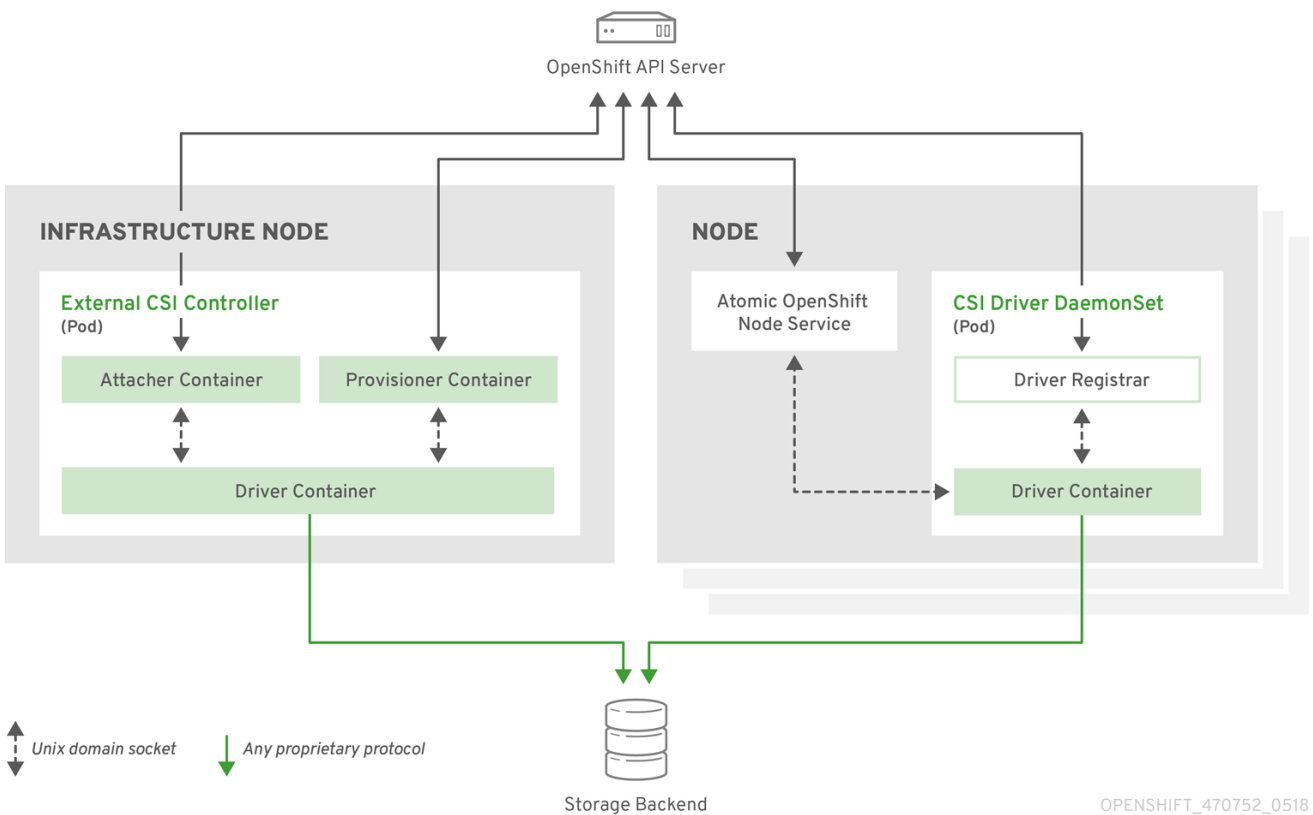
OpenShift Container Platform does not ship with any CSI drivers. It is recommended to use the CSI drivers provided by [community or storage vendors](#).

Installation instructions differ by driver, and are found in each driver’s documentation. Follow the instructions provided by the CSI driver.

4.1.1. CSI Architecture

CSI drivers are typically shipped as container images. These containers are not aware of OpenShift Container Platform where they run. To use CSI-compatible storage back end in OpenShift Container Platform, the cluster administrator must deploy several components that serve as a bridge between OpenShift Container Platform and the storage driver.

The following diagram provides a high-level overview about the components running in pods in the OpenShift Container Platform cluster.



It is possible to run multiple CSI drivers for different storage back ends. Each driver needs its own external controllers deployment and daemon set with the driver and CSI registrar.

4.1.1.1. External CSI controllers

External CSI Controllers is a deployment that deploys one or more pods with three containers:

- An external CSI attacher container translates **attach** and **detach** calls from OpenShift Container Platform to respective **ControllerPublish** and **ControllerUnpublish** calls to the CSI driver.
- An external CSI provisioner container that translates **provision** and **delete** calls from OpenShift Container Platform to respective **CreateVolume** and **DeleteVolume** calls to the CSI driver.
- A CSI driver container

The CSI attacher and CSI provisioner containers communicate with the CSI driver container using UNIX Domain Sockets, ensuring that no CSI communication leaves the pod. The CSI driver is not accessible from outside of the pod.



NOTE

attach, **detach**, **provision**, and **delete** operations typically require the CSI driver to use credentials to the storage backend. Run the CSI controller pods on infrastructure nodes so the credentials are never leaked to user processes, even in the event of a catastrophic security breach on a compute node.



NOTE

The external attacher must also run for CSI drivers that do not support third-party **attach** or **detach** operations. The external attacher will not issue any **ControllerPublish** or **ControllerUnpublish** operations to the CSI driver. However, it still must run to implement the necessary OpenShift Container Platform attachment API.

4.1.1.2. CSI driver daemon set

The CSI driver daemon set runs a pod on every node that allows OpenShift Container Platform to mount storage provided by the CSI driver to the node and use it in user workloads (pods) as persistent volumes (PVs). The pod with the CSI driver installed contains the following containers:

- A CSI driver registrar, which registers the CSI driver into the **openshift-node** service running on the node. The **openshift-node** process running on the node then directly connects with the CSI driver using the UNIX Domain Socket available on the node.
- A CSI driver.

The CSI driver deployed on the node should have as few credentials to the storage back end as possible. OpenShift Container Platform will only use the node plug-in set of CSI calls such as **NodePublish/NodeUnpublish** and **NodeStage/NodeUnstage**, if these calls are implemented.

4.1.2. Dynamic provisioning

Dynamic provisioning of persistent storage depends on the capabilities of the CSI driver and underlying storage back end. The provider of the CSI driver should document how to create a storage class in OpenShift Container Platform and the parameters available for configuration.

The created storage class can be configured to enable dynamic provisioning.

Procedure

- Create a default storage class that ensures all PVCs that do not require any special storage class are provisioned by the installed CSI driver.

```
# oc create -f - << EOF
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: <storage-class> 1
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: <provisioner-name> 2
parameters:
EOF
```

- 1 The name of the storage class that will be created.
- 2 The name of the CSI driver that has been installed

4.1.3. Example using the CSI driver

The following example installs a default MySQL template without any changes to the template.

Prerequisites

- The CSI driver has been deployed.
- A storage class has been created for dynamic provisioning.

Procedure

- Create the MySQL template:

```
# oc new-app mysql-persistent
--> Deploying template "openshift/mysql-persistent" to project default
...

# oc get pvc
NAME          STATUS  VOLUME                                     CAPACITY
ACCESS MODES STORAGECLASS AGE
mysql         Bound   kubernetes-dynamic-pv-3271ffc4e1811e8  1Gi
RWO           cinder  3s
```

4.2. CSI VOLUME SNAPSHOTS

This document describes how to use volume snapshots with supported Container Storage Interface (CSI) drivers to help protect against data loss in OpenShift Container Platform. Familiarity with [persistent volumes](#) is suggested.



IMPORTANT

CSI volume snapshot is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see <https://access.redhat.com/support/offerings/techpreview/>.

4.2.1. Overview of CSI volume snapshots

A *snapshot* represents the state of the storage volume in a cluster at a particular point in time. Volume snapshots can be used to provision a new volume.

OpenShift Container Platform supports CSI volume snapshots by default. However, a specific CSI driver is required.

With CSI volume snapshots, a cluster administrator can:

- Deploy a third-party CSI driver that supports snapshots.
- Create a new persistent volume claim (PVC) from an existing volume snapshot.
- Take a snapshot of an existing PVC.
- Restore a snapshot as a different PVC.
- Delete an existing volume snapshot.

With CSI volume snapshots, an app developer can:

- Use volume snapshots as building blocks for developing application- or cluster-level storage backup solutions.
- Rapidly rollback to a previous development version.
- Use storage more efficiently by not having to make a full copy each time.

Be aware of the following when using volume snapshots:

- Support is only available for CSI drivers. In-tree and FlexVolumes are not supported.
- OpenShift Container Platform does not ship with any CSI drivers. It is recommended to use the CSI drivers provided by [community or storage vendors](#). Follow the installation instructions provided by the CSI driver.
- CSI drivers may or may not have implemented the volume snapshot functionality. CSI drivers that have provided support for volume snapshots will likely use the **csi-external-snapshotter** sidecar. See documentation provided by the CSI driver for details.
- OpenShift Container Platform 4.4 supports version 1.1.0 of the [CSI specification](#).

4.2.2. CSI snapshot controller and sidecar

OpenShift Container Platform provides a snapshot controller that is deployed into the control plane. In addition, your CSI driver vendor provides the CSI snapshot sidecar as a helper container that is installed during the CSI driver installation.

The CSI snapshot controller and sidecar provide volume snapshotting through the OpenShift Container Platform API. These external components run in the cluster.

The external controller is deployed by the CSI Snapshot Controller Operator.

4.2.2.1. External controller

The CSI snapshot controller binds **VolumeSnapshot** and **VolumeSnapshotContent** objects. The controller manages dynamic provisioning by creating and deleting **VolumeSnapshotContent** objects.

4.2.2.2. External sidecar

Your CSI driver vendor provides the **csi-external-snapshotter** sidecar. This is a separate helper container that is deployed with the CSI driver. The sidecar manages snapshots by triggering **CreateSnapshot** and **DeleteSnapshot** operations. Follow the installation instructions provided by your vendor.

4.2.3. About the CSI Snapshot Controller Operator

The CSI Snapshot Controller Operator runs in the **openshift-cluster-storage-operator** namespace. It is installed by the Cluster Version Operator (CVO) in all clusters by default.

The CSI Snapshot Controller Operator installs the CSI snapshot controller, which runs in the **csi-snapshot-controller** namespace.

4.2.3.1. Volume snapshot CRDs

During OpenShift Container Platform installation, the CSI Snapshot Controller Operator creates the following snapshot custom resource definitions (CRDs) in the **snapshot.storage.k8s.io** API group:

VolumeSnapshotContent

A snapshot taken of a volume in the cluster that has been provisioned by a cluster administrator. Similar to the **PersistentVolume** CRD, the **VolumeSnapshotContent** CRD is a cluster resource that points to a real snapshot in the storage back end.

For manually pre-provisioned snapshots, a cluster administrator creates a number of **VolumeSnapshotContent** objects. These carry the details of the real volume snapshot in the storage system.

The **VolumeSnapshotContent** CRD is not namespaced and is for use by a cluster administrator.

VolumeSnapshot

Similar to the **PersistentVolumeClaim** CRD, the **VolumeSnapshot** CRD defines a developer request for a snapshot. The CSI Snapshot Controller Operator runs the CSI snapshot controller, which handles the binding of a **VolumeSnapshot** object with an appropriate **VolumeSnapshotContent** object. The binding is a one-to-one mapping.

The **VolumeSnapshot** CRD is namespaced. A developer uses the CRD as a distinct request for a snapshot.

VolumeSnapshotClass

Allows a cluster administrator to specify different attributes belonging to a **VolumeSnapshot** object. These attributes may differ among snapshots taken of the same volume on the storage system, in which case they would not be expressed by using the same storage class of a persistent volume claim.

The **VolumeSnapshotClass** CRD defines the parameters for the **csi-external-snapshotter** sidecar to use when creating a snapshot. This allows the storage back end to know what kind of snapshot to dynamically create if multiple options are supported.

Dynamically provisioned snapshots use the **VolumeSnapshotClass** CRD to specify storage-provider-specific parameters to use when creating a snapshot.

The **VolumeSnapshotContentClass** CRD is not namespaced and is for use by a cluster administrator to enable global configuration options for their storage back end.

4.2.4. Volume snapshot provisioning

There are two ways to provision snapshots: dynamically and manually.

4.2.4.1. Dynamic provisioning

Instead of using a preexisting snapshot, you can request that a snapshot be taken dynamically from a persistent volume claim. Parameters are specified using a **VolumeSnapshotClass** CRD.

4.2.4.2. Manual provisioning

As a cluster administrator, you can manually pre-provision a number of **VolumeSnapshotContent** objects. These carry the real volume snapshot details available to cluster users.

4.2.5. Creating a volume snapshot

When you create a **VolumeSnapshot** object, OpenShift Container Platform creates a volume snapshot.

Prerequisites

- Logged in to a running OpenShift Container Platform cluster.
- A PVC created using a CSI driver that supports **VolumeSnapshot** objects.
- A storage class to provision the storage back end.
- No pods are using the persistent volume claim (PVC) that you want to take a snapshot of.



NOTE

Do not create a volume snapshot of a PVC if a pod is using it. Doing so might cause data corruption because the PVC is not quiesced (paused). Be sure to first tear down a running pod to ensure consistent snapshots.

Procedure

To dynamically create a volume snapshot:

1. Create a file with the **VolumeSnapshotClass** object described by the following YAML:

volumesnapshotclass.yaml

```

apiVersion: snapshot.storage.k8s.io
kind: VolumeSnapshotClass 1
metadata:
  name: csi-hostpath-snap
driver: hostpath.csi.k8s.io
deletionPolicy: Delete

```

1 Allows you to specify different attributes belonging to a volume snapshot.

2. Create the object you saved in the previous step by entering the following command:

```
$ oc create -f volumesnapshotclass.yaml
```

3. Create a **VolumeSnapshot** object:

volumesnapshot-dynamic.yaml

```

apiVersion: snapshot.storage.k8s.io
kind: VolumeSnapshot
metadata:
  name: mysnap
spec:
  volumeSnapshotClassName: csi-hostpath-snap 1
  source:
    persistentVolumeClaimName: myclaim 2

```

1 The request for a particular class by the volume snapshot. If **volumeSnapshotClassName** is empty, then no snapshot is created.

2 The name of the **PersistentVolumeClaim** object bound to a persistent volume. This defines what you want to create a snapshot of. Required for dynamically provisioning a snapshot.

4. Create the object you saved in the previous step by entering the following command:

```
$ oc create -f volumesnapshot-dynamic.yaml
```

To manually provision a snapshot:

1. Provide a value for the **volumeSnapshotContentName** parameter as the source for the snapshot, in addition to defining volume snapshot class as shown above.

volumesnapshot-manual.yaml

```

apiVersion: snapshot.storage.k8s.io
kind: VolumeSnapshot
metadata:
  name: snapshot-demo

```

```
spec:
  source:
    volumeSnapshotContentName: mycontent 1
```

- 1 The **volumeSnapshotContentName** parameter is required for pre-provisioned snapshots.

2. Create the object you saved in the previous step by entering the following command:

```
$ oc create -f volumesnapshot-manual.yaml
```

Verification steps

After the snapshot has been created in the cluster, additional details about the snapshot are available.

1. To display details about the volume snapshot that was created, enter the following command:

```
$ oc describe volumesnapshot mysnap
```

The following example displays details about the **mysnap** volume snapshot:

volumesnapshot.yaml

```
apiVersion: snapshot.storage.k8s.io
kind: VolumeSnapshot
metadata:
  name: mysnap
spec:
  source:
    persistentVolumeClaimName: myclaim
    volumeSnapshotClassName: csi-hostpath-snap
status:
  boundVolumeSnapshotContentName: snapcontent-1af4989e-a365-4286-96f8-
  d5dcd65d78d6 1
  creationTime: "2020-01-29T12:24:30Z" 2
  readyToUse: true 3
  restoreSize: 500Mi
```

- 1 The pointer to the actual storage content that was created by the controller.
- 2 The time when the snapshot was created. The snapshot contains the volume content that was available at this indicated time.
- 3 If the value is set to **true**, the snapshot can be used to restore as a new PVC. If the value is set to **false**, the snapshot was created. However, the storage back end needs to perform additional tasks to make the snapshot usable so that it can be restored as a new volume. For example, Amazon Elastic Block Store data might be moved to a different, less expensive location, which can take several minutes.

2. To verify that the volume snapshot was created, enter the following command:

```
$ oc get volumesnapshotcontent
```

The pointer to the actual content is displayed. If the **boundVolumeSnapshotContentName** field is populated, a **VolumeSnapshotContent** object exists and the snapshot was created.

3. To verify that the snapshot is ready, confirm that the **VolumeSnapshot** object has **readyToUse: true**.

4.2.6. Deleting a volume snapshot

You can configure how OpenShift Container Platform deletes volume snapshots.

Procedure

To enable deletion of a volume snapshot in a cluster:

1. Specify the deletion policy that you require in the **VolumeSnapshotClass** object, as shown in the following example:

volumesnapshot.yaml

```
apiVersion: snapshot.storage.k8s.io
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snap
driver: hostpath.csi.k8s.io
deletionPolicy: Delete 1
```

- 1** If the **Delete** value is set, the underlying snapshot will be deleted, along with the **VolumeSnapshotContent** object. If the **Retain** value is set, both the underlying snapshot and **VolumeSnapshotContent** object remain. If the **Retain** value is set, and the **VolumeSnapshot** object is deleted without deleting the corresponding **VolumeSnapshotContent** object, then the content will remain. The snapshot itself is also retained in the storage back end.

4.2.7. Restoring a volume snapshot

After your **VolumeSnapshot** object is bound, you can use that object to provision a new volume that is pre-populated with data from the snapshot.

The volume snapshot content object is used to restore the existing volume to a previous state.

Prerequisites

- Logged in to a running OpenShift Container Platform cluster.
- A persistent volume claim (PVC) created using a Container Storage Interface (CSI) driver that supports volume snapshots.
- A storage class to provision the storage back end.

Procedure

1. Specify a **VolumeSnapshot** data source on a PVC as shown in the following:

pvc-restore.yaml


```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim-restore
spec:
  storageClassName: csi-hostpath-sc
  dataSource:
    name: mysnap 1
    kind: VolumeSnapshot 2
    apiGroup: snapshot.storage.k8s.io 3
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

```

- 1** Name of the **VolumeSnapshot** object representing the snapshot to use as source.
- 2** Must be set to the **VolumeSnapshot** value.
- 3** Must be set to the **snapshot.storage.k8s.io** value.

2. Create a PVC by entering the following command:

```
$ oc create -f pvc-restore.yaml
```

3. Verify that the restored PVC has been created by entering the following command:

```
$ oc get pvc
```

Two different PVCs are displayed.

4.3. CSI VOLUME CLONING

Volume cloning duplicates an existing persistent volume to help protect against data loss in OpenShift Container Platform. This feature is only available with supported Container Storage Interface (CSI) drivers. You should be familiar with [persistent volumes](#) before you provision a CSI volume clone.



IMPORTANT

CSI volume cloning is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see <https://access.redhat.com/support/offerings/techpreview/>.

4.3.1. Overview of CSI volume cloning

A Container Storage Interface (CSI) volume clone is a duplicate of an existing persistent volume at a particular point in time.

Volume cloning is similar to volume snapshots, although it is more efficient. For example, a cluster administrator can duplicate a cluster volume by creating another instance of the existing cluster volume.

Cloning creates an exact duplicate of the specified volume on the back-end device, rather than creating a new empty volume. After dynamic provisioning, you can use a volume clone just as you would use any standard volume.

No new API objects are required for cloning. The existing **dataSource** field in the **PersistentVolumeClaim** object is expanded so that it can accept the name of an existing PersistentVolumeClaim in the same namespace.

4.3.1.1. Support limitations

By default, OpenShift Container Platform supports CSI volume cloning with these limitations:

- The destination persistent volume claim (PVC) must exist in the same namespace as the source PVC.
- The source and destination storage class must be the same.
- Support is only available for CSI drivers. In-tree and FlexVolumes are not supported.
- OpenShift Container Platform does not include any CSI drivers. Use the CSI drivers provided by [community or storage vendors](#). Follow the installation instructions provided by the CSI driver.
- CSI drivers might not have implemented the volume cloning functionality. For details, see the CSI driver documentation.
- OpenShift Container Platform 4.4 supports version 1.1.0 of the [CSI specification](#).

4.3.2. Provisioning a CSI volume clone

When you create a cloned persistent volume claim (PVC) API object, you trigger the provisioning of a CSI volume clone. The clone pre-populates with the contents of another PVC, adhering to the same rules as any other persistent volume. The one exception is that you must add a **dataSource** that references an existing PVC in the same namespace.

Prerequisites

- You are logged in to a running OpenShift Container Platform cluster.
- Your PVC is created using a CSI driver that supports volume cloning.
- Your storage back end is configured for dynamic provisioning. Cloning support is not available for static provisioners.

Procedure

To clone a PVC from an existing PVC:

1. Create and save a file with the **PersistentVolumeClaim** object described by the following YAML:

```
pvc-clone.yaml
```

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-1-clone
  namespace: mynamespace
spec:
  storageClassName: csi-cloning ❶
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  dataSource:
    kind: PersistentVolumeClaim
    name: pvc-1

```

- ❶ The name of the storage class that provisions the storage back end. The default storage class can be used and **storageClassName** can be omitted in the spec.

2. Create the object you saved in the previous step by running the following command:

```
$ oc create -f pvc-clone.yaml
```

A new PVC **pvc-1-clone** is created.

3. Verify that the volume clone was created and is ready by running the following command:

```
$ oc get pvc pvc-1-clone
```

The **pvc-1-clone** shows that it is **Bound**.

You are now ready to use the newly cloned PVC to configure a pod.

4. Create and save a file with the **Pod** object described by the YAML. For example:

```

kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-1-clone ❶

```

- ❶ The cloned PVC created during the CSI volume cloning operation.

The created **Pod** object is now ready to consume, clone, snapshot, or delete your cloned PVC independently of its original **dataSource** PVC.

CHAPTER 5. EXPANDING PERSISTENT VOLUMES

5.1. ENABLING VOLUME EXPANSION SUPPORT

Before you can expand persistent volumes, the **StorageClass** object must have the **allowVolumeExpansion** field set to **true**.

Procedure

- Edit the **StorageClass** object and add the **allowVolumeExpansion** attribute. The following example demonstrates adding this line at the bottom of the storage class configuration.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
...
parameters:
  type: gp2
  reclaimPolicy: Delete
  allowVolumeExpansion: true 1
```

- 1** Setting this attribute to **true** allows PVCs to be expanded after creation.

5.2. EXPANDING CSI VOLUMES

You can use the Container Storage Interface (CSI) to expand storage volumes after they have already been created.

OpenShift Container Platform supports CSI volume expansion by default. However, a specific CSI driver is required.

OpenShift Container Platform does not ship with any CSI drivers. It is recommended to use the CSI drivers provided by [community or storage vendors](#). Follow the installation instructions provided by the CSI driver.

OpenShift Container Platform 4.4 supports version 1.1.0 of the [CSI specification](#).



IMPORTANT

Expanding CSI volumes is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see <https://access.redhat.com/support/offerings/techpreview/>.

5.3. EXPANDING FLEXVOLUME WITH A SUPPORTED DRIVER

When using FlexVolume to connect to your back-end storage system, you can expand persistent storage volumes after they have already been created. This is done by manually updating the persistent volume claim (PVC) in OpenShift Container Platform.

FlexVolume allows expansion if the driver is set with **RequiresFSResize** to **true**. The FlexVolume can be expanded on pod restart.

Similar to other volume types, FlexVolume volumes can also be expanded when in use by a pod.

Prerequisites

- The underlying volume driver supports resize.
- The driver is set with the **RequiresFSResize** capability to **true**.
- Dynamic provisioning is used.
- The controlling **StorageClass** object has **allowVolumeExpansion** set to **true**.

Procedure

- To use resizing in the FlexVolume plugin, you must implement the **ExpandableVolumePlugin** interface using these methods:

RequiresFSResize

If **true**, updates the capacity directly. If **false**, calls the **ExpandFS** method to finish the filesystem resize.

ExpandFS

If **true**, calls **ExpandFS** to resize filesystem after physical volume expansion is done. The volume driver can also perform physical volume resize together with filesystem resize.



IMPORTANT

Because OpenShift Container Platform does not support installation of FlexVolume plugins on master nodes, it does not support control-plane expansion of FlexVolume.

5.4. EXPANDING PERSISTENT VOLUME CLAIMS (PVCs) WITH A FILE SYSTEM

Expanding PVCs based on volume types that need file system resizing, such as GCE PD, EBS, and Cinder, is a two-step process. This process involves expanding volume objects in the cloud provider, and then expanding the file system on the actual node.

Expanding the file system on the node only happens when a new pod is started with the volume.

Prerequisites

- The controlling **StorageClass** object must have **allowVolumeExpansion** set to **true**.

Procedure

1. Edit the PVC and request a new size by editing **spec.resources.requests**. For example, the following expands the **ebs** PVC to 8 Gi.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
```

```

name: ebs
spec:
  storageClass: "storageClassWithFlagSet"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi 1

```

1 Updating **spec.resources.requests** to a larger amount will expand the PVC.

2. Once the cloud provider object has finished resizing, the PVC is set to **FileSystemResizePending**. The following command is used to check the condition:

```
$ oc describe pvc <pvc_name>
```

3. When the cloud provider object has finished resizing, the **PersistentVolume** object reflects the newly requested size in **PersistentVolume.Spec.Capacity**. At this point, you can create or recreate a new pod from the PVC to finish the file system resizing. Once the pod is running, the newly requested size is available and the **FileSystemResizePending** condition is removed from the PVC.

5.5. RECOVERING FROM FAILURE WHEN EXPANDING VOLUMES

If expanding underlying storage fails, the OpenShift Container Platform administrator can manually recover the persistent volume claim (PVC) state and cancel the resize requests. Otherwise, the resize requests are continuously retried by the controller without administrator intervention.

Procedure

1. Mark the persistent volume (PV) that is bound to the PVC with the **Retain** reclaim policy. This can be done by editing the PV and changing **persistentVolumeReclaimPolicy** to **Retain**.
2. Delete the PVC. This will be recreated later.
3. To ensure that the newly created PVC can bind to the PV marked **Retain**, manually edit the PV and delete the **claimRef** entry from the PV specs. This marks the PV as **Available**.
4. Re-create the PVC in a smaller size, or a size that can be allocated by the underlying storage provider.
5. Set the **volumeName** field of the PVC to the name of the PV. This binds the PVC to the provisioned PV only.
6. Restore the reclaim policy on the PV.

CHAPTER 6. DYNAMIC PROVISIONING

6.1. ABOUT DYNAMIC PROVISIONING

The **StorageClass** resource object describes and classifies storage that can be requested, as well as provides a means for passing parameters for dynamically provisioned storage on demand.

StorageClass objects can also serve as a management mechanism for controlling different levels of storage and access to the storage. Cluster Administrators (**cluster-admin**) or Storage Administrators (**storage-admin**) define and create the **StorageClass** objects that users can request without needing any intimate knowledge about the underlying storage volume sources.

The OpenShift Container Platform persistent volume framework enables this functionality and allows administrators to provision a cluster with persistent storage. The framework also gives users a way to request those resources without having any knowledge of the underlying infrastructure.

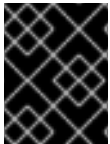
Many storage types are available for use as persistent volumes in OpenShift Container Platform. While all of them can be statically provisioned by an administrator, some types of storage are created dynamically using the built-in provider and plug-in APIs.

6.2. AVAILABLE DYNAMIC PROVISIONING PLUG-INS

OpenShift Container Platform provides the following provisioner plug-ins, which have generic implementations for dynamic provisioning that use the cluster's configured provider's API to create new storage resources:

Storage type	Provisioner plug-in name	Notes
Red Hat OpenStack Platform (RHOSP) Cinder	kubernetes.io/cinder	
AWS Elastic Block Store (EBS)	kubernetes.io/aws-ebs	For dynamic provisioning when using multiple clusters in different zones, tag each node with Key=kubernetes.io/cluster/<cluster_name>,Value=<cluster_id> where <cluster_name> and <cluster_id> are unique per cluster.
AWS Elastic File System (EFS)		Dynamic provisioning is accomplished through the EFS provisioner pod and not through a provisioner plug-in.
Azure Disk	kubernetes.io/azure-disk	
Azure File	kubernetes.io/azure-file	The persistent-volume-binder service account requires permissions to create and get secrets to store the Azure storage account and keys.

Storage type	Provisioner plug-in name	Notes
GCE Persistent Disk (gcePD)	kubernetes.io/gce-pd	In multi-zone configurations, it is advisable to run one OpenShift Container Platform cluster per GCE project to avoid PVs from being created in zones where no node in the current cluster exists.
VMware vSphere	kubernetes.io/vsphere-volume	



IMPORTANT

Any chosen provisioner plug-in also requires configuration for the relevant cloud, host, or third-party provider as per the relevant documentation.

6.3. DEFINING A STORAGE CLASS

StorageClass objects are currently a globally scoped object and must be created by **cluster-admin** or **storage-admin** users.



IMPORTANT

The Cluster Storage Operator might install a default storage class depending on the platform in use. This storage class is owned and controlled by the operator. It cannot be deleted or modified beyond defining annotations and labels. If different behavior is desired, you must define a custom storage class.

The following sections describe the basic definition for a **StorageClass** object and specific examples for each of the supported plug-in types.

6.3.1. Basic StorageClass object definition

The following resource shows the parameters and default values that you use to configure a storage class. This example uses the AWS ElasticBlockStore (EBS) object definition.

Sample StorageClass definition

```
kind: StorageClass 1
apiVersion: storage.k8s.io/v1 2
metadata:
  name: gp2 3
  annotations: 4
    storageclass.kubernetes.io/is-default-class: 'true'
```

```

...
provisioner: kubernetes.io/aws-ebs 5
parameters: 6
  type: gp2
...

```

- 1** (required) The API object type.
- 2** (required) The current apiVersion.
- 3** (required) The name of the storage class.
- 4** (optional) Annotations for the storage class.
- 5** (required) The type of provisioner associated with this storage class.
- 6** (optional) The parameters required for the specific provisioner, this will change from plug-in to plug-in.

6.3.2. Storage class annotations

To set a storage class as the cluster-wide default, add the following annotation to your storage class metadata:

```
storageclass.kubernetes.io/is-default-class: "true"
```

For example:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
...

```

This enables any persistent volume claim (PVC) that does not specify a specific volume to automatically be provisioned through the default storage class.



NOTE

The beta annotation **storageclass.beta.kubernetes.io/is-default-class** is still working; however, it will be removed in a future release.

To set a storage class description, add the following annotation to your storage class metadata:

```
kubernetes.io/description: My Storage Class Description
```

For example:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:

```

```

annotations:
  kubernetes.io/description: My Storage Class Description
...

```

6.3.3. RHOSP Cinder object definition

cinder-storageclass.yaml

```

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  type: fast 1
  availability: nova 2
  fsType: ext4 3

```

- 1** Volume type created in Cinder. Default is empty.
- 2** Availability Zone. If not specified, volumes are generally round-robined across all active zones where the OpenShift Container Platform cluster has a node.
- 3** File system that is created on dynamically provisioned volumes. This value is copied to the **fsType** field of dynamically provisioned persistent volumes and the file system is created when the volume is mounted for the first time. The default value is **ext4**.

6.3.4. AWS Elastic Block Store (EBS) object definition

aws-ebs-storageclass.yaml

```

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1 1
  iopsPerGB: "10" 2
  encrypted: "true" 3
  kmsKeyId: keyvalue 4
  fsType: ext4 5

```

- 1** (required) Select from **io1**, **gp2**, **sc1**, **st1**. The default is **gp2**. See the [AWS documentation](#) for valid Amazon Resource Name (ARN) values.
- 2** (optional) Only for **io1** volumes. I/O operations per second per GiB. The AWS volume plug-in multiplies this with the size of the requested volume to compute IOPS of the volume. The value cap is 20,000 IOPS, which is the maximum supported by AWS. See the [AWS documentation](#) for further details.
- 3** (optional) Denotes whether to encrypt the EBS volume. Valid values are **true** or **false**.

- 4 (optional) The full ARN of the key to use when encrypting the volume. If none is supplied, but **encrypted** is set to **true**, then AWS generates a key. See the [AWS documentation](#) for a valid ARN
- 5 (optional) File system that is created on dynamically provisioned volumes. This value is copied to the **fsType** field of dynamically provisioned persistent volumes and the file system is created when the volume is mounted for the first time. The default value is **ext4**.

6.3.5. Azure Disk object definition

azure-advanced-disk-storageclass.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-premium
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/azure-disk
volumeBindingMode: WaitForFirstConsumer 1
allowVolumeExpansion: true
parameters:
  kind: Managed 2
  storageaccounttype: Premium_LRS 3
reclaimPolicy: Delete
```

- 1 Using **WaitForFirstConsumer** is strongly recommended. This provisions the volume while allowing enough storage to schedule the pod on a free worker node from an available zone.
- 2 Possible values are **Shared** (default), **Managed**, and **Dedicated**.



IMPORTANT

Red Hat only supports the use of **kind: Managed** in the storage class.

With **Shared** and **Dedicated**, Azure creates unmanaged disks, while OpenShift Container Platform creates a managed disk for machine OS (root) disks. But because Azure Disk does not allow the use of both managed and unmanaged disks on a node, unmanaged disks created with **Shared** or **Dedicated** cannot be attached to OpenShift Container Platform nodes.

- 3 Azure storage account SKU tier. Default is empty. Note that Premium VMs can attach both **Standard_LRS** and **Premium_LRS** disks, Standard VMs can only attach **Standard_LRS** disks, Managed VMs can only attach managed disks, and unmanaged VMs can only attach unmanaged disks.
 - a. If **kind** is set to **Shared**, Azure creates all unmanaged disks in a few shared storage accounts in the same resource group as the cluster.
 - b. If **kind** is set to **Managed**, Azure creates new managed disks.
 - c. If **kind** is set to **Dedicated** and a **storageAccount** is specified, Azure uses the specified storage account for the new unmanaged disk in the same resource group as the cluster. For this to work:

- The specified storage account must be in the same region.
 - Azure Cloud Provider must have write access to the storage account.
- d. If **kind** is set to **Dedicated** and a **storageAccount** is not specified, Azure creates a new dedicated storage account for the new unmanaged disk in the same resource group as the cluster.

6.3.6. Azure File object definition

The Azure File storage class uses secrets to store the Azure storage account name and the storage account key that are required to create an Azure Files share. These permissions are created as part of the following procedure.

Procedure

1. Define a **ClusterRole** object that allows access to create and view secrets:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # name: system:azure-cloud-provider
  name: <persistent-volume-binder-role> 1
rules:
- apiGroups: [""]
  resources: ['secrets']
  verbs: ['get','create']
```

- 1 The name of the cluster role to view and create secrets.

2. Add the cluster role to the service account:

```
$ oc adm policy add-cluster-role-to-user <persistent-volume-binder-role>
system:serviceaccount:kube-system:persistent-volume-binder
```

3. Create the Azure File **StorageClass** object:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: <azure-file> 1
provisioner: kubernetes.io/azure-file
parameters:
  location: eastus 2
  skuName: Standard_LRS 3
  storageAccount: <storage-account> 4
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

- 1 Name of the storage class. The persistent volume claim uses this storage class for provisioning the associated persistent volumes.

- 2 Location of the Azure storage account, such as **eastus**. Default is empty, meaning that a new Azure storage account will be created in the OpenShift Container Platform cluster's location.
- 3 SKU tier of the Azure storage account, such as **Standard_LRS**. Default is empty, meaning that a new Azure storage account will be created with the **Standard_LRS** SKU.
- 4 Name of the Azure storage account. If a storage account is provided, then **skuName** and **location** are ignored. If no storage account is provided, then the storage class searches for any storage account that is associated with the resource group for any accounts that match the defined **skuName** and **location**.

6.3.6.1. Considerations when using Azure File

The following file system features are not supported by the default Azure File storage class:

- Symlinks
- Hard links
- Extended attributes
- Sparse files
- Named pipes

Additionally, the owner user identifier (UID) of the Azure File mounted directory is different from the process UID of the container. The **uid** mount option can be specified in the **StorageClass** object to define a specific user identifier to use for the mounted directory.

The following **StorageClass** object demonstrates modifying the user and group identifier, along with enabling symlinks for the mounted directory.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azure-file
mountOptions:
  - uid=1500 1
  - gid=1500 2
  - mfsymlinks 3
provisioner: kubernetes.io/azure-file
parameters:
  location: eastus
  skuName: Standard_LRS
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

- 1 Specifies the user identifier to use for the mounted directory.
- 2 Specifies the group identifier to use for the mounted directory.
- 3 Enables symlinks.

6.3.7. GCE PersistentDisk (gcePD) object definition

gce-pd-storageclass.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard 1
  replication-type: none
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
reclaimPolicy: Delete
```

- 1 Select either **pd-standard** or **pd-ssd**. The default is **pd-standard**.

6.3.8. VMware vSphere object definition

vsphere-storageclass.yaml

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/vsphere-volume 1
parameters:
  diskformat: thin 2
```

- 1 For more information about using VMware vSphere with OpenShift Container Platform, see the [VMware vSphere documentation](#).
- 2 **diskformat: thin, zeroedthick** and **eagerzeroedthick** are all valid disk formats. See vSphere docs for additional details regarding the disk format types. The default value is **thin**.

6.3.9. Red Hat OpenShift Container Storage object definition

When using Red Hat OpenShift Container Storage, the storage classes for dynamic volume provisioning are created when Red Hat OpenShift Container Storage 4.4 is deployed from the Operator Hub as described in [Verify that the storage classes are created and listed](#).

6.4. CHANGING THE DEFAULT STORAGE CLASS

If you are using AWS, use the following process to change the default storage class. This process assumes you have two storage classes defined, **gp2** and **standard**, and you want to change the default storage class from **gp2** to **standard**.

1. List the storage class:

```
$ oc get storageclass
```

NAME	TYPE
gp2 (default)	kubernetes.io/aws-ebs 1
standard	kubernetes.io/aws-ebs

1 **(default)** denotes the default storage class.

2. Change the value of the annotation **storageclass.kubernetes.io/is-default-class** to **false** for the default storage class:

```
$ oc patch storageclass gp2 -p '{"metadata": {"annotations": {"storageclass.kubernetes.io/is-default-class": "false"}}}'
```

3. Make another storage class the default by adding or modifying the annotation as **storageclass.kubernetes.io/is-default-class=true**.

```
$ oc patch storageclass standard -p '{"metadata": {"annotations": {"storageclass.kubernetes.io/is-default-class": "true"}}}'
```

4. Verify the changes:

```
$ oc get storageclass
```

NAME	TYPE
gp2	kubernetes.io/aws-ebs
standard (default)	kubernetes.io/aws-ebs