



OpenShift Container Platform 4.1

Storage

Configuring and managing storage in OpenShift Container Platform 4.1

OpenShift Container Platform 4.1 Storage

Configuring and managing storage in OpenShift Container Platform 4.1

Legal Notice

Copyright © 2020 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 PERSISTENT STORAGE	4
1.1. PERSISTENT STORAGE OVERVIEW	4
1.2. LIFECYCLE OF A VOLUME AND CLAIM	4
1.2.1. Provision storage	4
1.2.2. Bind claims	4
1.2.3. Use Pods and claimed PVs	5
1.2.4. Storage Object in Use Protection	5
1.2.5. Release volumes	5
1.2.6. Reclaim volumes	5
1.3. PERSISTENT VOLUMES	6
1.3.1. Types of PVs	6
1.3.2. Capacity	7
1.3.3. Access modes	7
1.3.4. Phase	8
1.3.4.1. Mount options	9
1.4. PERSISTENT VOLUME CLAIMS	10
1.4.1. Storage classes	10
1.4.2. Access modes	11
1.4.3. Resources	11
1.4.4. Claims as volumes	11
1.5. BLOCK VOLUME SUPPORT	11
1.5.1. Block volume examples	12
CHAPTER 2. CONFIGURING PERSISTENT STORAGE	15
2.1. PERSISTENT STORAGE USING AWS ELASTIC BLOCK STORE	15
2.1.1. Creating the EBS Storage Class	15
2.1.2. Creating the Persistent Volume Claim	15
2.1.3. Volume Format	16
2.1.4. Maximum Number of EBS Volumes on a Node	16
2.2. PERSISTENT STORAGE USING FIBRE CHANNEL	16
2.2.1. Provisioning	17
2.2.1.1. Enforcing disk quotas	17
2.2.1.2. Fibre Channel volume security	18
2.3. PERSISTENT STORAGE USING NFS	18
2.3.1. Provisioning	18
2.3.2. Enforcing disk quotas	19
2.3.3. NFS volume security	19
2.3.3.1. Group IDs	20
2.3.3.2. User IDs	21
2.3.3.3. SELinux	22
2.3.3.4. Export settings	22
2.3.4. Reclaiming resources	23
2.3.5. Additional configuration and troubleshooting	23
2.4. PERSISTENT STORAGE USING HOSTPATH	24
2.4.1. Overview	24
2.4.2. Statically provisioning hostPath volumes	24
2.4.3. Mounting the hostPath share in a privileged Pod	25
2.5. PERSISTENT STORAGE USING ISCSI	26
2.5.1. Provisioning	27
2.5.2. Enforcing Disk Quotas	27
2.5.3. iSCSI Volume Security	27

2.5.3.1. Challenge Handshake Authentication Protocol (CHAP) configuration	27
2.5.4. iSCSI Multipathing	28
2.5.5. iSCSI Custom Initiator IQN	29
2.6. PERSISTENT STORAGE USING THE CONTAINER STORAGE INTERFACE (CSI)	29
2.6.1. CSI Architecture	30
2.6.1.1. External CSI controllers	30
2.6.1.2. CSI Driver DaemonSet	31
2.6.2. Dynamic Provisioning	31
2.6.3. Example using the CSI driver	32
2.7. PERSISTENT STORAGE USING VMWARE VSPHERE VOLUMES	32
2.7.1. Dynamically provisioning VMware vSphere volumes	32
2.7.1.1. Dynamically provisioning VMware vSphere volumes using the UI	33
2.7.1.2. Dynamically provisioning VMware vSphere volumes using the CLI	33
2.7.2. Statically provisioning VMware vSphere volumes	34
2.7.2.1. Formatting VMware vSphere volumes	35
2.8. PERSISTENT STORAGE USING VOLUME SNAPSHOTS	35
2.8.1. About snapshots	36
2.8.2. External controller and provisioner	36
2.8.2.1. Running the external controller and provisioner	36
2.8.2.2. AWS and GCE authentication	37
2.8.2.2.1. AWS authentication	37
2.8.2.2.2. GCE authentication	39
2.8.2.3. Managing snapshot users	39
2.8.3. Creating and deleting snapshots	40
2.8.3.1. Create snapshot	40
2.8.3.2. Restore snapshot	42
2.8.3.3. Delete snapshot	43
CHAPTER 3. EXPANDING PERSISTENT VOLUMES	44
3.1. ENABLING VOLUME EXPANSION SUPPORT	44
3.2. EXPANDING PERSISTENT VOLUME CLAIMS (PVC) WITH A FILE SYSTEM	44
3.3. RECOVERING FROM FAILURE WHEN EXPANDING VOLUMES	45
CHAPTER 4. DYNAMIC PROVISIONING	46
4.1. ABOUT DYNAMIC PROVISIONING	46
4.2. AVAILABLE DYNAMIC PROVISIONING PLUG-INS	46
4.3. DEFINING A STORAGECLASS	46
4.3.1. Basic StorageClass object definition	47
4.3.2. StorageClass annotations	47
4.3.3. AWS Elastic Block Store (EBS) object definition	48
4.3.4. VMware vSphere object definition	49
4.4. CHANGING THE DEFAULT STORAGECLASS	49

CHAPTER 1. UNDERSTANDING PERSISTENT STORAGE

1.1. 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.

1.2. 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.

1.2.1. 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.

1.2.2. 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 PV volume might exceed your requested volume. 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.

1.2.3. 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.

1.2.4. 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.

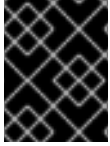
1.2.5. Release volumes

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.

1.2.6. Reclaim volumes

The reclaim policy of a **PersistentVolume** tells the cluster what to do with the volume after it is released. Volumes reclaim policy can either 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.

1.3. PERSISTENT VOLUMES

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

PV object definition example

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001 1
spec:
  capacity:
    storage: 5Gi 2
  accessModes:
    - ReadWriteOnce 3
  persistentVolumeReclaimPolicy: Retain 4
  ...
status:
  ...

```

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

1.3.1. Types of PVs

OpenShift Container Platform supports the following **PersistentVolume** plug-ins:

- AWS Elastic Block Store (EBS)
- Fibre Channel
- HostPath
- iSCSI

- NFS
- VMware vSphere

1.3.2. Capacity

Generally, a PV has a specific storage capacity. This is set by using the PV's **capacity** attribute.

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

1.3.3. Access modes

A **PersistentVolume** 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 1.1. 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

A volume's **AccessModes** are descriptors of the volume's 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 1.2. Supported access modes for PVs

Volume Plug-in	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS	■	-	-
Fibre Channel	■	■	-
HostPath	■	-	-
iSCSI	■	■	-
NFS	■	■	■
VMware vSphere	■	-	-



NOTE

Use a recreate deployment strategy for Pods that rely on AWS EBS.

1.3.4. Phase

Volumes can be found in one of the following phases:

Table 1.3. Volume phases

Phase	Description
Available	A free resource not yet bound to a claim.
Bound	The volume is bound to a claim.

Phase	Description
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>
```

1.3.4.1. 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)
- iSCSI
- NFS
- VMware vSphere

**NOTE**

Fibre Channel and HostPath PVs do not support mount options.

1.4. PERSISTENT VOLUME CLAIMS

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

PVC 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

1.4.1. 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 ClusterStorageOperator may install a default StorageClass depending on the platform in use. This StorageClass 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 StorageClass.

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 StorageClass is marked as default, a PVC can only be created if the **storageClassName** is explicitly specified. Therefore, only one StorageClass should be set as the default.

1.4.2. Access modes

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

1.4.3. 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.

1.4.4. 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

1.5. BLOCK VOLUME SUPPORT

You can statically provision raw block volumes by including API fields in your PV and PVC specifications.

**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 1.4. Block volume support

Volume Plug-in	Manually provisioned	Dynamically provisioned
AWS EBS	■	■
Fibre Channel	■	
HostPath		
iSCSI	■	
NFS		
VMware vSphere	■	■



IMPORTANT

Block volume support 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/>.

1.5.1. 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** field indicating 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** field indicating 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**, similar to **volumeMounts**, is used for block devices and can only be used with **PersistentVolumeClaim** sources.
- 2** **devicePath**, similar to **mountPath**, represents the path to the physical device.
- 3** The volume source must be of type **persistentVolumeClaim** and must match the name of the PVC as expected.

Table 1.5. Accepted values for **VolumeMode**

Value	Default
Filesystem	Yes
Block	No

Table 1.6. 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 2. CONFIGURING PERSISTENT STORAGE

2.1. 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 References

- [Amazon EC2](#)

2.1.1. Creating the EBS Storage Class

StorageClasses 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.

2.1.2. 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.

2.1.3. 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.

2.1.4. Maximum Number of EBS Volumes on a Node

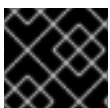
By default, OpenShift Container Platform supports a maximum of 39 EBS volumes attached to one node. This limit is consistent with the [AWS volume limits](#).

OpenShift Container Platform can be configured to have a higher limit by setting the environment variable **KUBE_MAX_PD_VOLS**. However, AWS requires a particular naming scheme ([AWS Device Naming](#)) for attached devices, which only supports a maximum of 52 volumes. This limits the number of volumes that can be attached to a node via OpenShift Container Platform to 52.

2.2. 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. PersistentVolumes are not bound to a single project or namespace; they can be shared across the OpenShift Container Platform cluster. PersistentVolumeClaims 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 references

- [Fibre Channel](#)

2.2.1. 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 PersistentVolume 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.

2.2.1.1. Enforcing disk quotas

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

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.

2.2.1.2. Fibre Channel volume security

Users request storage with a PersistentVolumeClaim. 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 PersistentVolume across a namespace causes the pod to fail.

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

2.3. 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 \(NFS\)](#)

2.3.1. 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
```

2.3.2. 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.

2.3.3. 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.

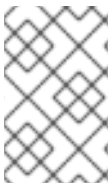
The **/etc/exports** file on the NFS server contains the accessible NFS directories. The target NFS directory has POSIX owner and group IDs. The OpenShift Container Platform NFS plug-in mounts the container's NFS directory with the same POSIX ownership and permissions found on the exported NFS directory. 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.

2.3.3.1. 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, use the **fsGroup** SCC strategy and the **fsGroup** value in the Pod's **securityContext**.



NOTE

It is generally preferable to use supplemental group IDs to gain access to persistent storage versus using 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 Pod's **securityContext** definition. For example:

```
spec:
  containers:
    - name:
      ...
      securityContext: ①
        supplementalGroups: [5555] ②
```

① **securityContext** must be defined at the Pod level, not under a specific container.

- 2 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's 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.

2.3.3.2. 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: 1
  - name:
    ...
    securityContext:
      runAsUser: 65534 2
```

- 1 Pods contain a **securityContext** 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 the **default** project and the **restricted** SCC, the Pod's requested user ID of **65534** is not allowed, and therefore the Pod fails. 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.

2.3.3.3. SELinux

By default, SELinux does not allow writing from a Pod to a remote NFS server. The NFS volume mounts correctly, but is read-only.

To enable writing to a remote NFS server, follow the below 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
```

2.3.3.4. 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.

2.3.4. 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.

2.3.5. 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>

2.4. 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.

2.4.1. 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 StorageClasses to set up dynamic provisioning.

A hostPath volume must be provisioned statically.

2.4.2. 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 PersistentVolumeClaims or Pods.
- 2 Used to bind PersistentVolumeClaim requests to this PersistentVolume.
- 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
```

2.4.3. Mounting the hostPath share in a privileged Pod

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

Prerequisites

- A PersistentVolumeClaim exists that is mapped to the underlying hostPath share.

Procedure

- Create a privileged Pod that mounts the existing PersistentVolumeClaim:

```

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 that has been previously created.

2.5. 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

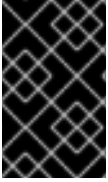
Persistent storage using iSCSI is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to 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/>.



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.

2.5.1. 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.

Example 2.1. Persistent Volume 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'
```

2.5.2. 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.

2.5.3. iSCSI Volume Security

Users request storage with a **PersistentVolumeClaim**. 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.

2.5.3.1. 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 Secrets object must be available in all namespaces that can use the referenced volume.

2.5.4. 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.

2.5.5. 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.

2.6. PERSISTENT STORAGE USING THE CONTAINER STORAGE INTERFACE (CSI)

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



IMPORTANT

Container Storage Interface 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/>.



NOTE

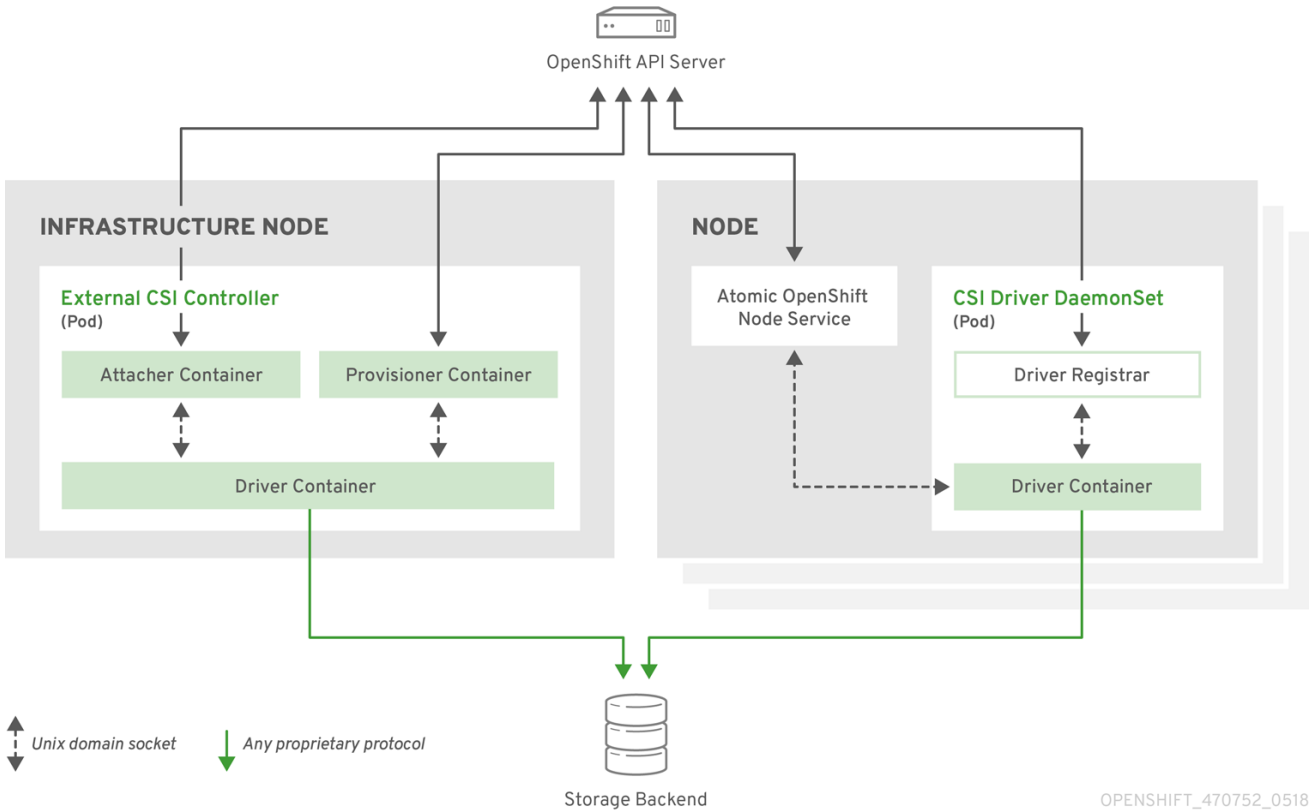
OpenShift Container Platform does not ship with any CSI drivers. It is recommended to use the CSI drivers provided by [community](#) or [storage vendors](#).

OpenShift Container Platform 4.1 supports version 1.0.0 of the [CSI specification](#).

2.6.1. 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 backend 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 backends. Each driver needs its own external controllers' deployment and DaemonSet with the driver and CSI registrar.

2.6.1.1. 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.

2.6.1.2. CSI Driver DaemonSet

The CSI driver DaemonSet 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 backend 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.

2.6.2. Dynamic Provisioning

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

As seen in the OpenStack Cinder example, you can deploy this StorageClass 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: cinder
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: csi-cinderplugin
parameters:
EOF
```

2.6.3. 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 StorageClass 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-3271ffcb4e1811e8  1Gi
RWO           cinder  3s
```

2.7. 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.

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

Additional references

- [VMware vSphere](#)

2.7.1. Dynamically provisioning VMware vSphere volumes

Dynamically provisioning VMware vSphere volumes is the recommended method.

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](#)

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

2.7.1.1. Dynamically provisioning VMware vSphere volumes using the UI

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

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** StorageClass.
 - 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 PersistentVolumeClaim and generate a PersistentVolume.

2.7.1.2. 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 PersistentVolumeClaim.
- 2** The PersistentVolumeClaim's access mode. With **ReadWriteOnce**, the volume can be mounted with read and write permissions by a single node.
- 3** The size of the PersistentVolumeClaim.

2. Create the PersistentVolumeClaim from the file:

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

2.7.2. 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/DatastoreName/volumes/<disk-name>.vmdk
```

- Create using **vmware-diskmanager**:

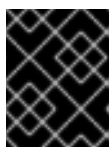
```
$ shell vmware-vdiskmanager -c -t 0 -s <size> -a lsilogic <disk-name>.vmdk
```

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

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv 1
spec:
  capacity:
    storage: 2Gi 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 PersistentVolumeClaims 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. 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 from the file:

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

2.7.2.1. 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.

2.8. PERSISTENT STORAGE USING VOLUME SNAPSHOTS

This document describes how to use VolumeSnapshots to protect against data loss in OpenShift Container Platform. Familiarity with [persistent volumes](#) is suggested.



IMPORTANT

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/>.

2.8.1. About snapshots

A volume snapshot is a snapshot taken from a storage volume in a cluster. The external snapshot controller and provisioner enable use of the feature in the OpenShift Container Platform cluster and handle volume snapshots through the OpenShift Container Platform API.

With volume snapshots, a cluster administrator can:

- Create a snapshot of a PersistentVolume bound to a PersistentVolumeClaim.
- List existing VolumeSnapshots.
- Delete an existing VolumeSnapshot.
- Create a new PersistentVolume from an existing VolumeSnapshot.

Supported PersistentVolume [types](#):

- AWS Elastic Block Store (EBS)
- Google Compute Engine (GCE) Persistent Disk (PD)

2.8.2. External controller and provisioner

The controller and provisioner provide volume snapshotting. These external components run in the cluster.

There are two external components that provide volume snapshotting:

External controller

Creates, deletes, and reports events on volume snapshots.

External provisioner

Creates new PersistentVolumes from VolumeSnapshots.

The external controller and provisioner services are distributed as container images and can be run in the OpenShift Container Platform cluster as usual.

2.8.2.1. Running the external controller and provisioner

The cluster administrator must configure access to run the external controller and provisioner.

Procedure

To allow the containers managing the API objects:

1. Create a ServiceAccount and ClusterRole:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: snapshot-controller-runner
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: snapshot-controller-role
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: ["list", "watch", "create", "update", "patch"]
- apiGroups: ["apiextensions.k8s.io"]
  resources: ["customresourcedefinitions"]
  verbs: ["create", "list", "watch", "delete"]
- apiGroups: ["volumesnapshot.external-storage.k8s.io"]
  resources: ["volumesnapshots"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups: ["volumesnapshot.external-storage.k8s.io"]
  resources: ["volumesnapshotdatas"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

```

- As the cluster administrator, provide the **hostNetwork** security context constraint (SCC):

```
# oc adm policy add-scc-to-user hostnetwork -z snapshot-controller-runner
```

This SCC controls access to the **snapshot-controller-runner** service account that the Pod is using.

- Bind the rules via ClusterRoleBinding:

```

apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: snapshot-controller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: snapshot-controller-role
subjects:
- kind: ServiceAccount
  name: snapshot-controller-runner
  namespace: default 1

```

- Specify the project name where the snapshot-controller resides.

2.8.2.2. AWS and GCE authentication

To authenticate the external controller and provisioner, your cloud provider may require the administrator to provide a secret.

2.8.2.2.1. AWS authentication

If the external controller and provisioner are deployed in Amazon Web Services (AWS), AWS must be able to authenticate using the access key.

To provide the credential to the Pod, the cluster administrator creates a new secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: awskeys
type: Opaque
data:
  access-key-id: <base64 encoded AWS_ACCESS_KEY_ID>
  secret-access-key: <base64 encoded AWS_SECRET_ACCESS_KEY>
```



IMPORTANT

When generating the base64 values required for the **awskeys** secret, remove any trailing newline character as follows:

```
$ echo -n "<aws_access_key_id>" | base64
$ echo -n "<aws_secret_access_key>" | base64
```

The following example displays the AWS deployment of the external controller and provisioner containers. Both Pod containers use the secret to access the AWS API.

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: snapshot-controller
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: snapshot-controller
    spec:
      serviceAccountName: snapshot-controller-runner
      hostNetwork: true
      containers:
        - name: snapshot-controller
          image: "registry.redhat.io/openshift3/snapshot-controller:latest"
          imagePullPolicy: "IfNotPresent"
          args: ["-cloudprovider", "aws"]
          env:
            - name: AWS_ACCESS_KEY_ID
              valueFrom:
                secretKeyRef:
                  name: awskeys
                  key: access-key-id
            - name: AWS_SECRET_ACCESS_KEY
              valueFrom:
                secretKeyRef:
```

```

    name: awskeys
    key: secret-access-key
- name: snapshot-provisioner
  image: "registry.redhat.io/openshift3/snapshot-provisioner:latest"
  imagePullPolicy: "IfNotPresent"
  args: ["-cloudprovider", "aws"]
  env:
  - name: AWS_ACCESS_KEY_ID
    valueFrom:
      secretKeyRef:
        name: awskeys
        key: access-key-id
  - name: AWS_SECRET_ACCESS_KEY
    valueFrom:
      secretKeyRef:
        name: awskeys
        key: secret-access-key

```

2.8.2.2.2. GCE authentication

For Google Compute Engine (GCE), there is no need to use secrets to access the GCE API.

The administrator can proceed with the deployment as shown in the following example:

```

kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: snapshot-controller
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: snapshot-controller
    spec:
      serviceAccountName: snapshot-controller-runner
      containers:
      - name: snapshot-controller
        image: "registry.redhat.io/openshift3/snapshot-controller:latest"
        imagePullPolicy: "IfNotPresent"
        args: ["-cloudprovider", "gce"]
      - name: snapshot-provisioner
        image: "registry.redhat.io/openshift3/snapshot-provisioner:latest"
        imagePullPolicy: "IfNotPresent"
        args: ["-cloudprovider", "gce"]

```

2.8.2.3. Managing snapshot users

Depending on the cluster configuration, it might be necessary to allow non-administrator users to manipulate the VolumeSnapshot objects on the API server. This can be done by creating a ClusterRole bound to a particular user or group.

For example, assume the user "alice" needs to work with snapshots in the cluster. The cluster administrator completes the following steps:

1. Define a new ClusterRole:

```
apiVersion: v1
kind: ClusterRole
metadata:
  name: volumesnapshot-admin
rules:
- apiGroups:
  - "volumesnapshot.external-storage.k8s.io"
  attributeRestrictions: null
  resources:
  - volumesnapshots
  verbs:
  - create
  - delete
  - deletecollection
  - get
  - list
  - patch
  - update
  - watch
```

2. Bind the cluster role to the user "alice" by creating a ClusterRoleBinding object:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: volumesnapshot-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: volumesnapshot-admin
subjects:
- kind: User
  name: alice
```



NOTE

This is only an example of API access configuration. The VolumeSnapshot objects behave similar to other OpenShift Container Platform API objects. See the [API access control documentation](#) for more information on managing the API RBAC.

2.8.3. Creating and deleting snapshots

Similar to how a persistent volume claim (PVC) binds to a persistent volume (PV) to provision a volume, VolumeSnapshotData and VolumeSnapshot are used to create a volume snapshot.

Volume snapshots must use a supported PersistentVolume type.

2.8.3.1. Create snapshot

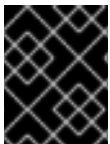
To take a snapshot of a PV, create a new `VolumeSnapshotData` object based on the `VolumeSnapshot`, as shown in the following example:

```
apiVersion: volumesnapshot.external-storage.k8s.io/v1
kind: VolumeSnapshot 1
metadata:
  name: snapshot-demo
spec:
  persistentVolumeClaimName: ebs-pvc 2
```

- 1** A `VolumeSnapshotData` object is automatically created based on the `VolumeSnapshot`.
- 2** **`persistentVolumeClaimName`** is the name of the `PersistentVolumeClaim` bound to a `PersistentVolume`. This particular PV is snapshotted.

Depending on the PV type, the create snapshot operation might go through several phases, which are reflected by the `VolumeSnapshot` status:

1. Create the new `VolumeSnapshot` object.
2. Start the controller. The snapshotted `PersistentVolume` might need to be frozen and the applications paused.
3. Create ("cut") the snapshot. The snapshotted `PersistentVolume` might return to normal operation, but the snapshot itself is not yet ready (status=**True**, type=**Pending**).
4. Create the new `VolumeSnapshotData` object, representing the actual snapshot.
5. The snapshot is complete and ready to use (status=**True**, type=**Ready**).



IMPORTANT

It is the user's responsibility to ensure data consistency (stop the Pod or application, flush caches, freeze the file system, and so on).



NOTE

In case of error, the `VolumeSnapshot` status is appended with an `Error` condition.

To display the `VolumeSnapshot` status:

```
$ oc get volumesnapshot -o yaml
```

The status is displayed, as shown in the following example:

```
apiVersion: volumesnapshot.external-storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  clusterName: ""
  creationTimestamp: 2017-09-19T13:58:28Z
  generation: 0
  labels:
    Timestamp: "1505829508178510973"
  name: snapshot-demo
```

```

namespace: default 1
resourceVersion: "780"
selfLink: /apis/volumesnapshot.external-
storage.k8s.io/v1/namespaces/default/volumesnapshots/snapshot-demo
uid: 9cc5da57-9d42-11e7-9b25-90b11c132b3f
spec:
  persistentVolumeClaimName: ebs-pvc
  snapshotDataName: k8s-volume-snapshot-9cc8813e-9d42-11e7-8bed-90b11c132b3f
status:
  conditions:
  - lastTransitionTime: null
    message: Snapshot created successfully
    reason: ""
    status: "True"
    type: Ready
  creationTimestamp: null

```

- 1** Specify the project name where the snapshot-controller resides.

2.8.3.2. Restore snapshot

A PVC is used to restore a snapshot. But first, the administrator must create a StorageClass to restore a PersistentVolume from an existing VolumeSnapshot.

1. Create a StorageClass:

```

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: snapshot-promoter
provisioner: volumesnapshot.external-storage.k8s.io/snapshot-promoter
parameters: 1
  encrypted: "true"
  type: gp2

```

- 1** If you are using AWS EBS storage with **gp2 encryption** configured, you must set the parameters for **encrypted** and **type**.

2. Create a PVC:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: snapshot-pv-provisioning-demo
  annotations:
    snapshot.alpha.kubernetes.io/snapshot: snapshot-demo 1
spec:
  storageClassName: snapshot-promoter 2
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi 3

```

- 1 The name of the VolumeSnapshot to be restored.
- 2 Created by the administrator for restoring VolumeSnapshots.
- 3 Storage size for a restored snapshot must be large enough to accommodate the original PV size.

A new PersistentVolume is created and bound to the PersistentVolumeClaim. The process might take several minutes depending on the PV type.

2.8.3.3. Delete snapshot

To delete a VolumeSnapshot:

```
$ oc delete volumesnapshot/<snapshot-name>
```

The VolumeSnapshotData bound to the VolumeSnapshot is automatically deleted.

CHAPTER 3. EXPANDING PERSISTENT VOLUMES

3.1. ENABLING VOLUME EXPANSION SUPPORT

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

Procedure

- Edit the StorageClass and add the **allowVolumeExpansion** attribute. The following example demonstrates adding this line at the bottom of the StorageClass's 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.

3.2. EXPANDING PERSISTENT VOLUME CLAIMS (PVC) 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 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 persistent volume 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.

3.3. 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 4. DYNAMIC PROVISIONING

4.1. 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.

4.2. 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
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.
VMware vSphere	kubernetes.io/vsphere-volume	

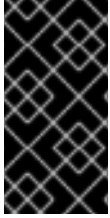


IMPORTANT

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

4.3. DEFINING A STORAGECLASS

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



IMPORTANT

The ClusterStorageOperator may install a default StorageClass depending on the platform in use. This StorageClass 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 StorageClass.

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

4.3.1. Basic StorageClass object definition

The following resource shows the parameters and default values that you use to configure a StorageClass. 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 StorageClass.
- 4** (optional) Annotations for the StorageClass
- 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.

4.3.2. StorageClass annotations

To set a StorageClass as the cluster-wide default, add the following annotation to your StorageClass's 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 StorageClass.



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 StorageClass description, add the following annotation to your StorageClass's metadata:

```
kubernetes.io/description: My StorageClass Description
```

For example:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    kubernetes.io/description: My StorageClass Description
...
```

4.3.3. 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**.

4.3.4. 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**.

4.4. CHANGING THE DEFAULT STORAGECLASS

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

1. List the StorageClass:

```
$ oc get storageclass

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

- 1 **(default)** denotes the default StorageClass.

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

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

3. Make another StorageClass 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