



OpenShift Container Platform 4.2

Storage

Configuring and managing storage in OpenShift Container Platform 4.2

OpenShift Container Platform 4.2 Storage

Configuring and managing storage in OpenShift Container Platform 4.2

Legal Notice

Copyright © 2019 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	5
1.1. PERSISTENT STORAGE OVERVIEW	5
1.2. LIFECYCLE OF A VOLUME AND CLAIM	5
1.2.1. Provision storage	5
1.2.2. Bind claims	5
1.2.3. Use Pods and claimed PVs	6
1.2.4. Storage Object in Use Protection	6
1.2.5. Release volumes	6
1.2.6. Reclaim volumes	6
1.3. PERSISTENT VOLUMES	7
1.3.1. Types of PVs	7
1.3.2. Capacity	8
1.3.3. Access modes	8
1.3.4. Phase	10
1.3.4.1. Mount options	10
1.4. PERSISTENT VOLUME CLAIMS	11
1.4.1. Storage classes	11
1.4.2. Access modes	12
1.4.3. Resources	12
1.4.4. Claims as volumes	12
1.5. BLOCK VOLUME SUPPORT	13
1.5.1. Block volume examples	14
CHAPTER 2. CONFIGURING PERSISTENT STORAGE	17
2.1. PERSISTENT STORAGE USING AWS ELASTIC FILE SYSTEM	17
2.1.1. Store the EFS variables in a ConfigMap	17
2.1.2. Configuring authorization for EFS volumes	18
2.1.3. Create the EFS StorageClass	20
2.1.4. Create the EFS provisioner	20
2.1.5. Create the EFS PersistentVolumeClaim	22
2.2. PERSISTENT STORAGE USING AWS ELASTIC BLOCK STORE	23
2.2.1. Creating the EBS Storage Class	23
2.2.2. Creating the Persistent Volume Claim	24
2.2.3. Volume format	24
2.2.4. Maximum Number of EBS Volumes on a Node	24
2.3. PERSISTENT STORAGE USING AZURE	25
2.3.1. Creating the Azure storage class	25
2.3.2. Creating the Persistent Volume Claim	25
2.3.3. Volume format	26
2.4. PERSISTENT STORAGE USING AZURE FILE	26
2.4.1. Create the Azure File share PersistentVolumeClaim	27
2.4.2. Mount the Azure File share in a Pod	28
2.5. PERSISTENT STORAGE USING CINDER	29
2.5.1. Manual provisioning with Cinder	29
2.5.1.1. Creating the persistent volume	29
2.5.1.2. Persistent volume formatting	30
2.5.1.3. Cinder volume security	30
2.6. PERSISTENT STORAGE USING FIBRE CHANNEL	31
2.6.1. Provisioning	31
2.6.1.1. Enforcing disk quotas	32
2.6.1.2. Fibre Channel volume security	32

2.7. PERSISTENT STORAGE USING GCE PERSISTENT DISK	32
2.7.1. Creating the GCE Storage Class	33
2.7.2. Creating the Persistent Volume Claim	33
2.7.3. Volume format	34
2.8. PERSISTENT STORAGE USING LOCAL VOLUMES	34
2.8.1. Installing the Local Storage Operator	34
2.8.2. Provision the local volumes	35
2.8.3. Create the local volume PersistentVolumeClaim	37
2.8.4. Attach the local claim	38
2.8.5. Deleting the Local Storage Operator's resources	39
2.8.5.1. Removing a local volume	39
2.8.5.2. Uninstalling the Local Storage Operator	40
2.9. PERSISTENT STORAGE USING NFS	41
2.9.1. Provisioning	41
2.9.2. Enforcing disk quotas	43
2.9.3. NFS volume security	43
2.9.3.1. Group IDs	43
2.9.3.2. User IDs	44
2.9.3.3. SELinux	45
2.9.3.4. Export settings	45
2.9.4. Reclaiming resources	46
2.9.5. Additional configuration and troubleshooting	47
2.10. PERSISTENT STORAGE USING ISCSI	47
2.10.1. Provisioning	48
2.10.2. Enforcing Disk Quotas	48
2.10.3. iSCSI Volume Security	48
2.10.3.1. Challenge Handshake Authentication Protocol (CHAP) configuration	49
2.10.4. iSCSI Multipathing	49
2.10.5. iSCSI Custom Initiator IQN	50
2.11. PERSISTENT STORAGE USING THE CONTAINER STORAGE INTERFACE (CSI)	50
2.11.1. CSI Architecture	50
2.11.1.1. External CSI controllers	51
2.11.1.2. CSI Driver DaemonSet	52
2.11.2. Dynamic Provisioning	52
2.11.3. Example using the CSI driver	53
2.12. PERSISTENT STORAGE USING VMWARE VSPHERE VOLUMES	53
2.12.1. Dynamically provisioning VMware vSphere volumes	53
2.12.1.1. Dynamically provisioning VMware vSphere volumes using the UI	53
2.12.1.2. Dynamically provisioning VMware vSphere volumes using the CLI	54
2.12.2. Statically provisioning VMware vSphere volumes	55
2.12.2.1. Formatting VMware vSphere volumes	56
2.13. PERSISTENT STORAGE USING VOLUME SNAPSHOTS	56
2.13.1. About snapshots	56
2.13.2. External controller and provisioner	57
2.13.2.1. Running the external controller and provisioner	57
2.13.2.2. AWS and GCE authentication	58
2.13.2.2.1. AWS authentication	58
2.13.2.2.2. GCE authentication	60
2.13.2.3. Managing snapshot users	60
2.13.3. Creating and deleting snapshots	61
2.13.3.1. Create snapshot	61
2.13.3.2. Restore snapshot	63
2.13.3.3. Delete snapshot	64

CHAPTER 3. EXPANDING PERSISTENT VOLUMES	65
3.1. ENABLING VOLUME EXPANSION SUPPORT	65
3.2. EXPANDING PERSISTENT VOLUME CLAIMS (PVC) WITH A FILE SYSTEM	65
3.3. RECOVERING FROM FAILURE WHEN EXPANDING VOLUMES	66
 CHAPTER 4. DYNAMIC PROVISIONING	 67
4.1. ABOUT DYNAMIC PROVISIONING	67
4.2. AVAILABLE DYNAMIC PROVISIONING PLUG-INS	67
4.3. DEFINING A STORAGECLASS	68
4.3.1. Basic StorageClass object definition	68
4.3.2. StorageClass annotations	69
4.3.3. OpenStack Cinder object definition	69
4.3.4. AWS Elastic Block Store (EBS) object definition	70
4.3.5. Azure Disk object definition	71
4.3.6. Azure File object definition	71
4.3.6.1. Considerations when using Azure File	72
4.3.7. GCE PersistentDisk (gcePD) object definition	73
4.3.8. VMware vSphere object definition	73
4.4. CHANGING THE DEFAULT STORAGECLASS	74

CHAPTER 1. UNDERSTANDING PERSISTENT STORAGE

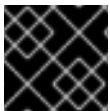
1.1. PERSISTENT STORAGE OVERVIEW

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 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.

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.

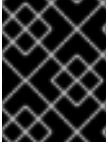
1.2.5. Release volumes

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

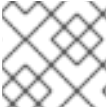
The reclaim policy of a **PersistentVolume** tells the cluster what to do with the volume after it is released. Volumes reclaim policy can either be **Retain**, **Recycle**, or **Delete**.

- **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 ❶
spec:
  capacity:
    storage: 5Gi ❷
  accessModes:
    - ReadWriteOnce ❸
  persistentVolumeReclaimPolicy: Retain ❹
  ...
status:
  ...
```

- ❶ Name of the persistent volume.
- ❷ The amount of storage available to the volume.
- ❸ The access mode, defining the read-write and mount permissions.
- ❹ 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)
- Azure Disk
- Azure File
- Cinder

- Fibre Channel
- GCE Persistent Disk
- 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	■	-	-
Azure File	■	■	■
Azure Disk	■	-	-
Cinder	■	-	-
Fibre Channel	■	■	-
GCE Persistent Disk	■	-	-
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.
Released	The claim was deleted, but the resource is not yet reclaimed by the cluster.
Failed	The volume has failed its automatic reclamation.

The CLI shows the name of the PVC bound to the PV.

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)

- Azure Disk
- Azure File
- Cinder
- GCE Persistent Disk
- 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

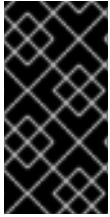
```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim ❶
spec:
  accessModes:
    - ReadWriteOnce ❷
  resources:
    requests:
      storage: 8Gi ❸
  storageClassName: gold ❹
status:
  ...
```

- ❶ Name of the PVC
- ❷ The access mode, defining the read-write and mount permissions
- ❸ The amount of storage available to the PVC
- ❹ 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

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.



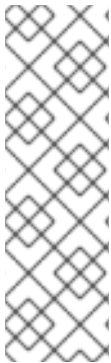
IMPORTANT

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

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

Table 1.4. Block volume support

Volume Plug-in	Manually provisioned	Dynamically provisioned	Fully supported
AWS EBS	■	■	■
Azure Disk	■	■	■
Fibre Channel	■		
GCP	■	■	■
HostPath			
iSCSI	■		
Local Volumes	■	■	■
NFS			
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/>.

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** 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: ❶
    - name: data
      devicePath: /dev/xvda ❷
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: block-pvc ❸

```

- ❶ **volumeDevices**, instead of **volumeMounts**, is used for block devices. Only **PersistentVolumeClaim** sources can be used with raw block volumes.
- ❷ **devicePath**, instead of **mountPath**, represents the path to the physical device where the raw block is mapped to the system.
- ❸ 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

PV VolumeMode	PVC VolumeMode	Binding Result
Block	Unspecified	No Bind
Filesystem	Block	No Bind
Block	Filesystem	No Bind



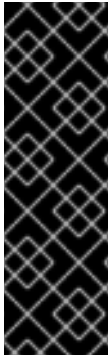
IMPORTANT

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

CHAPTER 2. CONFIGURING PERSISTENT STORAGE

2.1. PERSISTENT STORAGE USING AWS ELASTIC 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 Block Store volumes can be provisioned dynamically. 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.

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 references

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

2.1.1. Store the EFS variables in a ConfigMap

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

Procedure

1. Define an OpenShift Container Platform ConfigMap 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 StorageClass.
- 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>
```

2.1.2. 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
```

2.1.3. Create the EFS StorageClass

Before PersistentVolumeClaims can be created, a StorageClass must exist in the OpenShift Container Platform cluster. The following instructions create the StorageClass for the EFS provisioner.

Procedure

1. Define an OpenShift Container Platform ConfigMap 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
```

2.1.4. 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 ConfigMap that defines the EFS environment variables.
- Create a service account that contains the necessary cluster and role permissions.
- Create a StorageClass 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
```

2.1.5. Create the EFS PersistentVolumeClaim

EFS PersistentVolumeClaims 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 PersistentVolumeClaims 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 StorageClass for the EFS provisioner.

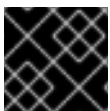
2. After the file has been configured, create it in your cluster by running the following command:

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

2.2. 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 References

- [Amazon EC2](#)

2.2.1. 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.

2.2.2. 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.

2.2.3. 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.

2.2.4. 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](#).

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.

2.3. 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 references

- [Microsoft Azure Disk](#)

2.3.1. Creating the Azure 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.

Additional References

- [Azure Disk Storage Class](#)

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**.
 - e. Enter additional parameters for the storage class as desired.
4. Click **Create** to create the storage class.

2.3.2. 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.

2.3.3. 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.

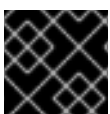
This allows using unformatted Azure volumes as persistent volumes, because OpenShift Container Platform formats them before the first use.

2.4. 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.

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 for use in applications.



IMPORTANT

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

Additional references

- [Azure Files](#)

2.4.1. Create the Azure File share PersistentVolumeClaim

To create the PersistentVolumeClaim, you must first define a Secret that contains the Azure account and key. This Secret is used in the PersistentVolume definition, and will be referenced by the PersistentVolumeClaim 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 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 that references the Secret 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 PersistentVolume.
- 2 The size of this PersistentVolume.
- 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 that maps to the PersistentVolume you created:

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
```

```

metadata:
  name: "claim1" ❶
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "5Gi" ❷
  storageClassName: azure-file-sc ❸
  volumeName: "pv0001" ❹

```

- ❶ The name of the PersistentVolumeClaim.
- ❷ The size of this PersistentVolumeClaim.
- ❸ The name of the StorageClass that is used to provision the PersistentVolume. Specify the StorageClass used in the PersistentVolume definition.
- ❹ The name of the existing PersistentVolume that references the Azure File share.

2.4.2. Mount the Azure File share in a 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 Azure File share.

Procedure

- Create a Pod that mounts the existing PersistentVolumeClaim:

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-name ❶
spec:
  containers:
    ...
    volumeMounts:
      - mountPath: "/data" ❷
        name: azure-file-share
  volumes:
    - name: azure-file-share
      persistentVolumeClaim:
        claimName: claim1 ❸

```

- ❶ The name of the Pod.
- ❷ The path to mount the Azure File share inside the Pod.
- ❸ The name of the PersistentVolumeClaim that has been previously created.

2.5. 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](#).

2.5.1. 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 OpenStack
- Cinder volume ID

2.5.1.1. Creating the persistent volume

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

Procedure

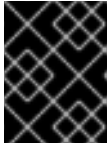
1. Save your object definition to a file.

cinder-persistentvolume.yaml

```
apiVersion: "v1"
kind: "PersistentVolume"
metadata:
  name: "pv0001" 1
spec:
  capacity:
    storage: "5Gi" 2
  accessModes:
    - "ReadWriteOnce"
  cinder: 3
    fsType: "ext3" 4
    volumeID: "f37a03aa-6212-4c62-a805-9ce139fab180" 5
```

- 1 The name of the volume that is used by persistent volume claims or pods.
- 2 The amount of storage allocated to this volume.

- 3 Indicates **cinder** for OpenStack Cinder volumes.
- 4 The file system that is created when the volume is mounted for the first time.
- 5 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
```

2.5.1.2. 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.

2.5.1.3. 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: ❹
    name: frontend ❺
spec:
  containers:
  - image: openshift/hello-openshift
    name: helloworld
    ports:
    - containerPort: 8080
      protocol: TCP
  restartPolicy: Always
  serviceAccountName: <service_account> ❻
  securityContext:
    fsGroup: 7777 ❼

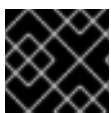
```

- ❶ The number of copies of the Pod to run.
- ❷ The label selector of the Pod to run.
- ❸ A template for the Pod that the controller creates.
- ❹ The labels on the Pod. They must include labels from the label selector.
- ❺ The maximum name length after expanding any parameters is 63 characters.
- ❻ Specifies the service account you created.
- ❼ Specifies an **fsGroup** for the Pods.

2.6. 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. 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](#)

2.6.1. 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 PersistentVolume and a LUN have a one-to-one mapping between them.

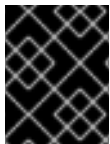
Prerequisites

- 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.

2.6.1.1. 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.

2.6.1.2. 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.

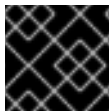
2.7. PERSISTENT STORAGE USING GCE PERSISTENT DISK

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 references

- [GCE Persistent Disk](#)

2.7.1. Creating the GCE 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/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.

2.7.2. 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.

2.7.3. 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.

2.8. 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.

2.8.1. 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.

Procedure

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

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

2. Install the Local Storage Operator from the web console.
 - a. Log in to the OpenShift Container Platform web console.
 - b. Navigate to **Operators → OperatorHub**.
 - c. Type **Local Storage** into the filter box to locate the Local Storage Operator.
 - d. Click **Install**.
 - e. On the **Create Operator Subscription** page, select **A specific namespace on the cluster**. Select **local-storage** from the drop-down menu.
 - f. Adjust the values for the **Update Channel** and **Approval Strategy** to the desired values.
 - g. Click **Subscribe**.
3. Once finished, the Local Storage Operator will be listed in the **Installed Operators** section of the web console.
4. Add the **cluster-admin** role to the ServiceAccount created by the Local Storage Operator, so that this Operator can manage the necessary resources:

```
$ oc adm policy add-cluster-role-to-user cluster-admin system:serviceaccount:local-storage:local-storage-operator
```

2.8.2. Provision the local volumes

Local volumes can not be created by dynamic provisioning. Instead, PersistentVolumes must be created by the Local Storage Operator. This provisioner will look for any devices, both Filesystem 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.

Example: Filesystem

```
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-10-0-140-183
    - ip-10-0-158-139
    - ip-10-0-164-33
  storageClassDevices:
    - storageClassName: "local-sc"
      volumeMode: Filesystem ❸
      fsType: xfs ❹
      devicePaths: ❺
      - /path/to/device

```

- 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 file system that is created when the local volume is mounted for the first time.
- ❺ The path to where the local volumes have been attached to the node.

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
      - /dev/xvdg

```

- 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.

- 3 The volume mode, either **Filesystem** or **Block**, defining the type of the local volumes.
 - 4 The path to where the local volumes have been attached to the node.
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 the provisioner was created, and the corresponding DaemonSets 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-lrvv4	1/1	Running	0	46m
pod/local-disks-local-diskmaker-phxdq	1/1	Running	0	46m
pod/local-storage-operator-54564d9988-vxvvh	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
daemonset.apps/local-disks-local-provisioner	3	3	3	3
daemonset.apps/local-disks-local-diskmaker	3	3	3	3

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 DaemonSet processes. If the desired count is **0**, it indicates the label selectors were invalid.

4. Verify that the PersistentVolumes were created:

```
$ oc get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
local-pv-1cec77cf	100Gi	RWO	Delete	Available	local-sc
local-pv-2ef7cd2a	100Gi	RWO	Delete	Available	local-sc
local-pv-3fa1c73	100Gi	RWO	Delete	Available	local-sc

2.8.3. Create the local volume PersistentVolumeClaim

Local volumes must be statically created as a PersistentVolumeClaim (PVC) to be accessed by the Pod.

Prerequisite

- PersistentVolumes have been created the local volume provisioner.

Procedure

1. Create the PVC using the corresponding StorageClass:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-pvc-name ❶
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem ❷
  resources:
    requests:
      storage: 100Gi ❸
  storageClassName: local-sc ❹
```

- ❶ Name of the PVC.
- ❷ The type of the PVC. Defaults to **Filesystem**.
- ❸ The amount of storage available to the PVC.
- ❹ Name of the StorageClass 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
```

2.8.4. 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
```

2.8.5. Deleting the Local Storage Operator's resources

2.8.5.1. 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 StorageClass, 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 PersistentVolume must be in a **Released** or **Available** state.



WARNING

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

Procedure

1. Edit the previously created LocalVolume 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 PersistentVolumes 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 StorageClass used to create the local volumes.

- d. Delete the symlink belonging to the removed device.

```
$ rm <symlink>
```

2.8.5.2. 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.

Prerequisites

- Access to the OpenShift Container Platform web console.

Procedure

1. 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.
2. Delete the Pods created by the Local Storage Operator:

```
$ oc delete pods --all -n local-storage
```

3. Delete the PersistentVolumes (PV) created by the Local Storage Operator. All of these PVs begin with **local-pv**.

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

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

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

2.9. 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 \(NFS\)](#)

2.9.1. 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>    5368709120  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
```

2.9.2. 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.

2.9.3. 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.

2.9.3.1. 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.

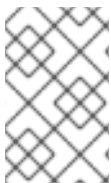
Because the group ID on the example target NFS directory is **5555**, the Pod can define that group ID using **supplementalGroups** under the Pod's **securityContext** definition. For example:

```
spec:
  containers:
    - name:
      ...
      securityContext: 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'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.

2.9.3.2. 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:

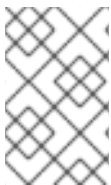
```
spec:
  containers: ❶
    - name:
      ...
      securityContext:
        runAsUser: 65534 ❷
```


- 1 Pods contain a **securityContext** 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.

2.9.3.3. 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
```

2.9.3.4. 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**) and port **111** (**portmapper**).

NFSv4

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

2.9.4. 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.

2.9.5. Additional configuration and troubleshooting

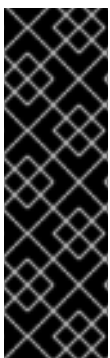
Depending on what version of NFS is being used and how it is configured, there may be additional configuration steps needed for proper export and security mapping. The following are some that may apply:

NFSv4 mount incorrectly shows all files with ownership of nobody:nobody	<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.
Disabling ID mapping on NFSv4	<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>

2.10. 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.



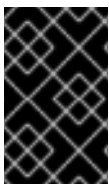
IMPORTANT

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.

2.10.1. 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.

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'
```

2.10.2. 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.

2.10.3. 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.

2.10.3.1. 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 ❶
    chapAuthSession: true ❷
    secretRef:
      name: chap-secret ❸
```

- ❶ Enable CHAP authentication of iSCSI discovery.
- ❷ Enable CHAP authentication of iSCSI session.
- ❸ Specify name of Secrets object with user name + password. This Secrets object must be available in all namespaces that can use the referenced volume.

2.10.4. 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'] ❶
    iqn: iqn.2016-04.test.com:storage.target00
```

```
lun: 0
fsType: ext4
readOnly: false
```

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

2.10.5. 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.

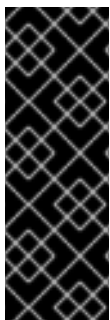
To specify a custom initiator IQN, use **initiatorName** field.

```
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.

2.11. 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

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.

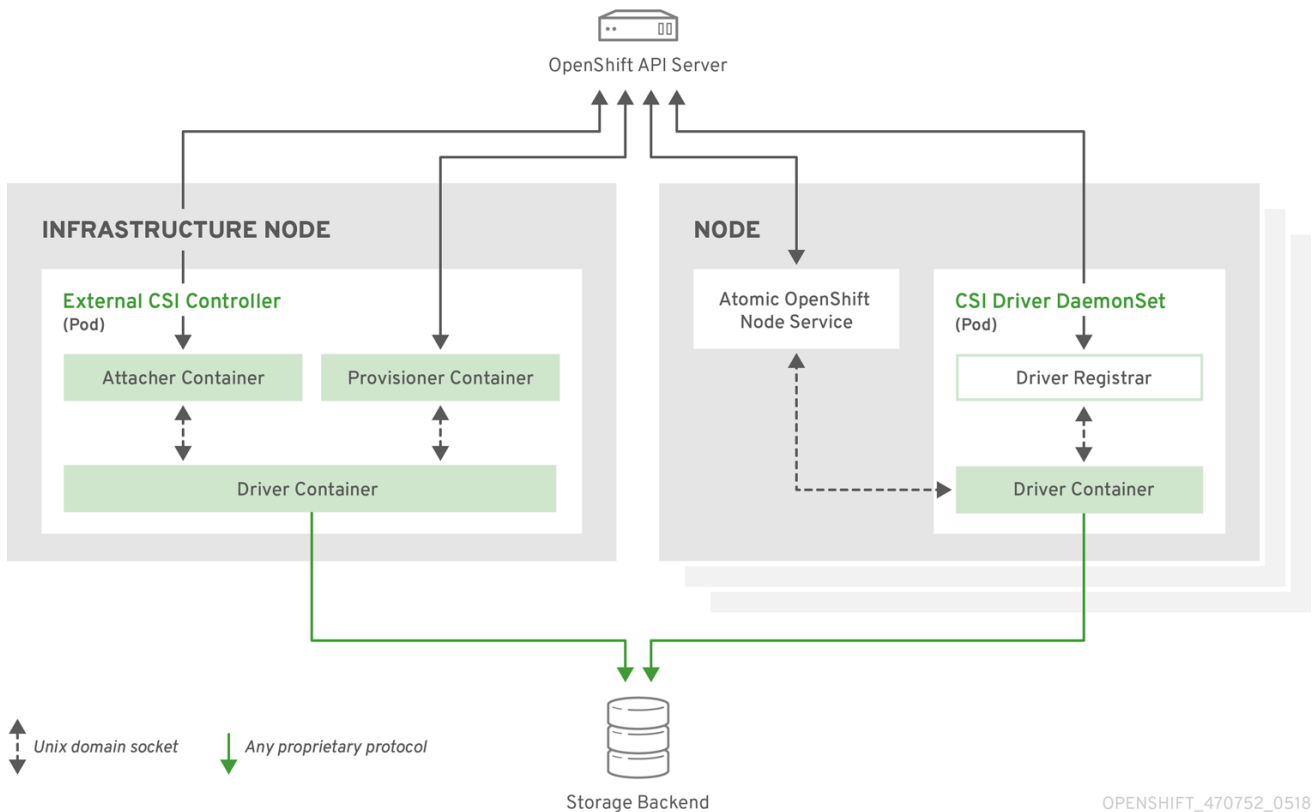
OpenShift Container Platform 4.2 supports version 1.1.0 of the [CSI specification](#).

2.11.1. 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.

2.11.1.1. External CSI controllers

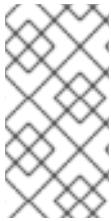
External CSI Controllers is a deployment that deploys one or more pods with three containers:

- 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.

2.11.1.2. 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.

2.11.2. 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.

The created StorageClass 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 StorageClass that will be created.

- 2 The name of the CSI driver that has been installed

2.11.3. 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
```

2.12. 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](#)

2.12.1. Dynamically provisioning VMware vSphere volumes

Dynamically provisioning VMware vSphere volumes is the recommended method. You can use either of the following procedures to dynamically provision these volumes using the default StorageClass.

2.12.1.1. 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.

2.12.1.2. 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
```

2.12.2. 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**:

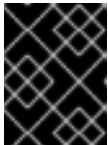
```
$ shell vmware-vdiskmanager -c -t 0 -s <size> -a lsilogic <disk-name>.vmdk
```

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:

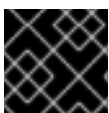
```
$ oc create -f pv.yaml
```

2.12.2.1. 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.

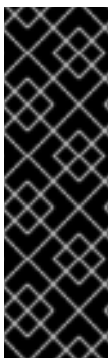
2.13. PERSISTENT STORAGE USING VOLUME SNAPSHOTS



IMPORTANT

Volume snapshot is deprecated in OpenShift Container Platform 4.2.

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/>.

2.13.1. 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)

2.13.2. 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.

2.13.2.1. 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"]

```

2. 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.

3. 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

```

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

2.13.2.2. AWS and GCE authentication

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

2.13.2.2.1. 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

```

2.13.2.2.2. 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"]

```

2.13.2.3. 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 ClusterRole binding 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.

2.13.3. 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.

2.13.3.1. 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.

2.13.3.2. 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.

2.13.3.3. 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.

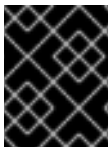
Many storage types are available for use as persistent volumes in OpenShift Container Platform. While all of them can be statically provisioned by an administrator, some types of storage are created dynamically using the built-in provider and plug-in APIs.

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.
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 ServiceAccount 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.

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. OpenStack 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-robin 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**.

4.3.4. 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 value.
- 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**.

4.3.5. Azure Disk object definition

azure-advanced-disk-storageclass.yaml

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  storageAccount: azure_storage_account_name ❶
  storageaccounttype: Standard_LRS ❷
  kind: Dedicated ❸
```

- ❶ Azure storage account name. This must reside in the same resource group as the cluster. If a storage account is specified, the **location** is ignored. If a storage account is not specified, a new storage account gets created in the same resource group as the cluster. If you are specifying a **storageAccount**, the value for **kind** must be **Dedicated**.
- ❷ 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.
- ❸ Possible values are **Shared** (default), **Dedicated**, and **Managed**.
 - 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 a 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.

4.3.6. Azure File object definition

The Azure File StorageClass 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 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> ❶
rules:
- apiGroups: [""]
  resources: ['secrets']
  verbs: ['get','create']
```

- ❶ The name of the ClusterRole to view and create secrets.

2. Add the ClusterRole to the ServiceAccount:

```
$ 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:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: <azure-file> ❶
provisioner: kubernetes.io/azure-file
parameters:
  location: eastus ❷
  skuName: Standard_LRS ❸
  storageAccount: <storage-account> ❹
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

- ❶ Name of the StorageClass. The PersistentVolumeClaim uses this StorageClass for provisioning the associated PersistentVolumes.
- ❷ 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.
- ❸ 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.
- ❹ 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 StorageClass searches for any storage account that is associated with the resource group for any accounts that match the defined **skuName** and **location**.

4.3.6.1. Considerations when using Azure File

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

- 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 to define a specific user identifier to use for the mounted directory.

The following StorageClass 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 ❶
  - gid=1500 ❷
  - myfsymlinks ❸
provisioner: kubernetes.io/azure-file
parameters:
  location: eastus
  skuName: Standard_LRS
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

- ❶ Specifies the user identifier to use for the mounted directory.
- ❷ Specifies the group identifier to use for the mounted directory.
- ❸ Enables symlinks.

4.3.7. GCE PersistentDisk (gcePD) object definition

gce-pd-storageclass.yaml

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard ❶
  replication-type: none
```

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

4.3.8. 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**.

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

