Red Hat build of MicroShift 4.15

Storage

Configuring and managing cluster storage

Last Updated: 2024-05-23

Red Hat build of MicroShift 4.15 Storage

Configuring and managing cluster storage

Legal Notice

Abstract

This document provides information about using storage for MicroShift.

Table of Contents

CHAPTER 1. STORAGE OVERVIEW

MicroShift supports multiple types of storage, both for on-premise and cloud providers. You can manage container storage for persistent and non-persistent data in a Red Hat build of MicroShift cluster.

1.1. STORAGE TYPES

MicroShift storage is broadly classified into two categories, namely ephemeral storage and persistent storage.

1.1.1. Ephemeral storage

Pods and containers are ephemeral or transient in nature and designed for stateless applications. Ephemeral storage allows administrators and developers to better manage the local storage for some of their operations. To read details about ephemeral storage, click Understanding ephemeral storage.

1.1.2. Persistent storage

Stateful applications deployed in containers require persistent storage. MicroShift uses a pre-provisioned storage framework called persistent volumes (PV) to allow cluster administrators to provision persistent storage. The data inside these volumes can exist beyond the lifecycle of an individual pod. Developers can use persistent volume claims (PVCs) to request storage requirements. For persistent storage details, read Understanding persistent storage.

1.1.3. Dynamic storage provisioning

Using dynamic provisioning allows you to create storage volumes on-demand, eliminating the need for pre-provisioned storage. For more information about how dynamic provisioning works in Red Hat build of MicroShift, read Dynamic provisioning.

CHAPTER 2. UNDERSTANDING EPHEMERAL STORAGE

Ephemeral storage is unstructured and temporary. It is often used with immutable applications. This guide discusses how ephemeral storage works for MicroShift.

2.1. OVERVIEW

In addition to persistent storage, pods and containers can require ephemeral or transient local storage for their operation. The lifetime of this ephemeral storage does not extend beyond the life of the individual pod, and this ephemeral storage cannot be shared across pods.

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

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

Unlike persistent volumes, ephemeral storage is unstructured and the space is shared between all pods running on the node, other uses by the system, and Red Hat build of MicroShift. The ephemeral storage framework allows pods to specify their transient local storage needs. It also allows Red Hat build of MicroShift to protect the node against excessive use of local storage.

While the ephemeral storage framework allows administrators and developers to better manage local storage, I/O throughput and latency are not directly effected.

2.2. TYPES OF EPHEMERAL STORAGE

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

Root

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

Runtime

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

2.3. EPHEMERAL STORAGE MANAGEMENT

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

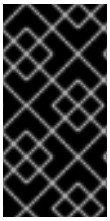
You can manage local ephemeral storage by specifying requests and limits. Each container in a pod can specify the following:

- **spec.containers[].resources.limits.ephemeral-storage**
- **spec.containers[].resources.requests.ephemeral-storage**

2.3.1. Ephemeral storage limits and requests units

Limits and requests for ephemeral storage are measured in byte quantities. You can express storage as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, k. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki.

For example, the following quantities all represent approximately the same value: 128974848, 129e6, 129M, and 123Mi.



IMPORTANT

The suffixes for each byte quantity are case-sensitive. Be sure to use the correct case. Use the case-sensitive "M", such as used in "400M" to set the request at 400 megabytes. Use the case-sensitive "400Mi" to request 400 mebibytes. If you specify "400m" of ephemeral storage, the storage requests is only 0.4 bytes.

2.3.2. Ephemeral storage requests and limits example

The following example configuration file shows a pod with two containers:

- Each container requests 2GiB of local ephemeral storage.
- Each container has a limit of 4GiB of local ephemeral storage.
- At the pod level, kubelet works out an overall pod storage limit by adding up the limits of all the containers in that pod.
  - In this case, the total storage usage at the pod level is the sum of the disk usage from all containers plus the pod's **emptyDir** volumes.
  - Therefore, the pod has a request of 4GiB of local ephemeral storage, and a limit of 8GiB of local ephemeral storage.

Example ephemeral storage configuration with quotas and limits

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        ephemeral-storage: "2Gi" 1
      limits:
        ephemeral-storage: "4Gi" 2
```

```
    volumeMounts:
    - name: ephemeral
      mountPath: "/tmp"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
    volumeMounts:
    - name: ephemeral
      mountPath: "/tmp"
  volumes:
    - name: ephemeral
      emptyDir: {}
```

1. Container request for local ephemeral storage.
2. Container limit for local ephemeral storage.

2.3.3. Ephemeral storage configuration effects pod eviction

The settings in the pod spec affect when kubelet evicts pods. At the container level, because the first container sets a resource limit, kubelet eviction manager measures the disk usage of this container and evicts the pod if the storage usage of the container exceeds its limit (4GiB). The kubelet eviction manager also marks the pod for eviction if the total usage exceeds the overall pod storage limit (8GiB).

NOTE

This policy is strictly for **emptyDir** volumes and is not applied to persistent storage. You can specify the **priorityClass** of pods to exempt the pod from eviction.

2.4. MONITORING EPHEMERAL STORAGE

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

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

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

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

Example output

```
Filesystem  Size  Used Avail Use% Mounted on
/dev/disk/by-partuuid/4cd1448a-01    69G   32G   34G  49% /
```

CHAPTER 3. GENERIC EPHEMERAL VOLUMES

Learn about ephemeral volumes for MicroShift, including their lifecycles, security, and naming.

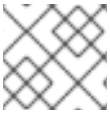
3.1. OVERVIEW

Generic ephemeral volumes are a type of ephemeral volume that can be provided by all storage drivers that support persistent volumes and dynamic provisioning. Generic ephemeral volumes are similar to **emptyDir** volumes in that they provide a per-pod directory for scratch data, which is usually empty after provisioning.

Generic ephemeral volumes are specified inline in the pod spec and follow the pod's lifecycle. They are created and deleted along with the pod.

Generic ephemeral volumes have the following features:

- Storage can be local or network-attached.
- Volumes can have a fixed size that pods are not able to exceed.
- Volumes might have some initial data, depending on the driver and parameters.
- Typical operations on volumes are supported, assuming that the driver supports them, including snapshotting, cloning, resizing, and storage capacity tracking.



NOTE

Generic ephemeral volumes do not support offline snapshots and resize.

3.2. LIFECYCLE AND PERSISTENT VOLUME CLAIMS

The parameters for a volume claim are allowed inside a volume source of a pod. Labels, annotations, and the whole set of fields for persistent volume claims (PVCs) are supported. When such a pod is created, the ephemeral volume controller then creates an actual PVC object (from the template shown in the *Creating generic ephemeral volumes* procedure) in the same namespace as the pod, and ensures that the PVC is deleted when the pod is deleted. This triggers volume binding and provisioning in one of two ways:

- Either immediately, if the storage class uses immediate volume binding.
  With immediate binding, the scheduler is forced to select a node that has access to the volume after it is available.
- When the pod is tentatively scheduled onto a node (**WaitForFirstConsumervolume** binding mode).
  This volume binding option is recommended for generic ephemeral volumes because then the scheduler can choose a suitable node for the pod.

In terms of resource ownership, a pod that has generic ephemeral storage is the owner of the PVCs that provide that ephemeral storage. When the pod is deleted, the Kubernetes garbage collector deletes the PVC, which then usually triggers deletion of the volume because the default reclaim policy of storage classes is to delete volumes. You can create quasi-ephemeral local storage by using a storage class with a reclaim policy of retain: the storage outlives the pod, and in this case, you must ensure that volume clean-up happens separately. While these PVCs exist, they can be used like any other PVC. In particular, they can be referenced as data source in volume cloning or snapshotting. The PVC object also holds the current status of the volume.

3.3. SECURITY

You can enable the generic ephemeral volume feature to allows users who can create pods to also create persistent volume claims (PVCs) indirectly. This feature works even if these users do not have permission to create PVCs directly. Cluster administrators must be aware of this. If this does not fit your security model, use an admission webhook that rejects objects such as pods that have a generic ephemeral volume.

The normal namespace quota for PVCs still applies, so even if users are allowed to use this new mechanism, they cannot use it to circumvent other policies.

3.4. PERSISTENT VOLUME CLAIM NAMING

Automatically created persistent volume claims (PVCs) are named by a combination of the pod name and the volume name, with a hyphen (-) in the middle. This naming convention also introduces a potential conflict between different pods, and between pods and manually created PVCs.

For example, **pod-a** with volume **scratch** and **pod** with volume **a-scratch** both end up with the same PVC name, **pod-a-scratch**.

Such conflicts are detected, and a PVC is only used for an ephemeral volume if it was created for the pod. This check is based on the ownership relationship. An existing PVC is not overwritten or modified, but this does not resolve the conflict. Without the right PVC, a pod cannot start.

IMPORTANT

Be careful when naming pods and volumes inside the same namespace so that naming conflicts do not occur.

3.5. CREATING GENERIC EPHEMERAL VOLUMES

Procedure

1. Create the **pod** object definition and save it to a file.
2. Include the generic ephemeral volume information in the file.

my-example-pod-with-generic-vols.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: my-app
spec:
  containers:
    - name: my-frontend
      image: busybox:1.28
      volumeMounts:
      - mountPath: "/mnt/storage"
        name: data
      command: [ "sleep", "1000000" ]
  volumes:
    - name: data (1)
      ephemeral:
```

```
    volumeClaimTemplate:
      metadata:
        labels:
          type: my-app-ephvol
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: "topolvm-provisioner"
        resources:
          requests:
            storage: 1Gi
```

1 Generic ephemeral volume claim.

CHAPTER 4. UNDERSTANDING PERSISTENT STORAGE

Managing storage is a distinct problem from managing compute resources. MicroShift 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.

4.1. PERSISTENT STORAGE OVERVIEW

PVCs are specific to a namespace, 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 namespace; they can be shared across the entire Red Hat build of MicroShift cluster and claimed from any namespace. 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.

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 plugins 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 LVM, the host filesystem such as hostpath, or raw block devices.

IMPORTANT

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

Like **PersistentVolumes**, **PersistentVolumeClaims** (PVCs) are API objects, 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. Access modes supported by OpenShift Container Platform are also definable in Red Hat build of MicroShift. However, because Red Hat build of MicroShift does not support multi-node deployments, only ReadWriteOnce (RWO) is pertinent.

4.2. ADDITIONAL RESOURCES

- Access modes for persistent storage

4.3. 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.

4.3.1. 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.

4.3.2. 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.

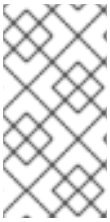
The size of all PVs might exceed your PVC size. This is especially true with manually provisioned PVs. To minimize the excess, Red Hat build of MicroShift binds to the smallest PV that matches all other criteria.

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.

4.3.3. 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.



NOTE

If you attach persistent volumes that have high file counts to pods, those pods can fail or can take a long time to start. For more information, see When using Persistent Volumes with high file counts in OpenShift, why do pods fail to start or take an excessive amount of time to achieve "Ready" state?.

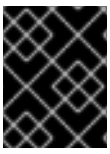
4.3.4. Release a persistent volume

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

4.3.5. Reclaim policy for persistent volumes

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

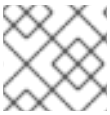
- **Retain** reclaim policy allows manual reclamation of the resource for those volume plugins that support it.
- **Recycle** reclaim policy recycles the volume back into the pool of unbound persistent volumes once it is released from its claim.



IMPORTANT

The **Recycle** reclaim policy is deprecated in Red Hat build of MicroShift 4. Dynamic provisioning is recommended for equivalent and better functionality.

- **Delete** reclaim policy deletes both the **PersistentVolume** object from Red Hat build of MicroShift and the associated storage asset in external infrastructure, such as Amazon Elastic Block Store (Amazon EBS) or VMware vSphere.



NOTE

Dynamically provisioned volumes are always deleted.

4.3.6. Reclaiming a persistent volume manually

When a persistent volume claim (PVC) is deleted, the underlying logical volume is handled according to the **reclaimPolicy**.

Procedure

To manually reclaim the PV as a cluster administrator:

1. Delete the PV.

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

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

2. Clean up the data on the associated storage asset.

3. Delete the associated storage asset. Alternately, to reuse the same storage asset, create a new PV with the storage asset definition.

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

4.3.7. Changing the reclaim policy of a persistent volume

To change the reclaim policy of a persistent volume:

1. List the persistent volumes in your cluster:

```
$ oc get pv
```

   Example output

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

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

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

3. Verify that your chosen persistent volume has the right policy:

```
$ oc get pv
```

   Example output

Example output

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

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

4.4. PERSISTENT VOLUMES

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

PersistentVolume object definition example

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

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

4.4.1. Capacity

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

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

4.4.2. Supported access modes

LVMS is the only CSI plugin Red Hat build of MicroShift supports. The hostPath and LVs built in to OpenShift Container Platform also support RWO.

4.4.3. Phase

Volumes can be found in one of the following phases:

Table 4.1. Volume phases

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

You can view the name of the PVC that is bound to the PV by running the following command:

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

4.4.3.1. Mount options

You can specify mount options while mounting a PV by using the attribute **mountOptions**.

For example:

Mount options example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
  name: topolvm-provisioner
mountOptions:
  - uid=1500
  - gid=1500
parameters:
  csi.storage.k8s.io/fstype: xfs
provisioner: topolvm.io
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
```

NOTE

mountOptions are not validated. Incorrect values will cause the mount to fail and an event to be logged to the PVC.

Additional resources

- Common mount options

4.5. PERSISTENT VOLUME CLAIMS

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

PersistentVolumeClaim object definition example

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

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

4.5.1. 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.

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

NOTE

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

4.5.2. Access modes

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

4.5.3. 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.

4.5.4. Claims as volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the pod 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. Do not mount to the container root, /, or any path that is the same in the host and the container. This can corrupt your host system if the container is sufficiently privileged, such as the host **/dev/pts** files. It is safe to mount the host by using **/host**.
3. Name of the PVC, that exists in the same namespace, to use.

4.6. USING FSGROUP TO REDUCE POD TIMEOUTS

If a storage volume contains many files (~1,000,000 or greater), you may experience pod timeouts.

This can occur because, by default, Red Hat build of MicroShift recursively changes ownership and permissions for the contents of each volume to match the **fsGroup** specified in a pod's **securityContext** when that volume is mounted. For large volumes, checking and changing ownership and permissions can be time consuming, slowing pod startup. You can use the **fsGroupChangePolicy** field inside a **securityContext** to control the way that Red Hat build of MicroShift checks and manages ownership and permissions for a volume.

fsGroupChangePolicy defines behavior for changing ownership and permission of the volume before being exposed inside a pod. This field only applies to volume types that support **fsGroup**-controlled ownership and permissions. This field has two possible values:

- **OnRootMismatch**: Only change permissions and ownership if permission and ownership of root directory does not match with expected permissions of the volume. This can help shorten the time it takes to change ownership and permission of a volume to reduce pod timeouts.
- **Always**: Always change permission and ownership of the volume when a volume is mounted.

fsGroupChangePolicy example

```
securityContext:
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
  fsGroupChangePolicy: "OnRootMismatch" 1
  ...
```

1 **OnRootMismatch** specifies skipping recursive permission change, thus helping to avoid pod timeout problems.

NOTE

The fsGroupChangePolicyfield has no effect on ephemeral volume types, such as secret, configMap, and emptydir.

CHAPTER 5. EXPANDING PERSISTENT VOLUMES

Learn how to expand persistent volumes in MicroShift.

5.1. EXPANDING CSI VOLUMES

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

CSI volume expansion does not support the following:

- Recovering from failure when expanding volumes
- Shrinking

Prerequisites

- The underlying CSI driver supports resize.
- Dynamic provisioning is used.
- The controlling **StorageClass** object has **allowVolumeExpansion** set to **true**. For more information, see "Enabling volume expansion support."

Procedure

1. For the persistent volume claim (PVC), set **.spec.resources.requests.storage** to the desired new size.
2. Watch the **status.conditions** field of the PVC to see if the resize has completed. Red Hat build of MicroShift adds the **Resizing** condition to the PVC during expansion, which is removed after expansion completes.

5.2. EXPANDING LOCAL VOLUMES

You can manually expand persistent volumes (PVs) and persistent volume claims (PVCs) created by using the local storage operator (LSO).

Procedure

1. Expand the underlying devices. Ensure that appropriate capacity is available on these devices.
2. Update the corresponding PV objects to match the new device sizes by editing the **.spec.capacity** field of the PV.
3. For the storage class that is used for binding the PVC to PVet, set **allowVolumeExpansion:true**.
4. For the PVC, set **.spec.resources.requests.storage** to match the new size.

Kubelet should automatically expand the underlying file system on the volume, if necessary, and update the status field of the PVC to reflect the new size.

5.3. EXPANDING PERSISTENT VOLUME CLAIMS (PVCS) WITH A FILE SYSTEM

Expanding PVCs based on volume types that need file system resizing, such as GCE Persistent Disk volumes (gcePD), AWS Elastic Block Store EBS (EBS), and Cinder, is a two-step process. First, expand the volume objects in the cloud provider. Second, expand the file system on the node.

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

Prerequisites

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

Procedure

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

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: ebs
spec:
  storageClass: "storageClassWithFlagSet"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi (1)
```

(1) Updating **spec.resources.requests** to a larger amount expands the PVC.

2. After the cloud provider object has finished resizing, the PVC is set to **FileSystemResizePending**. Check the condition by entering the following command:

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

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

5.4. RECOVERING FROM FAILURE WHEN EXPANDING VOLUMES

If expanding underlying storage fails, the Red Hat build of MicroShift 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.

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.

3. Manually edit the PV and delete the **claimRef** entry from the PV specs to ensure that the newly created PVC can bind to the PV marked **Retain**. This marks the PV as **Available**.

4. Re-create the PVC in a smaller size, or a size that can be allocated by the underlying storage provider.

5. Set the **volumeName** field of the PVC to the name of the PV. This binds the PVC to the provisioned PV only.

6. Restore the reclaim policy on the PV.

CHAPTER 6. DYNAMIC STORAGE USING THE LVMS PLUGIN

MicroShift enables dynamic storage provisioning that is ready for immediate use with the logical volume manager storage (LVMS) Container Storage Interface (CSI) provider. The LVMS plugin is the Red Hat downstream version of TopoLVM, a CSI plugin for managing logical volume management (LVM) logical volumes (LVs) for Kubernetes.

LVMS provisions new LVM logical volumes for container workloads with appropriately configured persistent volume claims (PVCs). Each PVC references a storage class that represents an LVM Volume Group (VG) on the host node. LVs are only provisioned for scheduled pods.

6.1. LVMS SYSTEM REQUIREMENTS

Using LVMS in MicroShift requires the following system specifications.

6.1.1. Volume group name

If you did not configure LVMS in an **lvmd.yaml** file placed in the **/etc/microshift/** directory, MicroShift attempts to assign a default volume group (VG) dynamically by running the **vgs** command.

- MicroShift assigns a default VG when only one VG is found.
- If more than one VG is present, the VG named **microshift** is assigned as the default.
- If a VG named **microshift** does not exist, LVMS is not deployed.

If there are no volume groups on the MicroShift host, LVMS is disabled.

If you want to use a specific VG, LVMS must be configured to select that VG. You can change the default name of the VG in the configuration file. For details, read the "Configuring the LVMS" section of this document.

You can change the default name of the VG in the configuration file. For details, read the "Configuring the LVMS" section of this document.

After MicroShift starts, you can update the **lvmd.yaml** to include or remove VGs. To implement changes, you must restart MicroShift. If the **lvmd.yaml** is deleted, MicroShift attempts to find a default VG again.

6.1.2. Volume size increments

The LVMS provisions storage in increments of 1 gigabyte (GB). Storage requests are rounded up to the nearest GB. When the capacity of a VG is less than 1 GB, the **PersistentVolumeClaim** registers a **ProvisioningFailed** event, for example:

Example output

```
Warning  ProvisioningFailed    3s (x2 over 5s)  topolvm.cybozu.com_topolvm-controller-858c78d96c-xttzp_0fa83aef-2070-4ae2-bcb9-163f818dcd9f failed to provision volume with StorageClass "topolvm-provisioner": rpc error: code = ResourceExhausted desc = no enough space left on VG: free=(BYTES_INT), requested=(BYTES_INT)
```

6.2. LVMS DEPLOYMENT

LVMS is automatically deployed on to the cluster in the **openshift-storage** namespace after MicroShift starts.

LVMS uses **StorageCapacity** tracking to ensure that pods with an LVMS PVC are not scheduled if the requested storage is greater than the free storage of the volume group. For more information about **StorageCapacity** tracking, read Storage Capacity.

6.3. CREATING AN LVMS CONFIGURATION FILE

When MicroShift runs, it uses LVMS configuration from **/etc/microshift/lvmd.yaml**, if provided. You must place any configuration files that you create into the **/etc/microshift/** directory.

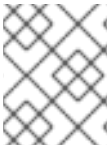
Procedure

- To create the **lvmd.yaml** configuration file, run the following command:

```
$ sudo cp /etc/microshift/lvmd.yaml.default /etc/microshift/lvmd.yaml
```

6.4. BASIC LVMS CONFIGURATION EXAMPLE

MicroShift supports passing through your LVM configuration and allows you to specify custom volume groups, thin volume provisioning parameters, and reserved unallocated volume group space. You can edit the LVMS configuration file you created at any time. You must restart MicroShift to deploy configuration changes after editing the file.



NOTE

If you need to take volume snapshots, you must use thin provisioning in your **lvmd.conf** file. If you do not need to take volume snapshots, you can use thick volumes.

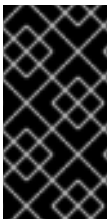
The following **lvmd.yaml** example file shows a basic LVMS configuration:

LVMS configuration example

```
socket-name: 1
device-classes: 2
  - name: "default" 3
    volume-group: "VGNAMEHERE" 4
    spare-gb: 0 5
    default: 6
```

1. String. The UNIX domain socket endpoint of gRPC. Defaults to '/run/lvmd/lvmd.socket'.
2. A list of maps for the settings for each **device-class**.
3. String. The name of the **device-class**.
4. String. The group where the **device-class** creates the logical volumes.
5. Unsigned 64-bit integer. Storage capacity in GB to be left unallocated in the volume group. Defaults to **0**.

6 Boolean. Indicates that the **device-class** is used by default. Defaults to **false**. At least one value must be entered in the YAML file values when this is set to **true**.



IMPORTANT

A race condition prevents LVMS from accurately tracking the allocated space and preserving the **spare-gb** for a device class when multiple PVCs are created simultaneously. Use separate volume groups and device classes to protect the storage of highly dynamic workloads from each other.

6.5. USING THE LVMS

The LVMS **StorageClass** is deployed with a default **StorageClass**. Any **PersistentVolumeClaim** objects without a **.spec.storageClassName** defined automatically has a **PersistentVolume** provisioned from the default **StorageClass**. Use the following procedure to provision and mount a logical volume to a pod.

Procedure

- To provision and mount a logical volume to a pod, run the following command:

```
$ cat <<EOF | oc apply -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-lv-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1G
---
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx
    image: nginx
    command: ["/usr/bin/sh", "-c"]
    args: ["sleep", "1h"]
    volumeMounts:
    - mountPath: /mnt
      name: my-volume
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
          - ALL
      runAsNonRoot: true
      seccompProfile:
        type: RuntimeDefault
```

```
  volumes:
    - name: my-volume
      persistentVolumeClaim:
        claimName: my-lv-pvc
EOF
```

6.5.1. Device classes

You can create custom device classes by adding a **device-classes** array to your logical volume manager storage (LVMS) configuration. Add the array to the **/etc/microshift/lvmd.yaml** configuration file. A single device class must be set as the default. You must restart MicroShift for configuration changes to take effect.

WARNING

Removing a device class while there are still persistent volumes or **VolumeSnapshotContent** objects connected to that device class breaks both thick and thin provisioning.

You can define multiple device classes in the **device-classes** array. These classes can be a mix of thick and thin volume configurations.

Example of a mixed device-class array

```
socket-name: /run/topolvm/lvmd.sock
device-classes:
  - name: ssd
    volume-group: ssd-vg
    spare-gb: 0 (1)
    default: true
  - name: hdd
    volume-group: hdd-vg
    spare-gb: 0
  - name: thin
    spare-gb: 0
    thin-pool:
      name: thin
      overprovision-ratio: 10
    type: thin
    volume-group: ssd
  - name: striped
    volume-group: multi-pv-vg
    spare-gb: 0
    stripe: 2
    stripe-size: "64"
    lvcreate-options: (2)
```

(1) When you set the spare capacity to anything other than **0**, more space can be allocated than expected.

2 Extra arguments to pass to the **lvcreate** command, such as **--type=<type>**. Neither MicroShift nor the LVMS verifies **lvcreate-options** values. These optional values are passed as is to the **lvcreate**

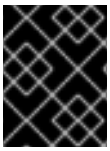
CHAPTER 7. WORKING WITH VOLUME SNAPSHOTS

Cluster administrators can use volume snapshots to help protect against data loss by using the supported MicroShift logical volume manager storage (LVMS) Container Storage Interface (CSI) provider. Familiarity with persistent volumes is required.

A snapshot represents the state of the storage volume in a cluster at a particular point in time. Volume snapshots can also be used to provision new volumes. Snapshots are created as read-only logical volumes (LVs) located on the same device as the original data.

A cluster administrator can complete the following tasks using CSI volume snapshots:

- Create a snapshot of an existing persistent volume claim (PVC).
- Back up a volume snapshot to a secure location.
- Restore a volume snapshot as a different PVC.
- Delete an existing volume snapshot.



IMPORTANT

Only the logical volume manager storage (LVMS) plugin CSI driver is supported by MicroShift.

Additional resources

- Understanding persistent volumes
- Configuring and managing logical volumes
- CSI snapshots: **VolumeSnapshot** APIs
- VolumeSnapshot API specification

7.1. ABOUT LVM THIN VOLUMES

To use advanced storage features such as creating volume snapshots or volume cloning, you must perform the following actions:

- Configure both the logical volume manager storage (LVMS) provider and the cluster.
- Provision a logical volume manager (LVM) thin-pool on the RHEL for Edge host.
- Attach LVM thin-pools to a volume group.

IMPORTANT

To create Container Storage Interface (CSI) snapshots, you must configure thin volumes on the RHEL for Edge host. The CSI does not support volume shrinking.

For LVMS to manage thin logical volumes (LVs), a thin-pool **device-class** array must be specified in the **etc/lvmd.yaml** configuration file. Multiple thin-pool device classes are permitted.

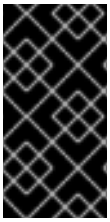
If additional storage pools are configured with device classes, then additional storage classes must also

exist to expose the storage pools to users and workloads. To enable dynamic provisioning on a thin-pool, a **StorageClass** resource must be present on the cluster. The **StorageClass** resource specifies the source **device-class** array in the **topolvm.io/device-class** parameter.

Example lvmd.yaml file that specifies a single device class for a thin-pool

```
socket-name: 1
device-classes: 2
  - name: thin 3
    default: true
    spare-gb: 0 4
    thin-pool:
      name: thin
      overprovision-ratio: 10 5
    type: thin 6
    volume-group: ssd 7
```

1. String. The UNIX domain socket endpoint of gRPC. Defaults to **/run/lvmd/lvmd.socket**.
2. A list of maps for the settings for each **device-class**.
3. String. The unique name of the **device-class**.
4. Unsigned 64-bit integer. Storage capacity in GB to be left unallocated in the volume group. Defaults to **0**.
5. If you have multiple thin-provisioned devices that share the same pool, then these devices can be over-provisioned. Over-provisioning requires a float value of 1 or greater.
6. Thin provisioning is required to create volume snapshots.
7. String. The group where the **device-class** creates the logical volumes.



IMPORTANT

When multiple PVCs are created simultaneously, a race condition prevents LVMS from accurately tracking the allocated space and preserving the storage capacity for a device class. Use separate volume groups and device classes to protect the storage of highly dynamic workloads from each other.

Additional resources

- To create a thin pool on the host, see Creating and managing thin provisioned volumes
- Creating thinly provisioned logical volumes
- Configuring and managing thin provisioned volumes
- Storage classes
- Storage device classes

7.1.1. Storage classes

Storage classes provide the workload layer interface for selecting a device class. The following storage class parameters are supported in MicroShift:

- The **csi.storage.k8s.io/fstype** parameter selects the file system types. Both **xfs** and **ext4** file system types are supported.
- The **topolvm.io/device-class** parameter is the name of the device class. If a device class is not provided, the default device class is assumed.

Multiple storage classes can refer to the same device class. You can provide varying sets of parameters for the same backing device class, such as **xfs** and **ext4** variants.

Example MicroShift default storage class resource

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" (1)
  name: topolvm-provisioner
parameters:
  "csi.storage.k8s.io/fstype": "xfs" (2)
provisioner: topolvm.io (3)
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer (4)
allowVolumeExpansion: (5)
```

(1) An example of the default storage class. If a PVC does not specify a storage class, this class is assumed. There can only be one default storage class in a cluster. Having no value assigned to this annotation is also supported.

(2) Specifies which file system to provision on the volume. Options are "xfs" and "ext4".

(3) Identifies which provisioner should manage this class.

(4) Specifies whether to provision the volume before a client pod is present or immediately. Options are **WaitForFirstConsumer** and **Immediate**. **WaitForFirstConsumer** is recommended to ensure that storage is only provisioned for pods that can be scheduled.

(5) Specifies if PVCs provisioned from the **StorageClass** permit expansion. The MicroShift LVMS CSI plugin does support volume expansion, but if this value is set to **false**, expansion is blocked.

Additional resources

- Defining storage classes
- Storage device classes

7.2. VOLUME SNAPSHOT CLASSES

Snapshotting is a CSI storage feature supported by LVMS. To enable dynamic snapshotting, at least one **VolumeSnapshotClass** configuration file must be present on the cluster.

IMPORTANT

You must enable thin logical volumes to take logical volume snapshots.

Example VolumeSnapshotClass configuration file

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: topolvm-snapclass
  annotations:
    snapshot.storage.kubernetes.io/is-default-class: "true" (1)
driver: topolvm.io (2)
deletionPolicy: Delete (3)
```

1. Determines which **VolumeSnapshotClass** configuration file to use when none is specified by **VolumeSnapshot**, which is a request for snapshot of a volume by a user.
2. Identifies which snapshot provisioner should manage the requests for snapshots of a volume by a user for this class.
3. Determines whether **VolumeSnapshotContent** objects and the backing snapshots are kept or deleted when a bound **VolumeSnapshot** is deleted. Valid values are **Retain** or **Delete**.

Additional resources

- OpenShift CSI volume snapshots

7.3. ABOUT VOLUME SNAPSHOTS

You can use volume snapshots with logical volume manager (LVM) thin volumes to help protect against data loss from applications running in a MicroShift cluster. MicroShift only supports the logical volume manager storage (LVMS) Container Storage Interface (CSI) provider.

NOTE

LVMS only supports the **volumeBindingMode** of the storage class being set to **WaitForFirstConsumer**. This setting means the storage volume is not provisioned until a pod is ready to mount it.

Example workload that deploys a single pod and PVC

```
$ oc apply -f - <<EOF
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-claim-thin
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
```

```
      storage: 1Gi
  storageClassName: topolvm-provisioner-thin
---
apiVersion: v1
kind: Pod
metadata:
  name: base
spec:
  containers:
  - command:
     - nginx
     - -g
     - 'daemon off;'
    image: registry.redhat.io/rhel8/nginx-
122@sha256:908ebb0dec0d669caaf4145a8a21e04fdf9ebffbba5fd4562ce5ab388bf41ab2
    name: test-container
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
    volumeMounts:
    - mountPath: /vol
      name: test-vol
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  volumes:
  - name: test-vol
    persistentVolumeClaim:
      claimName: test-claim-thin
EOF
```

7.3.1. Creating a volume snapshot

To create a snapshot of a MicroShift storage volume, you must first configure RHEL for Edge and the cluster. In the following example procedure, the pod that the source volume is mounted to is deleted. Deleting the pod prevents data from being written to it during snapshot creation. Ensuring that no data is being written during a snapshot is crucial to creating a viable snapshot.

Prerequisites

- User has root access to a MicroShift cluster.
- A MicroShift cluster is running.
- A device class defines an LVM thin-pool.
- A **volumeSnapshotClass** specifies **driver: topolvm.io**.
- Any workload attached to the source PVC is paused or deleted. This helps avoid data corruption.

IMPORTANT

All writes to the volume must be halted while you are creating the snapshot. If you do not halt writes, your data might be corrupted.

Procedure

1. Prevent data from being written to the volume during snapshotting by using one of the two following steps:

   a. Delete the pod to ensure that no data is written to the volume during snapshotting by running the following command:

   ```
   $ oc delete my-pod
   ```

   b. Scale the replica count to zero on a pod that is managed with a replication controller. Setting the count to zero prevents the instant creation of a new pod when one is deleted.

2. After all writes to the volume are halted, run a command similar to the example that follows. Insert your own configuration details.

   #### Example snapshot configuration

   ```
   # oc apply -f <<EOF
   apiVersion: snapshot.storage.k8s.io/v1
   kind: VolumeSnapshot (1)
   metadata:
     name: <snapshot_name> (2)
   spec:
     volumeSnapshotClassName: topolvm-snapclass (3)
     source:
       persistentVolumeClaimName: test-claim-thin (4)
   EOF
   ```

   (1) Create a **VolumeSnapshot** object.

   (2) The name that you specify for the snapshot.

   (3) Specify the desired name of the **VolumeSnapshotClass** object.

   (4) Specify either **persistentVolumeClaimName** or **volumeSnapshotContentName**. In this example, a snapshot is created from a PVC named **test-claim-thin**.

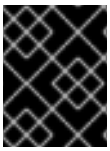
3. Wait for the storage driver to finish creating the snapshot by running the following command:

   ```
   $ oc wait volumesnapshot/<snapshot_name> --for=jsonpath\='{.status.readyToUse}=true'
   ```

Next steps

1. When the **volumeSnapshot** object is in a **ReadyToUse** state, you can restore it as a volume for future PVCs. Restart the pod or scale the replica count back up to the desired number.

2. After you have created the volume snapshot, you can remount the source PVC to a new pod.



IMPORTANT

Volume snapshots are located on the same devices as the original data. To use the volume snapshots as backups, move the snapshots to a secure location.

7.3.2. Backing up a volume snapshot

Snapshots of data from applications running on a MicroShift cluster are created as read-only logical volumes (LVs) located on the same devices as the original data. You must manually mount local volumes before they can be copied as persistent volumes (PVs) and used as backup copies. To use a snapshot of a MicroShift storage volume as a backup, find it on the local host and then move it to a secure location.

To find specific snapshots and copy them, use the following procedure.

Prerequisites

- You have root access to the host machine.
- You have an existing volume snapshot.

Procedure

1. Get the name of the volume snapshot by running the following command:

```
$ oc get volumesnapshot -n <namespace> <snapshot_name> -o 'jsonpath=
{.status.volumeSnapshotContentName}'
```

2. Get the unique identity of the volume created on the storage backend by using the following command and inserting the name retrieved in the previous step:

```
$ oc get volumesnapshotcontent snapcontent-<retrieved_volume_identity> -o 'jsonpath=
{.status.snapshotHandle}'
```

3. Display the snapshots by using the unique identity of the volume you retrieved in the previous step to determine which one you want to backup by running the following command:

```
$ sudo lvdisplay <retrieved_snapshot_handle>
```

Example output

```
--- Logical volume ---
LV Path                /dev/rhel/732e45ff-f220-49ce-859e-87ccca26b14c
LV Name                732e45ff-f220-49ce-859e-87ccca26b14c
VG Name                rhel
LV UUID                6Ojwc0-YTfp-nKJ3-F9FO-PvMR-Ic7b-LzNGSx
LV Write Access        read only
LV Creation host, time rhel-92.lab.local, 2023-08-07 14:45:26 -0500
LV Pool name           thinpool
LV Thin origin name    a2d2dcdc-747e-4572-8c83-56cd873d3b07
LV Status              available
# open                 0
LV Size                1.00 GiB
Mapped size            1.04%
Current LE             256
```

```
Segments              1
Allocation            inherit
Read ahead sectors    auto
- currently set to    256
Block device          253:11
```

4. Create a directory to use for mounting the LV by running the following command:

```
$ sudo mkdir /mnt/snapshot
```

5. Mount the LV using the device name for the retrieved snapshot handle by running the following command:

```
$ sudo mount /dev/<retrieved_snapshot_handle> /mnt/snapshot
```

6. Copy the files from the mounted location and store them in a secure location by running the following command:

```
$ sudo cp -r /mnt/snapshot <destination>
```

7.3.3. Restoring a volume snapshot

The following workflow demonstrates snapshot restoration. In this example, the verification steps are also given to ensure that data written to a source persistent volume claim (PVC) is preserved and restored on a new PVC.

IMPORTANT

A snapshot must be restored to a PVC of exactly the same size as the source volume of the snapshot. You can resize the PVC after the snapshot is restored successfully if a larger PVC is needed.

Procedure

1. Restore a snapshot by specifying the **VolumeSnapshot** object as the data source in a persistent volume claim by entering the following command:

```
$ oc apply -f <<EOF
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: snapshot-restore
spec:
  accessModes:
  - ReadWriteOnce
  dataSource:
    apiGroup: snapshot.storage.k8s.io
    kind: VolumeSnapshot
    name: my-snap
  resources:
    requests:
      storage: 1Gi
  storageClassName: topolvm-provisioner-thin
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: base
spec:
  containers:
  - command:
      - nginx
     - -g
     - 'daemon off;'
    image: registry.redhat.io/rhel8/nginx-
122@sha256:908ebb0dec0d669caaf4145a8a21e04fdf9ebffbba5fd4562ce5ab388bf41ab2
    name: test-container
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
    volumeMounts:
    - mountPath: /vol
      name: test-vol
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  volumes:
  - name: test-vol
    persistentVolumeClaim:
      claimName: snapshot-restore
EOF
```

Verification

1. Wait for the pod to reach the **Ready** state:

```
$ oc wait --for=condition=Ready pod/base
```

2. When the new pod is ready, verify that the data from your application is correct in the snapshot.

Additional resources

- Restoring a volume snapshot

7.3.4. Deleting a volume snapshot

You can configure how Red Hat build of MicroShift deletes volume snapshots.

Procedure

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

   volumesnapshotclass.yaml

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snap
driver: hostpath.csi.k8s.io
deletionPolicy: Delete (1)
```

(1) When deleting the volume snapshot, if the **Delete** value is set, the underlying snapshot is deleted along with the **VolumeSnapshotContent** object. If the **Retain** value is set, both the underlying snapshot and **VolumeSnapshotContent** object remain.
If the **Retain** value is set and the **VolumeSnapshot** object is deleted without deleting the corresponding **VolumeSnapshotContent** object, the content remains. The snapshot itself is also retained in the storage back end.

2. Delete the volume snapshot by entering the following command:

```
$ oc delete volumesnapshot <volumesnapshot_name>
```

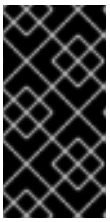
Example output

```
volumesnapshot.snapshot.storage.k8s.io "mysnapshot" deleted
```

3. If the deletion policy is set to **Retain**, delete the volume snapshot content by entering the following command:

```
$ oc delete volumesnapshotcontent <volumesnapshotcontent_name>
```

4. Optional: If the **VolumeSnapshot** object is not successfully deleted, enter the following command to remove any finalizers for the leftover resource so that the delete operation can continue:



IMPORTANT

Only remove the finalizers if you are confident that there are no existing references from either persistent volume claims or volume snapshot contents to the **VolumeSnapshot** object. Even with the **--force** option, the delete operation does not delete snapshot objects until all finalizers are removed.

```
$ oc patch -n $PROJECT volumesnapshot/$NAME --type=merge -p '{"metadata":
{"finalizers":null}}'
```

Example output

```
volumesnapshotclass.snapshot.storage.k8s.io "csi-ocs-rbd-snapclass" deleted
```

The finalizers are removed and the volume snapshot is deleted.

7.4. ABOUT LVM VOLUME CLONING

The logical volume manager storage (LVMS) supports persistent volume claim (PVC) cloning for logical

volume manager (LVM) thin volumes. A clone is a duplicate of an existing volume that can be used like any other volume. When provisioned, an exact duplicate of the original volume is created if the data source references a source PVC in the same namespace. After a cloned PVC is created, it is considered a new object and completely separate from the source PVC. The clone represents the data from the source at the moment in time it was created.

NOTE

Cloning is only possible when the source and destination PVCs are in the same namespace. To create PVC clones, you must configure thin volumes on the RHEL for Edge host.

Additional resources

- CSI volume cloning
- LVMS volume cloning for Single-Node OpenShift
- To configure the host to enable cloning, see About LVM thin volumes

CHAPTER 8. STORAGE MIGRATION USING THE KUBE STORAGE VERSION MIGRATOR

Storage version migration is used to update existing objects in the cluster from their current version to the latest version. The Kube Storage Version Migrator embedded controller is used in MicroShift to migrate resources without having to recreate those resources. Either you or a controller can create a **StorageVersionMigration** custom resource (CR) that will request a migration through the Migrator Controller.

8.1. MAKING A STORAGE MIGRATION REQUEST

Storage migration is the process of updating stored data to the latest storage version, for example from **v1beta1** to **v1beta2**. To update your storage version, use the following procedure.

Procedure

- Either you or any controller that has support for the **StorageVersionMigration** API must trigger a migration request. Use the following example request for reference:

  #### Example request

  ```
  apiVersion: migration.k8s.io/v1alpha1
  kind: StorageVersionMigration
  metadata:
    name: snapshot-v1
  spec:
    resource:
      group: snapshot.storage.k8s.io
      resource: volumesnapshotclasses (1)
      version: v1 (2)
  ```

  (1) You must use the plural name of the resource.

  (2) Version being updated to.

- The progress of the migration is posted to the **StorageVersionMigration** status.

NOTE

- Failures can occur because of a misnamed group or resource.
- Migration failures can also occur when there is an incompatibility between the previous and latest versions.