Persistent Volumes

This document describes *persistent volumes* in Kubernetes. Familiarity with volumes, StorageClasses and VolumeAttributesClasses is suggested.

Introduction

Managing storage is a distinct problem from managing compute instances. The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: PersistentVolume and PersistentVolumeClaim.

A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. 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 independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A *PersistentVolumeClaim* (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany, ReadWriteMany, or ReadWriteOncePod, see AccessModes).

While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the *StorageClass* resource.

See the detailed walkthrough with working examples.

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 follows this lifecycle:

Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

Static

A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses: the PVC must request a storage class and the administrator must have created and configured that class for dynamic provisioning to occur. Claims that request the class `""` effectively disable dynamic provisioning for themselves.

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the `DefaultStorageClass` admission controller on the API server. This can be done, for example, by ensuring that `DefaultStorageClass` is among the comma-delimited, ordered list of values for the `--enable-admission-plugins` flag of the API server component. For more information on API server command-line flags, check kube-apiserver documentation.

Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the control plane watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC.

Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a ClaimRef which is a bi-directional binding between the PersistentVolume and the PersistentVolumeClaim.

Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a `persistentVolumeClaim` section in a Pod's `volumes` block. See Claims As Volumes for more details on this.

Storage Object in Use Protection

The purpose of the Storage Object in Use Protection feature is to ensure that PersistentVolumeClaims (PVCs) in active use by a Pod and PersistentVolume (PVs) that are bound to PVCs are not removed from the system, as this may result in data loss.

> **Note:**
> PVC is in active use by a Pod when a Pod object exists that is using the PVC.

If a user deletes a PVC 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 an 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.

You can see that a PVC is protected when the PVC's status is `Terminating` and the `Finalizers` list includes `kubernetes.io/pvc-protection`:

```
kubectl describe pvc hostpath
Name:          hostpath
Namespace:     default
StorageClass:  example-hostpath
Status:        Terminating
Volume:
Labels:        <none>
Annotations:   volume.beta.kubernetes.io/storage-class=example-hostpath
               volume.beta.kubernetes.io/storage-provisioner=example.com/hostpath
Finalizers:    [kubernetes.io/pvc-protection]
...
```

You can see that a PV is protected when the PV's status is `Terminating` and the `Finalizers` list includes `kubernetes.io/pv-protection` too:

```
kubectl describe pv task-pv-volume
Name:            task-pv-volume
Labels:          type=local
Annotations:     <none>
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    standard
Status:          Terminating
Claim:
Reclaim Policy:  Delete
Access Modes:    RWO
Capacity:        1Gi
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/data
    HostPathType:
Events:            <none>
```

Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be Retained, Recycled, or Deleted.

Retain

The `Retain` reclaim policy allows for manual reclamation of the resource. When the PersistentVolumeClaim is deleted, the PersistentVolume still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume with the following steps.

1. Delete the PersistentVolume. The associated storage asset in external infrastructure still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset.

If you want to reuse the same storage asset, create a new PersistentVolume with the same storage asset definition.

Delete

For volume plugins that support the `Delete` reclaim policy, deletion removes both the PersistentVolume object from Kubernetes, as well as the associated storage asset in the external infrastructure. Volumes that were dynamically provisioned inherit the reclaim policy of their StorageClass, which defaults to `Delete` . The administrator should configure the StorageClass according to users' expectations; otherwise, the PV must be edited or patched after it is created. See Change the Reclaim Policy of a PersistentVolume.

Recycle

> **Warning:**
>
> The `Recycle` reclaim policy is deprecated. Instead, the recommended approach is to use dynamic provisioning.

If supported by the underlying volume plugin, the `Recycle` reclaim policy performs a basic scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim.

However, an administrator can configure a custom recycler Pod template using the Kubernetes controller manager command line arguments as described in the reference. The custom recycler Pod template must contain a `volumes` specification, as shown in the example below:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pv-recycler
  namespace: default
spec:
  restartPolicy: Never
  volumes:
  - name: vol
    hostPath:
      path: /any/path/it/will/be/replaced
  containers:
  - name: pv-recycler
    image: "registry.k8s.io/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]* /scrub/*  && test -z \"$(ls -A /scrub)\" ||
    volumeMounts:
    - name: vol
      mountPath: /scrub
```

However, the particular path specified in the custom recycler Pod template in the `volumes` part is replaced with the particular path of the volume that is being recycled.

PersistentVolume deletion protection finalizer

FEATURE STATE: Kubernetes v1.33 [stable](enabled by default)

Finalizers can be added on a PersistentVolume to ensure that PersistentVolumes having `Delete` reclaim policy are deleted only after the backing storage are deleted.

The finalizer `external-provisioner.volume.kubernetes.io/finalizer` (introduced in v1.31) is added to both dynamically provisioned and statically provisioned CSI volumes.

The finalizer `kubernetes.io/pv-controller` (introduced in v1.31) is added to dynamically provisioned in-tree plugin volumes and skipped for statically provisioned in-tree plugin volumes.

The following is an example of dynamically provisioned in-tree plugin volume:

```
kubectl describe pv pvc-74a498d6-3929-47e8-8c02-078c1ece4d78
Name:            pvc-74a498d6-3929-47e8-8c02-078c1ece4d78
Labels:          <none>
Annotations:     kubernetes.io/createdby: vsphere-volume-dynamic-provisioner
                 pv.kubernetes.io/bound-by-controller: yes
                 pv.kubernetes.io/provisioned-by: kubernetes.io/vsphere-volume
Finalizers:      [kubernetes.io/pv-protection kubernetes.io/pv-controller]
StorageClass:    vcp-sc
Status:          Bound
Claim:           default/vcp-pvc-1
Reclaim Policy:  Delete
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        1Gi
Node Affinity:   <none>
Message:
Source:
    Type:               vSphereVolume (a Persistent Disk resource in vSphere)
    VolumePath:         [vsanDatastore] d49c4a62-166f-ce12-c464-020077ba5d46/kubernetes-dynamic-pvc-74a498d6-3929-47e8-8c02-07
    FSType:             ext4
    StoragePolicyName:  vSAN Default Storage Policy
Events:                 <none>
```

The finalizer `external-provisioner.volume.kubernetes.io/finalizer` is added for CSI volumes. The following is an example:

```
Name:            pvc-2f0bab97-85a8-4552-8044-eb8be45cf48d
Labels:          <none>
Annotations:     pv.kubernetes.io/provisioned-by: csi.vsphere.vmware.com
Finalizers:      [kubernetes.io/pv-protection external-provisioner.volume.kubernetes.io/finalizer]
StorageClass:    fast
Status:          Bound
Claim:           demo-app/nginx-logs
Reclaim Policy:  Delete
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        200Mi
Node Affinity:   <none>
Message:
Source:
    Type:              CSI (a Container Storage Interface (CSI) volume source)
    Driver:            csi.vsphere.vmware.com
    FSType:            ext4
    VolumeHandle:      44830fa8-79b4-406b-8b58-621ba25353fd
    ReadOnly:          false
    VolumeAttributes:      storage.kubernetes.io/csiProvisionerIdentity=1648442357185-8081-csi.vsphere.vmware.com
                           type=vSphere CNS Block Volume
Events:                <none>
```

When the `CSIMigration{provider}` feature flag is enabled for a specific in-tree volume plugin, the `kubernetes.io/pv-controller` finalizer is replaced by the `external-provisioner.volume.kubernetes.io/finalizer` finalizer.

The finalizers ensure that the PV object is removed only after the volume is deleted from the storage backend provided the reclaim policy of the PV is `Delete`. This also ensures that the volume is deleted from storage backend irrespective of the order of deletion of PV and PVC.

Reserving a PersistentVolume

The control plane can bind PersistentVolumeClaims to matching PersistentVolumes in the cluster. However, if you want a PVC to bind to a specific PV, you need to pre-bind them.

By specifying a PersistentVolume in a PersistentVolumeClaim, you declare a binding between that specific PV and PVC. If the PersistentVolume exists and has not reserved PersistentVolumeClaims through its `claimRef` field, then the PersistentVolume and PersistentVolumeClaim will be bound.

The binding happens regardless of some volume matching criteria, including node affinity. The control plane still checks that storage class, access modes, and requested storage size are valid.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: foo-pvc
  namespace: foo
spec:
  storageClassName: "" # Empty string must be explicitly set otherwise default StorageClass will be set
  volumeName: foo-pv
  ...
```

This method does not guarantee any binding privileges to the PersistentVolume. If other PersistentVolumeClaims could use the PV that you specify, you first need to reserve that storage volume. Specify the relevant PersistentVolumeClaim in the `claimRef` field of the PV so that other PVCs can not bind to it.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: foo-pv
spec:
  storageClassName: ""
  claimRef:
    name: foo-pvc
    namespace: foo
  ...
```

This is useful if you want to consume PersistentVolumes that have their `persistentVolumeReclaimPolicy` set to `Retain`, including cases where you are reusing an existing PV.

Expanding Persistent Volumes Claims

ⓘ **FEATURE STATE:** Kubernetes v1.24 [stable]

Support for expanding PersistentVolumeClaims (PVCs) is enabled by default. You can expand the following types of volumes:

- csi (including some CSI migrated volme types)
- flexVolume (deprecated)
- portworxVolume (deprecated)

You can only expand a PVC if its storage class's `allowVolumeExpansion` field is set to true.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: example-vol-default
provisioner: vendor-name.example/magicstorage
parameters:
  resturl: "http://192.168.10.100:8080"
  restuser: ""
  secretNamespace: ""
  secretName: ""
allowVolumeExpansion: true
```

To request a larger volume for a PVC, edit the PVC object and specify a larger size. This triggers expansion of the volume that backs the underlying PersistentVolume. A new PersistentVolume is never created to satisfy the claim. Instead, an existing volume is resized.

> **Warning:**
> Directly editing the size of a PersistentVolume can prevent an automatic resize of that volume. If you edit the capacity of a PersistentVolume, and then edit the `.spec` of a matching PersistentVolumeClaim to make the size of the PersistentVolumeClaim match the PersistentVolume, then no storage resize happens. The Kubernetes control plane will see that the desired state of both resources matches, conclude that the backing volume size has been manually increased and that no resize is necessary.

CSI Volume expansion

ⓘ **FEATURE STATE:** Kubernetes v1.24 [stable]

Support for expanding CSI volumes is enabled by default but it also requires a specific CSI driver to support volume expansion. Refer to documentation of the specific CSI driver for more information.

Resizing a volume containing a file system

You can only resize volumes containing a file system if the file system is XFS, Ext3, or Ext4.

When a volume contains a file system, the file system is only resized when a new Pod is using the PersistentVolumeClaim in `ReadWrite` mode. File system expansion is either done when a Pod is starting up or when a Pod is running and the underlying file system supports online expansion.

FlexVolumes (deprecated since Kubernetes v1.23) allow resize if the driver is configured with the `RequiresFSResize` capability to `true`. The FlexVolume can be resized on Pod restart.

Resizing an in-use PersistentVolumeClaim

ⓘ **FEATURE STATE:** Kubernetes v1.24 [stable]

In this case, you don't need to delete and recreate a Pod or deployment that is using an existing PVC. Any in-use PVC automatically becomes available to its Pod as soon as its file system has been expanded. This feature has no effect on PVCs that are not in use by a Pod or deployment. You must create a Pod that uses the PVC before the expansion can complete.

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

> **Note:**
> FlexVolume resize is possible only when the underlying driver supports resize.

Recovering from Failure when Expanding Volumes

If a user specifies a new size that is too big to be satisfied by underlying storage system, expansion of PVC will be continuously retried until user or cluster administrator takes some action. This can be undesirable and hence Kubernetes provides following methods of recovering from such failures.

Manually with Cluster Administrator access | By requesting expansion to smaller size

If expanding underlying storage fails, the cluster 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.

1. Mark the PersistentVolume(PV) that is bound to the PersistentVolumeClaim(PVC) with `Retain` reclaim policy.
2. Delete the PVC. Since PV has `Retain` reclaim policy - we will not lose any data when we recreate the PVC.
3. Delete the `claimRef` entry from PV specs, so as new PVC can bind to it. This should make the PV `Available`.
4. Re-create the PVC with smaller size than PV and set `volumeName` field of the PVC to the name of the PV. This should bind new PVC to existing PV.
5. Don't forget to restore the reclaim policy of the PV.

Types of Persistent Volumes

PersistentVolume types are implemented as plugins. Kubernetes currently supports the following plugins:

- `csi` - Container Storage Interface (CSI)
- `fc` - Fibre Channel (FC) storage
- `hostPath` - HostPath volume (for single node testing only; WILL NOT WORK in a multi-node cluster; consider using `local` volume instead)
- `iscsi` - iSCSI (SCSI over IP) storage
- `local` - local storage devices mounted on nodes.
- `nfs` - Network File System (NFS) storage

The following types of PersistentVolume are deprecated but still available. If you are using these volume types except for `flexVolume`, `cephfs` and `rbd`, please install corresponding CSI drivers.

- `awsElasticBlockStore` - AWS Elastic Block Store (EBS) (**migration on by default** starting v1.23)
- `azureDisk` - Azure Disk (**migration on by default** starting v1.23)
- `azureFile` - Azure File (**migration on by default** starting v1.24)
- `cinder` - Cinder (OpenStack block storage) (**migration on by default** starting v1.21)
- `flexVolume` - FlexVolume (**deprecated** starting v1.23, no migration plan and no plan to remove support)
- `gcePersistentDisk` - GCE Persistent Disk (**migration on by default** starting v1.23)
- `portworxVolume` - Portworx volume (**migration on by default** starting v1.31)
- `vsphereVolume` - vSphere VMDK volume (**migration on by default** starting v1.25)

Older versions of Kubernetes also supported the following in-tree PersistentVolume types:

- `cephfs` (**not available** starting v1.31)
- `flocker` - Flocker storage. (**not available** starting v1.25)
- `glusterfs` - GlusterFS storage. (**not available** starting v1.26)
- `photonPersistentDisk` - Photon controller persistent disk. (**not available** starting v1.15)
- `quobyte` - Quobyte volume. (**not available** starting v1.25)
- `rbd` - Rados Block Device (RBD) volume (**not available** starting v1.31)
- `scaleIO` - ScaleIO volume. (**not available** starting v1.21)
- `storageos` - StorageOS volume. (**not available** starting v1.25)

Persistent Volumes

Each PV contains a spec and status, which is the specification and status of the volume. The name of a PersistentVolume object must be a valid DNS subdomain name.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

> **Note:**
> Helper programs relating to the volume type may be required for consumption of a PersistentVolume within a cluster. In this example, the PersistentVolume is of type NFS and the helper program /sbin/mount.nfs is required to support the mounting of NFS filesystems.

Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV's `capacity` attribute which is a Quantity value.

Currently, storage size is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

Volume Mode

FEATURE STATE: Kubernetes v1.18 [stable]

Kubernetes supports two `volumeModes` of PersistentVolumes: `Filesystem` and `Block`.

`volumeMode` is an optional API parameter. `Filesystem` is the default mode used when `volumeMode` parameter is omitted.

A volume with `volumeMode: Filesystem` is *mounted* into Pods into a directory. If the volume is backed by a block device and the device is empty, Kubernetes creates a filesystem on the device before mounting it for the first time.

You can set the value of `volumeMode` to `Block` to use a volume as a raw block device. Such volume is presented into a Pod as a block device, without any filesystem on it. This mode is useful to provide a Pod the fastest possible way to access a volume, without any filesystem layer between the Pod and the volume. On the other hand, the application running in the Pod must know how to handle a raw block device. See Raw Block Volume Support for an example on how to use a volume with `volumeMode: Block` in a Pod.

Access Modes

A PersistentVolume can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will 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.

The access modes are:

`ReadWriteOnce`

the volume can be mounted as read-write by a single node. ReadWriteOnce access mode still can allow multiple pods to access (read from or write to) that volume when the pods are running on the same node. For single pod access, please see ReadWriteOncePod.

`ReadOnlyMany`

the volume can be mounted as read-only by many nodes.

`ReadWriteMany`

the volume can be mounted as read-write by many nodes.

`ReadWriteOncePod`

FEATURE STATE: Kubernetes v1.29 [stable]

the volume can be mounted as read-write by a single Pod. Use ReadWriteOncePod access mode if you want to ensure that only one pod across the whole cluster can read that PVC or write to it.

> **Note:**
> The `ReadWriteOncePod` access mode is only supported for CSI volumes and Kubernetes version 1.22+. To use this feature you will need to update the following CSI sidecars to these versions or greater:
>
> - csi-provisioner:v3.0.0+
> - csi-attacher:v3.3.0+
> - csi-resizer:v1.3.0+

In the CLI, the access modes are abbreviated to:

- RWO - ReadWriteOnce
- ROX - ReadOnlyMany
- RWX - ReadWriteMany
- RWOP - ReadWriteOncePod

> **Note:**
> Kubernetes uses volume access modes to match PersistentVolumeClaims and PersistentVolumes. In some cases, the volume access modes also constrain where the PersistentVolume can be mounted. Volume access modes do **not** enforce write protection once the storage has been mounted. Even if the access modes are specified as ReadWriteOnce, ReadOnlyMany, or ReadWriteMany, they don't set any constraints on the volume. For example, even if a PersistentVolume is created as ReadOnlyMany, it is no guarantee that it will be read-only. If the access modes are specified as ReadWriteOncePod, the volume is constrained and can be mounted on only a single Pod.

> **Important!** A volume can only be mounted using one access mode at a time, even if it supports many.

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWriteOncePod
AzureFile	✓	✓	✓	-
CephFS	✓	✓	✓	-
CSI	depends on the driver	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-	-
FlexVolume	✓	✓	depends on the driver	-
HostPath	✓	-	-	-
iSCSI	✓	✓	-	-
NFS	✓	✓	✓	-
RBD	✓	✓	-	-
VsphereVolume	✓	-	- (works when Pods are collocated)	-
PortworxVolume	✓	-	✓	-

Class

A PV can have a class, which is specified by setting the `storageClassName` attribute to the name of a StorageClass. A PV of a particular class can only be bound to PVCs requesting that class. A PV with no `storageClassName` has no class and can only be bound to PVCs that request no particular class.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of the `storageClassName` attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

Reclaim Policy

Current reclaim policies are:

- Retain -- manual reclamation
- Recycle -- basic scrub (`rm -rf /thevolume/*`)
- Delete -- delete the volume

For Kubernetes 1.35, only `nfs` and `hostPath` volume types support recycling.

Mount Options

A Kubernetes administrator can specify additional mount options for when a Persistent Volume is mounted on a node.

> **Note:**
> Not all Persistent Volume types support mount options.

The following volume types support mount options:

- `csi` (including CSI migrated volume types)
- `iscsi`
- `nfs`

Mount options are not validated. If a mount option is invalid, the mount fails.

In the past, the annotation `volume.beta.kubernetes.io/mount-options` was used instead of the `mountOptions` attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

Node Affinity

> **Note:**
> For most volume types, you do not need to set this field. You need to explicitly set this for local volumes.

A PV can specify node affinity to define constraints that limit what nodes this volume can be accessed from. Pods that use a PV will only be scheduled to nodes that are selected by the node affinity. To specify node affinity, set `nodeAffinity` in the `.spec` of a PV. The PersistentVolume API reference has more details on this field.

Updates to node affinity

FEATURE STATE: Kubernetes v1.35 [alpha](disabled by default)

If the `MutablePVNodeAffinity` feature gate is enabled in your cluster, the `.spec.nodeAffinity` field of a PersistentVolume is mutable. This allows cluster administrators or external storage controller to update the node affinity of a PersistentVolume when the data is migrated, without interrupting the running pods.

When updating the node affinity, you should ensure that the new node affinity still matches the nodes where the volume is currently in use. For the pods violating the new affinity, if the pod is already running, it may continue to run. But Kubernetes does not support this configuration. You should terminate the violating pods soon. Due to in memory caching, the pods created after the update may still be scheduled according to the old node affinity for a short period of time.

To use this feature, you should enable the `MutablePVNodeAffinity` feature gate on the following components:

- `kube-apiserver`
- `kubelet`

Phase

A PersistentVolume will be in one of the following phases:

`Available`

a free resource that is not yet bound to a claim

`Bound`

the volume is bound to a claim

`Released`

the claim has been deleted, but the associated storage resource is not yet reclaimed by the cluster

`Failed`

the volume has failed its (automated) reclamation

You can see the name of the PVC bound to the PV using `kubectl describe persistentvolume <name>`.

Phase transition timestamp

ⓘ **FEATURE STATE:** Kubernetes v1.31 [stable](enabled by default)

The `.status` field for a PersistentVolume can include an alpha `lastPhaseTransitionTime` field. This field records the timestamp of when the volume last transitioned its phase. For newly created volumes the phase is set to `Pending` and `lastPhaseTransitionTime` is set to the current time.

PersistentVolumeClaims

Each PVC contains a spec and status, which is the specification and status of the claim. The name of a PersistentVolumeClaim object must be a valid DNS subdomain name.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

Access Modes

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

Volume Modes

Claims use the same convention as volumes to indicate the consumption of the volume as either a filesystem or block device.

Volume Name

Claims can use the `volumeName` field to explicitly bind to a specific PersistentVolume. You can also leave `volumeName` unset, indicating that you'd like Kubernetes to set up a new PersistentVolume that matches the claim. If the specified PV is already bound to another PVC, the binding will be stuck in a pending state.

Resources

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

> **Note:**
> For `Filesystem` volumes, the storage request refers to the "outer" volume size (i.e. the allocated size from the storage backend). This means that the writeable size may be slightly lower for providers that build a filesystem on top of a block device, due to filesystem overhead. This is especially visible with XFS, where many metadata features are enabled by default.

Selector

Claims can specify a label selector to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

- `matchLabels` - the volume must have a label with this value
- `matchExpressions` - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include `In`, `NotIn`, `Exists`, and `DoesNotExist`.

All of the requirements, from both `matchLabels` and `matchExpressions`, are ANDed together – they must all be satisfied in order to match.

Class

A claim can request a particular class by specifying the name of a StorageClass using the attribute `storageClassName`. Only PVs of the requested class, ones with the same `storageClassName` as the PVC, can be bound to the PVC.

PVCs don't necessarily have to request a class. A PVC with its `storageClassName` set equal to `""` is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to `""`). A PVC with no `storageClassName` is not quite the same and is treated differently by the cluster, depending on whether the `DefaultStorageClass` admission plugin is turned on.

- If the admission plugin is turned on, the administrator may specify a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs of that default. Specifying a default StorageClass is done by setting the annotation `storageclass.kubernetes.io/is-default-class` equal to `true` in a StorageClass object. If the administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default StorageClass is specified, the newest default is used when the PVC is dynamically provisioned.
- If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have `storageClassName` set to `""` can be bound only to PVs that have `storageClassName` also set to `""`. However, PVCs with missing `storageClassName` can be updated later once default StorageClass becomes available. If the PVC gets updated it will no longer bind to PVs that have `storageClassName` also set to `""`.

See retroactive default StorageClass assignment for more details.

Depending on installation method, a default StorageClass may be deployed to a Kubernetes cluster by addon manager during installation.

When a PVC specifies a `selector` in addition to requesting a StorageClass, the requirements are ANDed together: only a PV of the requested class and with the requested labels may be bound to the PVC.

> **Note:**
> Currently, a PVC with a non-empty `selector` can't have a PV dynamically provisioned for it.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of `storageClassName` attribute. This annotation is still working; however, it won't be supported in a future Kubernetes release.

Retroactive default StorageClass assignment

ⓘ **FEATURE STATE:** Kubernetes v1.28 [stable]

You can create a PersistentVolumeClaim without specifying a `storageClassName` for the new PVC, and you can do so even when no default StorageClass exists in your cluster. In this case, the new PVC creates as you defined it, and the `storageClassName` of that PVC remains unset until default becomes available.

When a default StorageClass becomes available, the control plane identifies any existing PVCs without `storageClassName`. For the PVCs that either have an empty value for `storageClassName` or do not have this key, the control plane then updates those PVCs to set `storageClassName` to match the new default StorageClass. If you have an existing PVC where the `storageClassName` is `""`, and you configure a default StorageClass, then this PVC will not get updated.

In order to keep binding to PVs with `storageClassName` set to `""` (while a default StorageClass is present), you need to set the `storageClassName` of the associated PVC to `""`.

This behavior helps administrators change default StorageClass by removing the old one first and then creating or setting another one. This brief window while there is no default causes PVCs without `storageClassName` created at that time to not have any default, but due to the retroactive default StorageClass assignment this way of changing defaults is safe.

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 then mounted to the host and into the Pod.

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

A Note on Namespaces

PersistentVolumes binds are exclusive, and since PersistentVolumeClaims are namespaced objects, mounting claims with "Many" modes (`ROX` , `RWX`) is only possible within one namespace.

PersistentVolumes typed `hostPath`

A `hostPath` PersistentVolume uses a file or directory on the Node to emulate network-attached storage. See an example of `hostPath` typed volume.

Raw Block Volume Support

ⓘ **FEATURE STATE:** Kubernetes v1.18 [stable]

The following volume plugins support raw block volumes, including dynamic provisioning where applicable:

- CSI (including some CSI migrated volume types)
- FC (Fibre Channel)
- iSCSI
- Local volume

PersistentVolume using a Raw Block Volume

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

PersistentVolumeClaim requesting a Raw Block Volume

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

Pod specification adding Raw Block Device path in container

```yaml
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:
        - name: data
          devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

> **Note:**
> When adding a raw block device for a Pod, you specify the device path in the container instead of a mount path.

Binding Block Volumes

If a user requests a raw block volume by indicating this using the `volumeMode` field in the PersistentVolumeClaim spec, the binding rules differ slightly from previous releases that didn't consider this mode as part of the spec. Listed is a table of possible combinations the user and admin might specify for requesting a raw block device. The table indicates if the volume will be bound or

not given the combinations: Volume binding matrix for statically provisioned volumes:

PV volumeMode	PVC volumeMode	Result
unspecified	unspecified	BIND
unspecified	Block	NO BIND
unspecified	Filesystem	BIND
Block	unspecified	NO BIND
Block	Block	BIND
Block	Filesystem	NO BIND
Filesystem	Filesystem	BIND
Filesystem	Block	NO BIND
Filesystem	unspecified	BIND

> **Note:**
> Only statically provisioned volumes are supported for alpha release. Administrators should take care to consider these values when working with raw block devices.

Volume Snapshot and Restore Volume from Snapshot Support

ⓘ **FEATURE STATE:** Kubernetes v1.20 [stable]

Volume snapshots only support the out-of-tree CSI volume plugins. For details, see Volume Snapshots. In-tree volume plugins are deprecated. You can read about the deprecated volume plugins in the Volume Plugin FAQ.

Create a PersistentVolumeClaim from a Volume Snapshot

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restore-pvc
spec:
  storageClassName: csi-hostpath-sc
  dataSource:
    name: new-snapshot-test
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Volume Cloning

Volume Cloning only available for CSI volume plugins.

Create PersistentVolumeClaim from an existing PVC

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cloned-pvc
spec:
  storageClassName: my-csi-plugin
  dataSource:
    name: existing-src-pvc-name
    kind: PersistentVolumeClaim
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Volume populators and data sources

FEATURE STATE: Kubernetes v1.24 [beta]

Kubernetes supports custom volume populators. To use custom volume populators, you must enable the `AnyVolumeDataSource` feature gate for the kube-apiserver and kube-controller-manager.

Volume populators take advantage of a PVC spec field called `dataSourceRef`. Unlike the `dataSource` field, which can only contain either a reference to another PersistentVolumeClaim or to a VolumeSnapshot, the `dataSourceRef` field can contain a reference to any object in the same namespace, except for core objects other than PVCs. For clusters that have the feature gate enabled, use of the `dataSourceRef` is preferred over `dataSource`.

Cross namespace data sources

FEATURE STATE: Kubernetes v1.26 [alpha]

Kubernetes supports cross namespace volume data sources. To use cross namespace volume data sources, you must enable the `AnyVolumeDataSource` and `CrossNamespaceVolumeDataSource` feature gates for the kube-apiserver and kube-controller-manager. Also, you must enable the `CrossNamespaceVolumeDataSource` feature gate for the csi-provisioner.

Enabling the `CrossNamespaceVolumeDataSource` feature gate allows you to specify a namespace in the dataSourceRef field.

> **Note:**
> When you specify a namespace for a volume data source, Kubernetes checks for a ReferenceGrant in the other namespace before accepting the reference. ReferenceGrant is part of the `gateway.networking.k8s.io` extension APIs. See ReferenceGrant in the Gateway API documentation for details. This means that you must extend your Kubernetes cluster with at least ReferenceGrant from the Gateway API before you can use this mechanism.

Data source references

The `dataSourceRef` field behaves almost the same as the `dataSource` field. If one is specified while the other is not, the API server will give both fields the same value. Neither field can be changed after creation, and attempting to specify different values for the two fields will result in a validation error. Therefore the two fields will always have the same contents.

There are two differences between the `dataSourceRef` field and the `dataSource` field that users should be aware of:

- The `dataSource` field ignores invalid values (as if the field was blank) while the `dataSourceRef` field never ignores values and will cause an error if an invalid value is used. Invalid values are any core object (objects with no apiGroup) except for PVCs.

- The `dataSourceRef` field may contain different types of objects, while the `dataSource` field only allows PVCs and VolumeSnapshots.

When the `CrossNamespaceVolumeDataSource` feature is enabled, there are additional differences:

- The `dataSource` field only allows local objects, while the `dataSourceRef` field allows objects in any namespaces.
- When namespace is specified, `dataSource` and `dataSourceRef` are not synced.

Users should always use `dataSourceRef` on clusters that have the feature gate enabled, and fall back to `dataSource` on clusters that do not. It is not necessary to look at both fields under any circumstance. The duplicated values with slightly different semantics exist only for backwards compatibility. In particular, a mixture of older and newer controllers are able to interoperate because the fields are the same.

Using volume populators

Volume populators are controllers that can create non-empty volumes, where the contents of the volume are determined by a Custom Resource. Users create a populated volume by referring to a Custom Resource using the `dataSourceRef` field:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: populated-pvc
spec:
  dataSourceRef:
    name: example-name
    kind: ExampleDataSource
    apiGroup: example.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Because volume populators are external components, attempts to create a PVC that uses one can fail if not all the correct components are installed. External controllers should generate events on the PVC to provide feedback on the status of the creation, including warnings if the PVC cannot be created due to some missing component.

You can install the alpha volume data source validator controller into your cluster. That controller generates warning Events on a PVC in the case that no populator is registered to handle that kind of data source. When a suitable populator is installed for a PVC, it's the responsibility of that populator controller to report Events that relate to volume creation and issues during the process.

Using a cross-namespace volume data source

FEATURE STATE: Kubernetes v1.26 [alpha]

Create a ReferenceGrant to allow the namespace owner to accept the reference. You define a populated volume by specifying a cross namespace volume data source using the `dataSourceRef` field. You must already have a valid ReferenceGrant in the source namespace:

```yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: ReferenceGrant
metadata:
  name: allow-ns1-pvc
  namespace: default
spec:
  from:
  - group: ""
    kind: PersistentVolumeClaim
    namespace: ns1
  to:
  - group: snapshot.storage.k8s.io
    kind: VolumeSnapshot
    name: new-snapshot-demo
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: foo-pvc
  namespace: ns1
spec:
  storageClassName: example
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  dataSourceRef:
    apiGroup: snapshot.storage.k8s.io
    kind: VolumeSnapshot
    name: new-snapshot-demo
    namespace: default
  volumeMode: Filesystem
```

Writing Portable Configuration

If you're writing configuration templates or examples that run on a wide range of clusters and need persistent storage, it is recommended that you use the following pattern:

- Include PersistentVolumeClaim objects in your bundle of config (alongside Deployments, ConfigMaps, etc).
- Do not include PersistentVolume objects in the config, since the user instantiating the config may not have permission to create PersistentVolumes.
- Give the user the option of providing a storage class name when instantiating the template.
  - If the user provides a storage class name, put that value into the `persistentVolumeClaim.storageClassName` field. This will cause the PVC to match the right storage class if the cluster has StorageClasses enabled by the admin.
  - If the user does not provide a storage class name, leave the `persistentVolumeClaim.storageClassName` field as nil. This will cause a PV to be automatically provisioned for the user with the default StorageClass in the cluster. Many cluster environments have a default StorageClass installed, or administrators can create their own default StorageClass.
- In your tooling, watch for PVCs that are not getting bound after some time and surface this to the user, as this may indicate that the cluster has no dynamic storage support (in which case the user should create a matching PV) or the cluster has no storage system (in which case the user cannot deploy config requiring PVCs).

What's next

- Learn more about Creating a PersistentVolume.
- Learn more about Creating a PersistentVolumeClaim.
- Read the Persistent Storage design document.

API references

Read about the APIs described in this page:

- `PersistentVolume`
- `PersistentVolumeClaim`

Feedback

Was this page helpful?

Yes No

Last modified October 20, 2025 at 12:16 PM PST: doc for MutablePVNodeAffinity (14eaf08222)