Persistent Volumes

This document describes the current state of *persistent volumes* in Kubernetes. Familiarity with volumes 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 or ReadWriteMany, 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 PersistentVolume**Claim** with a specific amount of storage requested and with certain access modes. A control loop in the master 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, PersistentVolume**Claim** binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a **Claim**Ref which is a bi-directional binding between the PersistentVolume and the PersistentVolume**Claim**.

Claims will remain unbound indefinitely if a matching volume does not exist. **Claim**s 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 **claim**s 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 **claim**ed PVs by including a `persistentVolume**Claim**` section in a Pod's `volumes` block. See **Claim**s 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 PersistentVolume**Claim**s (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 (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) 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, or if you want to reuse the same storage asset, create a new PersistentVolume with the 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, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume. 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: "k8s.gcr.io/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]* /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.

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
  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 `claimPolicy` set to `Retain`, including cases where you are reusing an existing PV.

Expanding Persistent Volumes Claims

FEATURE STATE: `Kubernetes v1.11 [beta]`

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

- gcePersistentDisk
- awsElasticBlockStore
- Cinder
- glusterfs
- rbd
- Azure File
- Azure Disk
- Portworx
- FlexVolumes
- CSI

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

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gluster-vol-default
provisioner: kubernetes.io/glusterfs
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.

CSI Volume expansion

FEATURE STATE: `Kubernetes v1.16 [beta]`

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 allow resize if the driver is set with the `RequiresFSResize` capability to `true`. The FlexVolume can be resized on Pod restart.

Resizing an in-use PersistentVolumeClaim

FEATURE STATE: `Kubernetes v1.15 [beta]`

> **Note:** Expanding in-use PVCs is available as beta since Kubernetes 1.15, and as alpha since 1.11. The `ExpandInUsePersistentVolumes` feature must be enabled, which is the case automatically for many clusters for beta features. Refer to the feature gate documentation for more information.

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.

> **Note:** Expanding EBS volumes is a time-consuming operation. Also, there is a per-volume quota of one modification every 6 hours.

Recovering from Failure when Expanding Volumes

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 PersistentVolume**Claim**(PVC) with `Retain` re**claim** policy.
2. Delete the PVC. Since PV has `Retain` re**claim** 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 re**claim** policy of the PV.

Types of Persistent Volumes

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

- `awsElasticBlockStore` - AWS Elastic Block Store (EBS)
- `azureDisk` - Azure Disk
- `azureFile` - Azure File
- `cephfs` - CephFS volume
- `csi` - Container Storage Interface (CSI)
- `fc` - Fibre Channel (FC) storage
- `flexVolume` - FlexVolume
- `gcePersistentDisk` - GCE Persistent Disk
- `glusterfs` - Glusterfs volume
- `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
- `portworxVolume` - Portworx volume
- `rbd` - Rados Block Device (RBD) volume
- `vsphereVolume` - vSphere VMDK volume

The following types of PersistentVolume are deprecated. This means that support is still available but will be removed in a future Kubernetes release.

- `cinder` - Cinder (OpenStack block storage) (**deprecated** in v1.18)
- `flocker` - Flocker storage (**deprecated** in v1.22)
- `quobyte` - Quobyte volume (**deprecated** in v1.22)
- `storageos` - StorageOS volume (**deprecated** in v1.22)

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

- `photonPersistentDisk` - Photon controller persistent disk. (**not available** after v1.15)
- `scaleIO` - ScaleIO volume (**not available** after v1.21)

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. See the Kubernetes Resource Model to understand the units expected by `capacity`.

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
- ReadOnlyMany -- the volume can be mounted read-only by many nodes
- ReadWriteMany -- the volume can be mounted as read-write by many nodes
- ReadWriteOncePod -- the volume can be mounted as read-write by a single Pod. This is only supported for CSI volumes and Kubernetes version 1.22+.

In the CLI, the access modes are abbreviated to:

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

> **Important!** A volume can only be mounted using one access mode at a time, even if it supports many. For example, a GCEPersistentDisk can be mounted as ReadWriteOnce by a single node or ReadOnlyMany by many nodes, but not at the same time.

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWriteOncePod
AWSElasticBlockStore	✓	-	-	-
AzureFile	✓	✓	✓	-
AzureDisk	✓	-	-	-
CephFS	✓	✓	✓	-
Cinder	✓	-	-	-
CSI	depends on the driver	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-	-
FlexVolume	✓	✓	depends on the driver	-
Flocker	✓	-	-	-
GCEPersistentDisk	✓	✓	-	-

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWriteOncePod
Glusterfs	✓	✓	✓	-
HostPath	✓	-	-	-
iSCSI	✓	✓	-	-
Quobyte	✓	✓	✓	-
NFS	✓	✓	✓	-
RBD	✓	✓	-	-
Vsphere Volume	✓	-	- (works when Pods are collocated)	-
PortworxVolume	✓	-	✓	-
Storage OS	✓	-	-	-

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 -- associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

Currently, only NFS and HostPath support recycling. AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.

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:

- AWSElasticBlockStore
- AzureDisk

- AzureFile
- CephFS
- Cinder (OpenStack block storage)
- GCEPersistentDisk
- Glusterfs
- NFS
- Quobyte Volumes
- RBD (Ceph Block Device)
- StorageOS
- VsphereVolume
- iSCSI

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. It is automatically populated for AWS EBS, GCE PD and Azure Disk volume block types. 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.

Phase

A volume 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 resource is not yet re**claim**ed by the cluster
- Failed -- the volume has failed its automatic reclamation

The CLI will show the name of the PVC bound to the PV.

PersistentVolume**Claim**s

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

```
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.

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 **claim**s.

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 is specified, the admission plugin forbids the creation of all PVCs.
- If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs that have no class. In this case, the PVCs that have no `storageClassName` are treated the same way as PVCs that have their `storageClassName` set to `""`.

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.

Claims As Volumes

Pods access storage by using the **claim** as a volume. **Claim**s 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 PersistentVolume**Claim**s are namespaced objects, mounting **claim**s 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:

- AWSElasticBlockStore
- AzureDisk
- CSI
- FC (Fibre Channel)
- GCEPersistentDisk
- iSCSI
- Local volume
- OpenStack Cinder
- RBD (Ceph Block Device)
- VsphereVolume

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 v1.22 [alpha]`

> **Note:**
> Kubernetes supports custom volume populators; this alpha feature was introduced in Kubernetes 1.18. Kubernetes 1.22 reimplemented the mechanism with a redesigned API. Check that you are reading the version of the Kubernetes documentation that matches your cluster. To check the version, enter `kubectl version`.
>
> 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 PersistentVolume**Claim** 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`.

Data source references

The `dataSourceRef` field behaves almost the same as the `dataSource` field. If either 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.

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.

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 August 04, 2021 at 10:52 PM PST : Link from PV / PVC concept to new API reference (1b3125353)