Volumes

Kubernetes *volumes* provide a way for containers in a pod to access and share data via the filesystem. There are different kinds of volume that you can use for different purposes, such as:

- populating a configuration file based on a ConfigMap or a Secret
- providing some temporary scratch space for a pod
- sharing a filesystem between two different containers in the same pod
- sharing a filesystem between two different pods (even if those Pods run on different nodes)
- durably storing data so that it stays available even if the Pod restarts or is replaced
- passing configuration information to an app running in a container, based on details of the Pod the container is in (for example: telling a sidecar container what namespace the Pod is running in)
- providing read-only access to data in a different container image

Data sharing can be between different local processes within a container, or between different containers, or between Pods.

Why volumes are important

- **Data persistence:** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped, the container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. After a crash, kubelet restarts the container with a clean state.

- **Shared storage:** Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to set up and access a shared filesystem across all of the containers.

The Kubernetes volume abstraction can help you to solve both of these problems.

Before you learn about volumes, PersistentVolumes and PersistentVolumeClaims, you should read up about Pods and make sure that you understand how Kubernetes uses Pods to run containers.

How volumes work

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime linked to a specific Pod, but persistent volumes exist beyond the lifetime of any individual pod. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. For any kind of volume in a given pod, data is preserved across container restarts.

At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, specify the volumes to provide for the Pod in `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[*].volumeMounts`.

When a pod is launched, a process in the container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes are mounted at specified paths within the container filesystem. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses.

Volumes cannot mount within other volumes (but see Using subPath for a related mechanism). Also, a volume cannot contain a hard link to anything in a different volume.

Types of volumes

Kubernetes supports several types of volumes.

awsElasticBlockStore (deprecated)

In Kubernetes 1.35, all operations for the in-tree `awsElasticBlockStore` type are redirected to the `ebs.csi.aws.com` CSI driver.

The AWSElasticBlockStore in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.27 release.

The Kubernetes project suggests that you use the [AWS EBS](#) third party storage driver instead.

azureDisk (deprecated)

In Kubernetes 1.35, all operations for the in-tree `azureDisk` type are redirected to the `disk.csi.azure.com` CSI driver.

The AzureDisk in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.27 release.

The Kubernetes project suggests that you use the [Azure Disk](#) third party storage driver instead.

azureFile (deprecated)

In Kubernetes 1.35, all operations for the in-tree `azureFile` type are redirected to the `file.csi.azure.com` CSI driver.

The AzureFile in-tree storage driver was deprecated in the Kubernetes v1.21 release and then removed entirely in the v1.30 release.

The Kubernetes project suggests that you use the [Azure File](#) third party storage driver instead.

cephfs (removed)

Kubernetes 1.35 does not include a `cephfs` volume type.

The `cephfs` in-tree storage driver was deprecated in the Kubernetes v1.28 release and then removed entirely in the v1.31 release.

cinder (deprecated)

In Kubernetes 1.35, all operations for the in-tree `cinder` type are redirected to the `cinder.csi.openstack.org` CSI driver.

The OpenStack Cinder in-tree storage driver was deprecated in the Kubernetes v1.11 release and then removed entirely in the v1.26 release.

The Kubernetes project suggests that you use the [OpenStack Cinder](#) third party storage driver instead.

configMap

A [ConfigMap](#) provides a way to inject configuration data into pods. The data stored in a ConfigMap can be referenced in a volume of type `configMap` and then consumed by containerized applications running in a pod.

When referencing a ConfigMap, you provide the name of the ConfigMap in the volume. You can customize the path to use for a specific entry in the ConfigMap. The following configuration shows how to mount the `log-config` ConfigMap onto a Pod called `configmap-pod`:
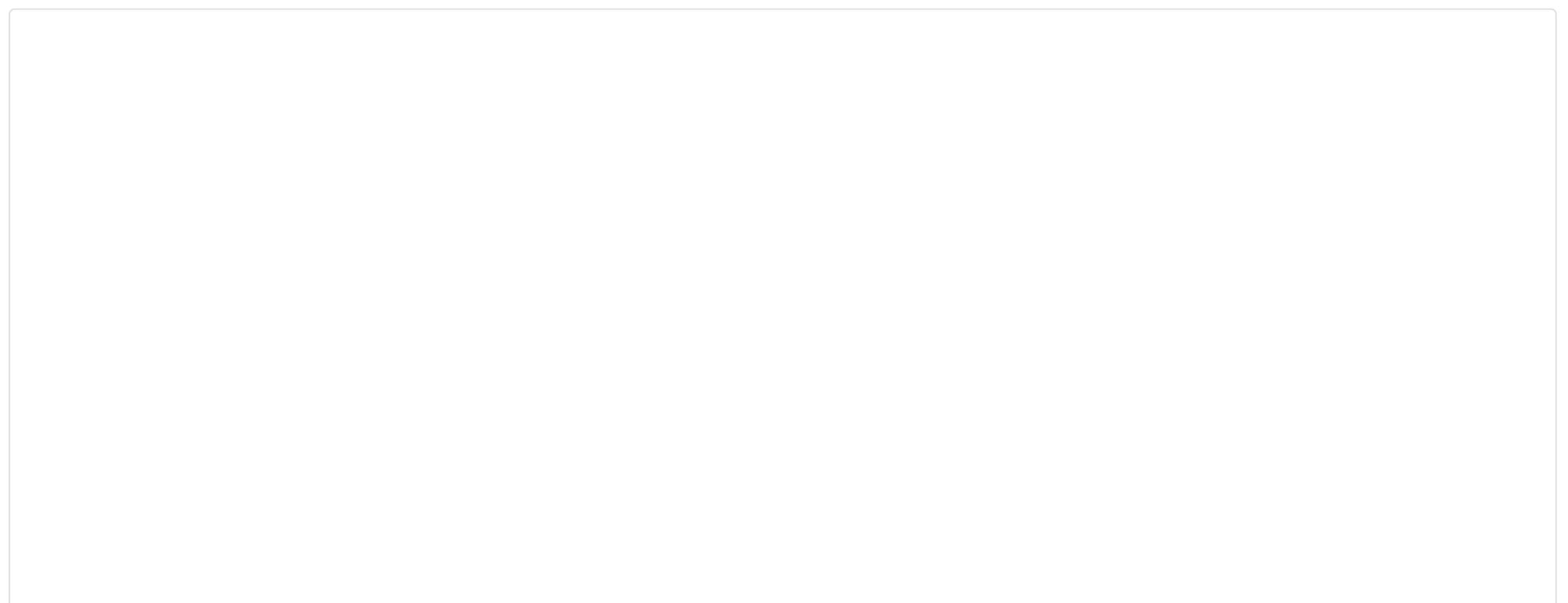
```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox:1.28
      command: ['sh', '-c', 'echo "The app is running!" && tail -f /dev/null']
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level.conf
```

The `log-config` ConfigMap is mounted as a volume, and all contents stored in its `log_level` entry are mounted into the Pod at path `/etc/config/log_level.conf`. Note that this path is derived from the volume's `mountPath` and the `path` keyed with `log_level`.

> **Note:**
> - You must [create a ConfigMap](#) before you can use it.
>
> - A ConfigMap is always mounted as `readOnly`.
>
> - A container using a ConfigMap as a `subPath` volume mount will not receive updates when the ConfigMap changes.
>
> - Text data is exposed as files using the UTF-8 character encoding. For other character encodings, use `binaryData`.

downwardAPI

A `downwardAPI` volume makes downward API data available to applications. Within the volume, you can find the exposed data as read-only files in plain text format.

> **Note:**
> A container using the downward API as a `subPath` volume mount does not receive updates when field values change.

See [Expose Pod Information to Containers Through Files](#) to learn more.

emptyDir

For a Pod that defines an `emptyDir` volume, the volume is created when the Pod is assigned to a node. As the name says, the `emptyDir` volume is initially empty. All containers in the Pod can read and write the same files in the `emptyDir` volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted permanently.

> **Note:**
> A container crashing does *not* remove a Pod from a node. The data in an `emptyDir` volume is safe across container crashes.

Some uses for an `emptyDir` are:

- scratch space, such as for a disk-based merge sort
- checkpointing a long computation for recovery from crashes
- holding files that a content-manager container fetches while a webserver container serves the data

The `emptyDir.medium` field controls where `emptyDir` volumes are stored. By default `emptyDir` volumes are stored on whatever medium that backs the node such as disk, SSD, or network storage, depending on your environment. If you set the `emptyDir.medium` field to `"Memory"`, Kubernetes mounts a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that, unlike disks, files you write count against the memory limit of the container that wrote them.

A size limit can be specified for the default medium, which limits the capacity of the `emptyDir` volume. The storage is allocated from [node ephemeral storage](). If that is filled up from another source (for example, log files or image overlays), the `emptyDir` may run out of capacity before this limit. If no size is specified, memory-backed volumes are sized to node allocatable memory.

> **Caution:**
>
> Please check [here]() for points to note in terms of resource management when using memory-backed `emptyDir`.

emptyDir configuration example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
```

emptyDir memory configuration example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
      medium: Memory
```

fc (fibre channel)

An `fc` volume type allows an existing fibre channel block storage volume to be mounted in a Pod. You can specify single or multiple target world wide names (WWNs) using the parameter `targetWWNs` in your Volume configuration. If multiple WWNs are specified, targetWWNs expect that those WWNs are from multi-path connections.

> **Note:**

> You must configure FC SAN Zoning to allocate and mask those LUNs (volumes) to the target WWNs beforehand so that Kubernetes hosts can access them.

gcePersistentDisk (deprecated)

In Kubernetes 1.35, all operations for the in-tree `gcePersistentDisk` type are redirected to the `pd.csi.storage.gke.io` CSI driver.

The `gcePersistentDisk` in-tree storage driver was deprecated in the Kubernetes v1.17 release and then removed entirely in the v1.28 release.

The Kubernetes project suggests that you use the [Google Compute Engine Persistent Disk CSI](#) third party storage driver instead.

gitRepo (deprecated)

> **Warning:**
>
> The `gitRepo` volume plugin is deprecated and is disabled by default.
>
> To provision a Pod that has a Git repository mounted, you can mount an `emptyDir` volume into an [init container](#) that clones the repo using Git, then mount the [EmptyDir](#) into the Pod's container.
>
> ---
>
> You can restrict the use of `gitRepo` volumes in your cluster using [policies](#), such as [ValidatingAdmissionPolicy](#). You can use the following Common Expression Language (CEL) expression as part of a policy to reject use of `gitRepo` volumes:
>
> ```
> !has(object.spec.volumes) || !object.spec.volumes.exists(v, has(v.gitRepo))
> ```

You can use this deprecated storage plugin in your cluster if you explicitly enable the `GitRepoVolumeDriver` [feature gate](#).

A `gitRepo` volume is an example of a volume plugin. This plugin mounts an empty directory and clones a git repository into this directory for your Pod to use.

Here is an example of a `gitRepo` volume:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /mypath
      name: git-volume
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

glusterfs (removed)

Kubernetes 1.35 does not include a `glusterfs` volume type.

The GlusterFS in-tree storage driver was deprecated in the Kubernetes v1.25 release and then removed entirely in the v1.26 release.

hostPath

A `hostPath` volume mounts a file or directory from the host node's filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

> **Warning:**
>
> Using the `hostPath` volume type presents many security risks. If you can avoid using a `hostPath` volume, you should. For example, define a [`local` PersistentVolume](#), and use that instead.
>
> If you are restricting access to specific directories on the node using admission-time validation, that restriction is only effective when you additionally require that any mounts of that `hostPath` volume are **read only**. If you allow a read-write mount of any host path by an untrusted Pod, the containers in that Pod may be able to subvert the read-write host mount.
>
> ---
>
> Take care when using `hostPath` volumes, whether these are mounted as read-only or as read-write, because:
>
> - Access to the host filesystem can expose privileged system credentials (such as for the kubelet) or privileged APIs (such as the container runtime socket) that can be used for container escape or to attack other parts of the cluster.
> - Pods with identical configuration (such as created from a PodTemplate) may behave differently on different nodes due to different files on the nodes.
> - `hostPath` volume usage is not treated as ephemeral storage usage. You need to monitor the disk usage by yourself because excessive `hostPath` disk usage will lead to disk pressure on the node.

Some uses for a `hostPath` are:

- running a container that needs access to node-level system components (such as a container that transfers system logs to a central location, accessing those logs using a read-only mount of `/var/log`)
- making a configuration file stored on the host system available read-only to a static pod; unlike normal Pods, static Pods cannot access ConfigMaps

hostPath volume types

In addition to the required `path` property, you can optionally specify a `type` for a `hostPath` volume.

The available values for `type` are:

Value	Behavior
`""`	Empty string (default) is for backward compatibility, which means that no checks will be performed before mounting the `hostPath` volume.
`DirectoryOrCreate`	If nothing exists at the given path, an empty directory will be created there as needed with permission set to 0755, having the same group and ownership with Kubelet.
`Directory`	A directory must exist at the given path.
`FileOrCreate`	If nothing exists at the given path, an empty file will be created there as needed with permission set to 0644, having the same group and ownership with Kubelet.
`File`	A file must exist at the given path.
`Socket`	A UNIX socket must exist at the given path.
`CharDevice`	*(Linux nodes only)* A character device must exist at the given path.
`BlockDevice`	*(Linux nodes only)* A block device must exist at the given path.

> **Caution:**

The `FileOrCreate` mode does **not** create the parent directory of the file. If the parent directory of the mounted file does not exist, the pod fails to start. To ensure that this mode works, you can try to mount directories and files separately, as shown in the [FileOrCreate example](#) for `hostPath`.

Some files or directories created on the underlying hosts might only be accessible by root. You then either need to run your process as root in a [privileged container](#) or modify the file permissions on the host to read from or write to a `hostPath` volume.

hostPath configuration example

Linux node | [Windows node](#)

```yaml
---
# This manifest mounts /data/foo on the host as /foo inside the
# single container that runs within the hostpath-example-linux Pod.
#
# The mount into the container is read-only.
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-example-linux
spec:
  os: { name: linux }
  nodeSelector:
    kubernetes.io/os: linux
  containers:
  - name: example-container
    image: registry.k8s.io/test-webserver
    volumeMounts:
    - mountPath: /foo
      name: example-volume
      readOnly: true
  volumes:
  - name: example-volume
    # mount /data/foo, but only if that directory already exists
    hostPath:
      path: /data/foo # directory location on host
      type: Directory # this field is optional
```

hostPath FileOrCreate configuration example

The following manifest defines a Pod that mounts `/var/local/aaa` inside the single container in the Pod. If the node does not already have a path `/var/local/aaa`, the kubelet creates it as a directory and then mounts it into the Pod.

If `/var/local/aaa` already exists but is not a directory, the Pod fails. Additionally, the kubelet attempts to make a file named `/var/local/aaa/1.txt` inside that directory (as seen from the host); if something already exists at that path and isn't a regular file, the Pod fails.

Here's the example manifest:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-webserver
spec:
  os: { name: linux }
  nodeSelector:
    kubernetes.io/os: linux
  containers:
  - name: test-webserver
    image: registry.k8s.io/test-webserver:latest
    volumeMounts:
    - mountPath: /var/local/aaa
      name: mydir
    - mountPath: /var/local/aaa/1.txt
      name: myfile
  volumes:
  - name: mydir
    hostPath:
      # Ensure the file directory is created.
      path: /var/local/aaa
      type: DirectoryOrCreate
  - name: myfile
    hostPath:
      path: /var/local/aaa/1.txt
      type: FileOrCreate
```

image

ⓘ **FEATURE STATE:** Kubernetes v1.35 [beta](enabled by default)

An `image` volume source represents an OCI object (a container image or artifact) which is available on the kubelet's host machine.

An example of using the `image` volume source is:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: image-volume
spec:
  containers:
  - name: shell
    command: ["sleep", "infinity"]
    image: debian
    volumeMounts:
    - name: volume
      mountPath: /volume
  volumes:
  - name: volume
    image:
      reference: quay.io/crio/artifact:v2
      pullPolicy: IfNotPresent
```

The volume is resolved at pod startup depending on which `pullPolicy` value is provided:

Always

The kubelet always attempts to pull the reference. If the pull fails, the kubelet sets the Pod to `Failed`.

Never

The kubelet never pulls the reference and only uses a local image or artifact. The Pod becomes `Failed` if any layers of the image aren't already present locally, or if the manifest for that image isn't already cached.

IfNotPresent

The kubelet pulls if the reference isn't already present on disk. The Pod becomes `Failed` if the reference isn't present and the pull fails.

The volume gets re-resolved if the pod gets deleted and recreated, which means that new remote content will become available on pod recreation. A failure to resolve or pull the image during pod startup will block containers from starting and may add significant latency. Failures will be retried using normal volume backoff and will be reported on the pod reason and message.

The types of objects that may be mounted by this volume are defined by the container runtime implementation on a host machine. At a minimum, they must include all valid types supported by the container image field. The OCI object gets mounted in a single directory (`spec.containers[*].volumeMounts[*].mountPath`) and will be mounted read-only.

Besides that:

- subPath or subPathExpr mounts for containers (`spec.containers[*].volumeMounts[*].subPath` , `spec.containers[*].volumeMounts[*].subPathExpr`) are only supported from Kubernetes v1.33.
- The field `spec.securityContext.fsGroupChangePolicy` has no effect on this volume type.
- The AlwaysPullImages Admission Controller does also work for this volume source like for container images.

The following fields are available for the `image` type:

reference

Artifact reference to be used. For example, you could specify `registry.k8s.io/conformance:v1.35.0` to load the files from the Kubernetes conformance test image. Behaves in the same way as `pod.spec.containers[*].image`. Pull secrets will be assembled in the same way as for the container image by looking up node credentials, service account image pull secrets, and pod spec image pull secrets. This field is optional to allow higher level config management to default or override container images in workload controllers like Deployments and StatefulSets. More info about container images.

pullPolicy

Policy for pulling OCI objects. Possible values are: `Always`, `Never` or `IfNotPresent`. Defaults to `Always` if `:latest` tag is specified, or `IfNotPresent` otherwise.

See the _Use an Image Volume With a Pod_ example for more details on how to use the volume source.

iscsi

An `iscsi` volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your Pod. Unlike `emptyDir` , which is erased when a Pod is removed, the contents of an `iscsi` volume are preserved and the volume is merely unmounted. This means that an iscsi volume can be pre-populated with data, and that data can be shared between pods.

> **Note:**
> You must have your own iSCSI server running with the volume created before you can use it.

A feature of iSCSI is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many Pods as you need. Unfortunately, iSCSI volumes can only be mounted by a single consumer in read-write mode. Simultaneous writers are not allowed.

local

A `local` volume represents a mounted local storage device such as a disk, partition or directory.

Local volumes can only be used as a statically created PersistentVolume. Dynamic provisioning is not supported.

Compared to `hostPath` volumes, `local` volumes are used in a durable and portable manner without manually scheduling pods to nodes. The system is aware of the volume's node constraints by looking at the node affinity on the PersistentVolume.

However, `local` volumes are subject to the availability of the underlying node and are not suitable for all applications. If a node becomes unhealthy, then the `local` volume becomes inaccessible to the pod. The pod using this volume is unable to run. Applications using `local` volumes must be able to tolerate this reduced availability, as well as potential data loss, depending on the

durability characteristics of the underlying disk.

The following example shows a PersistentVolume using a `local` volume and `nodeAffinity`:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

You must set a PersistentVolume `nodeAffinity` when using `local` volumes. The Kubernetes scheduler uses the PersistentVolume `nodeAffinity` to schedule these Pods to the correct node.

PersistentVolume `volumeMode` can be set to "Block" (instead of the default value "Filesystem") to expose the local volume as a raw block device.

When using local volumes, it is recommended to create a StorageClass with `volumeBindingMode` set to `WaitForFirstConsumer`. For more details, see the local [StorageClass](#) example. Delaying volume binding ensures that the PersistentVolumeClaim binding decision will also be evaluated with any other node constraints the Pod may have, such as node resource requirements, node selectors, Pod affinity, and Pod anti-affinity.

An external static provisioner can be run separately for improved management of the local volume lifecycle. Note that this provisioner does not support dynamic provisioning yet. For an example on how to run an external local provisioner, see the [local volume provisioner user guide](#).

> **Note:**
> The local PersistentVolume requires manual cleanup and deletion by the user if the external static provisioner is not used to manage the volume lifecycle.

nfs

An `nfs` volume allows an existing NFS (Network File System) share to be mounted into a Pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `nfs` volume are preserved and the volume is merely unmounted. This means that an NFS volume can be pre-populated with data, and that data can be shared between pods. NFS can be mounted by multiple writers simultaneously.
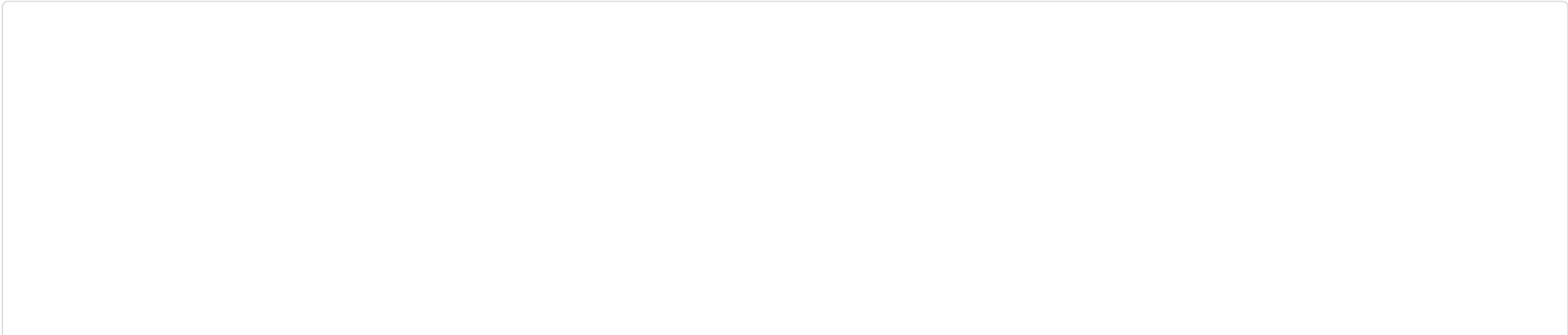
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /my-nfs-data
      name: test-volume
  volumes:
  - name: test-volume
    nfs:
      server: my-nfs-server.example.com
      path: /my-nfs-volume
      readOnly: true
```

> **Note:**
> You must have your own NFS server running with the share exported before you can use it.
>
> Also note that you can't specify NFS mount options in a Pod spec. You can either set mount options server-side or use
> [/etc/nfsmount.conf](). You can also mount NFS volumes via PersistentVolumes which do allow you to set mount options.

persistentVolumeClaim

A `persistentVolumeClaim` volume is used to mount a [PersistentVolume]() into a Pod. PersistentVolumeClaims are a way for users to "claim" durable storage (such as an iSCSI volume) without knowing the details of the particular cloud environment.

See the information about [PersistentVolumes]() for more details.

portworxVolume (deprecated)

ⓘ **FEATURE STATE:** Kubernetes v1.25 [deprecated]

A `portworxVolume` is an elastic block storage layer that runs hyperconverged with Kubernetes. [Portworx]() fingerprints storage in a server, tiers based on capabilities, and aggregates capacity across multiple servers. Portworx runs in-guest in virtual machines or on bare metal Linux nodes.

A `portworxVolume` can be dynamically created through Kubernetes or it can also be pre-provisioned and referenced inside a Pod. Here is an example Pod referencing a pre-provisioned Portworx volume:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-portworx-volume-pod
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /mnt
      name: pxvol
  volumes:
  - name: pxvol
    # This Portworx volume must already exist.
    portworxVolume:
      volumeID: "pxvol"
      fsType: "<fs-type>"
```

> **Note:**
> Make sure you have an existing PortworxVolume with name `pxvol` before using it in the Pod.

Portworx CSI migration

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

In Kubernetes 1.35, all operations for the in-tree Portworx volumes are redirected to the `pxd.portworx.com` Container Storage Interface (CSI) Driver by default.
[Portworx CSI Driver](#) must be installed on the cluster.

projected

A projected volume maps several existing volume sources into the same directory. For more details, see [projected volumes](#).

rbd (removed)

Kubernetes 1.35 does not include a `rbd` volume type.

The [Rados Block Device](#) (RBD) in-tree storage driver and its csi migration support were deprecated in the Kubernetes v1.28 release and then removed entirely in the v1.31 release.

secret

A `secret` volume is used to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by pods without coupling to Kubernetes directly. `secret` volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

> **Note:**
> - You must create a Secret in the Kubernetes API before you can use it.
>
> - A Secret is always mounted as `readOnly`.
>
> - A container using a Secret as a `subPath` volume mount will not receive Secret updates.

For more details, see [Configuring Secrets](#).

vsphereVolume (deprecated)

In Kubernetes 1.35, all operations for the in-tree `vsphereVolume` type are redirected to the `csi.vsphere.vmware.com` CSI driver.

The `vsphereVolume` in-tree storage driver was deprecated in the Kubernetes v1.19 release and then removed entirely in the v1.30 release.

The Kubernetes project suggests that you use the [vSphere CSI](#) third party storage driver instead.

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single pod. The `volumeMounts[*].subPath` property specifies a sub-path inside the referenced volume instead of its root.

The following example shows how to configure a Pod with a LAMP stack (Linux Apache MySQL PHP) using a single, shared volume. This sample `subPath` configuration is not recommended for production use.

The PHP application's code and assets map to the volume's `html` folder and the MySQL database is stored in the volume's `mysql` folder. For example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "rootpasswd"
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php:7.0-apache
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

Using subPath with expanded environment variables

ⓘ **FEATURE STATE:** Kubernetes v1.17 [stable]

Use the `subPathExpr` field to construct `subPath` directory names from downward API environment variables. The `subPath` and `subPathExpr` properties are mutually exclusive.

In this example, a `Pod` uses `subPathExpr` to create a directory `pod1` within the `hostPath` volume `/var/log/pods`. The `hostPath` volume takes the `Pod` name from the `downwardAPI`. The host directory `/var/log/pods/pod1` is mounted at `/logs` in the container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  containers:
  - name: container1
    env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
    image: busybox:1.28
    command: [ "sh", "-c", "while [ true ]; do echo 'Hello'; sleep 10; done | tee -a /logs/hello.txt" ]
    volumeMounts:
    - name: workdir1
      mountPath: /logs
      # The variable expansion uses round brackets (not curly brackets).
      subPathExpr: $(POD_NAME)
  restartPolicy: Never
  volumes:
  - name: workdir1
    hostPath:
      path: /var/log/pods
```

Resources

The storage medium (such as Disk or SSD) of an `emptyDir` volume is determined by the medium of the filesystem holding the kubelet root dir (typically `/var/lib/kubelet`). There is no limit on how much space an `emptyDir` or `hostPath` volume can consume, and no isolation between containers or pods.

To learn about requesting space using a resource specification, see [how to manage resources](#).

Out-of-tree volume plugins

The out-of-tree volume plugins include Container Storage Interface (CSI), and also FlexVolume (which is deprecated). These plugins enable storage vendors to create custom storage plugins without adding their plugin source code to the Kubernetes repository.

Previously, all volume plugins were "in-tree". The "in-tree" plugins were built, linked, compiled, and shipped with the core Kubernetes binaries. This meant that adding a new storage system to Kubernetes (a volume plugin) required checking code into the core Kubernetes code repository.

Both CSI and FlexVolume allow volume plugins to be developed independently of the Kubernetes code base, and deployed (installed) on Kubernetes clusters as extensions.

For storage vendors looking to create an out-of-tree volume plugin, please refer to the [volume plugin FAQ](#).

csi

[Container Storage Interface](#) (CSI) defines a standard interface for container orchestration systems (like Kubernetes) to expose arbitrary storage systems to their container workloads.

Please read the [CSI design proposal](#) for more information.

> **Note:**
> Support for CSI spec versions 0.2 and 0.3 is deprecated in Kubernetes v1.13 and will be removed in a future release.

> **Note:**
> CSI drivers may not be compatible across all Kubernetes releases. Please check the specific CSI driver's documentation for supported deployments steps for each Kubernetes release and a compatibility matrix.

Once a CSI-compatible volume driver is deployed on a Kubernetes cluster, users may use the `csi` volume type to attach or mount the volumes exposed by the CSI driver.

A `csi` volume can be used in a Pod in three different ways:

- through a reference to a [PersistentVolumeClaim](#)
- with a [generic ephemeral volume](#)
- with a [CSI ephemeral volume](#) if the driver supports that

The following fields are available to storage administrators to configure a CSI persistent volume:

- `driver`: A string value that specifies the name of the volume driver to use. This value must correspond to the value returned in the `GetPluginInfoResponse` by the CSI driver as defined in the [CSI spec](#). It is used by Kubernetes to identify which CSI driver to call out to, and by CSI driver components to identify which PV objects belong to the CSI driver.
- `volumeHandle`: A string value that uniquely identifies the volume. This value must correspond to the value returned in the `volume.id` field of the `CreateVolumeResponse` by the CSI driver as defined in the [CSI spec](#). The value is passed as `volume_id` in all calls to the CSI volume driver when referencing the volume.
- `readOnly`: An optional boolean value indicating whether the volume is to be "ControllerPublished" (attached) as read only. Default is false. This value is passed to the CSI driver via the `readonly` field in the `ControllerPublishVolumeRequest`.
- `fsType`: If the PV's `VolumeMode` is `Filesystem`, then this field may be used to specify the filesystem that should be used to mount the volume. If the volume has not been formatted and formatting is supported, this value will be used to format the volume. This value is passed to the CSI driver via the `VolumeCapability` field of `ControllerPublishVolumeRequest`, `NodeStageVolumeRequest`, and `NodePublishVolumeRequest`.

- `volumeAttributes` : A map of string to string that specifies static properties of a volume. This map must correspond to the map returned in the `volume.attributes` field of the `CreateVolumeResponse` by the CSI driver as defined in the [CSI spec](). The map is passed to the CSI driver via the `volume_context` field in the `ControllerPublishVolumeRequest` , `NodeStageVolumeRequest` , and `NodePublishVolumeRequest` .

- `controllerPublishSecretRef` : A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `ControllerPublishVolume` and `ControllerUnpublishVolume` calls. This field is optional, and may be empty if no secret is required. If the Secret contains more than one secret, all secrets are passed.

- `nodeExpandSecretRef` : A reference to the secret containing sensitive information to pass to the CSI driver to complete the CSI `NodeExpandVolume` call. This field is optional and may be empty if no secret is required. If the object contains more than one secret, all secrets are passed. When you have configured secret data for node-initiated volume expansion, the kubelet passes that data via the `NodeExpandVolume()` call to the CSI driver. All supported versions of Kubernetes offer the `nodeExpandSecretRef` field, and have it available by default. Kubernetes releases prior to v1.25 did not include this support.

- Enable the [feature gate]() named `CSINodeExpandSecret` for each kube-apiserver and for the kubelet on every node. Since Kubernetes version 1.27, this feature has been enabled by default and no explicit enablement of the feature gate is required. You must also be using a CSI driver that supports or requires secret data during node-initiated storage resize operations.

- `nodePublishSecretRef` : A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `NodePublishVolume` call. This field is optional and may be empty if no secret is required. If the secret object contains more than one secret, all secrets are passed.

- `nodeStageSecretRef` : A reference to the secret object containing sensitive information to pass to the CSI driver to complete the CSI `NodeStageVolume` call. This field is optional and may be empty if no secret is required. If the Secret contains more than one secret, all secrets are passed.

CSI raw block volume support

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

Vendors with external CSI drivers can implement raw block volume support in Kubernetes workloads.

You can set up your [PersistentVolume/PersistentVolumeClaim with raw block volume support]() as usual, without any CSI-specific changes.

CSI ephemeral volumes

ⓘ **FEATURE STATE:** Kubernetes v1.25 [stable]

You can directly configure CSI volumes within the Pod specification. Volumes specified in this way are ephemeral and do not persist across pod restarts. See [Ephemeral Volumes]() for more information.

For more information on how to develop a CSI driver, refer to the [kubernetes-csi documentation]()

Windows CSI proxy

ⓘ **FEATURE STATE:** Kubernetes v1.22 [stable]

CSI node plugins need to perform various privileged operations like scanning of disk devices and mounting of file systems. These operations differ for each host operating system. For Linux worker nodes, containerized CSI node plugins are typically deployed as privileged containers. For Windows worker nodes, privileged operations for containerized CSI node plugins is supported using [csi-proxy](), a community-managed, stand-alone binary that needs to be pre-installed on each Windows node.

For more details, refer to the deployment guide of the CSI plugin you wish to deploy.

Migrating to CSI drivers from in-tree plugins

ⓘ **FEATURE STATE:** Kubernetes v1.25 [stable]

The `CSIMigration` feature directs operations against existing in-tree plugins to corresponding CSI plugins (which are expected to be installed and configured). As a result, operators do not have to make any configuration changes to existing Storage Classes, PersistentVolumes or PersistentVolumeClaims (referring to in-tree plugins) when transitioning to a CSI driver that supersedes an in-tree plugin.

> **Note:**
> Existing PVs created by an in-tree volume plugin can still be used in the future without any configuration changes, even after the migration to CSI is completed for that volume type, and even after you upgrade to a version of Kubernetes that doesn't have compiled-in support for that kind of storage.
>
> As part of that migration, you - or another cluster administrator - **must** have installed and configured the appropriate CSI driver for that storage. The core of Kubernetes does not install that software for you.
>
> ---
>
> After that migration, you can also define new PVCs and PVs that refer to the legacy, built-in storage integrations. Provided you have the appropriate CSI driver installed and configured, the PV creation continues to work, even for brand new volumes. The actual storage management now happens through the CSI driver.

The operations and features that are supported include: provisioning/delete, attach/detach, mount/unmount and resizing of volumes.

In-tree plugins that support `CSIMigration` and have a corresponding CSI driver implemented are listed in [Types of Volumes](#).

flexVolume (deprecated)

ⓘ **FEATURE STATE:** Kubernetes v1.23 [deprecated]

FlexVolume is an out-of-tree plugin interface that uses an exec-based model to interface with storage drivers. The FlexVolume driver binaries must be installed in a pre-defined volume plugin path on each node and in some cases the control plane nodes as well.

Pods interact with FlexVolume drivers through the `flexVolume` in-tree volume plugin.

The following FlexVolume [plugins](#), deployed as PowerShell scripts on the host, support Windows nodes:

- [SMB](#)
- [iSCSI](#)

> **Note:**
> FlexVolume is deprecated. Using an out-of-tree CSI driver is the recommended way to integrate external storage with Kubernetes.
>
> Maintainers of FlexVolume driver should implement a CSI Driver and help to migrate users of FlexVolume drivers to CSI. Users of FlexVolume should move their workloads to use the equivalent CSI Driver.

Mount propagation

> **Caution:**
> Mount propagation is a low-level feature that does not work consistently on all volume types. The Kubernetes project recommends only using mount propagation with `hostPath` or memory-backed `emptyDir` volumes. See [Kubernetes issue #95049](#) for more context.

Mount propagation allows for sharing volumes mounted by a container to other containers in the same pod, or even to other pods on the same node.

Mount propagation of a volume is controlled by the `mountPropagation` field in `containers[*].volumeMounts`. Its values are:

- `None` - This volume mount will not receive any subsequent mounts that are mounted to this volume or any of its subdirectories by the host. In similar fashion, no mounts created by the container will be visible on the host. This is the default mode.

  This mode is equal to `rprivate` mount propagation as described in [mount(8)](#)

  However, the CRI runtime may choose `rslave` mount propagation (i.e., `HostToContainer`) instead, when `rprivate` propagation is not applicable. cri-dockerd (Docker) is known to choose `rslave` mount propagation when the mount source contains the Docker daemon's root directory (`/var/lib/docker`).

- `HostToContainer` - This volume mount will receive all subsequent mounts that are mounted to this volume or any of its subdirectories.

  In other words, if the host mounts anything inside the volume mount, the container will see it mounted there.

  Similarly, if any Pod with `Bidirectional` mount propagation to the same volume mounts anything there, the container with `HostToContainer` mount propagation will see it.

  This mode is equal to `rslave` mount propagation as described in the [mount(8)](#)

- `Bidirectional` - This volume mount behaves the same the `HostToContainer` mount. In addition, all volume mounts created by the container will be propagated back to the host and to all containers of all pods that use the same volume.

  A typical use case for this mode is a Pod with a FlexVolume or CSI driver or a Pod that needs to mount something on the host using a `hostPath` volume.

  This mode is equal to `rshared` mount propagation as described in the [mount(8)](#)

> **Warning:**
>
> `Bidirectional` mount propagation can be dangerous. It can damage the host operating system and therefore it is allowed only in privileged containers. Familiarity with Linux kernel behavior is strongly recommended. In addition, any volume mounts created by containers in pods must be destroyed (unmounted) by the containers on termination.

Read-only mounts

A mount can be made read-only by setting the `.spec.containers[*].volumeMounts[*].readOnly` field to `true`. This does not make the volume itself read-only, but that specific container will not be able to write to it. Other containers in the Pod may mount the same volume as read-write.

On Linux, read-only mounts are not recursively read-only by default. For example, consider a Pod which mounts the hosts `/mnt` as a `hostPath` volume. If there is another filesystem mounted read-write on `/mnt/<SUBMOUNT>` (such as tmpfs, NFS, or USB storage), the volume mounted into the container(s) will also have a writeable `/mnt/<SUBMOUNT>`, even if the mount itself was specified as read-only.

Recursive read-only mounts

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

Recursive read-only mounts can be enabled by setting the `RecursiveReadOnlyMounts` [feature gate](#) for kubelet and kube-apiserver, and setting the `.spec.containers[*].volumeMounts[*].recursiveReadOnly` field for a pod.

The allowed values are:

- `Disabled` (default): no effect.

- `Enabled`: makes the mount recursively read-only. Needs all the following requirements to be satisfied:

  - `readOnly` is set to `true`
  - `mountPropagation` is unset, or, set to `None`
  - The host is running with Linux kernel v5.12 or later
  - The [CRI-level](#) container runtime supports recursive read-only mounts
  - The OCI-level container runtime supports recursive read-only mounts.

  It will fail if any of these is not true.

- `IfPossible` : attempts to apply `Enabled` , and falls back to `Disabled` if the feature is not supported by the kernel or the runtime class.

Example:

```yaml
                                                              storage/rro.yaml  ⧉
apiVersion: v1
kind: Pod
metadata:
  name: rro
spec:
  volumes:
    - name: mnt
      hostPath:
        # tmpfs is mounted on /mnt/tmpfs
        path: /mnt
  containers:
    - name: busybox
      image: busybox
      args: ["sleep", "infinity"]
      volumeMounts:
        # /mnt-rro/tmpfs is not writable
        - name: mnt
          mountPath: /mnt-rro
          readOnly: true
          mountPropagation: None
          recursiveReadOnly: Enabled
        # /mnt-ro/tmpfs is writable
        - name: mnt
          mountPath: /mnt-ro
          readOnly: true
        # /mnt-rw/tmpfs is writable
        - name: mnt
          mountPath: /mnt-rw
```

When this property is recognized by kubelet and kube-apiserver, the `.status.containerStatuses[*].volumeMounts[*].recursiveReadOnly` field is set to either `Enabled` or `Disabled` .

Implementations

> **Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the [content guide](#) before submitting a change. [More information.](#)

The following container runtimes are known to support recursive read-only mounts.

CRI-level:

- [containerd](#), since v2.0
- [CRI-O](#), since v1.30

OCI-level:

- [runc](#), since v1.1
- [crun](#), since v1.8.6

What's next

Follow an example of [deploying WordPress and MySQL with Persistent Volumes](#).

Feedback

Was this page helpful?

Yes   No

Last modified December 11, 2025 at 12:09 PM PST: Remove noexec description from image volume (6dafe92409)