Volumes

On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers. First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state. Second, when running Containers together in a Pod it is often necessary to share files between those Containers. The Kubernetes Volume abstraction solves both of these problems.

Familiarity with [Pods](https://kubernetes.io/docs/user-guide/pods) is suggested.

* [**Background**](https://kubernetes.io/docs/concepts/storage/volumes/#background)
* [**Types of Volumes**](https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes)
* [**Using subPath**](https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath)
* [**Resources**](https://kubernetes.io/docs/concepts/storage/volumes/#resources)
* [**Out-of-Tree Volume Plugins**](https://kubernetes.io/docs/concepts/storage/volumes/#out-of-tree-volume-plugins)
* [**Mount propagation**](https://kubernetes.io/docs/concepts/storage/volumes/#mount-propagation)
* [**What's next**](https://kubernetes.io/docs/concepts/storage/volumes/#what-s-next)

Background

Docker also has a concept of [volumes](https://docs.docker.com/engine/admin/volumes/), though it is somewhat looser and less managed. In Docker, a volume is simply a directory on disk or in another Container. Lifetimes are not managed and until very recently there were only local-disk-backed volumes. Docker now provides volume drivers, but the functionality is very limited for now (e.g. as of Docker 1.7 only one volume driver is allowed per Container and there is no way to pass parameters to volumes).

A Kubernetes volume, on the other hand, has an explicit lifetime - the same as the Pod that encloses it. Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

At its core, a volume is just 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, a Pod specifies what volumes to provide for the Pod (the .spec.volumes field) and where to mount those into Containers (the .spec.containers[\*].volumeMounts field).

A process in a container sees a filesystem view composed from their Docker image and volumes. The [Docker image](https://docs.docker.com/userguide/dockerimages/) is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image. Volumes can not mount onto other volumes or have hard links to other volumes. Each Container in the Pod must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of Volumes:

* [awsElasticBlockStore](https://kubernetes.io/docs/concepts/storage/volumes/#awselasticblockstore)
* [azureDisk](https://kubernetes.io/docs/concepts/storage/volumes/#azuredisk)
* [azureFile](https://kubernetes.io/docs/concepts/storage/volumes/#azurefile)
* [cephfs](https://kubernetes.io/docs/concepts/storage/volumes/#cephfs)
* [cinder](https://kubernetes.io/docs/concepts/storage/volumes/#cinder)
* [configMap](https://kubernetes.io/docs/concepts/storage/volumes/#configmap)
* [csi](https://kubernetes.io/docs/concepts/storage/volumes/#csi)
* [downwardAPI](https://kubernetes.io/docs/concepts/storage/volumes/#downwardapi)
* [emptyDir](https://kubernetes.io/docs/concepts/storage/volumes/#emptydir)
* [fc (fibre channel)](https://kubernetes.io/docs/concepts/storage/volumes/#fc)
* [flexVolume](https://kubernetes.io/docs/concepts/storage/volumes/#flexVolume)
* [flocker](https://kubernetes.io/docs/concepts/storage/volumes/#flocker)
* [gcePersistentDisk](https://kubernetes.io/docs/concepts/storage/volumes/#gcepersistentdisk)
* [gitRepo (deprecated)](https://kubernetes.io/docs/concepts/storage/volumes/#gitrepo)
* [glusterfs](https://kubernetes.io/docs/concepts/storage/volumes/#glusterfs)
* [hostPath](https://kubernetes.io/docs/concepts/storage/volumes/#hostpath)
* [iscsi](https://kubernetes.io/docs/concepts/storage/volumes/#iscsi)
* [local](https://kubernetes.io/docs/concepts/storage/volumes/#local)
* [nfs](https://kubernetes.io/docs/concepts/storage/volumes/#nfs)
* [persistentVolumeClaim](https://kubernetes.io/docs/concepts/storage/volumes/#persistentvolumeclaim)
* [projected](https://kubernetes.io/docs/concepts/storage/volumes/#projected)
* [portworxVolume](https://kubernetes.io/docs/concepts/storage/volumes/#portworxvolume)
* [quobyte](https://kubernetes.io/docs/concepts/storage/volumes/#quobyte)
* [rbd](https://kubernetes.io/docs/concepts/storage/volumes/#rbd)
* [scaleIO](https://kubernetes.io/docs/concepts/storage/volumes/#scaleio)
* [secret](https://kubernetes.io/docs/concepts/storage/volumes/#secret)
* [storageos](https://kubernetes.io/docs/concepts/storage/volumes/#storageos)
* [vsphereVolume](https://kubernetes.io/docs/concepts/storage/volumes/#vspherevolume)

We welcome additional contributions.

awsElasticBlockStore

An awsElasticBlockStore volume mounts an Amazon Web Services (AWS) [EBS Volume](http://aws.amazon.com/ebs/) into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted. This means that an EBS volume can be pre-populated with data, and that data can be “handed off” between Pods.

**Caution:** You must create an EBS volume using aws ec2 create-volume or the AWS API before you can use it.

There are some restrictions when using an awsElasticBlockStore volume:

* the nodes on which Pods are running must be AWS EC2 instances
* those instances need to be in the same region and availability-zone as the EBS volume
* EBS only supports a single EC2 instance mounting a volume

Creating an EBS volume

Before you can use an EBS volume with a Pod, you need to create it.

aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2

Make sure the zone matches the zone you brought up your cluster in. (And also check that the size and EBS volume type are suitable for your use!)

AWS EBS Example configuration

apiVersion: v1

kind: Pod

metadata:

name: test-ebs

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-ebs

name: test-volume

volumes:

- name: test-volume

*# This AWS EBS volume must already exist.*

awsElasticBlockStore:

volumeID: <volume-id*>*

*fsType: ext4*

CSI Migration

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/docs/concepts/storage/volumes/)

The CSI Migration feature for awsElasticBlockStore, when enabled, shims all plugin operations from the existing in-tree plugin to the ebs.csi.aws.com Container Storage Interface (CSI) Driver. In order to use this feature, the [AWS EBS CSI Driver](https://github.com/kubernetes-sigs/aws-ebs-csi-driver) must be installed on the cluster and the CSIMigration and CSIMigrationAWS Alpha features must be enabled.

azureDisk

A azureDisk is used to mount a Microsoft Azure [Data Disk](https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-about-disks-vhds/) into a Pod.

More details can be found [here](https://github.com/kubernetes/examples/tree/master/staging/volumes/azure_disk/README.md).

CSI Migration

**FEATURE STATE:** Kubernetes v1.15 [alpha](https://kubernetes.io/docs/concepts/storage/volumes/)

The CSI Migration feature for azureDisk, when enabled, shims all plugin operations from the existing in-tree plugin to the disk.csi.azure.com Container Storage Interface (CSI) Driver. In order to use this feature, the [Azure Disk CSI Driver](https://github.com/kubernetes-sigs/azuredisk-csi-driver) must be installed on the cluster and the CSIMigration and CSIMigrationAzureDisk Alpha features must be enabled.

azureFile

A azureFile is used to mount a Microsoft Azure File Volume (SMB 2.1 and 3.0) into a Pod.

More details can be found [here](https://github.com/kubernetes/examples/tree/master/staging/volumes/azure_file/README.md).

CSI Migration

**FEATURE STATE:** Kubernetes v1.15 [alpha](https://kubernetes.io/docs/concepts/storage/volumes/)

The CSI Migration feature for azureFile, when enabled, shims all plugin operations from the existing in-tree plugin to the file.csi.azure.com Container Storage Interface (CSI) Driver. In order to use this feature, the [Azure File CSI Driver](https://github.com/kubernetes-sigs/azurefile-csi-driver) must be installed on the cluster and the CSIMigration and CSIMigrationAzureFile Alpha features must be enabled.

cephfs

A cephfs volume allows an existing CephFS volume to be mounted into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of a cephfs volume are preserved and the volume is merely unmounted. This means that a CephFS volume can be pre-populated with data, and that data can be “handed off” between Pods. CephFS can be mounted by multiple writers simultaneously.

**Caution:** You must have your own Ceph server running with the share exported before you can use it.

See the [CephFS example](https://github.com/kubernetes/examples/tree/master/volumes/cephfs/) for more details.

cinder

**Note:** Prerequisite: Kubernetes with OpenStack Cloud Provider configured. For cloudprovider configuration please refer [cloud provider openstack](https://kubernetes.io/docs/concepts/cluster-administration/cloud-providers/#openstack).

cinder is used to mount OpenStack Cinder Volume into your Pod.

Cinder Volume Example configuration

apiVersion: v1

kind: Pod

metadata:

name: test-cinder

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-cinder-container

volumeMounts:

- mountPath: /test-cinder

name: test-volume

volumes:

- name: test-volume

*# This OpenStack volume must already exist.*

cinder:

volumeID: <volume-id*>*

*fsType: ext4*

CSI Migration

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/docs/concepts/storage/volumes/)

The CSI Migration feature for Cinder, when enabled, shims all plugin operations from the existing in-tree plugin to the cinder.csi.openstack.org Container Storage Interface (CSI) Driver. In order to use this feature, the [Openstack Cinder CSI Driver](https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/using-cinder-csi-plugin.md) must be installed on the cluster and the CSIMigration and CSIMigrationOpenStack Alpha features must be enabled.

configMap

The [configMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/) resource provides a way to inject configuration data into Pods. The data stored in a ConfigMap object can be referenced in a volume of type configMap and then consumed by containerized applications running in a Pod.

When referencing a configMap object, you can simply provide its name in the volume to reference it. You can also customize the path to use for a specific entry in the ConfigMap. For example, to mount the log-config ConfigMap onto a Pod called configmap-pod, you might use the YAML below:

apiVersion: v1

kind: Pod

metadata:

name: configmap-pod

spec:

containers:

- name: test

image: busybox

volumeMounts:

- name: config-vol

mountPath: /etc/config

volumes:

- name: config-vol

configMap:

name: log-config

items:

- key: log\_level

path: log\_level

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”. Note that this path is derived from the volume’s mountPath and the path keyed with log\_level.

**Caution:** You must create a [ConfigMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/) before you can use it.

**Note:** A Container using a ConfigMap as a [subPath](https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath) volume mount will not receive ConfigMap updates.

downwardAPI

A downwardAPI volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

**Note:** A Container using Downward API as a [subPath](https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath) volume mount will not receive Downward API updates.

See the [downwardAPI volume example](https://kubernetes.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/) for more details.

emptyDir

An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node. As the name says, it is initially empty. Containers in the Pod can all 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 forever.

**Note:** A Container crashing does *NOT* remove a Pod from a node, so 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

By default, emptyDir volumes are stored on whatever medium is backing the node - that might be disk or SSD or network storage, depending on your environment. However, you can set the emptyDir.medium field to "Memory" to tell Kubernetes to mount a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that unlike disks, tmpfs is cleared on node reboot and any files you write will count against your Container’s memory limit.

Example Pod

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /cache

name: cache-volume

volumes:

- name: cache-volume

emptyDir: {}

fc (fibre channel)

An fc volume allows an existing fibre channel volume to be mounted in a Pod. You can specify single or multiple target World Wide Names using the parameter targetWWNs in your volume configuration. If multiple WWNs are specified, targetWWNs expect that those WWNs are from multi-path connections.

**Caution:** 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.

See the [FC example](https://github.com/kubernetes/examples/tree/master/staging/volumes/fibre_channel) for more details.

flocker

[Flocker](https://github.com/ClusterHQ/flocker) is an open-source clustered Container data volume manager. It provides management and orchestration of data volumes backed by a variety of storage backends.

A flocker volume allows a Flocker dataset to be mounted into a Pod. If the dataset does not already exist in Flocker, it needs to be first created with the Flocker CLI or by using the Flocker API. If the dataset already exists it will be reattached by Flocker to the node that the Pod is scheduled. This means data can be “handed off” between Pods as required.

**Caution:** You must have your own Flocker installation running before you can use it.

See the [Flocker example](https://github.com/kubernetes/examples/tree/master/staging/volumes/flocker) for more details.

gcePersistentDisk

A gcePersistentDisk volume mounts a Google Compute Engine (GCE) [Persistent Disk](http://cloud.google.com/compute/docs/disks) into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of a PD are preserved and the volume is merely unmounted. This means that a PD can be pre-populated with data, and that data can be “handed off” between Pods.

**Caution:** You must create a PD using gcloud or the GCE API or UI before you can use it.

There are some restrictions when using a gcePersistentDisk:

* the nodes on which Pods are running must be GCE VMs
* those VMs need to be in the same GCE project and zone as the PD

A feature of PD is that they can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a PD with your dataset and then serve it in parallel from as many Pods as you need. Unfortunately, PDs can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

Using a PD on a Pod controlled by a ReplicationController will fail unless the PD is read-only or the replica count is 0 or 1.

Creating a PD

Before you can use a GCE PD with a Pod, you need to create it.

gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk

Example Pod

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-pd

name: test-volume

volumes:

- name: test-volume

*# This GCE PD must already exist.*

gcePersistentDisk:

pdName: my-data-disk

fsType: ext4

Regional Persistent Disks

**FEATURE STATE:** Kubernetes v1.10 [beta](https://kubernetes.io/docs/concepts/storage/volumes/)

The [Regional Persistent Disks](https://cloud.google.com/compute/docs/disks/#repds) feature allows the creation of Persistent Disks that are available in two zones within the same region. In order to use this feature, the volume must be provisioned as a PersistentVolume; referencing the volume directly from a pod is not supported.

Manually provisioning a Regional PD PersistentVolume

Dynamic provisioning is possible using a [StorageClass for GCE PD](https://kubernetes.io/docs/concepts/storage/storage-classes/#gce). Before creating a PersistentVolume, you must create the PD:

gcloud beta compute disks create --size=500GB my-data-disk

--region us-central1

--replica-zones us-central1-a,us-central1-b

Example PersistentVolume spec:

apiVersion: v1

kind: PersistentVolume

metadata:

name: test-volume

labels:

failure-domain.beta.kubernetes.io/zone: us-central1-a\_\_us-central1-b

spec:

capacity:

storage: 400Gi

accessModes:

- ReadWriteOnce

gcePersistentDisk:

pdName: my-data-disk

fsType: ext4

CSI Migration

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/docs/concepts/storage/volumes/)

The CSI Migration feature for GCE PD, when enabled, shims all plugin operations from the existing in-tree plugin to the pd.csi.storage.gke.io Container Storage Interface (CSI) Driver. In order to use this feature, the [GCE PD CSI Driver](https://github.com/kubernetes-sigs/gcp-compute-persistent-disk-csi-driver) must be installed on the cluster and the CSIMigration and CSIMigrationGCE Alpha features must be enabled.

gitRepo (deprecated)

**Warning:** The gitRepo volume type is deprecated. To provision a container with a git repo, mount an [EmptyDir](https://kubernetes.io/docs/concepts/storage/volumes/#emptydir) into an InitContainer that clones the repo using git, then mount the [EmptyDir](https://kubernetes.io/docs/concepts/storage/volumes/#emptydir) into the Pod’s container.

A gitRepo volume is an example of what can be done as a volume plugin. It mounts an empty directory and clones a git repository into it for your Pod to use. In the future, such volumes may be moved to an even more decoupled model, rather than extending the Kubernetes API for every such use case.

Here is an example of gitRepo volume:

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

A glusterfs volume allows a [Glusterfs](http://www.gluster.org/) (an open source networked filesystem) volume to be mounted into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of a glusterfs volume are preserved and the volume is merely unmounted. This means that a glusterfs volume can be pre-populated with data, and that data can be “handed off” between Pods. GlusterFS can be mounted by multiple writers simultaneously.

**Caution:** You must have your own GlusterFS installation running before you can use it.

See the [GlusterFS example](https://github.com/kubernetes/examples/tree/master/volumes/glusterfs) for more details.

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.

For example, some uses for a hostPath are:

* running a Container that needs access to Docker internals; use a hostPath of /var/lib/docker
* running cAdvisor in a Container; use a hostPath of /sys
* allowing a Pod to specify whether a given hostPath should exist prior to the Pod running, whether it should be created, and what it should exist as

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

The supported values for field 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 | A character device must exist at the given path |
| BlockDevice | A block device must exist at the given path |

Watch out when using this type of volume, because:

* Pods with identical configuration (such as created from a podTemplate) may behave differently on different nodes due to different files on the nodes
* when Kubernetes adds resource-aware scheduling, as is planned, it will not be able to account for resources used by a hostPath
* the files or directories created on the underlying hosts are only writable by root. You either need to run your process as root in a [privileged Container](https://kubernetes.io/docs/user-guide/security-context) or modify the file permissions on the host to be able to write to a hostPath volume

Example Pod

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-pd

name: test-volume

volumes:

- name: test-volume

hostPath:

*# directory location on host*

path: /data

*# this field is optional*

type: Directory

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 “handed off” between Pods.

**Caution:** 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 - no simultaneous writers allowed.

See the [iSCSI example](https://github.com/kubernetes/examples/tree/master/volumes/iscsi) for more details.

local

**FEATURE STATE:** Kubernetes v1.14 [stable](https://kubernetes.io/docs/concepts/storage/volumes/)

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 yet.

Compared to hostPath volumes, local volumes can be used in a durable and portable manner without manually scheduling Pods to nodes, as the system is aware of the volume’s node constraints by looking at the node affinity on the PersistentVolume.

However, local volumes are still subject to the availability of the underlying node and are not suitable for all applications. If a node becomes unhealthy, then the local volume will also become inaccessible, and a Pod using it will not be able 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 is an example of PersistentVolume spec using a local volume and nodeAffinity:

apiVersion: v1

kind: PersistentVolume

metadata:

name: example-pv

spec:

capacity:

storage: 100Gi

*# volumeMode field requires BlockVolume Alpha feature gate to be enabled.*

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

PersistentVolume nodeAffinity is required when using local volumes. It enables the Kubernetes scheduler to correctly schedule Pods using local volumes to the correct node.

PersistentVolume volumeMode can now be set to “Block” (instead of the default value “Filesystem”) to expose the local volume as a raw block device. The volumeMode field requires BlockVolume Alpha feature gate to be enabled.

When using local volumes, it is recommended to create a StorageClass with volumeBindingMode set to WaitForFirstConsumer. See the [example](https://kubernetes.io/docs/concepts/storage/storage-classes/#local). 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](https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner).

**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 your 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 “handed off” between Pods. NFS can be mounted by multiple writers simultaneously.

**Caution:** You must have your own NFS server running with the share exported before you can use it.

See the [NFS example](https://github.com/kubernetes/examples/tree/master/staging/volumes/nfs) for more details.

persistentVolumeClaim

A persistentVolumeClaim volume is used to mount a [PersistentVolume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) into a Pod. PersistentVolumes are a way for users to “claim” durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

See the [PersistentVolumes example](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) for more details.

projected

A projected volume maps several existing volume sources into the same directory.

Currently, the following types of volume sources can be projected:

* [secret](https://kubernetes.io/docs/concepts/storage/volumes/#secret)
* [downwardAPI](https://kubernetes.io/docs/concepts/storage/volumes/#downwardapi)
* [configMap](https://kubernetes.io/docs/concepts/storage/volumes/#configmap)
* serviceAccountToken

All sources are required to be in the same namespace as the Pod. For more details, see the [all-in-one volume design document](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/all-in-one-volume.md).

The projection of service account tokens is a feature introduced in Kubernetes 1.11 and promoted to Beta in 1.12. To enable this feature on 1.11, you need to explicitly set the TokenRequestProjection [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to True.

Example Pod with a secret, a downward API, and a configmap.

apiVersion: v1

kind: Pod

metadata:

name: volume-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: all-in-one

mountPath: "/projected-volume"

readOnly: **true**

volumes:

- name: all-in-one

projected:

sources:

- secret:

name: mysecret

items:

- key: username

path: my-group/my-username

- downwardAPI:

items:

- path: "labels"

fieldRef:

fieldPath: metadata.labels

- path: "cpu\_limit"

resourceFieldRef:

containerName: container-test

resource: limits.cpu

- configMap:

name: myconfigmap

items:

- key: config

path: my-group/my-config

Example Pod with multiple secrets with a non-default permission mode set.

apiVersion: v1

kind: Pod

metadata:

name: volume-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: all-in-one

mountPath: "/projected-volume"

readOnly: **true**

volumes:

- name: all-in-one

projected:

sources:

- secret:

name: mysecret

items:

- key: username

path: my-group/my-username

- secret:

name: mysecret2

items:

- key: password

path: my-group/my-password

mode: 511

Each projected volume source is listed in the spec under sources. The parameters are nearly the same with two exceptions:

* For secrets, the secretName field has been changed to name to be consistent with ConfigMap naming.
* The defaultMode can only be specified at the projected level and not for each volume source. However, as illustrated above, you can explicitly set the mode for each individual projection.

When the TokenRequestProjection feature is enabled, you can inject the token for the current [service account](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#service-account-tokens) into a Pod at a specified path. Below is an example:

apiVersion: v1

kind: Pod

metadata:

name: sa-token-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: token-vol

mountPath: "/service-account"

readOnly: **true**

volumes:

- name: token-vol

projected:

sources:

- serviceAccountToken:

audience: api

expirationSeconds: 3600

path: token

The example Pod has a projected volume containing the injected service account token. This token can be used by Pod containers to access the Kubernetes API server, for example. The audience field contains the intended audience of the token. A recipient of the token must identify itself with an identifier specified in the audience of the token, and otherwise should reject the token. This field is optional and it defaults to the identifier of the API server.

The expirationSeconds is the expected duration of validity of the service account token. It defaults to 1 hour and must be at least 10 minutes (600 seconds). An administrator can also limit its maximum value by specifying the --service-account-max-token-expiration option for the API server. The path field specifies a relative path to the mount point of the projected volume.

**Note:** A Container using a projected volume source as a [subPath](https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath) volume mount will not receive updates for those volume sources.

portworxVolume

A portworxVolume is an elastic block storage layer that runs hyperconverged with Kubernetes. [Portworx](https://portworx.com/use-case/kubernetes-storage/) 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 Kubernetes Pod. Here is an example Pod referencing a pre-provisioned PortworxVolume:

apiVersion: v1

kind: Pod

metadata:

name: test-portworx-volume-pod

spec:

containers:

- image: k8s.gcr.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>"

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

More details and examples can be found [here](https://github.com/kubernetes/examples/tree/master/staging/volumes/portworx/README.md).

quobyte

A quobyte volume allows an existing [Quobyte](http://www.quobyte.com/) volume to be mounted into your Pod.

**Caution:** You must have your own Quobyte setup running with the volumes created before you can use it.

Quobyte supports the [Container Storage Interface](https://kubernetes.io/docs/concepts/storage/volumes/#csi). CSI is the recommended plugin to use Quobyte volumes inside Kubernetes. Quobyte’s GitHub project has [instructions](https://github.com/quobyte/quobyte-csi#quobyte-csi) for deploying Quobyte using CSI, along with examples.

rbd

An rbd volume allows a [Rados Block Device](http://ceph.com/docs/master/rbd/rbd/) volume to be mounted into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of a rbd volume are preserved and the volume is merely unmounted. This means that a RBD volume can be pre-populated with data, and that data can be “handed off” between Pods.

**Caution:** You must have your own Ceph installation running before you can use RBD.

A feature of RBD 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, RBD volumes can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

See the [RBD example](https://github.com/kubernetes/examples/tree/master/volumes/rbd) for more details.

scaleIO

ScaleIO is a software-based storage platform that can use existing hardware to create clusters of scalable shared block networked storage. The scaleIO volume plugin allows deployed Pods to access existing ScaleIO volumes (or it can dynamically provision new volumes for persistent volume claims, see [ScaleIO Persistent Volumes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#scaleio)).

**Caution:** You must have an existing ScaleIO cluster already setup and running with the volumes created before you can use them.

The following is an example of Pod configuration with ScaleIO:

apiVersion: v1

kind: Pod

metadata:

name: pod-0

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: pod-0

volumeMounts:

- mountPath: /test-pd

name: vol-0

volumes:

- name: vol-0

scaleIO:

gateway: https://localhost:443/api

system: scaleio

protectionDomain: sd0

storagePool: sp1

volumeName: vol-0

secretRef:

name: sio-secret

fsType: xfs

For further detail, please see the [ScaleIO examples](https://github.com/kubernetes/examples/tree/master/staging/volumes/scaleio).

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.

**Caution:** You must create a secret in the Kubernetes API before you can use it.

**Note:** A Container using a Secret as a [subPath](https://kubernetes.io/docs/concepts/storage/volumes/#using-subpath) volume mount will not receive Secret updates.

Secrets are described in more detail [here](https://kubernetes.io/docs/user-guide/secrets).

storageOS

A storageos volume allows an existing [StorageOS](https://www.storageos.com/) volume to be mounted into your Pod.

StorageOS runs as a Container within your Kubernetes environment, making local or attached storage accessible from any node within the Kubernetes cluster. Data can be replicated to protect against node failure. Thin provisioning and compression can improve utilization and reduce cost.

At its core, StorageOS provides block storage to Containers, accessible via a file system.

The StorageOS Container requires 64-bit Linux and has no additional dependencies. A free developer license is available.

**Caution:** You must run the StorageOS Container on each node that wants to access StorageOS volumes or that will contribute storage capacity to the pool. For installation instructions, consult the [StorageOS documentation](https://docs.storageos.com/).

apiVersion: v1

kind: Pod

metadata:

labels:

name: redis

role: master

name: test-storageos-redis

spec:

containers:

- name: master

image: kubernetes/redis:v1

env:

- name: MASTER

value: "true"

ports:

- containerPort: 6379

volumeMounts:

- mountPath: /redis-master-data

name: redis-data

volumes:

- name: redis-data

storageos:

*# The `redis-vol01` volume must already exist within StorageOS in the `default` namespace.*

volumeName: redis-vol01

fsType: ext4

For more information including Dynamic Provisioning and Persistent Volume Claims, please see the [StorageOS examples](https://github.com/kubernetes/examples/blob/master/volumes/storageos).

vsphereVolume

**Note:** Prerequisite: Kubernetes with vSphere Cloud Provider configured. For cloudprovider configuration please refer [vSphere getting started guide](https://vmware.github.io/vsphere-storage-for-kubernetes/documentation/).

A vsphereVolume is used to mount a vSphere VMDK Volume into your Pod. The contents of a volume are preserved when it is unmounted. It supports both VMFS and VSAN datastore.

**Caution:** You must create VMDK using one of the following methods before using with Pod.

Creating a VMDK volume

Choose one of the following methods to create a VMDK.

* [Create using vmkfstools](https://kubernetes.io/docs/concepts/storage/volumes/#tabs-volumes-0)
* [Create using vmware-vdiskmanager](https://kubernetes.io/docs/concepts/storage/volumes/#tabs-volumes-1)

First ssh into ESX, then use the following command to create a VMDK:

vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk

vSphere VMDK Example configuration

apiVersion: v1

kind: Pod

metadata:

name: test-vmdk

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-vmdk

name: test-volume

volumes:

- name: test-volume

*# This VMDK volume must already exist.*

vsphereVolume:

volumePath: "[DatastoreName] volumes/myDisk"

fsType: ext4

More examples can be found [here](https://github.com/kubernetes/examples/tree/master/staging/volumes/vsphere).

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single Pod. The volumeMounts.subPath property can be used to specify a sub-path inside the referenced volume instead of its root.

Here is an example of a Pod with a LAMP stack (Linux Apache Mysql PHP) using a single, shared volume. The HTML contents are mapped to its html folder, and the databases will be stored in its mysql folder:

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 v1.15 [beta](https://kubernetes.io/docs/concepts/storage/volumes/)

Use the subPathExpr field to construct subPath directory names from Downward API environment variables. This feature requires the VolumeSubpathEnvExpansion [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to be enabled. It is enabled by default starting with Kubernetes 1.15. 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, using the pod name from the Downward API. The host directory /var/log/pods/pod1 is mounted at /logs in the container.

apiVersion: v1

kind: Pod

metadata:

name: pod1

spec:

containers:

- name: container1

env:

- name: POD\_NAME

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: metadata.name

image: busybox

command: [ "sh", "-c", "while [ true ]; do echo 'Hello'; sleep 10; done | tee -a /logs/hello.txt" ]

volumeMounts:

- name: workdir1

mountPath: /logs

subPathExpr: $(POD\_NAME)

restartPolicy: Never

volumes:

- name: workdir1

hostPath:

path: /var/log/pods

Resources

The storage media (Disk, SSD, etc.) 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 between Pods.

In the future, we expect that emptyDir and hostPath volumes will be able to request a certain amount of space using a [resource](https://kubernetes.io/docs/user-guide/compute-resources) specification, and to select the type of media to use, for clusters that have several media types.

Out-of-Tree Volume Plugins

The Out-of-tree volume plugins include the Container Storage Interface (CSI) and FlexVolume. They enable storage vendors to create custom storage plugins without adding them to the Kubernetes repository.

Before the introduction of CSI and FlexVolume, all volume plugins (like volume types listed above) were “in-tree” meaning they were built, linked, compiled, and shipped with the core Kubernetes binaries and extend the core Kubernetes API. 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 independent 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 [this FAQ](https://github.com/kubernetes/community/blob/master/sig-storage/volume-plugin-faq.md).

CSI

[Container Storage Interface](https://github.com/container-storage-interface/spec/blob/master/spec.md) (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](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/storage/container-storage-interface.md) for more information.

CSI support was introduced as alpha in Kubernetes v1.9, moved to beta in Kubernetes v1.10, and is GA in Kubernetes v1.13.

**Note:** Support for CSI spec versions 0.2 and 0.3 are 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, mount, etc. the volumes exposed by the CSI driver.

The csi volume type does not support direct reference from Pod and may only be referenced in a Pod via a PersistentVolumeClaim object.

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](https://github.com/container-storage-interface/spec/blob/master/spec.md#getplugininfo). 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](https://github.com/container-storage-interface/spec/blob/master/spec.md#createvolume). The value is passed as volume\_id on 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](https://github.com/container-storage-interface/spec/blob/master/spec.md#createvolume). The map is passed to the CSI driver via the volume\_attributes 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 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 object contains more than one secret, all secrets are passed.
* 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.

CSI raw block volume support

**FEATURE STATE:** Kubernetes v1.14 [beta](https://kubernetes.io/docs/concepts/storage/volumes/)

Starting with version 1.11, CSI introduced support for raw block volumes, which relies on the raw block volume feature that was introduced in a previous version of Kubernetes. This feature will make it possible for vendors with external CSI drivers to implement raw block volumes support in Kubernetes workloads.

CSI block volume support is feature-gated, but enabled by default. The two feature gates which must be enabled for this feature are BlockVolume and CSIBlockVolume.

Learn how to [setup your PV/PVC with raw block volume support](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#raw-block-volume-support).

CSI ephemeral volumes

**FEATURE STATE:** Kubernetes v1.16 [beta](https://kubernetes.io/docs/concepts/storage/volumes/)

This feature allows CSI volumes to be directly embedded in the Pod specification instead of a PersistentVolume. Volumes specified in this way are ephemeral and do not persist across Pod restarts.

Example:

kind: Pod

apiVersion: v1

metadata:

name: my-csi-app

spec:

containers:

- name: my-frontend

image: busybox

volumeMounts:

- mountPath: "/data"

name: my-csi-inline-vol

command: [ "sleep", "1000000" ]

volumes:

- name: my-csi-inline-vol

csi:

driver: inline.storage.kubernetes.io

volumeAttributes:

foo: bar

This feature requires CSIInlineVolume feature gate to be enabled. It is enabled by default starting with Kubernetes 1.16.

CSI ephemeral volumes are only supported by a subset of CSI drivers. Please see the list of CSI drivers [here](https://kubernetes-csi.github.io/docs/drivers.html).

Developer resources

For more information on how to develop a CSI driver, refer to the [kubernetes-csi documentation](https://kubernetes-csi.github.io/docs/)

Migrating to CSI drivers from in-tree plugins

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/docs/concepts/storage/volumes/)

The CSI Migration feature, when enabled, directs operations against existing in-tree plugins to corresponding CSI plugins (which are expected to be installed and configured). The feature implements the necessary translation logic and shims to re-route the operations in a seamless fashion. As a result, operators do not have to make any configuration changes to existing Storage Classes, PVs or PVCs (referring to in-tree plugins) when transitioning to a CSI driver that supersedes an in-tree plugin.

In the alpha state, the operations and features that are supported include provisioning/delete, attach/detach, mount/unmount and resizing of volumes.

In-tree plugins that support CSI Migration and have a corresponding CSI driver implemented are listed in the “Types of Volumes” section above.

FlexVolume

FlexVolume is an out-of-tree plugin interface that has existed in Kubernetes since version 1.2 (before CSI). It uses an exec-based model to interface with drivers. FlexVolume driver binaries must be installed in a pre-defined volume plugin path on each node (and in some cases master).

Pods interact with FlexVolume drivers through the flexvolume in-tree plugin. More details can be found [here](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-storage/flexvolume.md).

Mount propagation

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 mountPropagation field in Container.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 private mount propagation as described in the [Linux kernel documentation](https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt)

* 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 [Linux kernel documentation](https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt)

* 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 [Linux kernel documentation](https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt)

**Caution:** 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.

Configuration

Before mount propagation can work properly on some deployments (CoreOS, RedHat/Centos, Ubuntu) mount share must be configured correctly in Docker as shown below.

Edit your Docker’s systemd service file. Set MountFlags as follows:

MountFlags=shared

Or, remove MountFlags=slave if present. Then restart the Docker daemon:

sudo systemctl daemon-reload

sudo systemctl restart docker