Storage

Ways to provide both long-term and temporary storage to Pods in your cluster.

1 - Volumes

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem is the loss of files when a container crashes. The kubelet restarts the container but with a clean state. A second problem occurs when sharing files between containers running together in a `Pod`. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested.

Background

Docker has a concept of volumes, though it is somewhat looser and less managed. A Docker volume is a directory on disk or in another container. Docker provides volume drivers, but the functionality is somewhat limited.

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod. Consequently, a volume outlives any containers that run within the pod, and data is preserved across container restarts. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes.

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`. A process in a container sees a filesystem view composed from their Docker image and volumes. The Docker image is at the root of the filesystem hierarchy. Volumes mount 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's configuration must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of volumes.

awsElasticBlockStore

An `awsElasticBlockStore` volume mounts an Amazon Web Services (AWS) EBS volume into your pod. Unlike `emptyDir`, which is erased when a pod is removed, the contents of an EBS volume are persisted and the volume is unmounted. This means that an EBS volume can be pre-populated with data, and that data can be shared between pods.

> **Note:** You must create an EBS volume by 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 AWS 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. Check that the size and EBS volume type are suitable for your use.

AWS EBS configuration example

```yaml
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
```

If the EBS volume is partitioned, you can supply the optional field `partition: "<partition number>"` to specify which parition to mount on.

AWS EBS CSI migration

FEATURE STATE: `Kubernetes v1.17 [beta]`

The `CSIMigration` feature for `awsElasticBlockStore`, when enabled, redirects 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 must be installed on the cluster and the `CSIMigration` and `CSIMigrationAWS` beta features must be enabled.

AWS EBS CSI migration complete

FEATURE STATE: `Kubernetes v1.17 [alpha]`

To disable the `awsElasticBlockStore` storage plugin from being loaded by the controller manager and the kubelet, set the `CSIMigrationAWSComplete` flag to `true`. This feature requires the `ebs.csi.aws.com` Container Storage Interface (CSI) driver installed on all worker nodes.

azureDisk

The `azureDisk` volume type mounts a Microsoft Azure Data Disk into a pod.

For more details, see the `azureDisk` volume plugin.

azureDisk CSI migration

FEATURE STATE: `Kubernetes v1.19 [beta]`

The `CSIMigration` feature for `azureDisk`, when enabled, redirects 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 must be installed on the cluster and the `CSIMigration` and `CSIMigrationAzureDisk` features must be enabled.

azureFile

The `azureFile` volume type mounts a Microsoft Azure File volume (SMB 2.1 and 3.0) into a pod.

For more details, see the `azureFile` volume plugin.

azureFile CSI migration

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

The `CSIMigration` feature for `azureFile`, when enabled, redirects 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 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 shared between pods. The `cephfs` volume can be mounted by multiple writers simultaneously.

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

See the CephFS example for more details.

cinder

> **Note:** Kubernetes must be configured with the OpenStack cloud provider.

The `cinder` volume type is used to mount the OpenStack Cinder volume into your pod.

Cinder volume configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-cinder
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-cinder-container
```

```yaml
    volumeMounts:
    - mountPath: /test-cinder
      name: test-volume
  volumes:
  - name: test-volume
    # This OpenStack volume must already exist.
    cinder:
      volumeID: "<volume id>"
      fsType: ext4
```

OpenStack CSI migration

FEATURE STATE: `Kubernetes v1.18 [beta]`

The `CSIMigration` feature for Cinder, when enabled, redirects 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 must be installed on the cluster and the `CSIMigration` and `CSIMigrationOpenStack` beta features must be enabled.

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`:

```yaml
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`.

Note:

- You must create a ConfigMap before you can use it.
- A container using a ConfigMap as a `subPath` volume mount will not receive ConfigMap updates.
- 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. It mounts a directory and writes the requested data in plain text files.

> **Note:** A container using the downward API as a subPath volume mount will not receive downward API updates.

See the downward API example 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, 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

Depending on your environment, `emptyDir` volumes are stored on whatever medium that backs the node such as disk or SSD, or network storage. However, 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, tmpfs is cleared on node reboot and any files you write count against your container's memory limit.

> **Note:** If the `SizeMemoryBackedVolumes` feature gate is enabled, you can specify a size for memory backed volumes. If no size is specified, memory backed volumes are sized to 50% of the memory on a Linux host.

emptyDir configuration example

```yaml
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 type allows an existing fibre channel block storage volume to mount 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.

See the fibre channel example for more details.

flocker (deprecated)

Flocker is an open-source, clustered container data volume manager. Flocker 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 shared between pods as required.

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

See the Flocker example for more details.

gcePersistentDisk

A `gcePersistentDisk` volume mounts a Google Compute Engine (GCE) persistent disk (PD) 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 shared between pods.

> **Note:** 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 persistent disk

One feature of GCE persistent disk is concurrent read-only access to a persistent disk. A `gcePersistentDisk` volume permits multiple consumers to simultaneously mount a persistent disk as read-only. 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. Simultaneous writers are not allowed.

Using a GCE persistent disk with a Pod controlled by a ReplicaSet will fail unless the PD is read-only or the replica count is 0 or 1.

Creating a GCE persistent disk

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

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

GCE persistent disk configuration example

```yaml
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

The Regional persistent disks 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. Before creating a PersistentVolume, you must create the persistent disk:

```
gcloud compute disks create --size=500GB my-data-disk
  --region us-central1
  --replica-zones us-central1-a,us-central1-b
```

Regional persistent disk configuration example

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
spec:
  capacity:
    storage: 400Gi
  accessModes:
  - ReadWriteOnce
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: failure-domain.beta.kubernetes.io/zone
          operator: In
          values:
          - us-central1-a
          - us-central1-b
```

GCE CSI migration

FEATURE STATE: `Kubernetes v1.17 [beta]`

The `CSIMigration` feature for GCE PD, when enabled, redirects 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 must be installed on the cluster and the `CSIMigration` and `CSIMigrationGCE` beta features must be enabled.

gitRepo (deprecated)

Warning: The `gitRepo` volume type is deprecated. To provision a container with a git repo, mount an EmptyDir into an InitContainer that clones the repo using git, then mount the EmptyDir into the Pod's container.

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

A `glusterfs` volume allows a Glusterfs (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 shared between pods. GlusterFS can be mounted by multiple writers simultaneously.

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

See the GlusterFS example 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, you 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.

Value	Behavior
`CharDevice`	A character device must exist at the given path
`File`	A file must exist at the given path
`Socket`	A UNIX socket 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
- 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 or modify the file permissions on the host to be able to write to a `hostPath` volume

hostPath configuration example

```
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
```

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 `FileOrCreateconfiguration`.

hostPath FileOrCreate configuration example

```
apiVersion: v1
kind: Pod
metadata:
  name: test-webserver
spec:
  containers:
  - name: test-webserver
    image: k8s.gcr.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
```

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.

See the iSCSI example for more details.

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 by 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`:

```
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.

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

See the NFS example for more details.

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 a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

See the information about PersistentVolumes for more details.

portworxVolume

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:

```
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>"
```

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

For more details, see the Portworx volume examples.

projected

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

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

- `secret`
- `downwardAPI`
- `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.

Example configuration with a secret, a downwardAPI, 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 configuration: 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 into a Pod at a specified path. For 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 a Pod's containers to access the Kubernetes API server. 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` volume mount will not receive updates for those volume sources.

quobyte

A `quobyte` volume allows an existing Quobyte volume to be mounted into your Pod.

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

Quobyte supports the Container Storage Interface. CSI is the recommended plugin to use Quobyte volumes inside Kubernetes. Quobyte's GitHub project has instructions for deploying Quobyte using CSI, along with examples.

rbd

An `rbd` volume allows a Rados Block Device (RBD) volume to mount into your Pod. Unlike `emptyDir`, which is erased when a pod is removed, the contents of an `rbd` volume are preserved and the volume is unmounted. This means that a RBD volume can be pre-populated with data, and that data can be shared between pods.

Note: You must have a 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. Simultaneous writers are not allowed.

See the RBD example for more details.

scaleIO (deprecated)

ScaleIO is a software-based storage platform that uses existing hardware to create clusters of scalable shared block networked storage. The `scaleIO` volume plugin allows deployed pods to access existing ScaleIO volumes. For information about dynamically provisioning new volumes for persistent volume claims, see ScaleIO persistent volumes.

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

The following example is a 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 details, see the ScaleIO examples.

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.

Note: A container using a Secret as a `subPath` volume mount will not receive Secret updates.

For more details, see Configuring Secrets.

storageOS

A `storageos` volume allows an existing StorageOS volume to mount 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 from 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.

The following example is a Pod configuration with StorageOS:

```
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 `defau
        volumeName: redis-vol01
        fsType: ext4
```

For more information about StorageOS, dynamic provisioning, and PersistentVolumeClaims, see the StorageOS examples.

vsphereVolume

Note: You must configure the Kubernetes vSphere Cloud Provider. For cloudprovider configuration, refer to the vSphere Getting Started guide.

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.

Note: You must create vSphere VMDK volume using one of the following methods before using with a Pod.

Creating a VMDK volume

Choose one of the following methods to create a VMDK.

Create using vmkfstools | Create using vmware-vdiskmanager

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

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

vSphere VMDK configuration example

```
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
```

For more information, see the vSphere volume examples.

vSphere CSI migration

FEATURE STATE: `Kubernetes v1.19 [beta]`

The `CSIMigration` feature for `vsphereVolume`, when enabled, redirects all plugin operations from the existing in-tree plugin to the `csi.vsphere.vmware.com` CSI driver. In order to use this feature, the vSphere CSI driver must be installed on the cluster and the `CSIMigration` and `CSIMigrationvSphere` feature gates must be enabled.

This also requires minimum vSphere vCenter/ESXi Version to be 7.0u1 and minimum HW Version to be VM version 15.

> **Note:**
> The following StorageClass parameters from the built-in `vsphereVolume` plugin are not supported by the vSphere CSI driver:
>
> - `diskformat`
> - `hostfailurestotolerate`
> - `forceprovisioning`
> - `cachereservation`
> - `diskstripes`
> - `objectspacereservation`
> - `iopslimit`
>
> Existing volumes created using these parameters will be migrated to the vSphere CSI driver, but new volumes created by the vSphere CSI driver will not be honoring these parameters.

vSphere CSI migration complete

FEATURE STATE: `Kubernetes v1.19 [beta]`

To turn off the `vsphereVolume` plugin from being loaded by the controller manager and the kubelet, you need to set this feature flag to `true`. You must install a `csi.vsphere.vmware.com` CSI driver on all worker nodes.

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:

```
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.

```
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
    volumeMounts:
    - name: workdir1
      mountPath: /logs
      subPathExpr: $(POD_NAME)
  restartPolicy: Never
  volumes:
  - name: workdir1
    hostPath:
      path: /var/log/pods
```

Resources

The storage media (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 between 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 FlexVolume. 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 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 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 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 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 (alpha feature)
- with a CSI ephemeral volume if the driver supports that (beta feature)

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` 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. 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.
- `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.
- `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 v1.18 [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 v1.16 [beta]`

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

Migrating to CSI drivers from in-tree plugins

FEATURE STATE: `Kubernetes v1.17 [beta]`

The `CSIMigration` feature, when enabled, 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.

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

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. 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. For more details, see the FlexVolume examples.

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 the `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

- `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

- `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

  > **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.

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
```

What's next

Follow an example of deploying WordPress and MySQL with Persistent Volumes.

2 - Persistent Volumes

This document describes the current state of *persistent volumes* in Kubernetes. Familiarity with volumes is suggested.

Introduction

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

A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

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

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

See the detailed walkthrough with working examples.

Lifecycle of a volume and claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle:

Provisioning

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

Static

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

Dynamic

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

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

Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a ClaimRef which is a bi-directional binding between the PersistentVolume and the PersistentVolumeClaim.

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

Using

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

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

Storage Object in Use Protection

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

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

If a user deletes a PVC in active use by a Pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods. Also, if an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

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

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

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

```
kubectl describe pv task-pv-volume
Name:            task-pv-volume
Labels:          type=local
```

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

Reclaiming

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

Retain

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

1. Delete the PersistentVolume. The associated storage asset in external infrastructure (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset, or if you want to reuse the same storage asset, create a new PersistentVolume with the storage asset definition.

Delete

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

Recycle

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

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

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pv-recycler
  namespace: default
```

```
spec:
  restartPolicy: Never
  volumes:
  - name: vol
    hostPath:
      path: /any/path/it/will/be/replaced
  containers:
  - name: pv-recycler
    image: "k8s.gcr.io/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]* /
    volumeMounts:
    - name: vol
      mountPath: /scrub
```

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

Reserving a PersistentVolume

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

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

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

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

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

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

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

Expanding Persistent Volumes Claims

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

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

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

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

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

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

CSI Volume expansion

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

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

Resizing a volume containing a file system

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

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

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

Resizing an in-use PersistentVolumeClaim

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

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

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

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

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

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

Recovering from Failure when Expanding Volumes

If expanding underlying storage fails, the cluster administrator can manually recover the Persistent Volume Claim (PVC) state and cancel the resize requests. Otherwise, the resize requests are continuously retried by the controller without administrator intervention.

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

Types of Persistent Volumes

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

- `awsElasticBlockStore` - AWS Elastic Block Store (EBS)
- `azureDisk` - Azure Disk
- `azureFile` - Azure File
- `cephfs` - CephFS volume
- `cinder` - Cinder (OpenStack block storage) (**deprecated**)
- `csi` - Container Storage Interface (CSI)
- `fc` - Fibre Channel (FC) storage
- `flexVolume` - FlexVolume
- `flocker` - Flocker storage
- `gcePersistentDisk` - GCE Persistent Disk
- `glusterfs` - Glusterfs volume
- `hostPath` - HostPath volume (for single node testing only; WILL NOT WORK in a multi-node cluster; consider using `local` volume instead)
- `iscsi` - iSCSI (SCSI over IP) storage
- `local` - local storage devices mounted on nodes.
- `nfs` - Network File System (NFS) storage
- `photonPersistentDisk` - Photon controller persistent disk. (This volume type no longer works since the removal of the corresponding cloud provider.)
- `portworxVolume` - Portworx volume
- `quobyte` - Quobyte volume
- `rbd` - Rados Block Device (RBD) volume
- `scaleIO` - ScaleIO volume (**deprecated**)
- `storageos` - StorageOS volume
- `vsphereVolume` - vSphere VMDK volume

Persistent Volumes

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

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

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

Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV's `capacity` attribute. See the Kubernetes Resource Model to understand the units expected by `capacity`.

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

Volume Mode

FEATURE STATE: `Kubernetes v1.18 [stable]`

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

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

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

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

Access Modes

A PersistentVolume can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

The access modes are:

- ReadWriteOnce -- the volume can be mounted as read-write by a single node
- ReadOnlyMany -- the volume can be mounted read-only by many nodes
- ReadWriteMany -- the volume can be mounted as read-write by many nodes

In the CLI, the access modes are abbreviated to:

- RWO - ReadWriteOnce
- ROX - ReadOnlyMany
- RWX - ReadWriteMany

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

Volume Plugin	**ReadWriteOnce**	**ReadOnlyMany**	**ReadWriteMany**
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
CSI	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-
FlexVolume	✓	✓	depends on the driver
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-
VsphereVolume	✓	-	- (works when Pods are collocated)
PortworxVolume	✓	-	✓

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
ScaleIO	✓	✓	-
StorageOS	✓	-	-

Class

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

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

Reclaim Policy

Current reclaim policies are:

- Retain -- manual reclamation
- Recycle -- basic scrub (`rm -rf /thevolume/*`)
- Delete -- associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

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

Mount Options

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

Note: Not all Persistent Volume types support mount options.

The following volume types support mount options:

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

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

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

Node Affinity

Note: For most volume types, you do not need to set this field. It is automatically populated for AWS EBS, GCE PD and Azure Disk volume block types. You need to explicitly set this for local volumes.

A PV can specify node affinity to define constraints that limit what nodes this volume can be accessed from. Pods that use a PV will only be scheduled to nodes that are selected by the node affinity.

Phase

A volume will be in one of the following phases:

- Available -- a free resource that is not yet bound to a claim
- Bound -- the volume is bound to a claim
- Released -- the claim has been deleted, but the resource is not yet reclaimed by the cluster
- Failed -- the volume has failed its automatic reclamation

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

PersistentVolumeClaims

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

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

Access Modes

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

Volume Modes

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

Resources

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

Selector

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

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

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

Class

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

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

- If the admission plugin is turned on, the administrator may specify a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs of that default. Specifying a default StorageClass is done by setting the annotation `storageclass.kubernetes.io/is-default-class` equal to `true` in a StorageClass object. If the administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default is specified, the admission plugin forbids the creation of all PVCs.
- If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have no `storageClassName` can be bound only to PVs that have no class. In this case, the PVCs that have no `storageClassName` are treated the same way as PVCs that have their `storageClassName` set to `""`.

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

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

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

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

Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the Pod using the claim. The cluster finds the claim in the Pod's namespace and uses it to get the PersistentVolume backing the claim. The volume is then mounted to the host and into the Pod.

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

A Note on Namespaces

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

PersistentVolumes typed `hostPath`

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

Raw Block Volume Support

FEATURE STATE: `Kubernetes v1.18 [stable]`

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

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

PersistentVolume using a Raw Block Volume

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

PersistentVolumeClaim requesting a Raw Block Volume

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

Pod specification adding Raw Block Device path in container

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]
      volumeDevices:
        - name: data
          devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

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

Binding Block Volumes

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

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

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

Volume Snapshot and Restore Volume from Snapshot Support

FEATURE STATE: `Kubernetes v1.20 [stable]`

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

Create a PersistentVolumeClaim from a Volume Snapshot

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

Volume Cloning

Volume Cloning only available for CSI volume plugins.

Create PersistentVolumeClaim from an existing PVC

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

Writing Portable Configuration

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

- Include PersistentVolumeClaim objects in your bundle of config (alongside Deployments, ConfigMaps, etc).
- Do not include PersistentVolume objects in the config, since the user instantiating the config may not have permission to create PersistentVolumes.

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

What's next

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

Reference

- PersistentVolume
- PersistentVolumeSpec
- PersistentVolumeClaim
- PersistentVolumeClaimSpec

3 - Volume Snapshots

In Kubernetes, a *VolumeSnapshot* represents a snapshot of a volume on a storage system. This document assumes that you are already familiar with Kubernetes persistent volumes.

Introduction

Similar to how API resources `PersistentVolume` and `PersistentVolumeClaim` are used to provision volumes for users and administrators, `VolumeSnapshotContent` and `VolumeSnapshot` API resources are provided to create volume snapshots for users and administrators.

A `VolumeSnapshotContent` is a snapshot taken from a volume in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a PersistentVolume is a cluster resource.

A `VolumeSnapshot` is a request for snapshot of a volume by a user. It is similar to a PersistentVolumeClaim.

`VolumeSnapshotClass` allows you to specify different attributes belonging to a `VolumeSnapshot`. These attributes may differ among snapshots taken from the same volume on the storage system and therefore cannot be expressed by using the same `StorageClass` of a `PersistentVolumeClaim`.

Volume snapshots provide Kubernetes users with a standardized way to copy a volume's contents at a particular point in time without creating an entirely new volume. This functionality enables, for example, database administrators to backup databases before performing edit or delete modifications.

Users need to be aware of the following when using this feature:

- API Objects `VolumeSnapshot`, `VolumeSnapshotContent`, and `VolumeSnapshotClass` are CRDs, not part of the core API.
- `VolumeSnapshot` support is only available for CSI drivers.
- As part of the deployment process of `VolumeSnapshot`, the Kubernetes team provides a snapshot controller to be deployed into the control plane, and a sidecar helper container called csi-snapshotter to be deployed together with the CSI driver. The snapshot controller watches `VolumeSnapshot` and `VolumeSnapshotContent` objects and is responsible for the creation and deletion of `VolumeSnapshotContent` object. The sidecar csi-snapshotter watches `VolumeSnapshotContent` objects and triggers `CreateSnapshot` and `DeleteSnapshot` operations against a CSI endpoint.
- There is also a validating webhook server which provides tightened validation on snapshot objects. This should be installed by the Kubernetes distros along with the snapshot controller and CRDs, not CSI drivers. It should be installed in all Kubernetes clusters that has the snapshot feature enabled.
- CSI drivers may or may not have implemented the volume snapshot functionality. The CSI drivers that have provided support for volume snapshot will likely use the csi-snapshotter. See CSI Driver documentation for details.
- The CRDs and snapshot controller installations are the responsibility of the Kubernetes distribution.

Lifecycle of a volume snapshot and volume snapshot content

`VolumeSnapshotContents` are resources in the cluster. `VolumeSnapshots` are requests for those resources. The interaction between `VolumeSnapshotContents` and `VolumeSnapshots` follow this lifecycle:

Provisioning Volume Snapshot

There are two ways snapshots may be provisioned: pre-provisioned or dynamically provisioned.

Pre-provisioned

A cluster administrator creates a number of `VolumeSnapshotContents`. They carry the details of the real volume snapshot on the storage system which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

Instead of using a pre-existing snapshot, you can request that a snapshot to be dynamically taken from a PersistentVolumeClaim. The VolumeSnapshotClass specifies storage provider-specific parameters to use when taking a snapshot.

Binding

The snapshot controller handles the binding of a `VolumeSnapshot` object with an appropriate `VolumeSnapshotContent` object, in both pre-provisioned and dynamically provisioned scenarios. The binding is a one-to-one mapping.

In the case of pre-provisioned binding, the VolumeSnapshot will remain unbound until the requested VolumeSnapshotContent object is created.

Persistent Volume Claim as Snapshot Source Protection

The purpose of this protection is to ensure that in-use PersistentVolumeClaim API objects are not removed from the system while a snapshot is being taken from it (as this may result in data loss).

While a snapshot is being taken of a PersistentVolumeClaim, that PersistentVolumeClaim is in-use. If you delete a PersistentVolumeClaim API object in active use as a snapshot source, the PersistentVolumeClaim object is not removed immediately. Instead, removal of the PersistentVolumeClaim object is postponed until the snapshot is readyToUse or aborted.

Delete

Deletion is triggered by deleting the `VolumeSnapshot` object, and the `DeletionPolicy` will be followed. If the `DeletionPolicy` is `Delete`, then the underlying storage snapshot will be deleted along with the `VolumeSnapshotContent` object. If the `DeletionPolicy` is `Retain`, then both the underlying snapshot and `VolumeSnapshotContent` remain.

VolumeSnapshots

Each VolumeSnapshot contains a spec and a status.

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
spec:
  volumeSnapshotClassName: csi-hostpath-snapclass
  source:
    persistentVolumeClaimName: pvc-test
```

`persistentVolumeClaimName` is the name of the PersistentVolumeClaim data source for the snapshot. This field is required for dynamically provisioning a snapshot.

A volume snapshot can request a particular class by specifying the name of a VolumeSnapshotClass using the attribute `volumeSnapshotClassName`. If nothing is set, then the default class is used if available.

For pre-provisioned snapshots, you need to specify a `volumeSnapshotContentName` as the source for the snapshot as shown in the following example. The `volumeSnapshotContentName` source field is required for pre-provisioned snapshots.

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: test-snapshot
spec:
  source:
    volumeSnapshotContentName: test-content
```

Volume Snapshot Contents

Each VolumeSnapshotContent contains a spec and status. In dynamic provisioning, the snapshot common controller creates `VolumeSnapshotContent` objects. Here is an example:

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: snapcontent-72d9a349-aacd-42d2-a240-d775650d2455
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    volumeHandle: ee0cfb94-f8d4-11e9-b2d8-0242ac110002
  volumeSnapshotClassName: csi-hostpath-snapclass
  volumeSnapshotRef:
    name: new-snapshot-test
    namespace: default
    uid: 72d9a349-aacd-42d2-a240-d775650d2455
```

`volumeHandle` is the unique identifier of the volume created on the storage backend and returned by the CSI driver during the volume creation. This field is required for dynamically provisioning a snapshot. It specifies the volume source of the snapshot.

For pre-provisioned snapshots, you (as cluster administrator) are responsible for creating the `VolumeSnapshotContent` object as follows.

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: new-snapshot-content-test
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    snapshotHandle: 7bdd0de3-aaeb-11e8-9aae-0242ac110002
  volumeSnapshotRef:
    name: new-snapshot-test
    namespace: default
```

`snapshotHandle` is the unique identifier of the volume snapshot created on the storage backend. This field is required for the pre-provisioned snapshots. It specifies the CSI snapshot id on the storage system that this `VolumeSnapshotContent` represents.

Provisioning Volumes from Snapshots

You can provision a new volume, pre-populated with data from a snapshot, by using the *dataSource* field in the `PersistentVolumeClaim` object.

For more details, see Volume Snapshot and Restore Volume from Snapshot.

4 - CSI Volume Cloning

This document describes the concept of cloning existing CSI Volumes in Kubernetes. Familiarity with Volumes is suggested.

Introduction

The CSI Volume Cloning feature adds support for specifying existing PVCs in the `dataSource` field to indicate a user would like to clone a Volume.

A Clone is defined as a duplicate of an existing Kubernetes Volume that can be consumed as any standard Volume would be. The only difference is that upon provisioning, rather than creating a "new" empty Volume, the back end device creates an exact duplicate of the specified Volume.

The implementation of cloning, from the perspective of the Kubernetes API, adds the ability to specify an existing PVC as a dataSource during new PVC creation. The source PVC must be bound and available (not in use).

Users need to be aware of the following when using this feature:

- Cloning support (`VolumePVCDataSource`) is only available for CSI drivers.
- Cloning support is only available for dynamic provisioners.
- CSI drivers may or may not have implemented the volume cloning functionality.
- You can only clone a PVC when it exists in the same namespace as the destination PVC (source and destination must be in the same namespace).
- Cloning is only supported within the same Storage Class.
  - Destination volume must be the same storage class as the source
  - Default storage class can be used and storageClassName omitted in the spec
- Cloning can only be performed between two volumes that use the same VolumeMode setting (if you request a block mode volume, the source MUST also be block mode)

Provisioning

Clones are provisioned like any other PVC with the exception of adding a dataSource that references an existing PVC in the same namespace.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: clone-of-pvc-1
    namespace: myns
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: cloning
  resources:
    requests:
      storage: 5Gi
  dataSource:
    kind: PersistentVolumeClaim
    name: pvc-1
```

Note: You must specify a capacity value for `spec.resources.requests.storage`, and the value you specify must be the same or larger than the capacity of the source volume.

The result is a new PVC with the name `clone-of-pvc-1` that has the exact same content as the specified source `pvc-1` .

Usage

Upon availability of the new PVC, the cloned PVC is consumed the same as other PVC. It's also expected at this point that the newly created PVC is an independent object. It can be consumed, cloned, snapshotted, or deleted independently and without consideration for it's original dataSource PVC. This also implies that the source is not linked in any way to the newly created clone, it may also be modified or deleted without affecting the newly created clone.

5 - Storage Classes

This document describes the concept of a StorageClass in Kubernetes. Familiarity with volumes and persistent volumes is suggested.

Introduction

A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called "profiles" in other storage systems.

The StorageClass Resource

Each StorageClass contains the fields `provisioner`, `parameters`, and `reclaimPolicy`, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned.

The name of a StorageClass object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating StorageClass objects, and the objects cannot be updated once they are created.

Administrators can specify a default StorageClass only for PVCs that don't request any particular class to bind to: see the PersistentVolumeClaim section for details.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

Provisioner

Each StorageClass has a provisioner that determines what volume plugin is used for provisioning PVs. This field must be specified.

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-

Volume Plugin	Internal Provisioner	Config Example
GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

You are not restricted to specifying the "internal" provisioners listed here (whose names are prefixed with "kubernetes.io" and shipped alongside Kubernetes). You can also run and specify external provisioners, which are independent programs that follow a specification defined by Kubernetes. Authors of external provisioners have full discretion over where their code lives, how the provisioner is shipped, how it needs to be run, what volume plugin it uses (including Flex), etc. The repository kubernetes-sigs/sig-storage-lib-external-provisioner houses a library for writing external provisioners that implements the bulk of the specification. Some external provisioners are listed under the repository kubernetes-sigs/sig-storage-lib-external-provisioner.

For example, NFS doesn't provide an internal provisioner, but an external provisioner can be used. There are also cases when 3rd party storage vendors provide their own external provisioner.

Reclaim Policy

PersistentVolumes that are dynamically created by a StorageClass will have the reclaim policy specified in the `reclaimPolicy` field of the class, which can be either `Delete` or `Retain`. If no `reclaimPolicy` is specified when a StorageClass object is created, it will default to `Delete`.

PersistentVolumes that are created manually and managed via a StorageClass will have whatever reclaim policy they were assigned at creation.

Allow Volume Expansion

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

PersistentVolumes can be configured to be expandable. This feature when set to `true`, allows the users to resize the volume by editing the corresponding PVC object.

The following types of volumes support volume expansion, when the underlying StorageClass has the field `allowVolumeExpansion` set to true.

Volume type	Required Kubernetes version
gcePersistentDisk	1.11
awsElasticBlockStore	1.11

Volume type	Required Kubernetes version
Cinder	1.11
glusterfs	1.11
rbd	1.11
Azure File	1.11
Azure Disk	1.11
Portworx	1.11
FlexVolume	1.13
CSI	1.14 (alpha), 1.16 (beta)

Note: You can only use the volume expansion feature to grow a Volume, not to shrink it.

Mount Options

PersistentVolumes that are dynamically created by a StorageClass will have the mount options specified in the `mountOptions` field of the class.

If the volume plugin does not support mount options but mount options are specified, provisioning will fail. Mount options are not validated on either the class or PV. If a mount option is invalid, the PV mount fails.

Volume Binding Mode

The `volumeBindingMode` field controls when volume binding and dynamic provisioning should occur.

By default, the `Immediate` mode indicates that volume binding and dynamic provisioning occurs once the PersistentVolumeClaim is created. For storage backends that are topology-constrained and not globally accessible from all Nodes in the cluster, PersistentVolumes will be bound or provisioned without knowledge of the Pod's scheduling requirements. This may result in unschedulable Pods.

A cluster administrator can address this issue by specifying the `WaitForFirstConsumer` mode which will delay the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created. PersistentVolumes will be selected or provisioned conforming to the topology that is specified by the Pod's scheduling constraints. These include, but are not limited to, resource requirements, node selectors, pod affinity and anti-affinity, and taints and tolerations.

The following plugins support `WaitForFirstConsumer` with dynamic provisioning:

- AWSElasticBlockStore
- GCEPersistentDisk
- AzureDisk

The following plugins support `WaitForFirstConsumer` with pre-created PersistentVolume binding:

- All of the above
- Local

FEATURE STATE: `Kubernetes v1.17 [stable]`

CSI volumes are also supported with dynamic provisioning and pre-created PVs, but you'll need to look at the documentation for a specific CSI driver to see its supported topology keys and examples.

Allowed Topologies

When a cluster operator specifies the `WaitForFirstConsumer` volume binding mode, it is no longer necessary to restrict provisioning to specific topologies in most situations. However, if still required, `allowedTopologies` can be specified.

This example demonstrates how to restrict the topology of provisioned volumes to specific zones and should be used as a replacement for the `zone` and `zones` parameters for the supported plugins.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
volumeBindingMode: WaitForFirstConsumer
allowedTopologies:
- matchLabelExpressions:
  - key: failure-domain.beta.kubernetes.io/zone
    values:
    - us-central1-a
    - us-central1-b
```

Parameters

Storage Classes have parameters that describe volumes belonging to the storage class. Different parameters may be accepted depending on the `provisioner`. For example, the value `io1`, for the parameter `type`, and the parameter `iopsPerGB` are specific to EBS. When a parameter is omitted, some default is used.

There can be at most 512 parameters defined for a StorageClass. The total length of the parameters object including its keys and values cannot exceed 256 KiB.

AWS EBS

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```

- `type`: `io1`, `gp2`, `sc1`, `st1`. See AWS docs for details. Default: `gp2`.
- `zone` (Deprecated): AWS zone. If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.
- `zones` (Deprecated): A comma separated list of AWS zone(s). If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.
- `iopsPerGB`: only for `io1` volumes. I/O operations per second per GiB. AWS volume plugin multiplies this with size of requested volume to compute IOPS of the volume and caps it at 20 000 IOPS (maximum supported by AWS, see AWS docs. A string is expected here, i.e. `"10"`, not `10`.

- `fsType`: fsType that is supported by kubernetes. Default: `"ext4"`.
- `encrypted`: denotes whether the EBS volume should be encrypted or not. Valid values are `"true"` or `"false"`. A string is expected here, i.e. `"true"`, not `true`.
- `kmsKeyId`: optional. The full Amazon Resource Name of the key to use when encrypting the volume. If none is supplied but `encrypted` is true, a key is generated by AWS. See AWS docs for valid ARN value.

> **Note:** `zone` and `zones` parameters are deprecated and replaced with allowedTopologies

GCE PD

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fstype: ext4
  replication-type: none
```

- `type`: `pd-standard` or `pd-ssd`. Default: `pd-standard`
- `zone` (Deprecated): GCE zone. If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.
- `zones` (Deprecated): A comma separated list of GCE zone(s). If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.
- `fstype`: `ext4` or `xfs`. Default: `ext4`. The defined filesystem type must be supported by the host operating system.
- `replication-type`: `none` or `regional-pd`. Default: `none`.

If `replication-type` is set to `none`, a regular (zonal) PD will be provisioned.

If `replication-type` is set to `regional-pd`, a Regional Persistent Disk will be provisioned. It's highly recommended to have `volumeBindingMode: WaitForFirstConsumer` set, in which case when you create a Pod that consumes a PersistentVolumeClaim which uses this StorageClass, a Regional Persistent Disk is provisioned with two zones. One zone is the same as the zone that the Pod is scheduled in. The other zone is randomly picked from the zones available to the cluster. Disk zones can be further constrained using `allowedTopologies`.

> **Note:** `zone` and `zones` parameters are deprecated and replaced with allowedTopologies

Glusterfs

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://127.0.0.1:8081"
  clusterid: "630372ccdc720a92c681fb928f27b53f"
  restauthenabled: "true"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"
```

- `resturl` : Gluster REST service/Heketi service url which provision gluster volumes on demand. The general format should be `IPaddress:Port` and this is a mandatory parameter for GlusterFS dynamic provisioner. If Heketi service is exposed as a routable service in openshift/kubernetes setup, this can have a format similar to `http://heketi-storage-project.cloudapps.mystorage.com` where the fqdn is a resolvable Heketi service url.

- `restauthenabled` : Gluster REST service authentication boolean that enables authentication to the REST server. If this value is `"true"`, `restuser` and `restuserkey` or `secretNamespace` + `secretName` have to be filled. This option is deprecated, authentication is enabled when any of `restuser`, `restuserkey`, `secretName` or `secretNamespace` is specified.

- `restuser` : Gluster REST service/Heketi user who has access to create volumes in the Gluster Trusted Pool.

- `restuserkey` : Gluster REST service/Heketi user's password which will be used for authentication to the REST server. This parameter is deprecated in favor of `secretNamespace` + `secretName`.

- `secretNamespace`, `secretName` : Identification of Secret instance that contains user password to use when talking to Gluster REST service. These parameters are optional, empty password will be used when both `secretNamespace` and `secretName` are omitted. The provided secret must have type `"kubernetes.io/glusterfs"`, for example created in this way:

  ```
  kubectl create secret generic heketi-secret \
    --type="kubernetes.io/glusterfs" --from-literal=key='opensesame' \
    --namespace=default
  ```

  Example of a secret can be found in glusterfs-provisioning-secret.yaml.

- `clusterid`: `630372ccdc720a92c681fb928f27b53f` is the ID of the cluster which will be used by Heketi when provisioning the volume. It can also be a list of clusterids, for example: `"8452344e2becec931ece4e33c4674e4e,42982310de6c63381718ccfa6d8cf397"`. This is an optional parameter.

- `gidMin`, `gidMax` : The minimum and maximum value of GID range for the StorageClass. A unique value (GID) in this range (gidMin-gidMax) will be used for dynamically provisioned volumes. These are optional values. If not specified, the volume will be provisioned with a value between 2000-2147483647 which are defaults for gidMin and gidMax respectively.

- `volumetype` : The volume type and its parameters can be configured with this optional value. If the volume type is not mentioned, it's up to the provisioner to decide the volume type.

  For example:

  - Replica volume: `volumetype: replicate:3` where '3' is replica count.
  - Disperse/EC volume: `volumetype: disperse:4:2` where '4' is data and '2' is the redundancy count.
  - Distribute volume: `volumetype: none`

  For available volume types and administration options, refer to the Administration Guide.

  For further reference information, see How to configure Heketi.

  When persistent volumes are dynamically provisioned, the Gluster plugin automatically creates an endpoint and a headless service in the name `gluster-dynamic-<claimname>`. The dynamic endpoint and service are automatically deleted when the persistent volume claim is deleted.

OpenStack Cinder

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  availability: nova
```

- `availability`: Availability Zone. If not specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node.

> **Note:**
>
> **FEATURE STATE:** `Kubernetes v1.11 [deprecated]`
>
> This internal provisioner of OpenStack is deprecated. Please use the external cloud provider for OpenStack.

vSphere

There are two types of provisioners for vSphere storage classes:

- CSI provisioner: `csi.vsphere.vmware.com`
- vCP provisioner: `kubernetes.io/vsphere-volume`

In-tree provisioners are deprecated. For more information on the CSI provisioner, see Kubernetes vSphere CSI Driver and vSphereVolume CSI migration.

CSI Provisioner

The vSphere CSI StorageClass provisioner works with Tanzu Kubernetes clusters. For an example, refer to the vSphere CSI repository.

vCP Provisioner

The following examples use the VMware Cloud Provider (vCP) StorageClass provisioner.

1. Create a StorageClass with a user specified disk format.

   ```yaml
   apiVersion: storage.k8s.io/v1
   kind: StorageClass
   metadata:
     name: fast
   provisioner: kubernetes.io/vsphere-volume
   parameters:
     diskformat: zeroedthick
   ```

   `diskformat`: `thin`, `zeroedthick` and `eagerzeroedthick`. Default: `"thin"`.

2. Create a StorageClass with a disk format on a user specified datastore.

   ```yaml
   apiVersion: storage.k8s.io/v1
   kind: StorageClass
   metadata:
     name: fast
   provisioner: kubernetes.io/vsphere-volume
   parameters:
       diskformat: zeroedthick
       datastore: VSANDatastore
   ```

`datastore`: The user can also specify the datastore in the StorageClass. The volume will be created on the datastore specified in the StorageClass, which in this case is `VSANDatastore`. This field is optional. If the datastore is not specified, then the volume will be created on the datastore specified in the vSphere config file used to initialize the vSphere Cloud Provider.

3. Storage Policy Management inside kubernetes

    - Using existing vCenter SPBM policy

      One of the most important features of vSphere for Storage Management is policy based Management. Storage Policy Based Management (SPBM) is a storage policy framework that provides a single unified control plane across a broad range of data services and storage solutions. SPBM enables vSphere administrators to overcome upfront storage provisioning challenges, such as capacity planning, differentiated service levels and managing capacity headroom.

      The SPBM policies can be specified in the StorageClass using the `storagePolicyName` parameter.

    - Virtual SAN policy support inside Kubernetes

      Vsphere Infrastructure (VI) Admins will have the ability to specify custom Virtual SAN Storage Capabilities during dynamic volume provisioning. You can now define storage requirements, such as performance and availability, in the form of storage capabilities during dynamic volume provisioning. The storage capability requirements are converted into a Virtual SAN policy which are then pushed down to the Virtual SAN layer when a persistent volume (virtual disk) is being created. The virtual disk is distributed across the Virtual SAN datastore to meet the requirements.

      You can see Storage Policy Based Management for dynamic provisioning of volumes for more details on how to use storage policies for persistent volumes management.

There are few vSphere examples which you try out for persistent volume management inside Kubernetes for vSphere.

Ceph RBD

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/rbd
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  userSecretNamespace: default
  fsType: ext4
  imageFormat: "2"
  imageFeatures: "layering"
```

- `monitors`: Ceph monitors, comma delimited. This parameter is required.
- `adminId`: Ceph client ID that is capable of creating images in the pool. Default is "admin".
- `adminSecretName`: Secret Name for `adminId`. This parameter is required. The provided secret must have type "kubernetes.io/rbd".
- `adminSecretNamespace`: The namespace for `adminSecretName`. Default is "default".

- `pool`: Ceph RBD pool. Default is "rbd".
- `userId`: Ceph client ID that is used to map the RBD image. Default is the same as `adminId`.
- `userSecretName`: The name of Ceph Secret for `userId` to map RBD image. It must exist in the same namespace as PVCs. This parameter is required. The provided secret must have type "kubernetes.io/rbd", for example created in this way:

```
kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \
  --from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9PQ=='
  --namespace=kube-system
```

- `userSecretNamespace`: The namespace for `userSecretName`.
- `fsType`: fsType that is supported by kubernetes. Default: `"ext4"`.
- `imageFormat`: Ceph RBD image format, "1" or "2". Default is "2".
- `imageFeatures`: This parameter is optional and should only be used if you set `imageFormat` to "2". Currently supported features are `layering` only. Default is "", and no features are turned on.

Quobyte

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: slow
provisioner: kubernetes.io/quobyte
parameters:
    quobyteAPIServer: "http://138.68.74.142:7860"
    registry: "138.68.74.142:7861"
    adminSecretName: "quobyte-admin-secret"
    adminSecretNamespace: "kube-system"
    user: "root"
    group: "root"
    quobyteConfig: "BASE"
    quobyteTenant: "DEFAULT"
```

- `quobyteAPIServer`: API Server of Quobyte in the format `"http(s)://api-server:7860"`
- `registry`: Quobyte registry to use to mount the volume. You can specify the registry as `<host>:<port>` pair or if you want to specify multiple registries, put a comma between them. `<host1>:<port>,<host2>:<port>,<host3>:<port>`. The host can be an IP address or if you have a working DNS you can also provide the DNS names.
- `adminSecretNamespace`: The namespace for `adminSecretName`. Default is "default".
- `adminSecretName`: secret that holds information about the Quobyte user and the password to authenticate against the API server. The provided secret must have type "kubernetes.io/quobyte" and the keys `user` and `password`, for example:

```
kubectl create secret generic quobyte-admin-secret \
  --type="kubernetes.io/quobyte" --from-literal=user='admin' --from-literal=pas
  --namespace=kube-system
```

- `user`: maps all access to this user. Default is "root".
- `group`: maps all access to this group. Default is "nfsnobody".

- `quobyteConfig`: use the specified configuration to create the volume. You can create a new configuration or modify an existing one with the Web console or the quobyte CLI. Default is "BASE".

- `quobyteTenant`: use the specified tenant ID to create/delete the volume. This Quobyte tenant has to be already present in Quobyte. Default is "DEFAULT".

Azure Disk

Azure Unmanaged Disk storage class

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  skuName: Standard_LRS
  location: eastus
  storageAccount: azure_storage_account_name
```

- `skuName`: Azure storage account Sku tier. Default is empty.
- `location`: Azure storage account location. Default is empty.
- `storageAccount`: Azure storage account name. If a storage account is provided, it must reside in the same resource group as the cluster, and `location` is ignored. If a storage account is not provided, a new storage account will be created in the same resource group as the cluster.

Azure Disk storage class (starting from v1.7.2)

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  storageaccounttype: Standard_LRS
  kind: Shared
```

- `storageaccounttype`: Azure storage account Sku tier. Default is empty.
- `kind`: Possible values are `shared` (default), `dedicated`, and `managed`. When `kind` is `shared`, all unmanaged disks are created in a few shared storage accounts in the same resource group as the cluster. When `kind` is `dedicated`, a new dedicated storage account will be created for the new unmanaged disk in the same resource group as the cluster. When `kind` is `managed`, all managed disks are created in the same resource group as the cluster.
- `resourceGroup`: Specify the resource group in which the Azure disk will be created. It must be an existing resource group name. If it is unspecified, the disk will be placed in the same resource group as the current Kubernetes cluster.

- Premium VM can attach both Standard_LRS and Premium_LRS disks, while Standard VM can only attach Standard_LRS disks.
- Managed VM can only attach managed disks and unmanaged VM can only attach unmanaged disks.

Azure File

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
parameters:
  skuName: Standard_LRS
  location: eastus
  storageAccount: azure_storage_account_name
```

- `skuName`: Azure storage account Sku tier. Default is empty.
- `location`: Azure storage account location. Default is empty.
- `storageAccount`: Azure storage account name. Default is empty. If a storage account is not provided, all storage accounts associated with the resource group are searched to find one that matches `skuName` and `location`. If a storage account is provided, it must reside in the same resource group as the cluster, and `skuName` and `location` are ignored.
- `secretNamespace`: the namespace of the secret that contains the Azure Storage Account Name and Key. Default is the same as the Pod.
- `secretName`: the name of the secret that contains the Azure Storage Account Name and Key. Default is `azure-storage-account-<accountName>-secret`
- `readOnly`: a flag indicating whether the storage will be mounted as read only. Defaults to false which means a read/write mount. This setting will impact the `ReadOnly` setting in VolumeMounts as well.

During storage provisioning, a secret named by `secretName` is created for the mounting credentials. If the cluster has enabled both RBAC and Controller Roles, add the `create` permission of resource `secret` for clusterrole `system:controller:persistent-volume-binder`.

In a multi-tenancy context, it is strongly recommended to set the value for `secretNamespace` explicitly, otherwise the storage account credentials may be read by other users.

Portworx Volume

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: portworx-io-priority-high
provisioner: kubernetes.io/portworx-volume
parameters:
  repl: "1"
  snap_interval:   "70"
  priority_io:  "high"

```

- `fs`: filesystem to be laid out: `none/xfs/ext4` (default: `ext4`).
- `block_size`: block size in Kbytes (default: `32`).
- `repl`: number of synchronous replicas to be provided in the form of replication factor `1..3` (default: `1`) A string is expected here i.e. `"1"` and not `1`.
- `priority_io`: determines whether the volume will be created from higher performance or a lower priority storage `high/medium/low` (default: `low`).
- `snap_interval`: clock/time interval in minutes for when to trigger snapshots. Snapshots are incremental based on difference with the prior snapshot, 0 disables snaps (default: `0`). A string is expected here i.e. `"70"` and not `70`.
- `aggregation_level`: specifies the number of chunks the volume would be distributed into, 0 indicates a non-aggregated volume (default: `0`). A string is expected here i.e. `"0"` and not `0`
- `ephemeral`: specifies whether the volume should be cleaned-up after unmount or should be persistent. `emptyDir` use case can set this value to true and `persistent volumes` use case such as for databases like Cassandra should set to false, `true/false` (default `false`). A string is expected here i.e. `"true"` and not `true`.

ScaleIO

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/scaleio
parameters:
  gateway: https://192.168.99.200:443/api
  system: scaleio
  protectionDomain: pd0
  storagePool: sp1
  storageMode: ThinProvisioned
  secretRef: sio-secret
  readOnly: false
  fsType: xfs
```

- `provisioner` : attribute is set to `kubernetes.io/scaleio`
- `gateway` : address to a ScaleIO API gateway (required)
- `system` : the name of the ScaleIO system (required)
- `protectionDomain` : the name of the ScaleIO protection domain (required)
- `storagePool` : the name of the volume storage pool (required)
- `storageMode` : the storage provision mode: `ThinProvisioned` (default) or `ThickProvisioned`
- `secretRef` : reference to a configured Secret object (required)
- `readOnly` : specifies the access mode to the mounted volume (default false)
- `fsType` : the file system to use for the volume (default ext4)

The ScaleIO Kubernetes volume plugin requires a configured Secret object. The secret must be created with type `kubernetes.io/scaleio` and use the same namespace value as that of the PVC where it is referenced as shown in the following command:

```
kubectl create secret generic sio-secret --type="kubernetes.io/scaleio" \
--from-literal=username=sioadmin --from-literal=password=d2NABDNjMA== \
--namespace=default
```

StorageOS

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/storageos
parameters:
  pool: default
  description: Kubernetes volume
  fsType: ext4
  adminSecretNamespace: default
  adminSecretName: storageos-secret
```

- `pool` : The name of the StorageOS distributed capacity pool to provision the volume from. Uses the `default` pool which is normally present if not specified.
- `description` : The description to assign to volumes that were created dynamically. All volume descriptions will be the same for the storage class, but different storage classes can be used to allow descriptions for different use cases. Defaults to `Kubernetes volume` .

- `fsType` : The default filesystem type to request. Note that user-defined rules within StorageOS may override this value. Defaults to `ext4` .
- `adminSecretNamespace` : The namespace where the API configuration secret is located. Required if adminSecretName set.
- `adminSecretName` : The name of the secret to use for obtaining the StorageOS API credentials. If not specified, default values will be attempted.

The StorageOS Kubernetes volume plugin can use a Secret object to specify an endpoint and credentials to access the StorageOS API. This is only required when the defaults have been changed. The secret must be created with type `kubernetes.io/storageos` as shown in the following command:

```
kubectl create secret generic storageos-secret \
--type="kubernetes.io/storageos" \
--from-literal=apiAddress=tcp://localhost:5705 \
--from-literal=apiUsername=storageos \
--from-literal=apiPassword=storageos \
--namespace=default
```

Secrets used for dynamically provisioned volumes may be created in any namespace and referenced with the `adminSecretNamespace` parameter. Secrets used by pre-provisioned volumes must be created in the same namespace as the PVC that references it.

Local

FEATURE STATE: `Kubernetes v1.14 [stable]`

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Local volumes do not currently support dynamic provisioning, however a StorageClass should still be created to delay volume binding until Pod scheduling. This is specified by the `WaitForFirstConsumer` volume binding mode.

Delaying volume binding allows the scheduler to consider all of a Pod's scheduling constraints when choosing an appropriate PersistentVolume for a PersistentVolumeClaim.

6 - Volume Snapshot Classes

This document describes the concept of VolumeSnapshotClass in Kubernetes. Familiarity with volume snapshots and storage classes is suggested.

Introduction

Just like StorageClass provides a way for administrators to describe the "classes" of storage they offer when provisioning a volume, VolumeSnapshotClass provides a way to describe the "classes" of storage when provisioning a volume snapshot.

The VolumeSnapshotClass Resource

Each VolumeSnapshotClass contains the fields `driver`, `deletionPolicy`, and `parameters`, which are used when a VolumeSnapshot belonging to the class needs to be dynamically provisioned.

The name of a VolumeSnapshotClass object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating VolumeSnapshotClass objects, and the objects cannot be updated once they are created.

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
driver: hostpath.csi.k8s.io
deletionPolicy: Delete
parameters:
```

Administrators can specify a default VolumeSnapshotClass for VolumeSnapshots that don't request any particular class to bind to by adding the `snapshot.storage.kubernetes.io/is-default-class: "true"` annotation:

```yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
  annotations:
    snapshot.storage.kubernetes.io/is-default-class: "true"
driver: hostpath.csi.k8s.io
deletionPolicy: Delete
parameters:
```

Driver

Volume snapshot classes have a driver that determines what CSI volume plugin is used for provisioning VolumeSnapshots. This field must be specified.

DeletionPolicy

Volume snapshot classes have a deletionPolicy. It enables you to configure what happens to a VolumeSnapshotContent when the VolumeSnapshot object it is bound to is to be deleted. The deletionPolicy of a volume snapshot class can either be `Retain` or `Delete`. This field must be specified.

If the deletionPolicy is `Delete`, then the underlying storage snapshot will be deleted along with the VolumeSnapshotContent object. If the deletionPolicy is `Retain`, then both the underlying snapshot and VolumeSnapshotContent remain.

Parameters

Volume snapshot classes have parameters that describe volume snapshots belonging to the volume snapshot class. Different parameters may be accepted depending on the `driver`.

7 - Dynamic Volume Provisioning

Dynamic volume provisioning allows storage volumes to be created on-demand. Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create `PersistentVolume` objects to represent them in Kubernetes. The dynamic provisioning feature eliminates the need for cluster administrators to pre-provision storage. Instead, it automatically provisions storage when it is requested by users.

Background

The implementation of dynamic volume provisioning is based on the API object `StorageClass` from the API group `storage.k8s.io`. A cluster administrator can define as many `StorageClass` objects as needed, each specifying a *volume plugin* (aka *provisioner*) that provisions a volume and the set of parameters to pass to that provisioner when provisioning. A cluster administrator can define and expose multiple flavors of storage (from the same or different storage systems) within a cluster, each with a custom set of parameters. This design also ensures that end users don't have to worry about the complexity and nuances of how storage is provisioned, but still have the ability to select from multiple storage options.

More information on storage classes can be found here.

Enabling Dynamic Provisioning

To enable dynamic provisioning, a cluster administrator needs to pre-create one or more StorageClass objects for users. StorageClass objects define which provisioner should be used and what parameters should be passed to that provisioner when dynamic provisioning is invoked. The name of a StorageClass object must be a valid DNS subdomain name.

The following manifest creates a storage class "slow" which provisions standard disk-like persistent disks.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

The following manifest creates a storage class "fast" which provisions SSD-like persistent disks.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Using Dynamic Provisioning

Users request dynamically provisioned storage by including a storage class in their `PersistentVolumeClaim`. Before Kubernetes v1.6, this was done via the `volume.beta.kubernetes.io/storage-class` annotation. However, this annotation is deprecated since v1.9. Users now can and should instead use the `storageClassName` field of the `PersistentVolumeClaim` object. The value of this field must match the name of a `StorageClass` configured by the administrator (see below).

To select the "fast" storage class, for example, a user would create the following PersistentVolumeClaim:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
  resources:
    requests:
      storage: 30Gi
```

This claim results in an SSD-like Persistent Disk being automatically provisioned. When the claim is deleted, the volume is destroyed.

Defaulting Behavior

Dynamic provisioning can be enabled on a cluster such that all claims are dynamically provisioned if no storage class is specified. A cluster administrator can enable this behavior by:

- Marking one `StorageClass` object as *default*;
- Making sure that the `DefaultStorageClass` admission controller is enabled on the API server.

An administrator can mark a specific `StorageClass` as default by adding the `storageclass.kubernetes.io/is-default-class` annotation to it. When a default `StorageClass` exists in a cluster and a user creates a `PersistentVolumeClaim` with `storageClassName` unspecified, the `DefaultStorageClass` admission controller automatically adds the `storageClassName` field pointing to the default storage class.

Note that there can be at most one *default* storage class on a cluster, or a `PersistentVolumeClaim` without `storageClassName` explicitly specified cannot be created.

Topology Awareness

In Multi-Zone clusters, Pods can be spread across Zones in a Region. Single-Zone storage backends should be provisioned in the Zones where Pods are scheduled. This can be accomplished by setting the Volume Binding Mode.

8 - Storage Capacity

Storage capacity is limited and may vary depending on the node on which a pod runs: network-attached storage might not be accessible by all nodes, or storage is local to a node to begin with.

FEATURE STATE: `Kubernetes v1.19 [alpha]`

This page describes how Kubernetes keeps track of storage capacity and how the scheduler uses that information to schedule Pods onto nodes that have access to enough storage capacity for the remaining missing volumes. Without storage capacity tracking, the scheduler may choose a node that doesn't have enough capacity to provision a volume and multiple scheduling retries will be needed.

Tracking storage capacity is supported for Container Storage Interface (CSI) drivers and needs to be enabled when installing a CSI driver.

API

There are two API extensions for this feature:

- CSIStorageCapacity objects: these get produced by a CSI driver in the namespace where the driver is installed. Each object contains capacity information for one storage class and defines which nodes have access to that storage.
- The `CSIDriverSpec.StorageCapacity` field: when set to `true`, the Kubernetes scheduler will consider storage capacity for volumes that use the CSI driver.

Scheduling

Storage capacity information is used by the Kubernetes scheduler if:

- the `CSIStorageCapacity` feature gate is true,
- a Pod uses a volume that has not been created yet,
- that volume uses a StorageClass which references a CSI driver and uses `WaitForFirstConsumer` volume binding mode, and
- the `CSIDriver` object for the driver has `StorageCapacity` set to true.

In that case, the scheduler only considers nodes for the Pod which have enough storage available to them. This check is very simplistic and only compares the size of the volume against the capacity listed in `CSIStorageCapacity` objects with a topology that includes the node.

For volumes with `Immediate` volume binding mode, the storage driver decides where to create the volume, independently of Pods that will use the volume. The scheduler then schedules Pods onto nodes where the volume is available after the volume has been created.

For CSI ephemeral volumes, scheduling always happens without considering storage capacity. This is based on the assumption that this volume type is only used by special CSI drivers which are local to a node and do not need significant resources there.

Rescheduling

When a node has been selected for a Pod with `WaitForFirstConsumer` volumes, that decision is still tentative. The next step is that the CSI storage driver gets asked to create the volume with a hint that the volume is supposed to be available on the selected node.

Because Kubernetes might have chosen a node based on out-dated capacity information, it is possible that the volume cannot really be created. The node selection is then reset and the Kubernetes scheduler tries again to find a node for the Pod.

Limitations

Storage capacity tracking increases the chance that scheduling works on the first try, but cannot guarantee this because the scheduler has to decide based on potentially out-dated information. Usually, the same retry mechanism as for scheduling without any storage capacity information handles scheduling failures.

One situation where scheduling can fail permanently is when a Pod uses multiple volumes: one volume might have been created already in a topology segment which then does not have enough capacity left for another volume. Manual intervention is necessary to recover from this, for example by increasing capacity or deleting the volume that was already created. Further work is needed to handle this automatically.

Enabling storage capacity tracking

Storage capacity tracking is an *alpha feature* and only enabled when the `CSIStorageCapacity` feature gate and the `storage.k8s.io/v1alpha1` API group are enabled. For details on that, see the `--feature-gates` and `--runtime-config` kube-apiserver parameters.

A quick check whether a Kubernetes cluster supports the feature is to list CSIStorageCapacity objects with:

```
kubectl get csistoragecapacities --all-namespaces
```

If your cluster supports CSIStorageCapacity, the response is either a list of CSIStorageCapacity objects or:

```
No resources found
```

If not supported, this error is printed instead:

```
error: the server doesn't have a resource type "csistoragecapacities"
```

In addition to enabling the feature in the cluster, a CSI driver also has to support it. Please refer to the driver's documentation for details.

What's next

- For more information on the design, see the Storage Capacity Constraints for Pod Scheduling KEP.
- For more information on further development of this feature, see the enhancement tracking issue #1472.
- Learn about Kubernetes Scheduler

9 - Ephemeral Volumes

This document describes *ephemeral volumes* in Kubernetes. Familiarity with volumes is suggested, in particular PersistentVolumeClaim and PersistentVolume.

Some application need additional storage but don't care whether that data is stored persistently across restarts. For example, caching services are often limited by memory size and can move infrequently used data into storage that is slower than memory with little impact on overall performance.

Other applications expect some read-only input data to be present in files, like configuration data or secret keys.

Ephemeral volumes are designed for these use cases. Because volumes follow the Pod's lifetime and get created and deleted along with the Pod, Pods can be stopped and restarted without being limited to where some persistent volume is available.

Ephemeral volumes are specified *inline* in the Pod spec, which simplifies application deployment and management.

Types of ephemeral volumes

Kubernetes supports several different kinds of ephemeral volumes for different purposes:

- emptyDir: empty at Pod startup, with storage coming locally from the kubelet base directory (usually the root disk) or RAM
- configMap, downwardAPI, secret: inject different kinds of Kubernetes data into a Pod
- CSI ephemeral volumes: similar to the previous volume kinds, but provided by special CSI drivers which specifically support this feature
- generic ephemeral volumes, which can be provided by all storage drivers that also support persistent volumes

`emptyDir`, `configMap`, `downwardAPI`, `secret` are provided as local ephemeral storage. They are managed by kubelet on each node.

CSI ephemeral volumes *must* be provided by third-party CSI storage drivers.

Generic ephemeral volumes *can* be provided by third-party CSI storage drivers, but also by any other storage driver that supports dynamic provisioning. Some CSI drivers are written specifically for CSI ephemeral volumes and do not support dynamic provisioning: those then cannot be used for generic ephemeral volumes.

The advantage of using third-party drivers is that they can offer functionality that Kubernetes itself does not support, for example storage with different performance characteristics than the disk that is managed by kubelet, or injecting different data.

CSI ephemeral volumes

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

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

> **Note:** CSI ephemeral volumes are only supported by a subset of CSI drivers. The Kubernetes CSI Drivers list shows which drivers support ephemeral volumes.

Conceptually, CSI ephemeral volumes are similar to `configMap`, `downwardAPI` and `secret` volume types: the storage is managed locally on each node and is created together with other local resources after a Pod has been scheduled onto a node. Kubernetes has no concept of rescheduling Pods anymore at this stage. Volume creation has to be unlikely to fail, otherwise Pod startup gets stuck. In particular, storage capacity aware Pod scheduling is *not* supported for these volumes. They are currently also not covered by the storage resource usage limits of a Pod, because that is something that kubelet can only enforce for storage that it manages itself.

Here's an example manifest for a Pod that uses CSI ephemeral storage:

```yaml
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
```

The `volumeAttributes` determine what volume is prepared by the driver. These attributes are specific to each driver and not standardized. See the documentation of each CSI driver for further instructions.

As a cluster administrator, you can use a PodSecurityPolicy to control which CSI drivers can be used in a Pod, specified with the `allowedCSIDrivers` field.

Generic ephemeral volumes

FEATURE STATE: `Kubernetes v1.19 [alpha]`

This feature requires the `GenericEphemeralVolume` feature gate to be enabled. Because this is an alpha feature, it is disabled by default.

Generic ephemeral volumes are similar to `emptyDir` volumes, except more flexible:

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

Example:

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: my-app
spec:
  containers:
    - name: my-frontend
      image: busybox
      volumeMounts:
      - mountPath: "/scratch"
        name: scratch-volume
      command: [ "sleep", "1000000" ]
  volumes:
    - name: scratch-volume
      ephemeral:
        volumeClaimTemplate:
          metadata:
            labels:
              type: my-frontend-volume
          spec:
            accessModes: [ "ReadWriteOnce" ]
            storageClassName: "scratch-storage-class"
            resources:
              requests:
                storage: 1Gi
```

Lifecycle and PersistentVolumeClaim

The key design idea is that the parameters for a volume claim are allowed inside a volume source of the Pod. Labels, annotations and the whole set of fields for a PersistentVolumeClaim are supported. When such a Pod gets created, the ephemeral volume controller then creates an actual PersistentVolumeClaim object in the same namespace as the Pod and ensures that the PersistentVolumeClaim gets deleted when the Pod gets deleted.

That triggers volume binding and/or provisioning, either immediately if the StorageClass uses immediate volume binding or when the Pod is tentatively scheduled onto a node (`WaitForFirstConsumer` volume binding mode). The latter is recommended for generic ephemeral volumes because then the scheduler is free to choose a suitable node for the Pod. With immediate binding, the scheduler is forced to select a node that has access to the volume once it is available.

In terms of resource ownership, a Pod that has generic ephemeral storage is the owner of the PersistentVolumeClaim(s) that provide that ephemeral storage. When the Pod is deleted, the Kubernetes garbage collector deletes the PVC, which then usually triggers deletion of the volume because the default reclaim policy of storage classes is to delete volumes. You can create quasi-ephemeral local storage using a StorageClass with a reclaim policy of `retain`: the storage outlives the Pod, and in this case you need to ensure that volume clean up happens separately.

While these PVCs exist, they can be used like any other PVC. In particular, they can be referenced as data source in volume cloning or snapshotting. The PVC object also holds the current status of the volume.

PersistentVolumeClaim naming

Naming of the automatically created PVCs is deterministic: the name is a combination of Pod name and volume name, with a hyphen (`-`) in the middle. In the example above, the PVC name will be `my-app-scratch-volume`. This deterministic naming makes it easier to interact with the PVC because one does not have to search for it once the Pod name and volume name are known.

The deterministic naming also introduces a potential conflict between different Pods (a Pod "pod-a" with volume "scratch" and another Pod with name "pod" and volume "a-scratch" both end up with the same PVC name "pod-a-scratch") and between Pods and manually created PVCs.

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

Caution: Take care when naming Pods and volumes inside the same namespace, so that these conflicts can't occur.

Security

Enabling the GenericEphemeralVolume feature allows users to create PVCs indirectly if they can create Pods, even if they do not have permission to create PVCs directly. Cluster administrators must be aware of this. If this does not fit their security model, they have two choices:

- Explicitly disable the feature through the feature gate, to avoid being surprised when some future Kubernetes version enables it by default.
- Use a Pod Security Policy where the `volumes` list does not contain the `ephemeral` volume type.

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

What's next

Ephemeral volumes managed by kubelet

See local ephemeral storage.

CSI ephemeral volumes

- For more information on the design, see the Ephemeral Inline CSI volumes KEP.
- For more information on further development of this feature, see the enhancement tracking issue #596.

Generic ephemeral volumes

- For more information on the design, see the Generic ephemeral inline volumes KEP.
- For more information on further development of this feature, see the enhancement tracking issue #1698.

10 - Node-specific Volume Limits

This page describes the maximum number of volumes that can be attached to a Node for various cloud providers.

Cloud providers like Google, Amazon, and Microsoft typically have a limit on how many volumes can be attached to a Node. It is important for Kubernetes to respect those limits. Otherwise, Pods scheduled on a Node could get stuck waiting for volumes to attach.

Kubernetes default limits

The Kubernetes scheduler has default limits on the number of volumes that can be attached to a Node:

Cloud service	Maximum volumes per Node
Amazon Elastic Block Store (EBS)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	16

Custom limits

You can change these limits by setting the value of the `KUBE_MAX_PD_VOLS` environment variable, and then starting the scheduler. CSI drivers might have a different procedure, see their documentation on how to customize their limits.

Use caution if you set a limit that is higher than the default limit. Consult the cloud provider's documentation to make sure that Nodes can actually support the limit you set.

The limit applies to the entire cluster, so it affects all Nodes.

Dynamic volume limits

FEATURE STATE: `Kubernetes v1.17 [stable]`

Dynamic volume limits are supported for following volume types.

- Amazon EBS
- Google Persistent Disk
- Azure Disk
- CSI

For volumes managed by in-tree volume plugins, Kubernetes automatically determines the Node type and enforces the appropriate maximum number of volumes for the node. For example:

- On Google Compute Engine, up to 127 volumes can be attached to a node, depending on the node type.
- For Amazon EBS disks on M5,C5,R5,T3 and Z1D instance types, Kubernetes allows only 25 volumes to be attached to a Node. For other instance types on Amazon Elastic Compute Cloud (EC2), Kubernetes allows 39 volumes to be attached to a Node.
- On Azure, up to 64 disks can be attached to a node, depending on the node type. For more details, refer to Sizes for virtual machines in Azure.
- If a CSI storage driver advertises a maximum number of volumes for a Node (using `NodeGetInfo`), the kube-scheduler honors that limit. Refer to the CSI specifications for details.

- For volumes managed by in-tree plugins that have been migrated to a CSI driver, the maximum number of volumes will be the one reported by the CSI driver.