W. Caban, *Architecting and Operating OpenShift Clusters*
https://doi.org/10.1007/978-1-4842-4985-7_4

4. Storage

William Caban [1]

(1) Columbia, MD, USA

Once the networking options are defined for Containers as described in Chapter 3 , another essential service is storage. Container storage is ephemeral by design. Initially, Containers were designed for immutable and stateless workloads. Later, the advantages of containerizing stateful applications became apparent. With that came the need to support persistent storage. A similar paradigm happened with Kubernetes; initially, it was designed for stateless applications, but it was rapidly extended to support stateful workloads. Supporting these new types of workloads drove the need to support multiple storage options. The storage options for Kubernetes and OpenShift environments are grouped under two classifications: ephemeral storage and persistent storage.

OpenShift Storage

With *Kubernetes* and *OpenShift*, the on-disk files representing the instance of a *Container* are ephemeral. Meaning, once the *Pod* is destroyed or reinstantiated (i.e., during rolling upgrade), any changes to files or data stored inside those *Container* are destroyed.

The default mount point for the ephemeral storage representing the filesystem and the data inside the Containers is determined by the *Container Runtime* in use. See Tables 4-1 and 4-2 for the default mount points used by *OpenShift* when using *Docker* runtime or *CRI-O* runtime.

Table 4-1 OpenShift Mount Points for OpenShift 3.11

Directory	Notes

Directory	Notes
/var/lib/docker	When Conta maint *Regist* It uses /var/ /var/ **Note:** to /va
/var/lib/containers	When active *Node* 1 *Regist* It uses /var/ conta /var/ id>

Directory	Notes
/var/lib/origin/openshift.local.volumes	This is includ runtin kube s storag It uses /var/ id>/c /var/ id>/v

Table 4-2 OpenShift Mount Points for OpenShift 4.0 [1]

Directory	Notes

Directory	Notes
/var/lib/containers	When using the CRI-O runtime (RHCOS), this is the mount poi Containers and Pods. This is th *Node* maintains a copy of *Conta Container Registry*. It uses the following naming fo /run/containers/storage containers/<pod-id> /var/lib/containers/sto id>
/var/lib/kubelet/pods	With Red Hat CoreOS (RHCOS) the ephemeral volume storage anything external that is moun runtime. This is also the moun variables, kube secrets, and an by a persistent storage volume It uses the following naming fo /var/lib/kubelet/pods/< uid>/volumes/<volume-ty

Beyond the default ephemeral storage of the on-disk files representing the instance of a Container, *Kubernetes* has the concept of a *Volume*. [2] A *Kubernetes Volume* is an object that provides a mechanism to provide persistent storage for the *Containers*. A *Volume* and the data on it are preserved across Container restarts and it even outlives any *Containers* within a *Pod*.

> **NOTE** A *Volume* is created to provide persistent storage for *Containers* in a *Pod.* There is a special Volume type, *emptyDir*, [3] that is ephemeral in nature as it is created when a *Pod* is assigned to a *Node* but is deleted when the *Pod* is removed from the *Node*.

Kubernetes Storage Constructs

Kubernetes maintains strict separations of concerns between the definitions of a *PersistentVolume (PV)* , making it available to the *Cluster* (see #1 and #12 in Figure 4-1), to the moment the *PV* is associated to a *Project or Namespace* through a *PersistentVolumeClaim (PVC)* (see #6 and #13 in Figure 4-1). Once the PVC is created associating the PV to the Project or Namespace, it then can be associated as a *Volume* and binds to a mount point in the *Container* (see #10 and #14 in Figure 4-1).

> **NOTE** A *PersistentVolume (PV)* is not tied to any *Namespace*. A *PersistentVolumeClaim (PVC)* is associated and created inside a *Project or Namespace*.

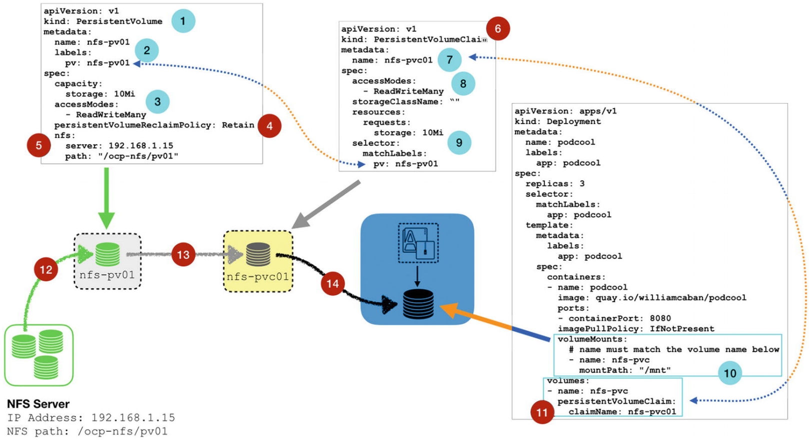

Figure 4-1 PersistentVolume, PersistentVolumeClaim, and Volumes

PersistentVolumes (PV) can be provisioned manually by the cluster administrator or the cluster administrator can enable dynamic provisioner plugins which take care of dynamically creating PVs for any PVC's definition configured in a *Namespace*.

> **TIP** A PVC storage size request (see #9 in Figure 4-1) can bind to a PV with equal or larger storage size (see #3 in Figure 4-1) defined by a PV.

> **CAUTION** If there is no PV capable of fulfilling the PVC storage size request, the PVC remain unbound indefinitely.

When the *Volume* is disconnected from the *Container,* the PVC is available for any other *Container* in the same *Namespace* to use. The data remains on the *Volume* and will be available to any future *Container* using the PVC.

When the PVC definition is deleted, the PV is considered to be *released*. The data is handled based on the *reclaimPolicy* of the PV.

PERSISTENTVOLUME STATUS

A PersistentVolume (PV) will be in one of the following status (see #5 in Figure 4-2):

- **Available**: The PV has not been claimed by a PVC.
- **Bound**: The PV is associated and claimed by a PVC.

- **Released**: The PVC was deleted but the resource has not been reclaimed by the cluster according to the *reclaimPolicy.*

- **Failed**: The automatic reclamation of the PV has failed.

```
$ oc get pv
NAME                                        CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS
nfs-pv01                                    10Mi      RWX           Retain          Bound
nfs-pv02                                    10Mi      RWX           Retain          Available
pvc-04ec0d5e-2721-11e9-87a2-001a4a160101   20Gi      RWO           Delete          Bound
pvc-1bb8a09c-2721-11e9-87a2-001a4a160101   2Gi       RWO           Delete          Bound
```

Figure 4-2 Output showing PV's Access Modes, reclaimPolicy, and Status

RECLAIM POLICY

PersistentVolumes (PV) have an associated *Reclaim Policy* (see #4 in Figure 4-2) which dictates how to handle data after the PV is not *Bound* to a PVC. Kubernetes supports the following Reclaim Policies [4] :

- **Retain**: With this policy the PV is kept after the PV is no longer Bound to a PVC and enables manual reclamation of the resources.

- **Recycle**: *(Depreciated in favor of dynamic provisioning)* This policy performs a basic scrub doing a "`rm -rf /<volume-path>/*`" on the *Volume,* then makes the *Volume* available again for new PVCs.

- **Delete**: This policy removes the PV and the associated storage asset (i.e., AWS EBS, GCE PD, Cinder Volume, Gluster Volume, etc.) when the PV is no longer *Bound* to a PVC.

> **NOTE** When no *reclaimPolicy* is specified or when using dynamically provisioned Volumes, the default reclaim policy is *Delete.*

ACCESS MODES

The access mode (see #3 in Figure 4-2) capabilities of a *PersistentVolume (PV)* are dependent on the modes supported by the provider of the storage resource. For example, NFS supports the three available access modes, while AWS EBS only supports one.

The available access modes are detailed in Table 4-3.

NOTE A *Volume Access Mode* describes the *Volume's* capability but does not enforce constraints. It is up to the storage provider to enforce this at runtime.

Table 4-3 Volume Access Modes

Access Mode	Abbreviation	Description
`ReadWriteOnce`	`RWO`	The volume can be mounted as read-write only by a single *Node* at a time.
`ReadOnlyMany`	`ROX`	The volume can be mounted as read-only by many *Nodes* at a time.
`ReadWriteMany`	`RWX`	The volume can be mounted as read-write by many *Nodes* at a time.

OpenShift PersistentVolume Plugins

OpenShift supports multiple storage plugins.[5] Some of these plugins and the access modes are listed in Table 4-4.

Table 4-4 OpenShift PersistentVolume (PV) Plugins and Supported Access Modes

PV Plugin Name	Access Mode	Mount Options
`NFS`	`RWO, ROX, RWX`	Yes
`HostPath`	`RWO`	No
`GlusterFS`	`RWO, ROX, RWX`	Yes
`Ceph RBD`	`RWO, ROX`	Yes

PV Plugin Name	Access Mode	Mount Options
OpenStack Cinder	RWO	Yes
AWS EBS	RWO	Yes
GCE Persistent Disk	RWO	Yes
iSCSI	RWO, ROX	Yes
FibreChannel	RWO, ROX	No
Azure Disk	RWO	Yes
Azure File	RWO, ROX, RWX	Yes

PV Plugin Name	Access Mode	Mount Options
`VMWare vSphere`	`RWO`	Yes
`Local`	`RWO`	No
`FlexVolume`	*FlexVolume* is an out-of-tree plugin interface that enables users to write their own drivers. Because of this, the supported *Access Modes* and *Mount Options* are implementation specific.	
`Container Storage Interface (CSI)`	*CSI* is an industry standard that enables vendors to develop storage plugins for container orchestration systems (i.e., Kubernetes) in a way that it is portable across CSI-compliant container orchestration systems. Because of this, the supported *Access Modes* and *Mount Options* are implementation specific.	

Since Kubernetes 1.8, the upstream Kubernetes project decided to stop accepting in-tree storage *Volume* plugins. Before this, Volume plugins were linked and distributed as part of the core binaries of Kubernetes. To enable vendors to develop Volume plugins independently from Kubernetes and with their own release cadence, nowadays, instead, it promoted the use of the FlexVolume plugin interface or the use of the Container Storage Interface (CSI) plugin.

The FlexVolume plugin interface has been available since Kubernetes 1.2. The Container Storage Interface (CSI) plugin was introduced in Kubernetes 1.9 and GA in 1.13. These two options are covered in detail in the following sections.

FlexVolume

FlexVolume is known as an out-of-tree plugin interface because it is developed outside the main *Kubernetes* source code. The *FlexVolume* interface enables users to write their own drivers. These drivers can be written in any programming or scripting language.

User-provided driver binaries must be installed in a predefined *Volume* plugin path [6] in every *Node* of the cluster (see #1 in Figure 4-3). The *FlexVolume* driver performing the attach and detach operations must be a self-contained executable with no external dependencies.

Kubernetes is shipped with a *FlexVolume* in-tree plugin that *kubelet* uses to interact with the user-provided drivers using an exec-based model (see #2 in Figure 4-3). When invoking the binary of the driver, the first command-line argument is an *operation* name followed by parameters for the operation.

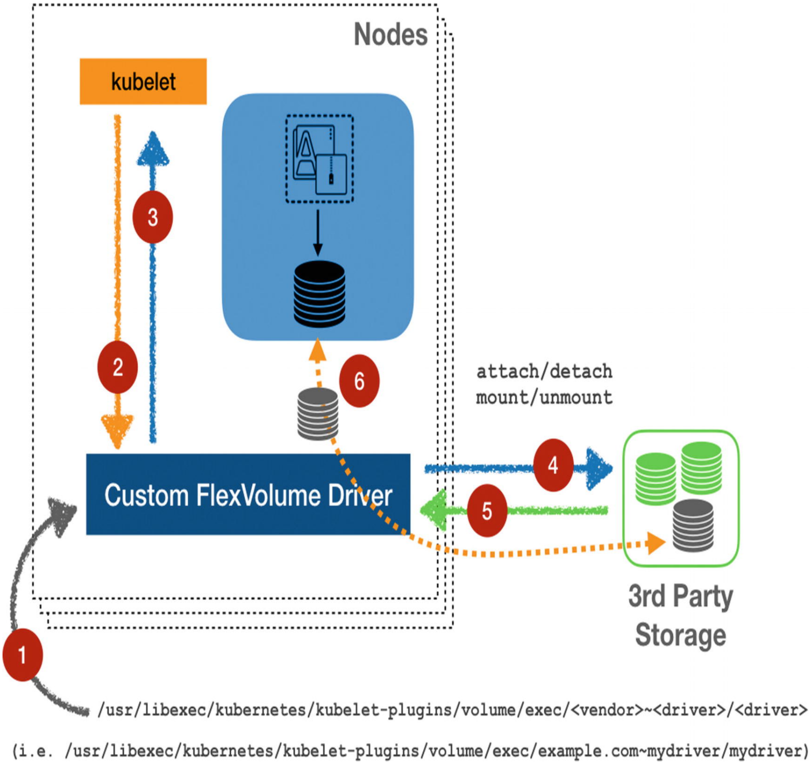

Figure 4-3 FlexVolume plugin architecture

The FlexVolume driver works in one of two modes:

- *FlexVolume* driver with master-initiated attach/detach operation
- *FlexVolume* driver without the master-initiated attach/detach operation

WITH MASTER-INITIATED ATTACH/DETACH

A *FlexVolume* driver with master-initiated attach/detach operation [7] must implement the following *operations*:

- **init**: Initializes the driver
- **getvolumename**: Returns the unique name of the volume
- **attach**: Attaches a volume to a given Node

- **waitforattach**: Waits until the Volume is attached to a Node and the device is recognized by the OS

- **detach**: Detaches the Volume from a Node

- **isattached**: Checks if a particular Volume is attached to a Node

- **mountdevice**: Mounts a Volume device to a directory in a Node

- **umountdevice**: Unmounts a Volume's device from a directory in a Node

WITHOUT MASTER-INITIATED ATTACH/DETACH

A *FlexVolume* driver that does not support master-initiated attach/detach operations [8] is only executing at the specific target *Node* and must implement the following *operations*:

- **init**: Initializes the driver.

- **mount**: Mounts a Volume to a directory in the *Node*. This operation is responsible for finding the device, attaching the device to the Node, and mounting the device to the correct mount point.

- **umount**: Unmounts a Volume from a directory in the Node. This operation should take care of cleaning up the Volume and detaching the device from the Node.

CSI

The *Container Storage Interface (CSI)* was designed to provide a way for vendors to develop storage plugins for any container orchestration platform following the *CSI* specification. This means these plugins are not tied to *Kubernetes* but any *CSI-compliant* platform. *CSI* was introduced into *Kubernetes* as a way to decouple plugin development from Kubernetes releases and prevent bugs from a plugin from affecting other *Kubernetes* critical components.

Contrary to *FlexVolume* plugins that use an exec-based API and assume plugins have access to the root filesystem, the *CSI* plugins use a *gRPC* interface over a unix domain socket.

To support *CSI* plugins, a CSI-compliant plugin interface recommended[9] architecture was defined (Figure 4-4). The CSI plugin interface was included starting in *Kubernetes 1.9* and was made GA in *Kubernetes 1.13*.

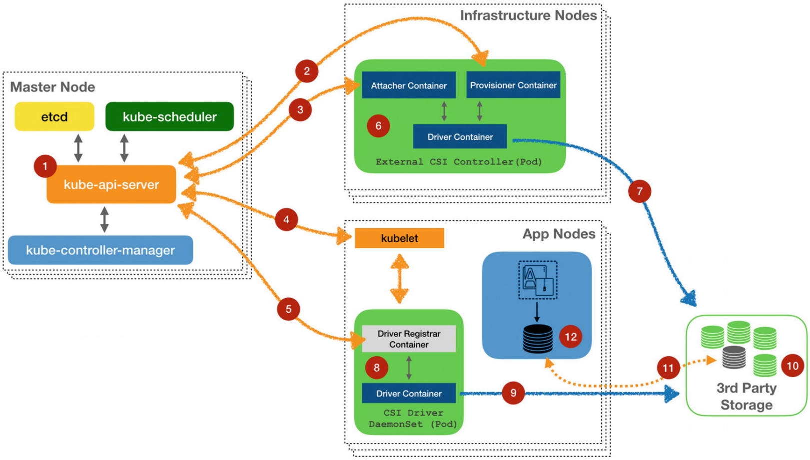

Figure 4-4 CSI plugin recommended architecture

The Kubernetes CSI volume plugin implements the following internal volume interfaces:

- **VolumePlugin**: Mount and unmount of a Volume to a specific path. During the mount operation, Kubernetes generates a unique path and passes it to the CSI Driver DaemonSet (see #4, #5, and #8 in Figure 4-4) for the CSI plugin to mount the volume (see #9 and #11 in Figure 4-4).

- **AttachableVolumePlugin**: Attach and detach of a volume to a given node. This action is handled by the CSI External Controller (see #2, #3, and #6 in Figure 4-4). It is up to the CSI external controller to determine when a CSI Volume must be attached or detached from a particular Node (see #7 and #10 in Figure 4-4). Once the CSI controller determines a Volume should be attached to a Node, it generates a *PersistentVolume (PV)* and eventually the corresponding PersistentVolumeClaim (PVC) to be consumed by the container (see #12 in Figure 4-4).

OpenShift Ephemeral

The *OpenShift Ephemeral* framework is a *Technology Preview (TechPreview)* capability to allow administrators to limit and manage the ephemeral local storage consumed by *Pods* and *Containers* running in the particular *Node*.

Without the *Ephemeral* framework, *Pods* are not aware how much local storage is available to be consumed by the Container's writable layers or *EmptyDir Volumes,* and the *Pod* cannot request guaranteed local storage. Because of this, if the *Node* runs out of local storage, *Pods* can be evicted, losing all the data stored in the ephemeral volumes.

Enabling this capability requires manually enabling the feature on the Master Nodes configurations and the ConfigMaps associated with all the other Nodes. The feature-specific capabilities require to set *LocalStorageCapacityIsolation=true.*[10]

OpenShift Container Storage

The *OpenShift Container Storage (OCS)*[11] brings the software-defined storage capabilities of the *Gluster*[12] *and Heketi*[13] open source projects as a native storage solution into *Containers* environments. It does this by adding a *REST API* interface to front end the *Gluster* services.

The *OpenShift Container Storage (OCS)* supports two deployment modes: converged mode and independent mode (see Figure 4-5).

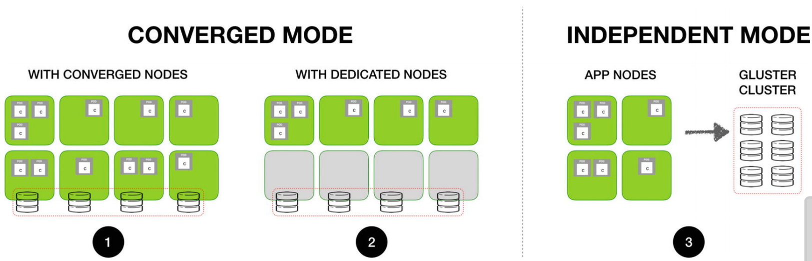

Figure 4-5 OpenShift Container Storage deployment modes

NOTE During the installation of OCS using the OpenShift advanced installer (openshift-ansible), only one of the OCS modes can be specified. Should both modes be required in a cluster, one of the modes can be installed with the Ansible workflow and the other must be manually configured.[14]

OCS CONVERGED MODE

The *OCS Converged Mode* deploys a hyperconverged environment with an end result where the *Nodes* are providing *Compute* and storage services to the cluster.

From the technical perspective, *OCS Converged Mode* deploys an environment where the *Gluster* storage *Containers* reside in *Nodes* where it mounts raw disks attached to these *Nodes* that are then used for the *Gluster* service (see #1 and #2 in Figure 4-5).

There are two common deployment patterns with *OCS Converged Mode*:

1. Worker *Nodes* running *OCS Pods* and also running application Pods (#1 in Figure 4-5)

2. Dedicated *OCS* worker Nodes (#2 in Figure 4-5)

In both of these deployment patterns, the Gluster services are deployed as Containers (see Figure 4-6). A minimum of three nodes are required for the Converged deployment.

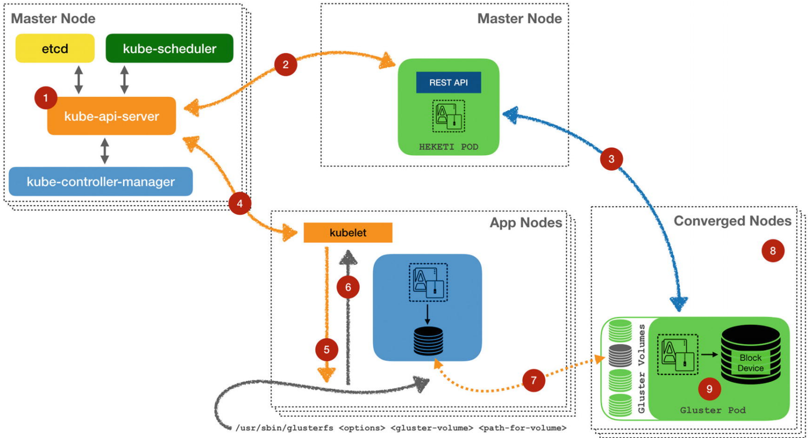

Figure 4-6 OCS Converged Mode

> **TIP** *OCS Converged Mode* is commonly illustrated using *Application Nodes* as the *Converged Nodes,* but it is not limited to those. With the proper planning and design considerations, another option is to deploy *OCS Converged Mode* to *Infrastructure Nodes* instead.

Raw Disks for OCS Converged Mode

The raw block devices for the Gluster service Pods can be provided by Kernel using any supported technology to provide raw block devices to the Node (see Figure 4-7).

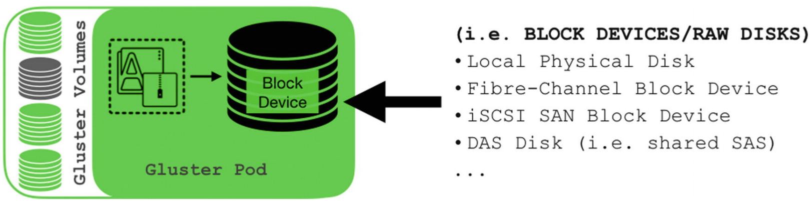

Figure 4-7 OCS Converged Mode block device

OCS INDEPENDENT MODE

OCS Independent Mode uses an external or standalone Gluster cluster managed by an instance of Heketi REST API (#3 and #8 Figure 4-8).

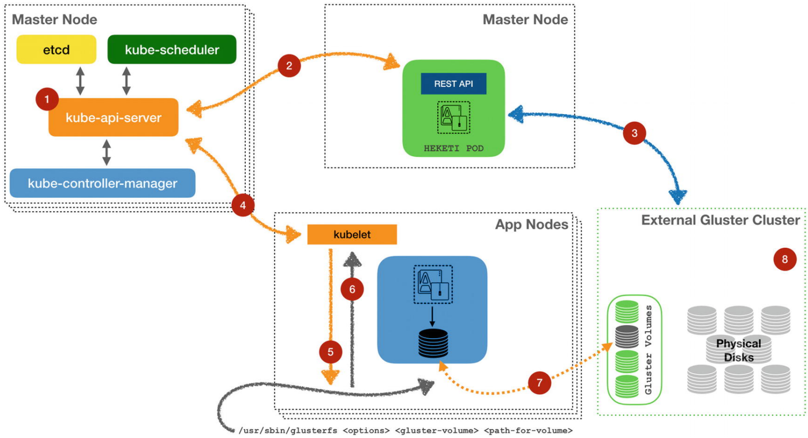

Figure 4-8 OCS Independent Mode

> **NOTE** Even when the *Heketi* service can run either as a regular system service or as a *Container*, the recommendation is for *Heketi* to be deployed as a *Pod* on *OpenShift* so it can benefit from the HA capabilities of the platform.

OCS STORAGE PROVISIONING

OCS supports static or dynamic *GlusterFS* storage volume provisioning. The desired provisioning mode is configured during the deployment of *OCS*. The *PVC* and *PV* provisioning workflow varies the configured provisioning mode. With static storage provisioning[15]:

1. The GlusterFS administrator creates a GlusterFS volume.

2. A user with cluster-admin privileges creates the corresponding GlusterFS Kubernetes Endpoints in the cluster.

3. A user with cluster-admin privileges creates a PV definition.

4. A user creates the corresponding PVC request.

With dynamic provisioning[16]:

1. (If dynamic provisioning was not selected during the deployment of OCS

or if doing a manual OCS deployment.) A cluster administrator creates a GlusterFS StorageClass.

2. A user creates a PVC request.

With dynamic provisioning enabled, when there is a creation of a *PVC* request, the *kube-api-server* sends a request for a new volume to the Heketi REST API (#2 in Figure 4-6 or Figure 4-8) which communicates with the *Gluster* service (#3 in Figure 4-6 or Figure 4-8) to create a new *Gluster Volume*. With the confirmation of the volume, the creation of the *kube-api-server* generates a *PV* which is bound to the *PVC* request.

When the *Kubelet* service (#4 in Figure 4-6 or Figure 4-8) receives the mount request, it invokes the *mount.glusterfs* system command (#5 and #6 in Figure 4-6 or Figure 4-8) with the appropriate parameters to mount the volume to the *Container*. When the *Kubelet* receives an unmount volume request, it uses the *umount* system command.

When the *PVC* is deleted, the *PV* is destroyed and a notification is sent to the Heketi service (#2 in Figure 4-6 or Figure 4-8) which in turn notifies *Gluster* service (#3 in Figure 4-6 or Figure 4-8).

> **NOTE** After the *PVC* and *PV* objects are destroyed and do not exist in the *Kubernetes* environment, from the *Gluster* cluster perspective, it might not be the case as the action of completely deleting and recycling a *Gluster* volume may take additional time.

Storage Classes

A StorageClass is a Kubernetes construct for cluster administrators to create storage profiles describing the storage options available for the platform. Cluster administrators are free to use the StorageClass to represent storage types, or backup policies, or quality-of-service levels, or replication policies, or encryption policies, or any other arbitrary characteristic or service determined relevant for the organization.

A *StorageClass*[17] configuration consists of a *YAML* file with the following options:

- **Provisioner**: (#3 in Figure 4-9) Determines the volume plugin to use for provisioning PVs under the specified StorageClass.
- **Reclaim Policy**: (#5 in Figure 4-9) Tells the cluster what to do with the *Volume* after it is released. The policy can be either *Delete*, *Retain*, *or Recycle*.[18] With dynamically provisioned volumes, the *Reclaim Policy* is *Delete*.
- **Mount Options** (optional): (#6 in Figure 4-9) Mount options for dynamically created *PVs*.
- **Volume Binding Mode**: (#7 in Figure 4-9) This parameter controls the *Volume* binding and dynamic *Volume* provisioning.
- **Allowed Topologies** (optional): Used to restrict provisioning to specific topologies.
- **Parameters** (optional): (#4 in Figure 4-9) This section is used to set *Provisioner*-specific parameters.

```
kind: StorageClass  1
apiVersion: storage.k8s.io/v1
metadata:
  name: mystorageclass  2
  annotations:
      ...                                    3
provisioner: kubernetes.io/plug-in-type
parameters:
  param1: value  4
  ...
  paramN: value
reclaimPolicy: Delete  5
mountOptions:
  - debug  6
  ...
volumeBindingMode: Immediate  7
```

Figure 4-9 Sample StorageClass definition

NOTE A *StorageClass* definition is required for enabling dynamic storage provisioning.

OpenShift with Third-Party Storage

Beyond the list[19] of supported OpenShift software-defined storage (SDS) plugins, because of the availability of the *FlexVolume* and *CSI* plugins, there are many third-party traditional or modern storage solutions supported for OpenShift. This section is a reference (nonexhaustive) list of additional third-party storage vendors. Additional vendors can be found at the OpenShift Primed[20] web site.

DRIVESCALE COMPOSABLE PLATFORM

The *DriveScale Composable Platform*[21] by *DriveScale* is a composable storage platform that aggregates JBOD chassis behind the *DriveScale Composer*. From there, the raw disks are presented as iSCSI targets.

DriveScale supports dynamic storage provisioning in *OpenShift*. At the moment of this writing, DriveScale has a FlexVolume and a CSI plugin. The *DriveScale FlexVolume* plugin is available at the *Red Hat ISV* registry[22] and the *CSI*[23] plugin is provided directly by them.

From the *OpenShift* perspective, at the creation of a new *PVC*, the *DriveScale FlexVolume* plugin interacts with the *DriveScale Composer* and dynamically allocates disks from the JBOD. It then proceeds to present them directly to the *Node* running the *Container* and mount them as a *Volume* into the *Container*. If the *Pod* is reinstantiated into another *Node*, the plugin takes care of unmounting the disk from the *Node* and mounting it into the new *Node*.

HPE 3PAR

The *HPE 3PAR*[24] storage by *HPE* is an all-flash or hybrid storage array platform with support for data services and quality of services guaranteed for the storage. The LUNs are presented to the *Nodes* over *FibreChannel (FC)* or *iSCSI* protocols.

HPE 3PAR supports dynamic storage provisioning in *OpenShift*. At the time of this writing, HPE provides a *FlexVolume* plugin[25] for *OpenShift*. The HPE *FlexVolume* driver is named *Dory*, and the dynamic provisioner is named *Doryd*. The configuration for the plugin can either be set for FibreChannel (FC) or iSCSI, not both at the time. The *FibreChannel (FC)* protocol is supported for OpenShift bare-metal deployments, and the *iSCSI* protocol is supported for *OpenShift* bare-metal or *OpenShift* over virtualization environments.

From the *OpenShift* perspective, at the creation of a new *PVC*, the *HPE 3PAR FlexVolume* plugin interacts with the *Doryd* and dynamically allocates LUNs from the HPE 3PAR storage array. *Dory* presents them directly to the *Node* running the *Container* and mounts them as a *Volume* into the *Container*. If the Pod is reinstantiated into another Node, the plugin takes care of unmounting the disk from the *Node* and mounting it into the new *Node.*

HPE NIMBLE

The *HPE Nimble*[26] storage by *HPE* is an all-flash high-performance storage platform with support for data-at-rest encryption, extreme availability, and sub-millisecond response time. The LUNs are presented to the *Nodes* over the *iSCSI* protocol.

HPE Nimble supports dynamic storage provisioning in *OpenShift*. At the time of this writing, HPE provides a *FlexVolume* plugin[27] for *OpenShift*. The HPE *FlexVolume* is available from the Red Hat ISV registry.[28]

From the *OpenShift* perspective, at the creation of a new *PVC*, the *HPE Nimble FlexVolume* plugin interacts with the *Nimble Dynamic Provisioner* and dynamically allocates LUNs from the HPE Nimble storage. This LUN is presented directly to the *Node* running the *Container* and mounts as a *Volume* into the *Container*. If the Pod is reinstantiated into another Node, the plugin takes care of unmounting the disk from the *Node* and mounting it into the new *Node.*

NETAPP TRIDENT

NetApp Trident[29] is an open source project maintained by *NetApp* designed to support the *NetApp* storage portfolio in *Docker* and *Kubernetes* environments. The plugin supports the NFS or iSCSI protocols.

NetApp Trident supports dynamic storage provisioning in *OpenShift*. At the time of this writing, by default, *NetApp Trident* provides a plugin which uses the native *Kubernetes iSCSI* and *NFS* plugins and provides an experimental *CSI* plugin[30] implementation.

From the *OpenShift* perspective at the creation of a new PVC, the NetApp Trident plugin provisions the corresponding *LUN* or *Volume* in the storage array and relies in the native *Kubernetes iSCSI* or *NFS* plugins for mounting the Volume into the *Container*.

OPENEBS (OSS, MAYADATA)

OpenEBS[31] is an open source project supported by MayaData to provide block storage with tiering and replica policies. While it can use any block devices as the backend storage, the OpenEBS Volumes are presented to the Nodes over the iSCSI protocol.

OpenEBS supports dynamic storage provisioning in *OpenShift*. At the time of this writing, *OpenEBS* provides a *FlexVolume* plugin available from the *Red Hat ISV registry*[32] or directly from the upstream[33] project.

From the OpenShift perspective at the creation of a *PVC*, the OpenEBS plugin creates a volume. A volume is represented by a series of Pods. First there is Pod that works as the iSCSI target[34] for the particular volume. This is the target that is presented to the *Node* running the *Container* and mounts as a Volume into the *Container*. Supporting the iSCSI target volume, there is one Pod per replica. For example, if the configuration is set to have three replicas, there will be three Pods, each one representing one of the replicas. This replica Pods provide the actual backend storage for the Volume. The backend storage can be supported by any block device.

Summary

The use of storage in *Kubernetes* and *OpenShift* environments can be grouped under two classifications: ephemeral storage and persistent storage. The different use cases of ephemeral storage rely on the underlying *Node* filesystem. When working with persistent storage, there are new constructs in play. OpenShift and Kubernetes provide an extensible plugin framework that enables third-party storage providers to onboard their solutions developing plugins at their own phase and independently, without having to coordinate releases with the *Kubernetes* core project.

There are many more persistent storage providers and plugins for OpenShift. The *OpenShift Primed* web site is good place to find additional ones understanding the ecosystem supporting OpenShift and Kubernetes is much larger than the list there.

Once the Containers have networking and storage services, containerized applications can start serving requests. To benefit from the HA capabilities of the platform, the traffic to these applications should consider the use of load balancers. Chapter 5 explores various configuration options to steer traffic to the cluster using load balancers.

Footnotes

1 This information applies to OpenShift 4.0 Beta release. Paths may be subject to change during development and may be different for final release.

2 Additional information and definitions of Volume from the upstream Kubernetes community are available at https://kubernetes.io/docs/concepts/storage/volumes/

3 For use cases and details about emptyDir, refer to the Kubernetes upstream documentation at https://kubernetes.io/docs/concepts/storage/volumes/#emptydir

4 Additional details and utilization of the Reclaim Policies are available at the upstream Kubernetes documentation: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#reclaiming

5 For an updated list of the supported plugins, visit https://docs.openshift.com/container-platform/3.11/install_config/persistent_storage/index.html#install-config-persistent-storage-index

6 The standard path for FlexVolume is `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/<vendor>~<driver>/<driver>`.

7 Additional details can be found at https://docs.openshift.com/container-platform/3.11/install_config/persistent_storage/persistent_storage_f…
volume-drivers-with-master-initiated-attach-detach

8 Additional details can be found at https://docs.openshift.com/container-platform/3.11/install_config/persistent_storage/persistent_storage_f…
volume-drivers-without-master-initiated-attach-detach

9 Details about recommended deployment mechanisms for CSI plugin on Kubernetes are available at https://github.com/kubernetes/community/blob/master/contributors/des…
proposals/storage/container-storage-interface.md#recommended-mechanism-for-deploying-csi-drivers-on-kubernetes

10 For the specific steps toward enabling the LocalStorageCapacityIsolation, refer to https://docs.openshift.com/container-platform/3.11/install_config/configuring_ephemeral.html#ephemeral-storage-enabling-ephemeral-storage

11 Additional information about OCS is available at (an active Red Hat subscription is required to access this link) https://access.redhat.com/documentation/en-us/red_hat_openshift_container_storage/3.11/

12 The upstream Gluster project is available at www.gluster.org (http://www.gluster.org)

13 The Heketi RESTful API for Gluster project is available at https://github.com/heketi/heketi

14 The Red Hat OpenShift Container Storage (OCS) Deployment Guide provides step-by-step instructions for manual installation of the OCS deployment modes (an active Red Hat subscription is required to access this link): https://access.redhat.com/documentation/en-us/red_hat_openshift_container_storage/3.11/html/deployment_guide/

15 Step-by-step instructions on how to configure OCS static provisioning are available at https://docs.openshift.com/container-platform/3.11/install_config/persistent_storage/persistent_storage_g static

16 Instructions for configuring OCS dynamic provisioning on an existing cluster are available at https://docs.openshift.com/container-

platform/3.11/install_config/persistent_storage/persistent_storage_g
dynamic

17 The details of StorageClass resources are described in the upstream Kubernetes documentation: https://kubernetes.io/docs/concepts/storage/storage-classes/#the-storageclass-resource

18 The Recycle Reclaim Policy is considered deprecated. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#recycle

19 OpenShift Persistent Volume plugins: https://docs.openshift.com/container-platform/3.11/install_config/persistent_storage/index.html

20 OpenShift Primed technical readiness: www.openshift.com/learn/partners/primed/ (http://www.openshift.com/learn/partners/primed/)

21 Additional information about the DriveScale Composable Platform is available at https://drivescale.com/composable-platform/

22 DriveScale Composable Platform FlexVolume plugin: https://access.redhat.com/containers/?tab=overview#/registry.connect.redhat.com/drivescale/flexvolume

23 DriveScale CSI plugin: https://github.com/DriveScale/k8s-plugins

24 Additional information about the HPE 3PAR storage is available at www.hpe.com/us/en/storage/3par.html (http://www.hpe.com/us/en/storage/3par.html)

25 Additional information about the HPE 3PAR FlexVolume plugin is available at https://github.com/hpe-storage/python-hpedockerplugin/blob/master/ansible_3par_docker_plugin/README.md

26 Additional information about the HPE 3PAR storage is available at www.hpe.com/us/en/storage/3par.html (http://www.hpe.com/us/en/storage/3par.html)

27 Additional information about the HPE 3PAR FlexVolume plugin is available at https://github.com/hpe-storage/python-hpedockerplugin/blob/master/ansible_3par_docker_plugin/README.md

28 The HPE Nimble Kube Storage Controller is available at https://access.redhat.com/containers/?tab=overview#/registry.connect.redhat.com/nimble/kube-storage-controller

29 Additional information about NetApp Trident is available in the upstream documentation: https://netapp-trident.readthedocs.io/en/stable-v19.01/

30 CSI Trident for Kubernetes: https://netapp-trident.readthedocs.io/en/stable-

v19.01/kubernetes/trident-csi.html?highlight=CSI#csi-trident-for-kubernetes

31 OpenEBS: www.openebs.io (http://www.openebs.io)

32 OpenEBS API Server and volume exporter: https://access.redhat.com/containers/#/product/54cd9cf908d9f6b7

33 OpenEBS project documentation: https://docs.openebs.io/docs/next/installation.html

34 For additional information around the constructs of OpenEBS, refer to the upstream documentation in GitHub: https://github.com/openebs/openebs/blob/master/contribute/design/REA
volume-container-aka-jiva-aka-data-plane