digi hunch

SERVICES ˅

Home » Kubernetes Storage Explained – from in-tree plugin to CSI

June 12, 2021 / Tech Reviews

Kubernetes Storage Explained – from in-tree plugin to CSI

To support a variety of storage backend, Kubernetes abstract storage issues with several objects (volume, persistent volume, persistent volume claim, storage class) and adopts container storage interface. Unfortunately, the documents are not very well organized to deliver the idea of these concepts, most likely because features are introduced at very different times. Hence this article. At the bottom of this article, I also go through five examples of using volumes in different ways, taking azure disk (SSD as an example).

The first to think about is whether we need just ephemeral storage or persistent storage. Generic volume with ephemeral storage lives and dies with the Pod and we don't really care where it is from. With persistent storage, we need to consider where it is from and how to create (provision) the storage. The storage can be created statically or dynamically.

PersistentVolume (PV) and PersistentVolumeClaim (PVC)

Regardlessly of static or dynamic storage provision, we first need to understand two objects before getting to that: Persistent Volume (PV) and Persistent Volume Claim (PVC).

ARCHIVES

Select Month

ABOUT ME

🍁 Travelling and swimmi
for a living 😟 . A seasoned
problem solver, and a 🐶 f
the long term. I like conne
minded professionals. Dro
comments.

CATEGORIE

- Basics & Standards (29)
- Field Notes (50)
- Miscellaneous (4)
- Product Reviews (16)
- Tech Reviews (49)
- Tutorials (20)

RECENT POS

Training and Inference in
April 20, 2025

SageMaker AI and Networ
April 1, 2025

The Leanest Web and Ema
February 27, 2025

Firewall Deployment Patte
November 16, 2024

Cloud Certifications for Le

1. We use PV object to represent external storage volume. A single external storage volume can be represented by a single PV. So PV goes with external volumes in 1 to 1 relationship. A 100G volume cannot be represented by two PVs each with 50G, unless the storage administrator divides it into two separate volumes, each with 50G.
2. PVC goes with Pod in 1 to 1 relationship. The Pods needs a PVC in order to claim ownership of a PV. A valid PVC allows a Pod to mount a PV as its volume.
3. Here we call storage volume external in relative to the pods. If the storage volume is mapped to a directory on the host file system, it is still considered an external storage.
4. A single PV can link to multiple PVCs, so long as the total request in PVCs does not exceed PV's capacity. So PV and PVC are in 1 to many relationship.
5. How PVC binds to PV is defined by Access Mode, with three options. Note that the options are effective for the entire PV. You cannot have different options for each PVC linked to a PV:
   - RWO (ReadWriteOnce): allowing the PV to be bound to a single PVC (for read write). This mode is typically used in block storage;
   - RWM (ReadWriteMany): allowing the PV to be bound to multiple PVCs (for read write). This mode is only supported by file (e.g. NFS) and object storage;
   - ROM (ReadOnlyMany): allowing the PV bound to multiple PVCs for read only.
6. When a PVC is released, what to do with the PV is defined as persistentVolumeReclaimPolicy, and the two options (effective at PV level) are:
   - Delete
   - Retain

Static Provisioning and Dynamic Provisioning

With static provisioning, the external storage volume must be pre-created. In this context, a PV object represents a pre-created external storage volume. So PVs must be explicit declared. The K8s literature also refers to such PVs as pre-created PV.

With dynamic provisioning, the external storage volume is provisioned dynamically. Therefore, you do not need to explicitly create PVs. By the same token, access mode does not apply.

Instead of PV, now we need to explicitly declare storage class, which specifies how to dynamically provision PVs, with the following properties:

1. volumeBindingMode defines when the binding and provisioning of a PersistentVolume occurs, with two options:
   - Immediate (default)
   - WaitForFirstConsumer (recommended): delays until a Pod using the PVC is created
2. reclaimPolicy (the equivalent of persistentVolumeReclaimPolicy for pre-created PV) with two options:
   - Delete (default)
   - Retain
3. provisioners: determines what volume plugin is used for provisioning PVs. There are two categories:
   - **Internal provisioner** (prefixed with kubernetes.io): common ones are listed here. Note that there isn't an internal provisioner for NFS any more. External NFS provisioner is needed.
   - **External provisioner**: third-party out-of-tree plugins compliant to CSI. For example: Dell XtremIO CSI plugin, Dell Isilon plugin, PureStorage CSI driver, Scality Artesca (launched in Apr 2021), and NetApp Trident CSI drivers, and NFS subdir provisioner in Kubernetes-sigs repo.
4. parameters: each provisioner has its own set of mandatory and optional parameters;
5. allowVolumeExpansion: can be set to true if the underlying storage class supports volume expansion;
6. mountOptions: specify only if the storage class supports it;

October 9, 2024

Debating between count a
Terraform

August 27, 2024

Test Open ID Connect Flo

August 5, 2024

IAM Roles for any workloa

July 14, 2024

Managing EC2 instances a
with Ansible

May 27, 2024

Public Key Infrastructure
Implementation

March 30, 2024

With the information above, we can simplify the rules as follows:

- In static provisioning, PV needs to be declared explicitly and SC is not needed
- In dynamic provisioning, SC is required so we can specify provisioner and the parameters needed by the provisioner. PV doesn't need to be explicitly declared, even though it exists in the interaction.

In real life however, you might come across the following edge cases which seems to contradict with the two generic rules above:

- Local volume, currently does not support dynamic provisioning. However a StorageClass should still be created to delay volume binding until Pod scheduling. The volume binding mode *WaitForFirstConsumer* should be specified.
- In dynamic provisioning, if a PVC does not explicitly define PVC, the administrator should have specified a default StorageClass in place for the cluster. You might also come across PVC with empty string ("") as storageClassName, which indicates that no storage class will be used (i.e. dynamic provisioning is disabled for the PVC). According to this post, in a PVC:
    - If storageClassName="", then it is static provisioning
    - If storageClassName is not specified, then the default storage class will be used.
    - If storageClassName is set to a specific value, then the matching storageClassName will be considered. If no corresponding storage class exists, the PVC will fail.

The confusing "Volumes"

We've discussed PersistentVolume, which is a K8s object that represents an external storage volume. When the word Volume stands by itself, it generally refers to the part of storage exposed to the Kubernetes cluster, no matter what type of storage it is or where it comes from. We can distinguish them in the following table:

	Generic **Volumes**	**Persistent Volumes**
Pod assignment	Bound to a single pod, declared as part of a Pod.	A standalone resource type decoupled from Pod and can be bound to single, or multiple Pods via PVC
Lifecycle	Volume is deleted as the owner Pod dies. Data on the volume may or may not persist.	Assuming PVC is gone with Pod, the PV persists. Data on PV may or may not persist depending on ReclaimPolicy.
Configuration	Pod creator (e.g. app developer) needs to know the details of storage resource in the cloud environment. (e.g volume ID)	Pod creator does not need the details of storage resource in the cloud environment. K8s Cluster administrator can provision PV, either statically or dynamically for Pod creator.

If you want to use PeristentVolume to back a Volume in Pod, you'd have to use PersistentVolumeClaim. This means, some types of volumes (including hostPath) can be both mounted as a persistent volume as well as a regular volume. To compare the two ways of mount volume (direct vs via PVC), we take a look at the Kubernetes configuration examples for Azure Disk. The examples are provided at the bottom of this post.

Note that, no matter which method of using the volumes, some types of volumes just work natively, and some requires plugin to operate. The table below summarizes the mechanism behind common volume types.

Volume Types	**Mechanism**	Mountable as **non-persistent volume**	mountable as **persistent volume** (through PVC or SC)

emptyDir	A native volume type, for temporary data only. Data is wiped along with volume. The storage media is determined by the medium of the filsystem holding the kubelet root dir (typically /var/lib/kubelet). You can even set emptyDir.medium to "Memory"	YES	NO. By definition, emptyDir is not persistent.
ConfigMap, Secret	Native volume type to store non-sensitive or sensitive configuration data. ConfigMap and Secrets are stored in etcd.	YES	NO. However, by nature, ConfigMap and Secret are stored persistently. There is no need to mount them as PV.
HostPath	A native volume type to mount a file or directory from the host node's filesystem into the Pod. In addition to path property, you may optionally specify a type for a hostPath volume (e.g. DirectoryOrCreate, Directory, FileOrCreate, etc). Note that there is also a type named empty string ("") which is the default value. It means means that no checks will be performed before mounting the hostPath volume. In addition to the caveat with using hostPath from the documentation, we also need to understand that: 1. HostPath gives Pod the ability to maliciously modify files on the host system, or simply fill up the host file system; 2. As the document suggests, you may end up with multiple Pods trying to write simultaneously to a host path.	YES. Read this.	YES. Check out PersistentVolumes typed hostPath
Local	It represents a mounted local storage device such as a disk, partition, or directory. 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 PV. You must set nodeAffinity on the PV when using local volumes. This also means local volumes are subject to the availability of the underlying node. Refer to this post. This is also referred to as Local persistent Volume.	NO	YES. Static provisioning only.
CephFS, NFS, GlusterFS, Ginder, RBD, FC, iSCSI......	These volume types are backed by legacy in-tree plugins. They are used to connect to external storage in self-hosted clusters.	YES	YES
awsElasticBlockStore, AzureDisk, AzureFile, GCEPersistentDisk	These volume types are backed by legacy in-tree plugins. They are used to connect to external storage in public cloud	YES	YES

Note that the table above does not list PersistenVolumeClaim as a volume type, because it obviously only support being mounted as persistent volume.

From in-tree plugins to out-of-tree CSI plugins

In the table above, the bottom two rows involves in-tree plugins (aka built-in plugins). In-tree means the volume plugins are built in the Kubernetes code repository. They were built, linked, compiled, and shipped with the core Kubernetes binaries. There has been 20+ in-tree plugins. The problems of this plugin development model are:

1. These in-tree plugins introduces risk to the stability of Kubernetes itself;
2. The maintenance and upgrade of plugin is tightly coupled with Kubernetes release
3. The Kubernetes community carries the burden of maintaining plugins for all storage backends.

4. Plugin developers have to open-source all their volume plugin code.

The Kubernetes community seeks better alternatives, and has stopped accepting any more in-tree plugins since GA 1.8. The first alternative paradigm for shipping storage plugin, is flexVolume, which existed since version 1.2. However, flexVolume is still not good enough. For example, some packages like Ceph requires dependency package (ceph-common), and the deployment of plugin requires elevated access to the worker node. For that reason, the community later shifted to the Container Storage Interface (CSI) paradigm. A CSI-compliant plugin allows the storage resource to be surfaced as volumes (be it persistent or not) in Kubernetes cluster. More details in this post and here is a list of supported CSI-compliant drivers.

Back to our azure disk example, this page provides examples for both dynamic and static provisioning.

CSI-compliant plugin development is more complicate but it offloads it the driver developer. The community hopes users to shift to CSI so the 20+ grandfathered in-tree plugins can eventually be phased out. With that as the goal, there are several types of volumes with the name "CSI migration", allowing users to migrate from in-tree volume plugins to CSI-based plugins.

All the CSI-based plugins are fairly recent. As of today, the document outlines three ways to use CSI volume in a Pod:

- through a reference to a PersistentVolumeClaim (examples 4 and 5 below)
- with a generic ephemeral volume (alpha feature)
- with a CSI ephemeral volume if the driver supports that (beta feature)

Examples

We'll go over five examples, as listed in the able below. Note that out of all the combinations, you cannot mount a csi-based plugin as a volume. No such volume type supported by CSI exist.

Plug-in mechanism	Mount method	Example
In-tree legacy volume plug-in	as volume	#1. using azureDisk property of Volume
	as PV (static)	#2. using azureDisk property of PersistentVolume
	as PV (dynamic)	#3. using kubernetes.io/azure-disk as provisioner for SC
Out-of-tree CSI volume plugin	as volume	This mode does not exist. Example is not available
	as PV (static)	#4. using disk.csi.azure.com as csi driver of PV
	as PV (dynamic)	#5 using disk.csi.azure.com as provisioner for SC

Now, let's take a look at the example code snippet. Some examples are from Azure documentation. Some are from the azure-disk-csi-driver repository. I've made minor modifications for conciseness.

Example 1 uses legacy in-tree plugin, and directly mount the volume. The example code is in Kubernetes repo.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - image: kubernetes/pause
    name: mypod
    volumeMounts:
      - name: azure
        mountPath: /mnt/azure
  volumes:
      - name: azure
        azureDisk:
          kind: Managed
          diskName: myAKSDisk
          diskURI: /subscriptions/&lt;subscriptionID>/resourceGroups/MC_myAKS
```

Example 2 uses legacy in-tree plugin, and mount the PV statically via PVC. No storage class is used (as indicated by empty string in storage class property)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: azure-disk-pv
spec:
  capacity:
    storage: 2Gi
  storageClassName: ""
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  azureDisk:
    kind: Managed
    diskName: &lt;enter-disk-name>
    diskURI:  &lt;enter-disk-resource-id>
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-disk-pvc
spec:
  storageClassName: ""
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
---
apiVersion: apps/v1
kind: Pod
metadata:
  name: logz-deployment
spec:
  containers:
  - name: pause
    image: kubernetes/pause
    volumeMounts:
    - name: azure-disk-vol
      mountPath: /mnt/logs
  volumes:
    - name: azure-disk-vol
      persistentVolumeClaim:
        claimName: azure-disk-pvc
```

Example 3 uses legacy in-tree plugin, and mount the PV dynamically and implicitly via SC. Note that Azure AKS will create several SCs for you by default so use existing ones whenever available.

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-premium
parameters:
  cachingmode: ReadOnly
  kind: Managed
  storageaccounttype: Premium_LRS
provisioner: kubernetes.io/azure-disk
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: azure-managed-disk
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
---
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: kubernetes/pause
    volumeMounts:
    - mountPath: "/mnt/azure"
      name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk
```

Example 4 uses CSI-based plugin, and mount the PV statically via PVC

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-azuredisk
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  csi:
    driver: disk.csi.azure.com
    readOnly: false
    volumeHandle: /subscriptions/{sub-id}/resourcegroups/{group-name}/provide
    volumeAttributes:
      fsType: ext4
      partition: "1"  # optional, remove this if there is no partition
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-azuredisk
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeName: pv-azuredisk
  storageClassName: ""
---
kind: Pod
apiVersion: v1
metadata:
  name: nginx-azuredisk
spec:
  nodeSelector:
    kubernetes.io/os: linux
  containers:
    - image: kubernetes/pause
      name: mypod
      volumeMounts:
        - name: azuredisk01
          mountPath: "/mnt/azuredisk"
  volumes:
    - name: azuredisk01
      persistentVolumeClaim:
        claimName: pvc-azuredisk
```

Example 5 uses CSI-based plugin, and mount the PV dynamically and implicitly via SC

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azuredisk-csi-waitforfirstconsumer
provisioner: disk.csi.azure.com
parameters:
  skuname: StandardSSD_LRS
allowVolumeExpansion: true
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-azuredisk
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: managed-csi
---
kind: Pod
```

```
apiVersion: v1
metadata:
  name: nginx-azuredisk
spec:
  nodeSelector:
    kubernetes.io/os: linux
  containers:
    - image: kubernetes/pause
      name: mypod
      volumeMounts:
        - name: azuredisk01
          mountPath: "/mnt/azuredisk"
  volumes:
    - name: azuredisk01
      persistentVolumeClaim:
        claimName: pvc-azuredisk
```

Bottomline

As of June 2021, the CSI support is still new. Generally, if a CSI-based plugin is available and in GA, you should consider using it. If you have existing legacy volume types using in-tree plugin, you should consider migration, and create a migration plan. Also, try to avoid the use case of mounting as generic volume (without PVC) because it is rare and not supported with CSI drivers. Without PVC, it also cannot take advantage of the **volumeClaimTemplates** property in StatefulSet object.