Create and attach persistent volume claims to pods.

What are Persistent Volumes?

Persistent Volumes are Kubernetes objects that represent storage resources in your cluster. PVs work in conjunction with Persistent Volume Claims (PVCs), another type of object which permits Pods to request access to PVs. To successfully utilize persistent storage in a cluster, you'll need a PV and a PVC that connects it to your Pod.

The primary role of PVs is to abstract away the differences between cluster storage implementations. Each PV is assigned a storage class which defines the type of storage it provisions. Storage classes allow the automatic provisioning of volumes through many different storage providers, including cloud-hosted block storage options such as AWS Elastic Block Store and GCE Persistent Disk.

What is the difference between Volume and Persistent Volume in Kubernetes?

Kubernetes also supports Volumes which can be used instead of, or in addition to, Persistent Volumes.

Volumes exist in the context of specific Pods. A Volume's lifecycle is coupled to the Pod that owns it. It allows safe use of storage by all the containers in the Pod, but the volume's data will still be lost when the Pod terminates.

Persistent Volumes build upon the foundations established by Volumes. They provide storage that's decoupled from Pods, allowing data to persist beyond the lifecycle of individual Pods. Unlike Volumes, Persistent Volumes are first-class objects within your cluster.

Persistent Volumes also distinguish between storage provisioning and utilization. Cluster admins can set up PVs for developers to access using PVCs. This facilitates controlled approaches to storage management.

Plain Volumes can be used when data should be shared between a Pod's containers but won't need to be accessible to any other Pods – including replacements of the original. PVs are the solution for most real-world storage use cases where data needs to be persisted independently of Pods.

Types of Persistent Volume

Kubernetes supports several Persistent Volume types that alter where and how your data is stored:

- local – Data is stored on devices mounted locally to your cluster's Nodes.
- hostPath – Stores data within a named directory on a Node (this is designed for testing purposes and doesn't work with multi-Node clusters).
- nfs – Used to access Network File System (NFS) mounts.
- iscsi – iSCSI (SCSI over IP) storage attachments.
- csi – Allows integration with storage providers that support the Container Storage Interface (CSI) specification, such as the block storage services provided by cloud platforms.
- cephfs – Allow the use of CephFS volumes.
- fc – Fibre Channel (FC) storage attachments.
- rbd – Rados Block Device (RBD) volumes.

In addition, there are volume types such as awsElasticBlockStore, azureDisk, and gcePersistentDisk that support built-in integrations with specific cloud providers. However, these are all now deprecated in favor of CSI-based volumes.

Storage classes handle storage provisioning operations for volumes. There are built-in storage classes for each of the supported PV types mentioned above. Additional storage classes can be added by installing plugins that implement the Kubernetes Volume Provisioning specification.

Persistent Volume access modes

Persistent Volumes support four different access modes that define how they're mounted to Nodes and Pods:

- **ReadWriteOnce (RWO)** – The volume is mounted with read-write access for a *single* Node in your cluster. Any of the Pods running on that Node can read and write the volume's contents.
- **ReadOnlyMany (ROX)** – The volume can be concurrently mounted to any of the Nodes in your cluster, with read-only access for any Pod.
- **ReadWriteMany (RWX)** – Similar to ReadOnlyMany, but with read-write access.
- **ReadWriteOncePod (RWOP)** – This new variant, introduced as a beta feature in Kubernetes v1.27, enforces that read-write access is provided to a *single* Pod. No other Pods in the cluster will be able to use the volume simultaneously.

The access modes that can be used with a specific PV depend on the type of storage that backs it. hostPath volumes support only RWO, for example, while nfs offers RWO, ROX, and RWX. For CSI integrations, the options are defined by the specific storage driver that's in use, such as AWS or GCE.

The lifecycles of PVs and PVCs

PVs and PVCs have their own lifecycles, which describe the current status of the volume and whether it's in use. There are four main stages:

1. Provisioning

At the Provisioning stage, the PV is created and its storage is allocated using the selected driver.

Provisioning can occur manually by creating a PersistentVolume object in your cluster, or dynamically, by adding a PVC that refers to an unknown PV. After provisioning, the PV will exist in your cluster, but won't be actively providing storage.

2. Binding

Binding occurs when a cluster user adds a PVC that claims the PV. The PV will enter this state automatically when dynamic provisioning is used, because you'll have already created the PVC.

Kubernetes automatically watches for new PVCs and binds them to the PVs they reference. Each PV can only be bound to a single PVC at a time. Once a PVC claims a PV, the volume will be Bound, but won't necessarily be used by a Pod.

3. Using

A volume enters use once its PVC is consumed by a Pod. When Pods reference PVCs, Kubernetes automatically mounts the correct volume into the Pod's filesystem. In this state, the PV is actively providing storage to an application in your cluster.

4. Reclaiming

Storage isn't always required indefinitely. Users can delete the PVC to relinquish access to the PV. When this happens, the storage used by the PV is "reclaimed."

The reclaim behavior is customizable and allows you to either delete the provisioned storage, recycle it by emptying its contents, or retain it as-is for future reuse.

You might also like:

- Enabling Seamless Collaboration in IaC
- How to Provision Kubernetes Cluster with Terraform
- Terraform vs. Kubernetes Comparison

Example: How to create and use a Persistent Volume

Ready to explore Persistent Volumes? Let's get started. Make sure you've got access to a Kubernetes cluster configured in Kubectl before you begin.

Step 1: Check your cluster's storage classes

First, find out which storage classes are available in your cluster:

```
$ kubectl get storageclass
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate           false                  7d2h
```

We're using a local Minikube cluster which includes a single built-in storage class called standard. It provisions volumes on the host machine.

We'll use the standard storage class in the following examples, but you might need to use an alternative depending on the options available in your cluster. For example, Azure clusters will have azurefile-csi available, while DigitalOcean supports do-block-storage.

Step 2: Create a Persistent Volume

Next, copy the following PV Kubernetes manifest and save it to pv.yaml:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: demo-pv
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
```

```
  storageClassName: standard
  hostPath:
    path: /tmp/demo-pv
```

This defines a 1Gi Persistent Volume that uses the standard storage class. The access mode is set to ReadWriteOnce, which is the only read-write mode supported by the storage class in this tutorial.

The hostPath field configures how the volume will be configured by the storage driver, which in this case uses the hostPath provisioner. It instructs the driver to store the volume's data at /tmp/demo-pv on the host. You will need to supply different configuration options if you're using a storage driver with an alternative provisioner.

Run the following command to create your Persistent Volume:

```
$ kubectl apply -f pv.yaml
persistentvolume/demo-pv created
```

See the difference between kubectl apply vs. kubectl create.

The PV will now exist in your cluster. It will show as Available as it's unused and unclaimed:

```
$ kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   REASON   AGE
demo-pv   1Gi        RWO            Retain           Available           standard                113s
```

Step 3: Create a Persistent Volume Claim

Next, create a PVC for your volume. Copy the following manifest to pvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: demo-pvc
spec:
```

```
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
  volumeName: demo-pv
```

It defines a claim for the volume named demo-pv, which we created above. It's necessary to repeat the PV's properties because Kubernetes requires the PV and PVC specs to match.

Add the PVC to your cluster:

```
$ kubectl apply -f pvc.yaml
persistentvolumeclaim/demo-pvc created
```

The PV will now show as Bound because it's claimed by the PVC:

```
$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
demo-pv   1Gi        RWO            Retain           Bound    default/demo-pvc   standard
6m40s
```

Step 4: Dynamically provisioning PVs

Static provisioning of PVs, as shown above, is cumbersome: you must create the PV, then the PVC while ensuring their properties match. You also have to supply the PV configuration data required by the storage provisioner.

Dynamic provisioning is simpler and the more popular option for real-world use. The PV is created for you based on your PVC's configuration.

Copy the following manifest to pvc-dynamic.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: pvc-dynamic
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

This manifest creates and claims a new 1Gi PV that will be backed by the standard storage class. Because the volumeName is omitted from the PVC's spec, the PV is dynamically provisioned:

```
$ kubectl apply -f pvc-dynamic.yaml
persistentvolumeclaim/pvc-dynamic created

$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS   CLAIM              STORAGECLASS   REASON   AGE
demo-pv                                    1Gi        RWO            Retain
Bound    default/demo-pvc   standard                33m
pvc-fd022049-eabf-415c-937a-94927c84ef6f   1Gi        RWO            Delete
Bound    default/pvc-dynamic   standard
```

Dynamically created PVs default to using the Delete retention policy. This means the PV is automatically deleted when the PVC is destroyed.

Step 5: Attach PVCs to Pods

Once you've provisioned and bound your PV, it's time to attach the PVC to a Pod. This will mount your PV into the Pod's filesystem, allowing the Pod to read and write files with full persistence.

Copy the following manifest to pod.yaml:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: pvc-pod
spec:
  containers:
    - name: pvc-pod-container
      image: nginx:latest
      volumeMounts:
        - mountPath: /data
          name: data
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: pvc-dynamic
```

What's happening here?

- First, the **spec.containers.volumeMounts** field sets up a volume for the Pod called data. It's mounted to /data within the Pod's containers.
- Next, the **spec.volumes** field defines the data volume as referring to the PVC called pvc-dynamic.
- This results in the PV claimed by pvc-dynamic being mounted to /data inside the container.

Create your Pod, then try writing some files to /data:

```
$ kubectl apply -f pod.yaml
pod/pvc-pod created

# Get a shell inside the Pod
$ kubectl exec -it pod/pvc-pod -- bash

# No files in the volume yet
```

```
root@pvc-pod:/# ls /data

# Write a file to the volume
root@pvc-pod:/# echo "bar" > /data/foo

# The file now shows in the volume
root@pvc-pod:/# ls /data
foo
```

The files written to the volume are now stored independently of the Pod. You can delete the Pod and recreate it – your data will still be intact within the PV:

```
$ kubectl delete pod/pvc-pod
pod "pvc-pod" deleted

$ kubectl apply -f pod.yaml
pod/pvc-pod created

$ kubectl exec -it pod/pvc-pod -- bash

root@pvc-pod:/# cat /data/foo
bar
```

Similarly, you can simultaneously attach the PVC to additional Pods to share files between them.

Persistent Volumes best practices

PVs and PVCs are sometimes confusing to newcomers. The terminology and lifecycle model are unique to Kubernetes, but the high level of abstraction provides great flexibility when configuring your storage.

Here are some best practices to follow.

- **Always specify a storage class for your PVs.** PVs without a storageClassName will use the default storage class in your cluster, which can cause unpredictable behavior if the value's changed or not set.
- **Anticipate future storage requirements.** It's usually beneficial to provision the largest amount of storage you think you'll need. It's not always possible to resize a volume after creation – your options will depend on the storage driver being used.
- **Avoid static provisioning of Persistent Volumes.** Statically provisioning and binding PVs increases the workload for administrators and can be more challenging to scale. Dynamic provisioning ensures usable PVs will be automatically created for your PVCs, simplifying the set up procedure.
- **Select an appropriate reclaim policy.** It's vital to select the correct reclaim policy for each of your PVs. Consider using the Retain policy for critical data. It means you'll need to manually delete the volume after it's reclaimed, but ensures your data will be recoverable in the event of accidental deletion.
- **Use RBAC to protect storage resources.** Configure your cluster's RBAC authorization policies to help protect your PVs and PVCs from accidental or malicious changes and deletions. Incorrect operations applied to storage objects are likely to cause irrecoverable data loss, unlike stateless Kubernetes object types, which you can rollback or recreate.

Key Points

Persistent Volumes represent storage resources in Kubernetes clusters. They abstract the details of how storage is provisioned and consumed.

Your Pods gain access to PVs by creating a Persistent Volume Claim, which represents a request to utilize storage. PVCs mount the requested volume into the Pod's filesystem, allowing your application to read and write files that will outlive the Pod itself.

Need a better way to control PV changes? Try out Spacelift to visualize your cluster's storage and set up guardrails that prevent accidental changes. It's good practice to require an approval before a PVC is deleted, for example, to help protect against inadvertent data loss. Spacelift can also detect and prevent drift to maintain a stable configuration and alert you when discrepancies occur.