Kubernetes Storage

Remko de Knikker
Client Developer Advocate

Stateless versus Persistent

State in a container on Kubernetes is epheremal. When a container crashes, kubelet will restart it but data is lost.

In Docker, a volume is simply a directory on disk or in another Container and there is no explicit lifetime. On Kubernetes however, a Volume has the same lifetime as the pod that encloses it. Consequently, a volume outlives containers that run within the Pod, and data is preserved across container restarts, but when a Pod ceases to exist, the volume will cease to exist.

Kubernetes supports several types of Volumes: configMap, emptyDir, glusterfs, hostPath, nfs, persistenVolumeClaim, secret et al.

PersistentVolumes and PersistentVolumeClaims

Managing storage is a distinct problem on Kubernetes. PersistentVolume and PersistentVolumeClaim abstract the details of how storage is provided from how it is consumed.

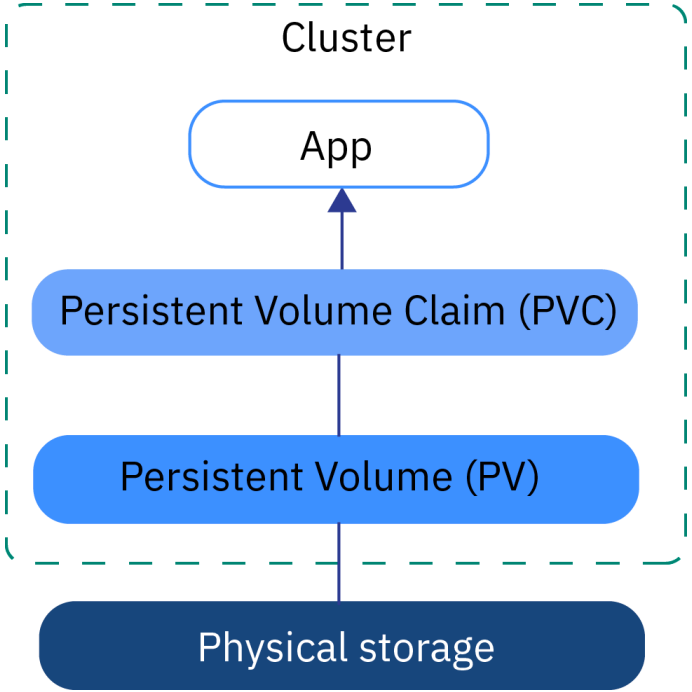
A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV.

A *PersistentVolumeClaim* (PVC) is a request for storage. PVCs consume PV resources. Claims can request specific size and AccessModes.

There are 2 ways to provision a PV:

- Statically, PVs defined by administrator, or
- Dynamically based on StorageClasses, and with a DefaultStorageClass enabled.

A control loop in the master watches for new PVCs, finds a matching PV and binds them together.



PersistentVolumes and PersistentVolumeClaims

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod.

When a user is done with their volume, they can delete the PVC. 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 be Retained, Recycled, or Deleted.

PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: guestbook-pv
 labels:
   app: guestbook
spec:
 storageClassName: manual
 capacity:
   storage: 10Gi
 accessModes:
   - ReadWriteOnce
 hostPath:
   path: "/tmp/data"
```

PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: guestbook-pvc
spec:
 storageClassName: manual
 accessModes:
 - ReadWriteOnce
 resources:
  requests:
    storage: 10Gi
 selector:
  matchLabels:
   app: guestbook
```

PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: couchdb-pv
 labels:
   app: couchdb
spec:
 capacity:
   storage: "10Gi"
 accessModes:
   - ReadWriteMany
 nfs:
   server: "fsf-wdc0701a-fz.adn.networklayer.com"
   path: "/IBM02SEV1624905_6/data01/couchdb/data"
```

PersistentVolumeClaim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: couchdb-pvc
 namespace: my-ns
 labels:
   app: couchdb
spec:
 accessModes:
  - ReadWriteMany
 resources:
  requests:
    storage: "10Gi"
 volumeName: couchdb-pv
```

Pod

```
apiVersion: v1
kind: Pod
metadata:
   name: guestbook
   namespace: default
spec:
   restartPolicy: Never
   containers:
   - name: guestbook
     image: ibmcom/guestbook
     volumeMounts:
     - name: guestbook-pv
       mountPath: /mnt/data
   volumes:
      - name: guestbook-pv
        hostPath:
           path: /tmp/data
```

Deployment

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: couchdb-deployment
  namespace: my-ns
  labels:
    app: couchdb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: couchdb
  template:
    metadata:
      labels:
        app: couchdb
    spec:
      containers:
      - name: couchdb
        image: apache/couchdb
        volumeMounts:
        - mountPath: /opt/couchdb/data
          name: couchdb-pv
        ...
        envFrom:
        - configMapRef:
          name: couchdb-configmap
      volumes:
        - name: couchdb-pv
          persistentVolumeClaim:
            claimName: couchdb-pvc
```

Container Storage Interface (CSI)

The Container Storage Interface (CSI) is a standard for exposing arbitrary block and file storage systems to containerized workloads on Container Orchestration Systems (COs) like Kubernetes. Using CSI third-party storage providers can write and deploy plugins exposing new storage systems in Kubernetes without ever having to touch the core Kubernetes code.

CSI defines APIs that enable:

- Dynamic provisioning and deprovisioning of a volume.
- Attaching or detaching a volume from a node.
- Mounting/unmounting a volume from a node.
- Consumption of both block and mountable volumes.
- Local storage providers.
- Creating and deleting a snapshot (source of the snapshot is a volume).
- Provisioning a new volume from a snapshot.