**Working with Kubernetes Storage Options**

* Kubernetes Shared Volumes
* Persistent Volumes
* Persistent Volume Claim
* Storage classes

**Kubernetes Shared Volumes**

* In a volatile environment, storage might be seen as a problem. How do you keep data in storage when the container or pod is transient and nodes can fail?
* While nodes represent the **compute capacity** of a Kubernetes cluster, a volume represents its **storage capacity**.

**Storage API Objects in Kubernetes**

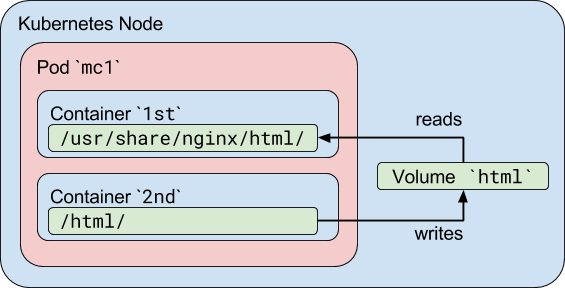
1. Kubernetes Shared Volume
2. Persistent Volume
3. Persistent Volume Claim
4. Storage Classes

**Kubernetes Volume:**

* A Kubernetes Volume is essentially a **directory** accessible to all containers running in a pod.
* Its deployed as part of the Pod spec.
* In contrast to the container-local filesystem, the data in volumes is preserved across container restarts.
* The medium backing a volume and its contents are determined by the volume type.

**Volume Types:**

* node-local types such as **emptyDir** or **hostPath**
* file-sharing types such as **nfs**
* cloud provider-specific types like **awsElasticBlockStore**, **azureDisk**, or **gcePersistentDisk**
* distributed file system types, for example **glusterfs** or **cephfs**
* special-purpose types like **configmap**, **secret**, **gitRepo**
* **. . .**



**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, it is initially empty (for every new pod). Containers in the Pod can all 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 forever.

Note: A container crashing does NOT remove a Pod from a node, so the data in an emptyDir volume is safe across Container crashes. Replacement containers can access data from predecessors.

apiVersion: v1

kind: Pod

metadata:

  name: sharevoldemo-pod

spec:

**containers**:

  - name: consumer-con

    image: nginx

    volumeMounts:

      - name: xchange

        mountPath: "/usr/share/nginx/html"

        readOnly: true

  - name: producer-con

    image: redis

    volumeMounts:

      - name: xchange

        mountPath: "/html"

        readOnly: false

**volumes**:

  - name: xchange

    emptyDir: {}

**Step1: Create Pod with two containers using the above YAML**

1. kubectl apply -f pods.yaml

**Step2: We first exec into one of the containers in the pod, producer-con, check the volume mount and generate some HTML files:**

1. kubectl exec -it sharevoldemo-pod -c producer-con -- bash
   1. cd /html
   2. echo "this is demo" > demo.html
   3. exit

**Step3:** When we now exec into consumer-con, the second container running in the pod, we can see the volume mounted at /html and are able to read the data created in the previous step:

1. kubectl exec -it sharevoldemo-pod -c consumer-con -- bash
2. cd /usr/share/nginx/html
3. ls #Note that demo.html exists
4. curl <http://localhost>
5. curl <http://localhost/demo.html>
6. kubectl get po -o wide #Note the IP Address of pod
7. kubectl run -it --rm --restart=Never mynginx-for-test --image=nginx -- /bin/sh  
   # curl http://<IpAddressOfPod>/demo.html

**hostPath:** This type of volume mounts a file or directory from the **host node’s** filesystem into your pod.

Data is physically stored on a specific node in cluster.

Available to replacement pods (but only if run on same node)

Can have issues:

➢ In cluster with multiple nodes, limits usefulness.

➢ If not careful, can expose large blocks of host data.

Note: If the container of a Pod uses HostPath for persistent storage, the data will not be available to New Pods if they are created on different node in the cluster.

**Pod.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: test-pd

spec:

**containers:**

  - image: nginx

    name: test-container

**volumeMounts**:

    - mountPath: /con-data

      name: test-volume

**volumes:**

  - name: test-volume

    hostPath:

      path: /host-data

      type: DirectoryOrCreate

**HOST Directory for Minikube:** **/var/lib/docker/volumes/minikube/\_data/data/hello.txt**

**Execute the commands in following order**

kubectl apply -f pod.yaml

kubectl exec -it test-pd -- sh

# cd /con-data

# echo Hello > hello.txt

# exit

kubectl delete -f pod.yaml

------------------------------------

kubectl apply -f pod.yaml

kubectl exec -it test-pd -- ls /con-data #Note the file hello.txt already exists assuming that its on a single node cluster.

**Cloud provider-specific types:**  **azureFile**

1. Create Azure Storage and File Share **images**
2. Note **Storage Account Name** and **Storage** **Access Key**
3. Create Kubernetes Secret

kubectl create **secret** generic **azure-storage-secret** --from-literal=**azurestorageaccountname**=<StorageAccountName> --from-literal=**azurestorageaccountkey**=<StorageAccountKey>

1. **Create POD with below YAML**

apiVersion: v1

kind: Pod

metadata:

  name: mypod

spec:

**containers**:

  - image:  nginx

    name: mypod

**volumeMounts**:

      - name: azure

        mountPath: /myimages

  volumes:

  - name: azure

**azureFile**:

      secretName: azure-storage-secret

      shareName: images

      readOnly: false

Note: The above YAML doesn’t work with Docker Desktop and Minikube.

Kubectl apply -f azure-demo.yaml

kubectl exec -it mypod -- /bin/sh

# cd azure

# echo Demo > demo.txt

**Note: Some linux Commands**

# df

You can see that the File Share is mounted as: //mydssdemostorage.file.core.windows.net/test

# cifs

**ConfigMap and Volumes**

* Configmap Can be presented as files inside directories in the container
* Uses volumes – making contents of ConfigMap available to pod
* Uses volume mounts – loads contents of ConfigMap volume into specific container path in pod
* Can contain settings to override defaults

Example1: ConfigMap from YAML

**ConfigMap1.yaml**

apiVersion: v1

kind: ConfigMap

metadata:

  name: mysettings-config4

  namespace: default

data:

  mysettings.json:  "{\"name\":\"sandeep soni\",\"location\":\"india\"}"

---

apiVersion: v1

kind: Pod

metadata:

  name: test-pod

spec:

  containers:

    - name: test-container

      image: nginx

      volumeMounts:

      - name: config-vol

        mountPath: "/data"

        readOnly: true

**volumes:**

    - name: config-vol

**configMap**:

        name: mysettings-config4

**When a ConfigMap is used as Volume Type, the Key(s) in ConfigMap are reflected as file(s) in the container mountedPath folder.**

**Execute the following commands and note that the file exists by name mysettings.properties.**

kubectl apply -f d:\Demos\Kubernetes\configmap1.yaml

kubectl exec -it pod/test-pod -- bash

# cat /data/mysettings.json

**Example2: Using env file.**

**Each configmap key is converted to a filename and value is added into that file.**

**mysettings.env**

name=sandeep

location=hyderabad

**Command to create a config map.**

kubectl create configmap mysettings-config3 **--from-env-file**=mysettings.env

**ConfigMap2.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: test-pod

spec:

  containers:

    - name: test-container

      image: nginx

      volumeMounts:

      - name: config-vol

        mountPath: "/data"

  volumes:

    - name: config-vol

**configMap**:

        name: mysettings-config3

**Execute the following commands and note that the file is created by name location and name.**

kubectl apply -f **ConfigMap2**.yaml

kubectl exec -it pod/test-pod – bash

# cd /data

# cat name

# cat location

**Challenges in using Volume:**

* Sharing Code across different deployments because the volume spec is tightly coupled to pod spec.
* Volume will have same lifecycle as Pod.

Table

Description automatically generated

**Persistent Volume, Persistent Volume Claim and StorageClasses**

**Persistent Volumes (PV)** are resources that need to be provisioned separately from rest of the objects in the Kubernetes cluster.

It has all the required details for the storage and thus it enables portability of your application configuration.

Kubernetes can use these resources but does not manage them.

It can exist beyond the lifetime of an individual pod.

A Kubernetes persistent volume has the following attributes

* It is provisioned either dynamically or by an administrator.
* Created with a particular filesystem.
* Has a particular size
* Has identifying characteristics such as volume IDs and a name.

**Persistent Volume Claim (PVC)**

* In order for pods to start using these volumes, they need to be claimed (**via a persistent volume claim**) and the claim referenced in the spec for a pod.
* A **Persistent Volume Claim** describes the amount and characteristics of the storage (**Size, Access Mode and Storage)** required by the pod, finds any matching persistent volumes and claims these.
* The cluster will map the PVC to a PV.

### 

**Status Provisioning Workflow**

1. As a Administrator, create a PersistentVolume (PV)
2. As a Developer
   1. Create a PersistentVolumeClaim(PVC) and bind to PV
   2. Define Volume in Pod Spec using PersistentVolumeClaim(PVC)

**Use Case:**

The most common use case for Persistent volumes in Kubernetes is for databases. Obviously a database needs to have access to its data at all times, and by leveraging PVs, we can start using databases like MySQL, Cassandra, CockroachDB and even MS SQL for our applications.

**Creating PersistentVolume and PersistentVolumeClaim**

apiVersion: v1

kind: PersistentVolume

metadata:

  name: mysql-pv

  labels:

    type: local

spec:

  storageClassName: manual #This is Optional

  capacity:

    storage: 10Gi

  accessModes:

    - ReadWriteOnce

  hostPath:

    path: "/data"

---

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: mysql-pv-claim

spec:

  storageClassName: manual

  accessModes:

    - ReadWriteOnce

  resources:

    requests:

      storage: 10Gi

Note: 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.

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.

**Creating MySQL Database Deployment**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: mysql-dep

spec:

  replicas: 1

  selector:

    matchLabels:

      app: "mysql-pod"

  template:

    metadata:

      labels:

        app: "mysql-pod"

    spec:

**containers:**

      - image: mysql:5.6

        name: mysql

        env:

        - name: MYSQL\_ROOT\_PASSWORD

          value: tiger1234

        ports:

        - containerPort: 3306

          name: mysql

        volumeMounts:

        - name: mysql-persistent-volume-claim

          mountPath: /var/lib/mysql

**volumes**:

      - name: mysql-persistent-volume-claim

**persistentVolumeClaim**:

          claimName: mysql-pv-claim

---

apiVersion: v1

kind: Service

metadata:

  name: mysql-srv

spec:

  ports:

  - port: 3306

  selector:

    app: "mysql-pod"

**Create a Pod to test MySQL Server**

kubectl apply -f mysql.yaml

kubectl run **mysql-client** -it --rm --restart=Never --image=mysql:5.6 -- bash

# **mysql** -h mysql-srv -uroot -ptiger1234

sql> create database demo;

sql> use demo;

sql> CREATE TABLE Persons (  
    PersonID int,  
    LastName varchar(255),  
    FirstName varchar(255),  
    Address varchar(255),  
    City varchar(255)  
);

sql> select \* from Persons;

sql> exit

# exit

kubectl **delete -f mysql.yaml**

kubectl apply -f mysql.yaml

kubectl run **mysql-client** -it –rm --restart=Never --image=mysql:5.6 -- bash

# **mysql** -h mysql-srv -uroot -ptiger1234

sql> use demo;

sql> select \* from Persons;

**Dynamic Creation of Persistent Volume using Storage Class**

**Storage classes:**

* A StorageClass provides a way for administrators to describe the "classes" of storage they offer.
* Storage classes give you the flexibility to choose multiple volume options for a Kubernetes cluster. You can use multiple storage solutions from various vendors, including Azure, Amazon Web Service, and Google Cloud Platform.
* Containers request storage from a cluster via a volume claim, which is a declarative claim for a specific type of volume access mode, capacity, and so on. The Kubernetes cluster evaluates this claim request, and then assigns a volume from its storage class.
* Storage class enables Dynamic Provisioning of Persistent Volume.

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

**Dynamic Provisioning:**

You create the persistent volume claim and let persistent volume be created on demand by cluster.

Can leverage storage classes for defining and matching types of storage.

**Storage Classes are defined by three properties:**

➢ **provisioner** – component that creates persistent volumes on demand

➢ **reclaimPolicy** – what to do with dynamically created volumes when claim is deleted

➢ **volumeBindingMode** – eager vs. lazy binding/creation (at same time as claim creation or only when pod using the claim get created)

**Provisioner:**

Each StorageClass has a provisioner that determines what volume plugin is used for provisioning PVs.

[**https://kubernetes.io/docs/concepts/storage/storage-classes/#azure-file**](https://kubernetes.io/docs/concepts/storage/storage-classes/#azure-file)

**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**.

* reclaimPolicy: Retain = After deleting PVC, retains the PV and corresponding Disk. PV will be marked as "released". But it is not yet available for another claim because the previous claimant's data remains on the volume.
* reclaimPolicy: Delete = After deleting PVC, deletes the PV and corresponding Disk.

**Lab: Dynamic Provisioning Workflow**

1. Create a StorageClass
2. Create a PersistentVolumeClaim
3. Define Volume in Pod Spec (Creates a PersistentVolume)

1. **Create a Custom Storage Class:**

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

  name: managed-standard-ssd-storageclass

volumeBindingMode: WaitForFirstConsumer #Default is **Immediate**

reclaimPolicy: Retain #or Delete

provisioner: kubernetes.io/azure-disk

parameters:

  cachingmode: ReadOnly

  kind: Managed

  storageaccounttype: StandardSSD\_LRS

1. **Create a PersistentVolumeClaim**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: pvc-azure-standard-ssd

spec:

  accessModes:

  - ReadWriteOnce

  storageClassName: managed-standard-ssd-storageclass

  resources:

    requests:

      storage: 10Gi

At this point, PV is automatically created and also Azure Disk resource is created.

Note: If storageClassName is not mentioned the PVC uses default storage class and the PV will be created only when PVC is used in any deployment/pod. This behaviour is because Default Storage Class VolumeBindingMode = WiatForFirstConsumer

1. **Create Deployment using persistent volume claim.**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

      containers:

      - name: nginx

        image: nginx

        ports:

        - containerPort: 80

        volumeMounts:

        - name: webcontent

          mountPath: "/usr/share/nginx/html/web-app"

      volumes:

      - name: webcontent

**persistentVolumeClaim:**

**claimName: pvc-azure-standard-ssd**