**Working with Kubernetes Storage Options**

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

**Kubernetes 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 persistent volume represents its **storage capacity**.

**Storage API Objects in Kubernetes**

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

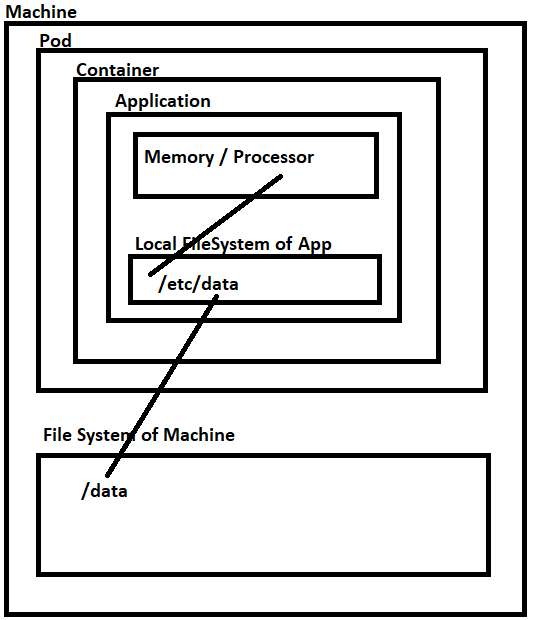
**Kubernetes Volume:**

* 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**
* **. . .**

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



**Pods.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: test-pd

spec:

**containers:**

  - image: nginx

    name: test-container

**volumeMounts**:

    - mountPath: /etc/data

      name: test-volume

**volumes:**

  - name: test-volume

    hostPath:

      path: /data

      type: DirectoryOrCreate

**HOST Directory:** **/var/lib/docker/volumes/minikube/\_data/data/Demo.txt**

**Execute the commands in following order**

1. **D:\>** docker run -it --rm -v /:/hostroot ubuntu bash (To Enter the Shell of Host Linux VM used by Docker)
2. **root@7b975680d414:/hostroot#** ls -l > demo.txt - This creates a file demo.txt in root of the Linux host VM.
3. **root@7b975680d414:/hostroot#** ls -l > exit
4. **D:\>** kubectl apply -f pods.yaml
5. **D:\>** kubectl get all
6. D:\> get logs pod/test-pd

**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 share 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.

apiVersion: v1

kind: Pod

metadata:

  name: sharevol

spec:

**containers**:

  - name: c1

    image: nginx

    volumeMounts:

      - name: xchange

        mountPath: "/data1"

  - name: c2

    image: redis

    volumeMounts:

      - name: xchange

        mountPath: "/data2"

**volumes**:

  - name: xchange

    emptyDir: {}

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

1. D:\Kubernetes>kubectl apply -f pods.yaml

**Step2: We first exec into one of the containers in the pod, c1, check the volume mount and generate some data:**

1. D:\Kubernetes> kubectl exec -it sharevol -c c1 -- bash
2. root@p1:/# cd /data1
3. root@p1:/data1# echo "this is demo" > demo.txt
4. root@p1:/data1# exit
5. exit

**Step3:** When we now exec into c2, the second container running in the pod, we can see the volume mounted at /data2 and are able to read the data created in the previous step:

1. D:\Kubernetes> kubectl exec -it sharevol -c c2 -- bash
2. root@p1:/# cd /data2
3. root@p1:/data2# ls

demo.txt

1. root@p1:/data2#

**ConfigMap and Volumes**

Example1: ConfigMap from YAML

**ConfigMap1.yaml**

apiVersion: v1

kind: ConfigMap

metadata:

  name: mysettings-config4

  namespace: default

data:

  mysettings.json: |-

{

"name":"rahul",

"location":"boston"

}

---

apiVersion: v1

kind: Pod

metadata:

  name: test-pod

spec:

  containers:

    - name: test-container

      image: nginx

      command: [ "cat", "/data/mysettings.json" ]

      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=John

location=boston

**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.

**Persistent Volume, Persistent Volume Claim**

**Persistent Volumes (PV)** are resources that need to be provisioned separately from 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, the volume needs 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-volume

  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 (RWO)** -- the volume can be mounted as read-write by a single node.
* **ReadWriteOncePod (RWOP)** -- the volume can be mounted as read-write by a single Pod (introduced from 1.22).
* **ReadOnlyMany (ROX)** -- the volume can be mounted read-only by many nodes.
* **ReadWriteMany (RWX)** -- the volume can be mounted as read-write by many nodes.

**Creating MySQL Database Deployment**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: mysql-ss

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 run -it --rm --image=mysql:5.6 **mysql-client** -- mysql -h mysql-srv -uroot -ptiger1234

**Note:**

* mysql -uroot -ptiger1234
* mysql - command to be executed
* -h is host(mysql-srv)
* mysql-srv is service name
* -u is user (root)
* -p is password (tiger1234)