DOCKER & KUBERNETES : PERSISTENT VOLUMES & PERSISTENT VOLUMES CLAIMS - HOSTPATH AND ANNOTATIONS

Kubernetes Volume

A Container's file system lives only as long as the Container does. So when a Container terminates and restarts, filesystem changes are lost. A Kubernetes volume, unlike the volume in Docker, has an explicit lifetime - the same as the Pod that encloses it.

Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too.

However, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously. For more consistent storage that is independent of the Container, we can use a Volume. This is especially important for stateful applications, such as key-value stores (such as Redis) and databases.

To use a volume, a Pod specifies what volumes to provide for the Pod (the .spec.volumes field) and where to mount those into Containers (the .spec.containers.volumeMounts field) as shown below (**busybox.yaml**):

kind: Pod

apiVersion: v1

metadata:

name: busybox-pod

**spec**:

**volumes**:

- name: cache

**containers**:

- name: bysybox-container

image: busybox

command: ['sleep', '3600']

**volumeMounts**:

- mountPath: "**/cache**"

name: cache

We can check the mounted volume, **cache**:

$ kubectl create -f busybox.yaml

pod/busybox-pod created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

busybox-pod 1/1 Running 0 18s

$ kubectl exec -it busybox-pod sh

/ # ls -la

total 48

drwxr-xr-x 1 root root 4096 Mar 25 17:35 .

drwxr-xr-x 1 root root 4096 Mar 25 17:35 ..

-rwxr-xr-x 1 root root 0 Mar 25 17:35 .dockerenv

drwxr-xr-x 2 root root 12288 Feb 14 18:58 bin

drwxrwxrwx 2 root root 4096 Mar 25 17:35 **cache**

...

Here, we created a Pod that runs one Container. This Pod has a Volume of type **emptyDir** that lasts for the life of the Pod, even if the Container terminates and restarts.

A process in a container sees a filesystem view composed from their Docker image and volumes. The Docker image is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image.

Each Container in the Pod must independently specify where to mount each volume.

Volume of type "emptyDir"

In this section, we'll create a Pod that runs one Container. This Pod has a Volume of type **emptyDir** that lasts for the life of the Pod, even if the Container terminates and restarts. Here is the configuration file for the Pod (**redis.yaml**):

apiVersion: v1

kind: Pod

metadata:

name: redis

spec:

containers:

- name: redis

image: redis

volumeMounts:

- name: redis-storage

mountPath: /data/redis

volumes:

- name: redis-storage

emptyDir: {}

We can check the mounted volume, **cache**:

$ kubectl create -f redis.yaml

pod/redis created

$ kubectl get pod redis --watch

NAME READY STATUS RESTARTS AGE

redis 1/1 Running 0 33s

In another terminal, get a shell to the running Container and create a file:

$ kubectl exec -it redis -- /bin/bash

# cd /data/redis/

root@redis:/data/redis# echo Hello > test-file

root@redis:/data/redis# ls

test-file

In the shell, list the running processes:

root@redis:/data/redis# apt-get update

root@redis:/data/redis# apt-get install procps

root@redis:/data/redis# ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

redis 1 0.1 0.1 50288 3068 ? Ssl 19:27 0:00 redis-server \*:

root 15 0.0 0.1 18136 2100 pts/0 Ss 19:29 0:00 /bin/bash

root 275 0.0 0.1 36636 2756 pts/0 R+ 19:32 0:00 ps aux

In our shell, kill the Redis process:

root@redis:/data/redis# kill 1

root@redis:/data/redis# command terminated with exit code 137

In our original terminal, watch for changes to the Redis Pod. Eventually, we will see something like this:

$ kubectl get pod redis --watch

NAME READY STATUS RESTARTS AGE

redis 1/1 Running 0 33s

redis 0/1 Completed 0 8m50s

redis 1/1 Running 1 8m53s

At this point, the Container has terminated and restarted. This is because the Redis Pod has a "restartPolicy" of "Always".

Let's get into a shell of the restarted Container:

kubectl exec -it redis -- /bin/bash

root@redis:/data# ls /data/redis

test-file

We can see that **test-file** we created before the container restarted is still there.

Note that in addition to the local disk storage provided by "emptyDir", Kubernetes supports many different **network-attached storage** solutions, including **PD** on GCE and **EBS** on EC2, which are preferred for critical data and will handle details such as mounting and unmounting the devices on the nodes.

PersistentVolume for Storage

In the following sections, we'll learn how to configure a Pod to use a **PersistentVolumeClaim** for storage. Here is a summary of the process:

1. A cluster administrator creates a **PersistentVolume** that is backed by physical storage. The administrator does not associate the volume with any Pod.
2. A cluster user creates a **PersistentVolumeClaim**, which gets automatically bound to a suitable **PersistentVolume**.
3. The user creates a Pod that uses the **PersistentVolumeClaim** as storage.

Create a hostPath PersistentVolume

In this section, we'll create a **hostPath PersistentVolume**. Kubernetes supports **hostPath** for development and testing on a single-node cluster. A hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, however, we would not use hostPath. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume. Cluster administrators can also use [StorageClasses](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.13/" \l "storageclass-v1-storage" \t "_blank) to set up [dynamic provisioning](https://kubernetes.io/blog/2016/10/dynamic-provisioning-and-storage-in-kubernetes).

Before we move on, we want to create an **index.html** file on our Node (minikube).

Let's open a shell on minikube and create the file:

$ minikube ssh

\_ \_

\_ \_ ( ) ( )

\_\_\_ \_\_\_ (\_) \_\_\_ (\_)| |/') \_ \_ | |\_ \_\_

/' \_ ` \_ `\| |/' \_ `\| || , < ( ) ( )| '\_`\ /'\_\_`\

| ( ) ( ) || || ( ) || || |\`\ | (\_) || |\_) )( \_\_\_/

(\_) (\_) (\_)(\_)(\_) (\_)(\_)(\_) (\_)`\\_\_\_/'(\_,\_\_/'`\\_\_\_\_)

$ sudo mkdir /mnt/data

$ echo 'Hello from Kubernetes storage' | sudo tee -a /mnt/data/index.html

Hello from Kubernetes storage

Here is the configuration file for the hostPath PersistentVolume (**pv-volume.yaml**):

kind: PersistentVolume

apiVersion: v1

metadata:

name: task-pv-volume

labels:

type: local

spec:

**storageClassName: manual**

capacity:

storage: 10Gi

accessModes:

- **ReadWriteOnce**

**hostPath**:

path: "/mnt/data"

The configuration file specifies that the volume is at **/mnt/data** on the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of **ReadWriteOnce**, which means the volume can be mounted as read-write by a single Node. It defines the **StorageClass** name **manual** for the PersistentVolume, which will be used to bind PersistentVolumeClaim requests to this PersistentVolume.

Let's create the PersistentVolume and view it:

$ kubectl create -f pv-volume.yaml

persistentvolume/task-pv-volume created

$ kubectl get pv task-pv-volume

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE

task-pv-volume 10Gi RWO Retain **Available** manual 24s

The output indicates that the PersistentVolume has a STATUS of **Available**. This means it has not yet been bound to a PersistentVolumeClaim.

Create a PersistentVolumeClaim

The next step is to create a PersistentVolumeClaim. Pods use PersistentVolumeClaims to request physical storage. In this section, we create a PersistentVolumeClaim that requests a volume of at least three gibibytes that can provide read-write access for at least one Node.

Here is the configuration file for the PersistentVolumeClaim (**pv-claim.yaml**):

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: task-pv-claim

spec:

storageClassName: manual

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 3Gi

Let's create the PersistentVolumeClaim:

$ kubectl create -f pv-claim.yaml

persistentvolumeclaim/task-pv-claim created

After we created the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim's requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Look again at the PersistentVolume:

$ kubectl get pv task-pv-volume

NAME CAPACITY ACCESS MODES RECLAIM POLICY **STATUS** CLAIM STORAGECLASS REASON AGE

task-pv-volume 10Gi RWO Retain **Bound** default/task-pv-claim manual 9m

Now the output shows a STATUS of **Bound**!

Then, let's look at the PersistentVolumeClaim:

$ kubectl get pvc task-pv-claim

NAME **STATUS** VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

**task-pv-claim** **Bound** **task-pv-volume** 10Gi RWO manual 4m

Create a Pod uses PersistentVolumeClaim as a volume

Now it's time to create a Pod that uses our PersistentVolumeClaim as a volume. The configuration file for the Pod looks like this (**pv-pod.yaml**):

kind: Pod

apiVersion: v1

metadata:

name: task-pv-pod

spec:

volumes:

- name: task-pv-storage

**persistentVolumeClaim**:

claimName: task-pv-claim

containers:

- name: task-pv-container

image: nginx

ports:

- containerPort: 80

name: "http-server"

volumeMounts:

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

name: task-pv-storage

One thing to notice is that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. That's because from the Pod's point of view, the claim is a volume.

Let's create the Pod:

$ kubectl create -f pv-pod.yaml

pod/task-pv-pod created

$ kubectl get pod task-pv-pod

NAME READY STATUS RESTARTS AGE

task-pv-pod 1/1 Running 0 59s

Get a shell to the Container running in our Pod and check if nginx is serving the index.html file from the hostPath volume:

$ kubectl exec -it task-pv-pod -- /bin/bash

root@task-pv-pod:/# apt-get update

root@task-pv-pod:/# apt-get install curl

root@task-pv-pod:/# curl localhost

Hello from Kubernetes storage

The output shows the text that we wrote to the index.html file on the hostPath volume!

Access control - annotations

Storage configured with a group ID (GID) allows writing only by Pods using the same GID. Mismatched or missing GIDs cause permission denied errors. To reduce the need for coordination with users, an administrator can **annotate** a PersistentVolume with a GID. Then the GID is automatically added to any Pod that uses the PersistentVolume.

kind: PersistentVolume

apiVersion: v1

metadata:

name: task-pv-volume

labels:

type: local

**annotations**:

**pv.beta.kubernetes.io/gid**: **"1001"**

spec:

storageClassName: manual

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/mnt/data"

So, when a Pod consumes a PersistentVolume that has a GID annotation, the annotated GID is applied to all Containers in the Pod in the same way that GIDs specified in the Pod's security context are. Every GID, whether it originates from a PersistentVolume annotation or the Pod’s specification, is applied to the first process run in each Container.