Configure a Pod to Use a PersistentVolume for Storage

This page shows you how to configure a Pod to use a PersistentVolumeClaim for storage. Here is a summary of the process:

1. You, as cluster administrator, create a PersistentVolume backed by physical storage. You do not associate the volume with any Pod.

2. You, now taking the role of a developer / cluster user, create a PersistentVolumeClaim that is automatically bound to a suitable PersistentVolume.

3. You create a Pod that uses the above PersistentVolumeClaim for storage.

Before you begin

- You need to have a Kubernetes cluster that has only one Node, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a single-node cluster, you can create one by using Minikube.

- Familiarize yourself with the material in Persistent Volumes.

Create an index.html file on your Node

Open a shell to the single Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering `minikube ssh`.

In your shell on that Node, create a `/mnt/data` directory:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo mkdir /mnt/data
```

In the `/mnt/data` directory, create an `index.html` file:

```
# This again assumes that your Node uses "sudo" to run commands
# as the superuser
sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/index.html"
```

Note: If your Node uses a tool for superuser access other than `sudo`, you can usually make this work if you replace `sudo` with the name of the other tool.

Test that the `index.html` file exists:

```
cat /mnt/data/index.html
```

The output should be:

```
Hello from Kubernetes storage
```

You can now close the shell to your Node.

Create a PersistentVolume

In this exercise, you 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, you 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 to set up dynamic provisioning.

Here is the configuration file for the hostPath PersistentVolume:

pods/storage/pv-volume.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
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.

Create the PersistentVolume:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml
```

View information about the PersistentVolume:

```
kubectl get pv task-pv-volume
```

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

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS      CLAIM     STOR
task-pv-volume   10Gi       RWO           Retain          Available             manu
```

Create a PersistentVolumeClaim

The next step is to create a PersistentVolumeClaim. Pods use PersistentVolumeClaims to request physical storage. In this exercise, you 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:

pods/storage/pv-claim.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-claim.yaml
```

After you create 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
```

Now the output shows a `STATUS` of `Bound`.

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS   CLAIM
task-pv-volume   10Gi       RWO           Retain          Bound    default/task-pv-
```

Look at the PersistentVolumeClaim:

```
kubectl get pvc task-pv-claim
```

The output shows that the PersistentVolumeClaim is bound to your PersistentVolume, `task-pv-volume`.

```
NAME            STATUS    VOLUME           CAPACITY   ACCESSMODES   STORAGECLASS   A
task-pv-claim   Bound     task-pv-volume   10Gi       RWO           manual         3
```

Create a Pod

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod:

pods/storage/pv-pod.yaml

```yaml
apiVersion: v1
kind: Pod
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
```

Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
```

Verify that the container in the Pod is running;

```
kubectl get pod task-pv-pod
```

Get a shell to the container running in your Pod:

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

In your shell, verify that nginx is serving the `index.html` file from the hostPath volume:

```
# Be sure to run these 3 commands inside the root shell that comes from
# running "kubectl exec" in the previous step
apt update
apt install curl
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file on the hostPath volume:

```
Hello from Kubernetes storage
```

If you see that message, you have successfully configured a Pod to use storage from a PersistentVolumeClaim.

Clean up

Delete the Pod, the PersistentVolumeClaim and the PersistentVolume:

```
kubectl delete pod task-pv-pod
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

If you don't already have a shell open to the Node in your cluster, open a new shell the same way that you did earlier.

In the shell on your Node, remove the file and directory that you created:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo rm /mnt/data/index.html
sudo rmdir /mnt/data
```

You can now close the shell to your Node.

Mounting the same persistentVolume in two places

pods/storage/pv-duplicate.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: test
      image: nginx
      volumeMounts:
        # a mount for site-data
        - name: config
          mountPath: /usr/share/nginx/html
          subPath: html
        # another mount for nginx config
        - name: config
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
  volumes:
    - name: config
      persistentVolumeClaim:
        claimName: test-nfs-claim
```

You can perform 2 volume mounts on your nginx container:

`/usr/share/nginx/html` for the static website `/etc/nginx/nginx.conf` for the default config

Access control

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.

Use the `pv.beta.kubernetes.io/gid` annotation as follows:

```yaml
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
  name: pv1
  annotations:
    pv.beta.kubernetes.io/gid: "1234"
```

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.

> **Note:** When a Pod consumes a PersistentVolume, the GIDs associated with the PersistentVolume are not present on the Pod resource itself.

What's next

- Learn more about PersistentVolumes.
- Read the Persistent Storage design document.

Reference

- PersistentVolume
- PersistentVolumeSpec
- PersistentVolumeClaim
- PersistentVolumeClaimSpec

Feedback

Was this page helpful?

Yes No

Last modified September 21, 2022 at 1:32 PM PST: Link to Dynamic Provisioning doc instead of blog (4be5a30965)