Create ReadWriteMany PersistentVolumeClaims on your Kubernetes Cluster

Lexa Follow
May 16, 2019 · 8 min read ★

Kubernetes allows us to provision our `PersistentVolumes` dynamically using `PersistentVolumeClaims`. Pods treat these claims as volumes. The access mode of the `PVC` determines how many nodes can establish a connection to it. We can refer to the resource provider's docs for their supported access modes.

> *A* **`PersistentVolume`** *can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.*
>
> *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*
>
> *In the CLI, the access modes are abbreviated to:*
>
> *- RWO — ReadWriteOnce*
> *- ROX — ReadOnlyMany*
> *- RWX — ReadWriteMany*

— Kubernetes Docs

I recently worked on a High Availability project that required sharing a volume between multiple pods. For learning purposes, I deployed this project to 3 different Kubernetes platforms (GKE, EKS, and DigitalOcean). If your project's mere requirement for your

volumes is to share the content with multiple pods, you can employ the following solutions depending on your cloud provider.

Amazon EKS: FileSystem Provisioner using EFS

> *The efs-provisioner allows you to mount EFS storage as PersistentVolumes in Kubernetes. It consists of a container that has access to an AWS EFS resource. The container reads a* `configmap` *which contains the EFS filesystem ID, the AWS region, and the name you want to use for your efs-provisioner. This name will be used later when you create a storage class.*

— EFS Provisioner Docs

We will be using the Helm chart for `efs-provisioner`.

> *This chart deploys the EFS Provisioner and a StorageClass for EFS volumes (optionally as the default). The EFS external storage provisioner runs in a Kubernetes cluster and will create persistent volumes in response to the PersistentVolumeClaim resources being created. These persistent volumes can then be mounted on containers. The persisent volumes are created as folders with in an AWS EFS filesystem.*

— EFS Helm Chart

Before we install our provisioner, we need to take install Helm and Tiller. Let's look at their application as defined by Helm docs:

> *Helm is a tool that streamlines installing and managing Kubernetes applications. Think of it like apt/yum/homebrew for Kubernetes.*
>
> *Helm has two parts: a client (* `helm` *) and a server (* `tiller` *)*
>
> *Tiller runs inside of your Kubernetes cluster, and manages releases (installations) of your charts.*

— Helm Docs

Now you can install Helm on your computer by running the following:

```
$ brew install kubernetes-helm
```

If you are not using MacOS, you can find other ways of installing Helm here.

Now that you have installed Helm (client) you will need to install Tiller (server) on your Kubernetes cluster.

In `rbac-config.yaml` insert the following:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

Now run the following to install Tiller on your cluster along with its role:

```
$ kubectl apply -f rbac-config.yaml
$ helm init --service-account tiller
```

Now open the Amazon EFS Management Console at https://console.aws.amazon.com/efs/ and create an EFS FileSystem using the tutorial found here. Once created you will need your FileSystem ID with the format `fs-xxxxxx` and the zone you created your EFS in.

Now replace the variables marked in bold and run the following command in your terminal to install the provisioner:

```
$ helm install --name efs-provisioner \
    --namespace default \
    --set  efsProvisioner.efsFileSystemId=fs-xxxxxx \
    --set efsProvisioner.awsRegion=us-east-1
    stable/efs-provisioner
```

You can now create a `ReadWriteMany` `PVC` using a manifest with the following template:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: <pvc name>
  namespace: <pvc namespace>
spec:
  storageClassName: "efs"
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: <pvc storage capacity>
```

Note that we are using the "efs" `storageClassName` here. This class is automatically created when you install the provisioner through Helm.

Google GKE: NFS Server using Google Disk

There are quite a few tutorials on how to use Google Disk for Kubernetes Persistent Volumes. Note that using Disks, Storage or FS services from the provider reduces the chances of losing our Volume data. The data may remain on these platforms even if the PVC is deleted.

The steps you need to take for this installation are:

1. Create a Google Disk instance with your desired capacity.
2. Create the `nfs-server` deployment and service.
3. Create your `PV` and the corresponding `PVC` with special specs.

You can find a detailed tutorial on how to deploy an `nfs-server` on your GKE cluster here. Follow this tutorial and you should be able to share a `PVC` created with the steps found in the link above with the access mode `ReadWriteMany`.

Another good tutorial on this subject can be found here.

DigitalOcean Kubernetes: S3-CLI Using Spaces

DigitalOcean Spaces is usually described as a storage service that compares to S3 from AWS. You can create buckets, much like AWS and operate it using Amazon's API for S3. Here we are aiming to create a bucket for every volume claim such that we can attach the claim using the `ReadWriteMany` access mode.

The existing CSI tool for Digital Ocean Spaces (`csi-digitalocean`) can only support single-node connection:

```
Only SINGLE_NODE_WRITER is supported ('accessModes.ReadWriteOnce' on
Kubernetes)
```

That's why I decided to experiment with the `s3-cli` tool and essentially make it work with my DigitalOcean Spaces.

Since this tool was not designed to work with DigitalOcean, there are a few steps to follow. The following content comes from the `s3-cli` GitHub repo with a few adjustments amended.

1. Create your credential secret

This step is directly taken from the `s3-cli` GitHub repo. Generate a new Spaces key pair on your DO dashboard at:

> *https://cloud.digitalocean.com/account/api/*

Take your keypair you just generated and create the following secret on your DO cluster (here we have chosen the region `nyc3` :

```
apiVersion: v1
kind: Secret
metadata:
  name: csi-s3-secret
stringData:
  accessKeyID: <YOUR_DO_SPACES_ACCESS_KEY_ID>
  secretAccessKey: <YOUR_DO_SPACES_SECRET_ACCES_KEY>
  endpoint: "https://nyc3.digitaloceanspaces.com"
  region: ""
  encryptionKey: ""
```

2. Change the namespace to default and create your resources

Now clone the `s3-cli` GitHub repo and from the `deploy/kubernetes` folder open the files `provisioner.yaml` , `attacher.yaml` and `csi-s3.yaml` . In these files, you are going to replace `kube-system` with `default` in all `namspace` values. This is because we have created the secret in the default namespace.

Once changed, create these resources using the following (source is `s3-cli` GitHub repo) :

```
kubectl create -f provisioner.yaml
kubectl create -f attacher.yaml
kubectl create -f csi-s3.yaml
```

3. Create your `storageclass` with the `goofys` mounter

You can use a few mounters for this tool to mount the bucket. I tried all of them to find the best one for DigitalOcean. The available mounters are `rclone`, `s3fs`, `goofys` , and `s3backer`. My conclusions are:

- `s3backer`: backs up data and allows for `readWriteMany` access, but for some reason creates a new backup every few minutes and clutters the bucket.
- `s3fs`: successfully creates the bucket but simply does not back up any stored files into the bucket. Basically dysfunctional in our case.
- `rclone`: creates a bucket, backs files up but does not attach to more than one pod.
- `goofys`: golden, does everything you want!

So I decided to go with the best option, `goofys`. Open your `storageclass.yaml` file inside the `deploy/kubernetes` directory, and change the mounter to `goofys`:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-s3
provisioner: ch.ctrox.csi.s3-driver
parameters:
  # specify which mounter to use
  # can be set to s3backer, s3ql, s3fs or goofys
  mounter: goofys
```

Now go ahead and apply this file. You can now create the PVC using the `csi-s3` storage class. There is one step left to do after you create your PVC.

4. Manually add the metadata file to your bucket

Now go ahead and create your PVC (this is not the last step):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-do-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-s3
```

After hours of getting errors about not finding the metadata file, I discovered that it might be some compatibility issue with DigitalOcean. I found the format of this file and manually created and added it to my bucket. This file has information about your bucket name, size, FS path and what mounter you are using. The only information that you will need to provide in this file to initiate your bucket is the size. Go ahead and create a file called `.metadata.json` , and put your bucket size is it. Note that the bucket size is in bytes so you will need to put a number higher than the actual number of bytes needed. For example, if I want a 1GB bucket, the actual size is 1024*1024*1024 = 1073741824 bytes. So I can go ahead and use 10000000000 which is almost 10 times as big. So my JSON file will look like this:

```
{
  "capacityBytes": 10000000000
}
```

Now go ahead and upload your file (`.metadata.json`) to your bucket on DO Spaces bucket that just got automatically created. Once the provisioner `recognizes this file, it will populate it further and the output file may look like this:

```
{
  "Name":"pvc-xxxx-xxxx",
  "Mounter":"goofys",
  "FSPath":"csi-fs",
  "CapacityBytes":1073741824
}
```

You do not have to make the file look like this, it will be automatically done for you. Now you have yourself a PVC with `ReadWriteMany` access permissions using DigitalOcean Spaces:

```
$ kubectl get pvc

NAME      STATUS VOLUME        CAPACITY  ACCESS   STORAGECLASS   AGE
my-do-pvc Bound  pvc-xxxx-xxxx 1Gi       RWX      csi-s3         22h
```

In order to monitor the creation of your volume and your files being backed up in the FS path in your bucket, follow the logs:

```
$ kubectl logs -l app=csi-provisioner-s3 -c csi-s3
```

Other Cloud Providers

There is always a way! Considering your cloud provider does not offer FS, Storage or Disk services, you can use tools that replicate an NFS Provisioner. A quite functional one can be installed using a Helm chart.

> *NFS Server Provisioner is an out-of-tree dynamic provisioner for Kubernetes. You can use it to quickly & easily deploy shared storage that works almost anywhere.*
>
> *This chart will deploy the Kubernetes external-storage projects `nfs` provisioner. This provisioner includes a built in NFS server, and is not intended for connecting to a pre-existing NFS server. If you have a pre-existing NFS Server, please consider using the NFS Client Provisioner instead.*

— NFS Server Provisioner Helm Chart

This is likely the easiest of the 3 methods discussed in this document. As mentioned in the docs, this tool merely enables the `ReadWriteMany` access mode on your claims, however, it does not go any further than Kubernetes' default volumes for storage.

You can install this pseudo-NFS-Provisioner on your cluster using Helm by running the following command:

```
$ helm install --name nfs \
    stable/nfs-server-provisioner
```

You can now create volumes using the created `storageClass`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: <pvc name>
  namespace: <pvc namespace>
spec:
  storageClassName: "nfs"    #the installed Chart just created this
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: <pvc storage capacity>
```

Enjoy!

Kubernetes Digitalocean Amazon Gke Persistent Volume