Using Storage Classes to Dynamically Provision Persistent Volumes

In this lesson, we will learn how to provision persistent volumes dynamically by using storage classes.

WE'LL COVER THE FOLLOWING

Static Volume Provisioning

So far, we used static PersistentVolumes. We had to create both EBS volumes and Kubernetes PersistentVolumes manually. Only after both became available we were able to deploy Pods that are mounting those volumes through PersistentVolumeClaims. We'll call this process static volume provisioning.

In some cases, static volume provisioning is a necessity. Our infrastructure might not be capable of creating dynamic volumes. That is often the case with on-premise infrastructure with volumes based on Network File System (NFS). Even then, with a few tools, a change in processes, and the right choices for supported volume types, we can often reach the point where volume provisioning is dynamic. Still, that might prove to be a challenge with legacy processes and infrastructure.

Since our cluster is in AWS, we cannot blame legacy infrastructure for

provisioning volumes manually. Indeed, we could have jumped straight into this section. After all, AWS is all about dynamic infrastructure management.

However, we felt that it will be easier to understand the processes by exploring manual provisioning first. The knowledge we obtained thus far will help us understand better what's coming next.

The second reason is that maybe you will run a Kubernetes cluster on infrastructure that has to be static. Even though we're using AWS for the examples, everything you learned this far can be implemented on static infrastructure. You'll only have to change EBS with NFS and go through the NFSVolumeSource documentation. There are only **three** NFS-specific fields so you should be up-and-running in no time.

Before we discuss how to enable dynamic persistent volume provisioning, we should understand that it will be used only if none of the static PersistentVolumes match our claims. In other words, Kubernetes will always select statically created PersistentVolumes over dynamic ones.

Dynamic Volume Provisioning

Dynamic volume provisioning allows us to create storage on-demand. Instead of manually pre-provisioning storage, we can provision it automatically when a resource requests it.

We can enable dynamic provisioning through the usage of StorageClasses from the `storage.k8s.io` API group. They allow us to describe the types of storage that can be claimed.

On the one hand, a cluster administrator can create as many StorageClasses as there are storage flavors. On the other hand, the users of the cluster do not have to worry about the details of each available external storage. It's a win-win situation where the administrators do not have to create PersistentVolumes in advance, and the users can simply claim the storage type they need.

StorageClasses

To enable dynamic provisioning, we need to create at least one StorageClass object. Luckily for us, kops already set up a few, so we might just as well take a look at the StorageClasses currently available in our cluster.

```
kubectl get sc
```

The **output** is as follows.

```
NAME          PROVISIONER           AGE
default       kubernetes.io/aws-ebs 44m
gp2 (default) kubernetes.io/aws-ebs 44m
```

We can see that there are two StorageClasses in our cluster. Both are using the same `aws-ebs` provisioner. Besides the names, the only difference, at least in this output, is that one of them is marked as `default`. We'll explore what that means a bit later. For now, we'll trust that kops configured those classes correctly and try to claim a PersistentVolume.

Claiming PersistantVolume

Let's take a quick look at yet another `jenkins` definition.

```
cat pv/jenkins-dynamic.yml
```

The **output**, limited to the relevant parts, is as follows.

```
...
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: jenkins
  namespace: jenkins
spec:
  storageClassName: gp2
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
...
```

This Jenkins definition is almost the same as the one we used before. The only difference is in the PersistentVolumeClaim that, this time, specified `gp2` as the StorageClassName. There is one more difference though. This time we do not have any PersistentVolume pre-provisioned. If everything works as expected, a new PersistentVolume will be created dynamically.

Creating Resources

```
kubectl apply \
    -f pv/jenkins-dynamic.yml \
    --record
```

We can see that some of the resources were re-configured, while others were created.

Next, we'll wait until the `jenkins` Deployment is rolled out successfully.

```
kubectl --namespace jenkins \
    rollout status \
    deployment jenkins
```

Now we should be able to see what happened through the `jenkins` Namespace events.

```
kubectl --namespace jenkins \
    get events
```

The **output**, limited to the last few lines, is as follows.

```
...
20s 20s 1 jenkins.... Deployment            Normal ScalingReplicaSet     deployment-controlle
20s 20s 1 jenkins.... PersistentVolumeClaim Normal ProvisioningSucceeded persistentvolume-cor
```

We can see that a new PersistentVolume was `successfully provisioned`.

Verification

Let's take a look at the status of the PersistentVolumeClaim.

```
kubectl --namespace jenkins get pvc
```

The **output** is as follows.

```
NAME    STATUS VOLUME  CAPACITY ACCESS MODES STORAGECLASS AGE
jenkins Bound  pvc-... 1Gi      RWO          gp2          1m
```

The part of the output that matters is the status. We can see that it `Bound` with the PersistentVolume thus confirming, again, that the volume was indeed created dynamically.

created dynamically.

To be on the safe side, we'll list the PersistentVolumes as well.

```
kubectl get pv
```

The **output** is as follows.

```
NAME    CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM           STORAGECLASS REASON AGE
pvc-... 1Gi      RWO          Delete         Bound  jenkins/jenkins gp2                 4m
```

As expected, the PersistentVolume was created, it is bound to the PersistentVolumeClaim, and its reclaim policy is `Delete`. We'll see the policy in action soon.

Finally, the last verification we'll perform is to confirm that the EBS volume was created as well.

```
aws ec2 describe-volumes \
    --filters 'Name=tag-key,Values="kubernetes.io/created-for/pvc/name"'
```

The **output**, limited to the relevant parts, is as follows.

```
{
  "Volumes": [
    {
      "AvailabilityZone": "us-east-2c",
      ...
      "VolumeType": "gp2",
      "VolumeId": "vol-0a4d5cfa4699e5c6f",
      "State": "in-use",
      ...
    }
  ]
}
```

We can see that a new EBS volume was created in the availability zone `us-east-2c`, that the type is `gp2`, and that its state is `in-use`.

Dynamic provisioning works! Given that we're using AWS, it is a much better solution than using static resources.

Deletion of Resources

Before we move into a next subject, we'll explore the effect of the reclaim

policy `Delete`. To do so, we'll delete the Deployment and the PersistentVolumeClaim.

```
kubectl --namespace jenkins \
    delete deploy,pvc jenkins
```

The **output** is as follows.

```
deployment "jenkins" deleted
persistentvolumeclaim "jenkins" deleted
```

Now that the claim to the volume was removed, we can check what happened with the dynamically provisioned PersistentVolumes.

```
kubectl get pv
```

After a while, the **output** shows that `no resources` were `found`, clearly indicating that the PersistentVolume that was created through the claim is now gone.

How about the AWS EBS volume? Was it removed as well?

```
aws ec2 describe-volumes \
    --filters 'Name=tag-key,Values="kubernetes.io/created-for/pvc/name"'
```

The **output** is as follows.

```
{
  "Volumes": []
}
```

We got an empty array proving that the EBS volume was removed as well.

Through dynamic volume provisioning, not only that volumes are created when resources claim them, but they are also removed when the claims are released. Dynamic removal is accomplished through the reclaim policy `Delete`.

In the next lesson, we will explore the default Storage Classes created by kops.