Hello All and welcome back. In this video, we will learn about Volumes and Persistent Storage in Kubernetes.

We know that containers, and the pods running them, are ephemeral. New pods can get created, old pods can be scaled down or rescheduled to other cluster nodes. Also, whenever there is an update to either the deployment object or you use another container image, the pods get recreated.

In such cases, the data that was saved in the container filesystem will be lost as the new container comes up with new writable layers. This needs to be handled because we cannot let the data just vanish.

So what’s the solution, it is to save the data outside the container file system. So now, the pods can come and go, but the data is safe outside. We can further have other backup mechanisms to safeguard the data which is stored outside of the Kubernetes cluster.

So, let’s understand the concept of Kubernetes Volumes. Volumes provide the storage which saves the data. There are various ways in which volumes can store data – it can save it on the host machine, over the network or on the cloud storage. At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. Volumes are then mounted to a container at a defined path and thus they become available to the containers.

How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used. Kubernetes supports several types of Volumes. In this video, we will cover a few types.

First let’s go through the emptyDir volume type. It is a simple volume type to store data on the node on which the pod is running. As the name suggests, it is empty when the pod starts. And once the pod is removed from the node, the data in the volume also gets deleted. Let’s create a simple deployment, with volume type as emptyDir.

Under volumes, we can see that there is a volume defined of type emptyDir with name as ubuntu-emptydir-storage.

This is a Deployment type object which runs 1 replica of ubuntu container. And we have specified the command tail -f /dev/null to keep the container alive. We have mounted this volume at /tmp/data.

Let’s run this deployment. We can check that the deployment and the pod is in running state. We can list the volumes used by this deployment by running the command “oc describe deployment/ubuntu-emptydir”. We can see here that the volume type is emptyDir which lasts only till the pod lasts.

Let’s exec into this container. Under /tmp/data, let's create a file. I will redirect the echo command output for that. Echo “file created from within the container” > file1.txt. We can cat the filename to verify that the file was actually created. We will come out of this container. Now lets delete this pod.

Since it is a deployment, a new pod would be immediately spun. Let's exec into this and go to directory /tmp/data. As expected the earlier created file is no longer here. That’s because the emptyDir volume type lasts only till the pod lasts.

A couple of use cases for emptyDir volumes are to store Cache or to store intermediate data as part of some computation.

Let’s delete the deployment before proceeding further.

In this video we understood the need for Persistent Storage in Kubernetes and we also went through the example of emptyDir volume type. We will understand more about Persistent volumes in the upcoming videos.

======

Welcome back. In the last video we went through the example of emptyDir volume type.

Let’s now understand and use the hostPath type of volume in our deployments. A hostPath volume mounts a file or directory from the host node’s file system into the Pod. The file or directory on the host machine will last even if the pod is removed or recreated.

In this example, ubuntu deployment is using hostPath volume.

Here the volume type specified is hostpath, with a path to a directory which is present on the host running the pod. We are using type as Directory or Create, which will create an empty directory if it doesn’t exist. This directory is then mounted at /tmp/data within the container.

Let’s run this deployment. The pod gets in running state. Let’s exec inside this pod. If we check now the dir is blank. Let’s create a file and add some text to it. Echo “text added to the file” >> /tmp/file.txt. We will list the pods so that their nodes are also listed.

I will connect to this node. We can see that under the directory we have the file listed. I will just display the contents, which are the same. We can add some content to a new file, and we can cat the same from inside the container as well.

Let’s now scale up the number of replicas for this deployment to 5. Out of the new pods, this pod has been created on the same node as the first one. If we see the contents of the mounted path inside this container, we can see the same contents of the file.

Now, let’s take up this pod which is on some other node. If we check the contents of this mounted path, we see that it is empty here. You would have probably got the reason, and that’s because the volume data is restricted to the node.

This volume type should primarily be used with clusters running a single node only. Because pods may behave differently on different nodes due to different files on the nodes. We will delete the deployment before proceeding further.

Some other commonly used Volume types are as follows -

AWS EBS CSI - The Container Storage Interface of AWS Elastic Block Store

Azure Disk CSI - for mounting Azure Disk

Azure File CSI - mounts a Microsoft Azure File volume into a pod

GCE Persistent Disk CSI

Portworx CSI volumes

NFS volume etc

That’s it about this video. We used Hostpath type of volumes in an example deployment.

==========

Persistent Volumes and Storage Classes

Hello and welcome back. In this video we will learn about Persistent Volumes, Claims and Storage Classes.

The Persistent Volumes provide an API for users and administrators that abstracts details of how storage is provided from how it is consumed. In production deployments or multi-tenant kubernetes clusters, it is quite possible that application developers won’t have permission to access or create volume. As volumes are external to the kubernetes cluster they would require a different set of permissions which the kubernetes developers might not have.

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a node is a cluster resource. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A PersistentVolumeClaim (PVC) is a request for storage by a user or application developer. So in a nutshell, cluster Administrators would create the Persistent Volumes and the developers would consume that Persistent Volume as a Persistent Volume Claim. In this manner, the PV details including authentication details are hidden from the developers.

Now instead of Administrators manually creating the Persistent Volumes everytime there is a requirement, Kubernetes provides a dynamic provisioning method which uses Storage Class. All the required details including the authentication are saved in the Storage Class and the developers use this Storage Class to dynamically create Persistent Volume Claims. Cluster administrators can offer a variety of PersistentVolumes through Storage Classes, without exposing users to the details of how those volumes are implemented. And the users can create PVCs based on those Storage Classes.

Pods use claims as volumes. Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping. Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available.

We can see the list of supported volumes at kubernetes documentation <https://kubernetes.io/docs/concepts/storage/volumes>. You will notice that quite a few volume types, for example AWS Elastic Block Store is now deprecated and is now migrated to the CSI implementation. Kubernetes CSI is a Kubernetes-specific implementation of the Container Storage Interface (CSI). The CSI standard determines how arbitrary blocks and file storage systems are exposed to workloads on containers.

Before the Container Storage Interface, Kubernetes only supported the use of in-tree k8s volume plugins, which had to be written and deployed using core Kubernetes binaries. This meant that storage providers had to check in the core codebase of their k8s plugins to enable support for new storage systems. It was very tightly coupled and it has been resolved by exposing the executable-based API to third-party plugins.Third-party storage vendors can use CSI to build plugins and deploy them to enable Kubernetes to work with new storage systems, without having to edit the core code of Kubernetes.

So, that’s all about this video. We understood the concept behind Static and Dynamic provisioning with Persistent Volumes. To summarize consider that we are using EBS volumes -

Static Provisioning - Associate an externally-created EBS volume with a PersistentVolume (PV) for consumption within Kubernetes.

Dynamic Provisioning - Automatically create EBS volumes and associated PersistentVolumes (PV) from PersistentVolumeClaims) (PVC). Parameters can be passed via a StorageClass for fine-grained control over volume creation.

We also went through the use case for Storage Classes and CSI drivers. In the upcoming videos we will go through examples to dive into this topic.

=====

Hello. In this video we will go through an example of configuring Static Provisioning for AWS Elastic Block Store in our Kubernetes cluster.

Firstly we need the CSI driver for AWS Elastic Block Store installed in our cluster. Driver provides a CSI interface used by Container Orchestrators to manage the lifecycle of Amazon EBS volumes.

You can refer to the CSI Driver installation for AWS EBS storage at <https://github.com/kubernetes-sigs/aws-ebs-csi-driver/blob/master/docs/install.md>. As mentioned here, you can either use Kustomize or Helm to install the Driver. Once the driver has been deployed, verify the pods are running.

Once we have the driver installed in the cluster, we need to provide appropriate AWS permissions to the driver, so that it can create, delete or update the volumes in our AWS account. Basically, the driver requires IAM permissions to talk to Amazon EBS and to manage the volume on user's behalf. This is already done on the cluster and let’s now proceed with creating the Persistent Volume.

Let’s assume you have taken up the role of Cluster Administrator in an organization and there is a requirement from the development team to have 5 GB of storage in Elastic volume, which can then be mounted to their application pods.

Firstly, we need to create the volume for which we will go to the AWS console. Select the option to create a new volume. I will just go with default values and specify the size as 5 GB and zone as us-east-2b.

Now let’s go through the manifest or yaml file for Persistent Volume. The Kind is Persistent Volume and PVs are not namespaced, hence you don’t see any namespace here.

Let’s talk a bit about the access Modes for PVs.

Access modes define how a volume can be used by the cluster. There are three possible ways –

ReadWriteOnce (RWO) - The volume can be mounted as read-write by a single node. So whatever pods are running on the same node can only mount the PV.

ReadOnlyMany (ROX) - The volume can be mounted by many nodes, but only as read-only.

ReadWriteMany (RWX) - The volume can be mounted as read-write by many nodes.

Back to the yaml; next we have specified the capacity as 5 Gi which was the requirement. Next we have the section for csi details. The driver is the csi driver for ebs. Next is the File system type that will be formatted during volume creation. We have selected ext4. Next, we have to specify the volume handle - so let’s copy it from the AWS console.

Next section is for nodeAffinity. A PV can specify node affinity to define constraints that limit what nodes this volume can be accessed from. Pods that use a PV will only be scheduled to nodes that are selected by the node affinity. Since our volume is in us-east-2b, we want only pods running on nodes in us-east-2b zone to connect to this volume. Let’s apply this volume.

We can check its status now, and we can see that the status is Available.

Next is the manifest for the persistent volume claim. PVCs are namespaced so that’s why you see the namespace in the metadata. The storage class name is empty which means that we are performing a static provisioning to the PV. And we have provided the name of the Persistent Volume that we created above. Access mode is again set to ReadWriteOnce and the size as 5 Gi. Let’s apply the yaml.

Now we can check the status of both PVC as well as the PV. Both of them are now in Bound state which is good.

Next we have the same deployment yaml which we had used for earlier volume types. Under volume, we have specified the type as Persistent Volume Claim and the name of the PVC resource. There is no mention of the PV, just the PVC is mentioned here. In addition we have also specified a node name here. This node is in the same zone as the PV and we have already put in the condition in the PV to only allow nodes in this zone to connect. So here we are just ensuring that the bound happens successfully.

Let’s apply this YAML. We see the pod is running now and we can connect a shell to this pod. We can run the command df-h to see the mounted volumes and here we can see the EBS volume. I will create a file again in this volume, which is created successfully. Exit out of the shell and delete the pod. A new pod will come up. We have connected again to this pod and we can verify that the file still exists.

So, that’s all about this video. We created a Persistent Volume for an AWS EBS volume, followed by a Persistent Volume Claim which was bound to the PV. Finally we used the PVC to mount storage to the application pod. Thank you.

====

Hello. In this video we will go through an example of configuring storage to an application pod in a Dynamic manner. We will again use AWS Elastic Block Store as the storage.

Let’s look at the manifest file for the storage class. Storage classes are again not namespaced. We have specified the CSI provisioner for ebs volume types.

We will understand the Volume Binding mode in Kubernetes. The volumeBindingMode field controls when volume binding and dynamic provisioning should occur.

The Immediate mode indicates that volume binding and dynamic provisioning occurs once the PersistentVolumeClaim is created. The PVC will immediately get in Bound state once created.

WaitForFirstConsumer mode delays the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created. So until there is a pod using the PVC, the PVC will remain in Available state and binding will not happen.

In our case, we are going ahead with Immediate binding.

The storage class can be created with several parameters. We have specified the File system type and the EBS volume type.

Next we have the allowedTopologies field. It is used to put constraints on the zone of the disks that will be dynamically created. Here we have added the constraint that the EBS volumes should be created in useast2b zone.

Let’s apply the Yaml. We can check the status of the storage classes in the cluster and we can verify that ebs-sc storage class is created successfully.

Next we have the PVC manifest. We don’t need to manually create the Persistent Volume as we are doing dynamic provisioning. The yaml file is the same as earlier, except for the storage class which we have now mentioned as ebs-sc.

Before applying the YAML, let’s take a look at the available EBS volumes in the AWS console. We see three volumes; one for each of the nodes.

Let’s apply the PVC yaml now. We get the status and we can see that it got in the Bound state. Let’s take a look at the AWS console; let me refresh. And we can see now that a new Volume has been created successfully.

Once the PVC is ready it can be used in the pod just like we have used in the earlier case.

That’s all about this video. We saw that Dynamic volume provisioning allows storage volumes to be created on-demand. Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create PersistentVolume objects to represent them in Kubernetes. Thank you.

====