Kubernetes Storage: Provision EKS with EBS CSI driver through Terraform. EmptyDir, Persistent volume, Storageclass PVC, Docker volume, mount

Check GitHub for helpful DevOps tools:

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

Ask Personal AI Document assistant to learn interactively (FASTER)!

1 Download PDF

2 Go to website

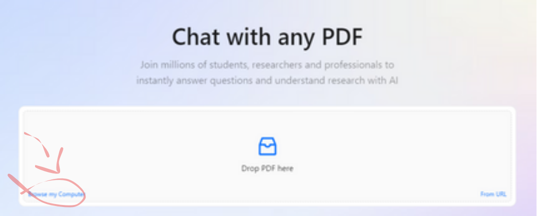
3 Browse file

4 Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesStorage.pdf

Click there to go to ChatPdf website 2

3



Chat with any PDF

Join millions of students, researchers and professionals to instantly answer questions and understand research with AI

Drop PDF here

Browse my Computer

From URL

Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Storage: Docker volume, mount and persistent storage

1) Build Image

Get app source code

```
git clone https://github.com/docker/getting-started-app.git
cd getting-started-app/
```

Create Dockerfile for app

```
touch Dockerfile
```

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Build docker image

```
docker build -t day02-todo .
```

2) Nature of layerd architecture

Docker images are built in layers, with each Dockerfile instruction creating a cached layer. During a rebuild, only the modified layers are rebuilt, while unchanged layers are reused from the cache, significantly reducing build time.

3) Container, multiple containers and Image

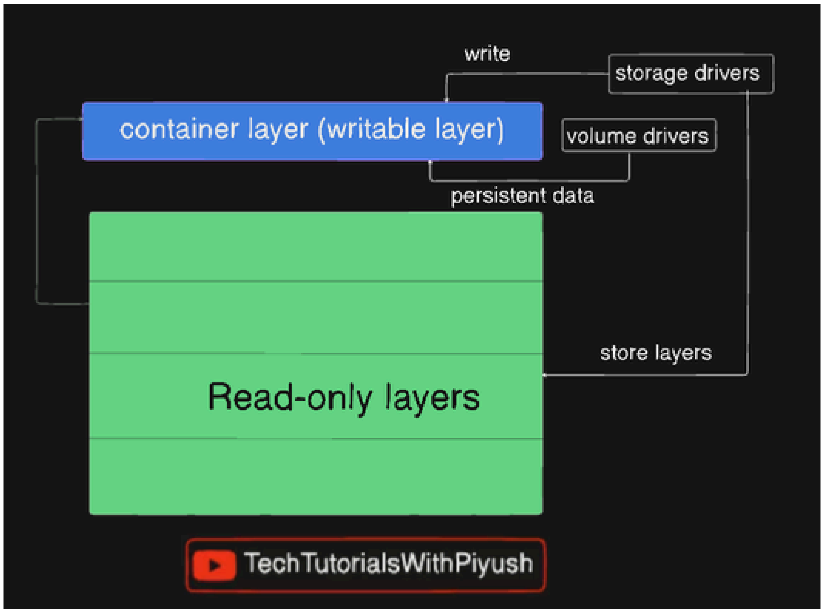
Image layers are read only. Each container gets its own unique writable layer on top of the shared image layers. If a container modifies a file from shared image layers, the **storage driver** copies it to the container's writable layer, where changes are made. Other containers using the same image remain unaffected, and unmodified files continue to be shared from image layers.

4) storage driver

A storage driver manages container file systems, handling data storage, copy-on-write for changes, and layer sharing to save space. Popular drivers like Overlay2 efficiently combine layers with minimal overhead.

5) volume drivers

A volume driver manages persistent storage independent of container layers, allowing data to persist across restarts or be shared between containers. Volumes offer durability, high I/O performance, and sharing across containers. Types include local (host-based) and remote (e.g., NFS, EFS)



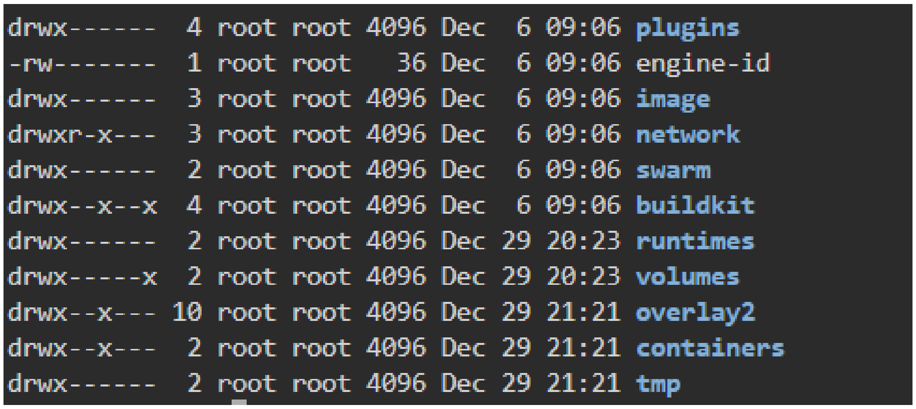
4) Volumes

create volume

```
sudo docker volume create data_vol
```

get into directory, where docker store its files

```
cd /var/lib/docker
ls -lrt
```



```
drwx------  4 root root 4096 Dec  6 09:06 plugins
-rw-------  1 root root   36 Dec  6 09:06 engine-id
drwx------  3 root root 4096 Dec  6 09:06 image
drwxr-x---  3 root root 4096 Dec  6 09:06 network
drwx------  2 root root 4096 Dec  6 09:06 swarm
drwx--x--x  4 root root 4096 Dec  6 09:06 buildkit
drwx------  2 root root 4096 Dec 29 20:23 runtimes
drwx-----x  2 root root 4096 Dec 29 20:23 volumes
drwx--x--- 10 root root 4096 Dec 29 21:21 overlay2
drwx--x---  2 root root 4096 Dec 29 21:21 containers
drwx------  2 root root 4096 Dec 29 21:21 tmp
```

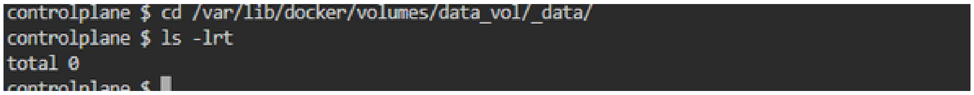
/var/lib/docker/overlay2: This directory stores the image layers and writable layers for containers when using the overlay2 storage driver, enabling efficient file system layering.

/var/lib/docker/containers: This directory contains data specific to each container, including configuration files, logs, and the container's unique writable layer.

/var/lib/docker/volumes: This directory holds persistent volumes, storing data that remains intact across container restarts and is independent of the container's lifecycle.

Check data_volume we created. It should have no data inside.

```
cd /var/lib/docker/volumes/data_vol/_data/

ls -lrt
```



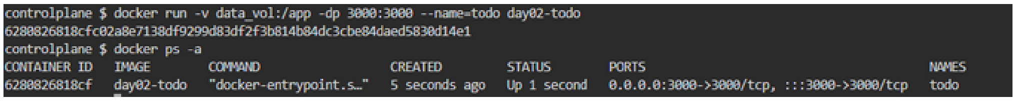
```
controlplane $ cd /var/lib/docker/volumes/data_vol/_data/
controlplane $ ls -lrt
total 0
controlplane $ 
```

create container and mount to it volume

```
docker run -v data_vol:/app -dp 3000:3000 --name=todo day02-todo
```

Check docker container id

```
docker ps
```



```
controlplane $ docker run -v data_vol:/app -dp 3000:3000 --name=todo day02-todo
6280826818cfc02a8e7138df9299d83df2f3b814b84dc3cbe84daed5830d14e1
controlplane $ docker ps -a
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS         PORTS                                       NAMES
6280826818cf   day02-todo   "docker-entrypoint.s…"   5 seconds ago   Up 1 second    0.0.0.0:3000->3000/tcp, :::3000->3000/tcp   todo
```

exec into it and create new dir

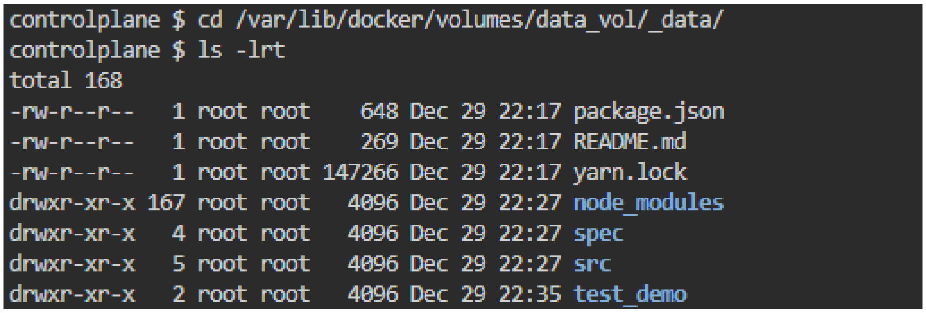
```
sudo docker exec -it 6280826818cf sh

mkdir test_demo
```

exit and delete container

```
exit

docker stop 6280826818cf

docker rm 6280826818cf
```

Check volume directory. It should have data, from deleted container.

```
cd /var/lib/docker/volumes/data_vol/_data/

ls -lrt
```

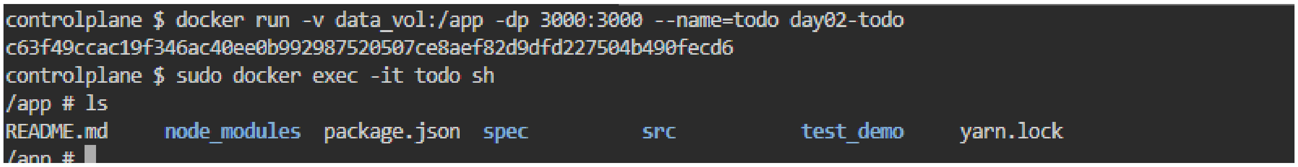


```
controlplane $ cd /var/lib/docker/volumes/data_vol/_data/
controlplane $ ls -lrt
total 168
-rw-r--r--   1 root root    648 Dec 29 22:17 package.json
-rw-r--r--   1 root root    269 Dec 29 22:17 README.md
-rw-r--r--   1 root root 147266 Dec 29 22:17 yarn.lock
drwxr-xr-x 167 root root   4096 Dec 29 22:27 node_modules
drwxr-xr-x   4 root root   4096 Dec 29 22:27 spec
drwxr-xr-x   5 root root   4096 Dec 29 22:27 src
drwxr-xr-x   2 root root   4096 Dec 29 22:35 test_demo
```

create new container and mount volume:

```
docker run -v data_vol:/app -dp 3000:3000 --name=todo day02-todo
```

Now enter container. It should have all data from volume dat_vol available.



```
controlplane $ docker run -v data_vol:/app -dp 3000:3000 --name=todo day02-todo
c63f49ccac19f346ac40ee0b992987520507ce8aef82d9dfd227504b490fecd6
controlplane $ sudo docker exec -it todo sh
/app # ls
README.md     node_modules  package.json  spec          src           test_demo     yarn.lock
/app # 
```

delete container

```
docker stop todo

docker rm todo
```

5) Bind mounts

A bind mount in Docker allows you to mount a file or directory from the host system into a container, enabling real-time synchronization and sharing of data between the host and container.

create file in host system

```
mkdir Test_Dir
cd Test_Dir
touch file_test_1.txt
pwd
```

create docker container:

```
docker run  -v /root/Test_Dir:/app/Test_Dir -dp 3000:3000 --name=todo day02-todo
```

Enter container and create file in Test_Dir. Created file should be created at host aswell

```
sudo docker exec -it todo sh
cd Test_Dir
touch file_test_2.txt
exit
```

Create file in host and enter container. Created file should be visible in container.

Kubernetes Storage: EmptyDir, Persistent volume and Persistent volume claim

1) Install & configure EKS cluster

Get EKS cluster terraform directory:

```
git clone https://github.com/MichaelRobotics/Kubernetes.git
cd Kubernetes/Storage/EKSwithCSI
```

Install terraform, kubectl and helm. Create AWS account and configure aws CLI on your machine. Karpenter demo explains it clearly:

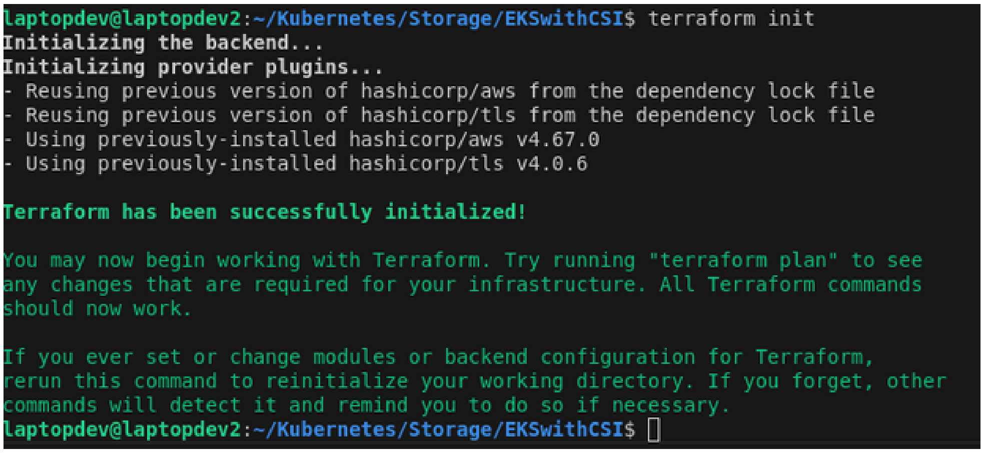
Getting Started with Karpenter

Set up a cluster and add Karpenter

https://karpenter.sh/docs/

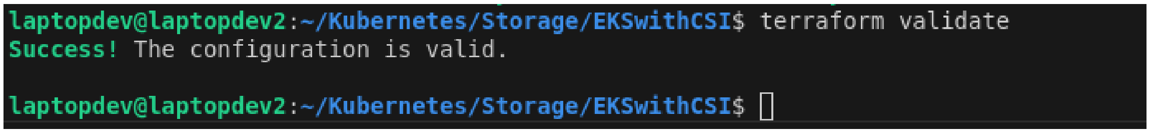
Init terraform, validate tf files, check plan and apply configuration.

```
terraform init
```



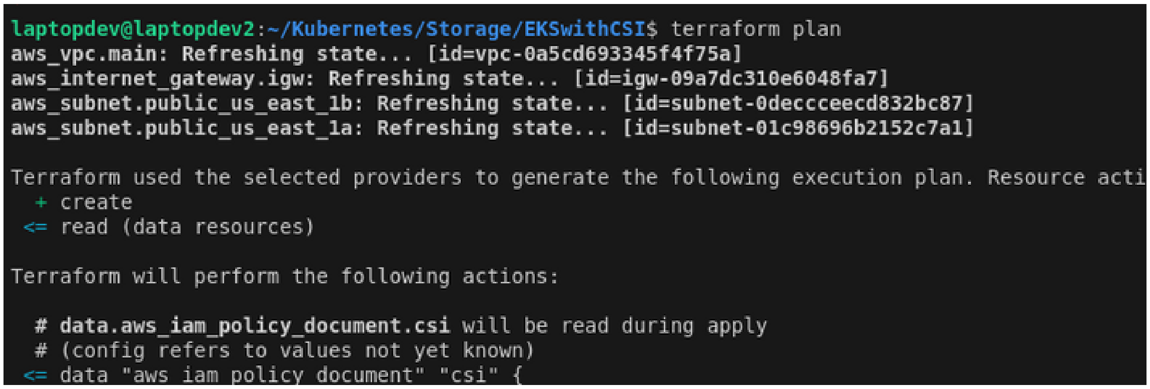
```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/tls from the dependency lock file
- Using previously-installed hashicorp/aws v4.67.0
- Using previously-installed hashicorp/tls v4.0.6

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI$ 
```

```
terraform validate
```



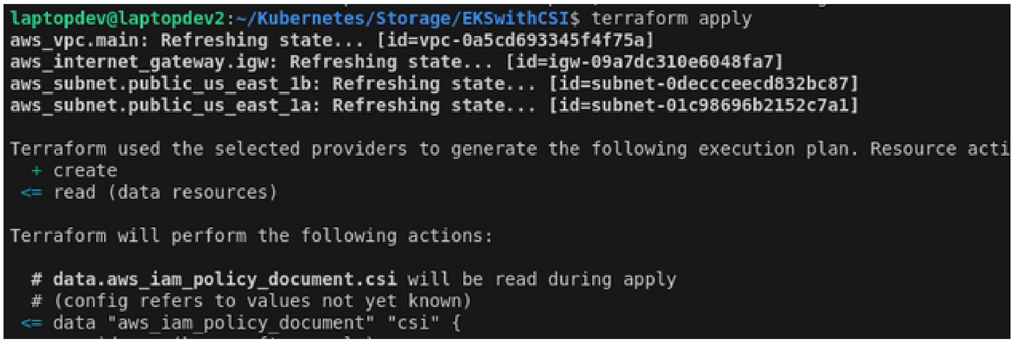
```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI$ terraform validate
Success! The configuration is valid.

laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI$ 
```

```
terraform plan
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI$ terraform plan
aws_vpc.main: Refreshing state... [id=vpc-0a5cd693345f4f75a]
aws_internet_gateway.igw: Refreshing state... [id=igw-09a7dc310e6048fa7]
aws_subnet.public_us_east_1b: Refreshing state... [id=subnet-0deccceecd832bc87]
aws_subnet.public_us_east_1a: Refreshing state... [id=subnet-01c98696b2152c7a1]

Terraform used the selected providers to generate the following execution plan. Resource acti
  + create
 <= read (data resources)

Terraform will perform the following actions:

  # data.aws_iam_policy_document.csi will be read during apply
  # (config refers to values not yet known)
 <= data "aws_iam_policy_document" "csi" {
```

```
terraform apply
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI$ terraform apply
aws_vpc.main: Refreshing state... [id=vpc-0a5cd693345f4f75a]
aws_internet_gateway.igw: Refreshing state... [id=igw-09a7dc310e6048fa7]
aws_subnet.public_us_east_1b: Refreshing state... [id=subnet-0deccceecd832bc87]
aws_subnet.public_us_east_1a: Refreshing state... [id=subnet-01c98696b2152c7a1]

Terraform used the selected providers to generate the following execution plan. Resource acti
  + create
 <= read (data resources)

Terraform will perform the following actions:

  # data.aws_iam_policy_document.csi will be read during apply
  # (config refers to values not yet known)
 <= data "aws_iam_policy_document" "csi" {
```

After EKS creation, switch kubectl context to EKS cluster:

```
aws eks --region us-east-1 update-kubeconfig --name demo
```

Get directory with kubernetes yamls

```
git clone https://github.com/vfarcic/kubernetes-demo
cd kubernetes-demo
```

Install traeffik

```
helm repo add traefik \
https://helm.traefik.io/traefik
```

```
helm repo update
helm upgrade --install traefik traefik/traefik \
--namespace traefik --create-namespace --wait
```

Install yq

```
curl -L https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64 -o
/usr/local/bin/yq
chmod +x /usr/local/bin/yq
```

Get LoadBalancer ip:

```
export INGRESS_HOSTNAME=$(kubectl --namespace traefik \
get svc traefik \
--output jsonpath="{.status.loadBalancer.ingress[0].hostname}")
export INGRESS_HOST=$(dig +short $INGRESS_HOSTNAME)
```

Show variable:

```
echo $INGRESS_HOST
```

Pass one of ip’s to variable:

```
export INGRESS_HOST=<IP>
```

Modify ingress.yaml host with right ip:

```
yq --inplace \
".spec.rules[0].host = \"silly-demo.$INGRESS_HOST.nip.io\"" \
service/ingress.yaml
```

2) EmptyDir

EmptyDir is an ephemeral volume type that exists only as long as the Pod does. It is created when the Pod starts and deleted when the Pod is removed, not when containers crash. If a container restarts within the same Pod, the emptyDir data remains.

EmptyDir is commonly used in local development, where persistent storage is not needed, and a simpler setup is sufficient for testing applications.

Download repo

```
git clone https://github.com/vfarcic/kubernetes-demo
cd kubernetes-demo
```

Check empty-dir.yml structure:

```
cat volume/empty-dir.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: silly-demo
 [...]
          env:
            - name: DB
              value: fs
          volumeMounts:
            - mountPath: /cache
              name: silly-cache
      volumes:
        - name: silly-cache
          emptyDir: {}
```

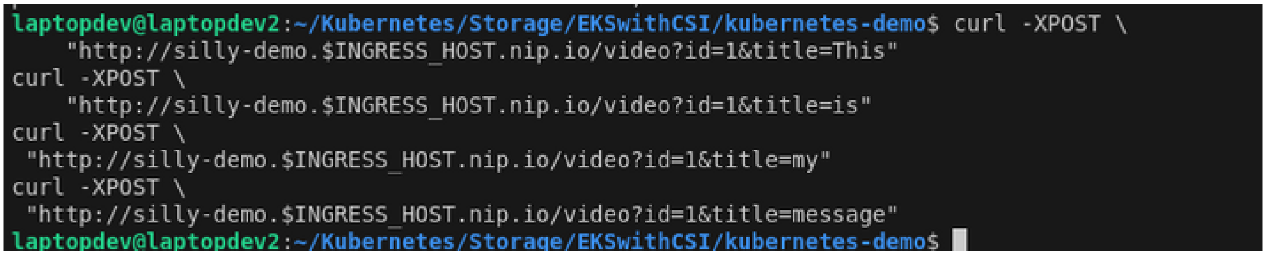
Pods, defined through the Deployment, can include volumes that are directories with a filesystem, either local or remote, permanent or ephemeral. Here, we're using an emptyDir volume, and the container mounts it as /cache, making the volume's files accessible at that path.

Deploy app:

```
kubectl apply -f service/base.yaml
kubectl apply -f volume/empty-dir.yaml
kubectl apply -f service/ingress.yaml
```

Next, we’ll send a POST request to the application to store data on the file system, allowing us to observe the mounted volume

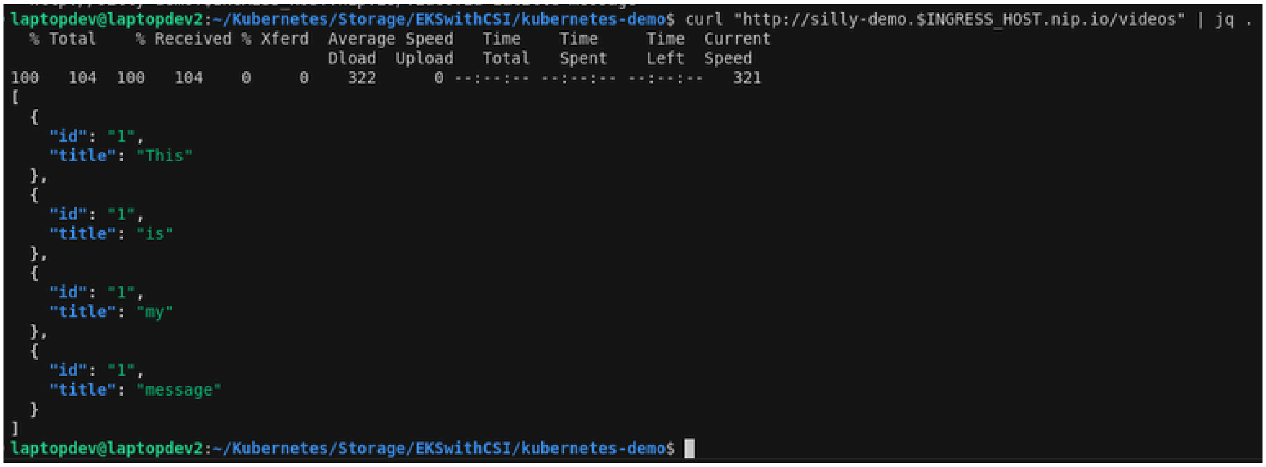
```
curl -XPOST \
   "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=This"
curl -XPOST \
   "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=is"
curl -XPOST \
 "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=my"
curl -XPOST \
 "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=message"
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ curl -XPOST \
    "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=This"
curl -XPOST \
    "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=is"
curl -XPOST \
 "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=my"
curl -XPOST \
 "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=message"
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$
```

Next, we'll send a request to the app which requires it to fetch data from the volume and send it back to us.

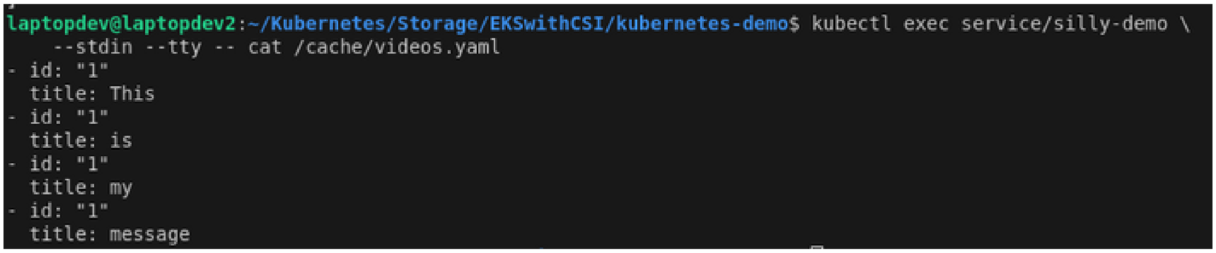
```
curl "http://silly-demo.$INGRESS_HOST.nip.io/videos" | jq .
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ curl "http://silly-demo.$INGRESS_HOST.nip.io/videos" | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   104  100   104    0     0    322      0 --:--:-- --:--:-- --:--:--   321
[
  {
    "id": "1",
    "title": "This"
  },
  {
    "id": "1",
    "title": "is"
  },
  {
    "id": "1",
    "title": "my"
  },
  {
    "id": "1",
    "title": "message"
  }
]
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$
```

The volume appears to work, with the app reading and writing files to it. To confirm, exec into the silly-demo container and list files in the mounted /cache directory.

```
kubectl exec service/silly-demo \

  --stdin --tty -- cat /cache/videos.yaml
```

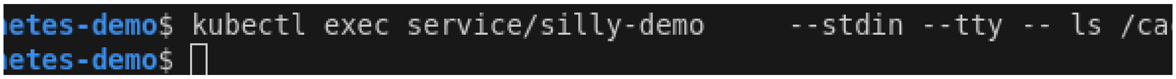


```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl exec service/silly-demo \
    --stdin --tty -- cat /cache/videos.yaml
- id: "1"
  title: This
- id: "1"
  title: is
- id: "1"
  title: my
- id: "1"
  title: message
```

Ephemeral volumes last only as long as the Pod. We can show this by deleting the Pod.

```
kubectl delete pod \

  --selector app.kubernetes.io/name=silly-demo
```

Since the Pod is managed by a ReplicaSet and Deployment, the ReplicaSet Controller will create a new Pod. Try check what is in /cache directory on pod.



```
etes-demo$ kubectl exec service/silly-demo     --stdin --tty -- ls /ca
etes-demo$ 
```

All data is gone. After pod deletion, cache got cleared.

3) Persitent Volume, Persistent volume claims

What is a PV?

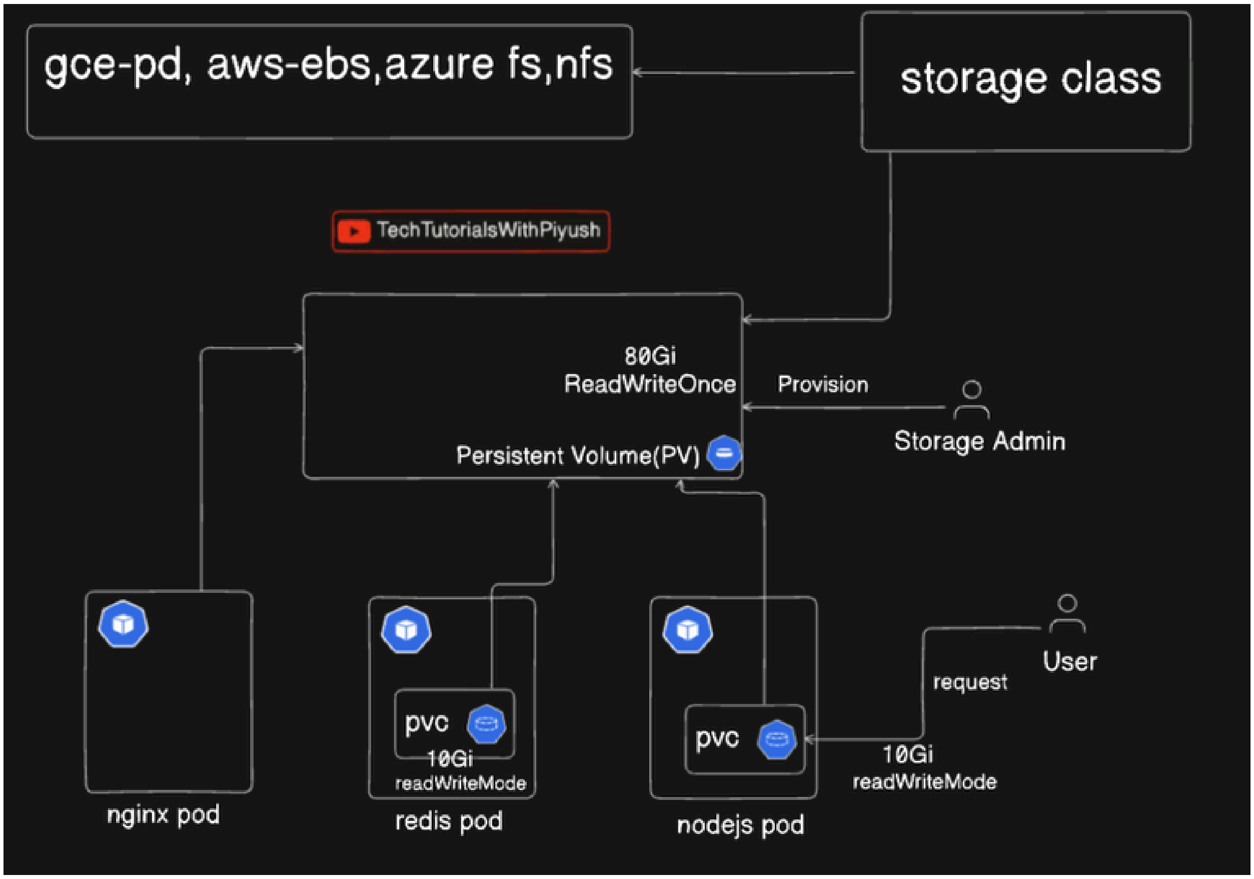
- A Persistent Volume (PV) is like a storage drive that Kubernetes can use.
- It’s created by the cluster admin and can be backed by things like cloud storage, network drives, or local disks.
- Think of it as the "storage resource" available to the cluster.

PVs have properties like capacity, access modes, and reclaim policies.

- Capacity: Defines the size of the volume.
- Access Modes: Specifies how the volume can be mounted (e.g., ReadWriteOnce, ReadOnlyMany, ReadWriteMany).
- Reclaim Policy: Determines what happens to the PV when it is released (e.g., Retain, Recycle, Delete).

What is a PVC?

- A Persistent Volume Claim (PVC) is a request for storage made by an application (or user).
- It’s like saying, "I need 10GB of space that I can read and write to."
- Kubernetes finds a matching PV and "reserves" it for the PVC.



How They Work Together?

- Admin sets up PVs (or configures dynamic provisioning with StorageClasses).
- User creates a PVC, specifying the size and access type (e.g., read/write).
- Kubernetes finds a PV that matches the PVC’s request and binds them.
- Pods use the PVC to mount the storage and access data.

Create PV on local machine

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath-home
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /home/username  # Replace 'username' with the actual user's home directory path
    type: Directory
```

make PVC claim:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-hostpath-home
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeName: pv-hostpath-home
```

Use PVC in deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app-container
          image: nginx:latest  # Replace with your application image
          volumeMounts:
            - mountPath: /app/data  # Path inside the container
              name: hostpath-volume
      volumes:
        - name: hostpath-volume
          persistentVolu
```

Nevertheless, using a PersistentVolume with hostPath ties your application to specific nodes, limiting portability, scalability, and high availability while exposing the host to security risks. It lacks dynamic provisioning and is less robust than alternatives like cloud-backed storage, making it suitable mainly for development or specialized use cases.

Kubernetes Storage: CSI Drivers and Storage Classes

1) CSI introduction

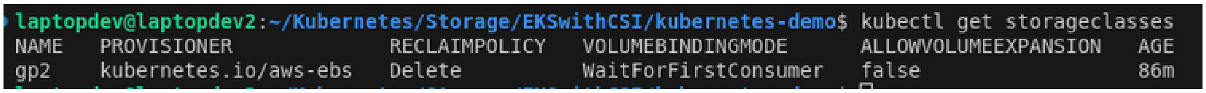
Kubernetes does not support specific storage options due to the vast variety available, which depends on the environment. AWS, Azure, Google Cloud, and on-premises datacenters each offer different storage solutions, along with universal options that differ from traditional block storage or NFS.

Rather than embedding specific storage solutions, Kubernetes defines the Container Storage Interface (CSI)—a standard that lets storage vendors create plugins to integrate with Kubernetes. CSI serves as a bridge, providing a unified interface for Kubernetes to communicate with storage systems and enabling vendors to expose their offerings to Kubernetes users.

For end users, this simplifies storage management. Most Kubernetes setups include at least one CSI driver, and additional drivers can be installed if needed. The key concept for users is storage classes, which represent the types of storage available in the cluster, typically defined by the CSI drivers.

lets check availavble storage classes:

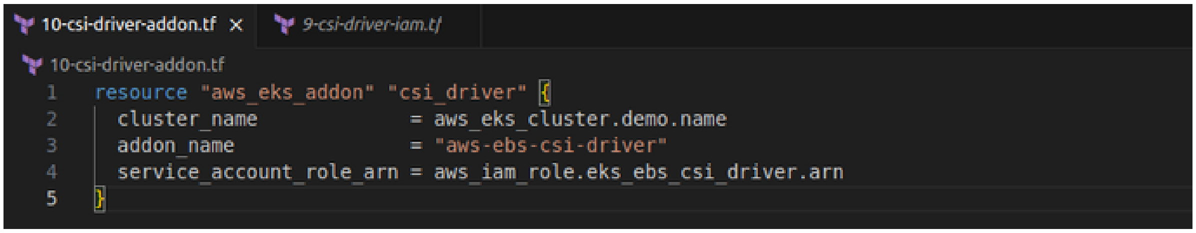
kubectl get storageclasses



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl get storageclasses
NAME   PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
gp2    kubernetes.io/aws-ebs   Delete          WaitForFirstConsumer   false                  86m
```

The PROVISIONER column indicates that all StorageClasses in your case use kubernetes.io/aws-ebs, meaning AWS EBS volumes will be provisioned for storage requests. The VOLUMEBINDINGMODE set to WaitForFirstConsumer ensures that the volume is only created when a pod using the associated PVC is scheduled, guaranteeing that the volume is provisioned in the same availability zone as the pod.

CSI provider was defined in terraform code in 10-csi-driver-addon file:



```
resource "aws_eks_addon" "csi_driver" {
  cluster_name             = aws_eks_cluster.demo.name
  addon_name               = "aws-ebs-csi-driver"
  service_account_role_arn = aws_iam_role.eks_ebs_csi_driver.arn
}
```

To provision storage on cloud, your cluster needs CCM cloud controller manager. EKS have its functionality build into control plane node.

StorageClass in EKS is created by default so you dont need to specify it. Check its structure:

```
kubectl get storageclass gp2 -o yaml
```

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"storage.k8s.io/v1","kind":"StorageClass","metadata":{"annotations":{},"name":"gp2"},"parameters":{"fsType":"ext4","type":"gp2"},"provisioner":"kubernetes.io/aws-ebs","volumeBindingMode":"WaitForFirstConsumer"}
  creationTimestamp: "2024-12-31T15:10:18Z"
  name: gp2
  resourceVersion: "273"
  uid: 08f427f9-2a63-48fa-8d9c-adbd7a5f9295
parameters:
  fsType: ext4
  type: gp2
provisioner: kubernetes.io/aws-ebs
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

This StorageClass defines storage provisioning for AWS EBS volumes of type gp2 with an ext4 file system. Using the kubernetes.io/aws-ebs provisioner, it creates volumes only when a pod is scheduled (WaitForFirstConsumer) and automatically deletes the volume when the PVC is removed (Delete reclaim policy).

2) Create storage with CSI

Look at PVC claim:

```
cat volume/persistent-volume-claim.yaml

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: silly-demo
  labels:
    app.kubernetes.io/name: silly-demo
spec:
  storageClassName: gp2
  resources:
    requests:
      storage: 1Gi
  accessModes:
  - ReadWriteOnce
```

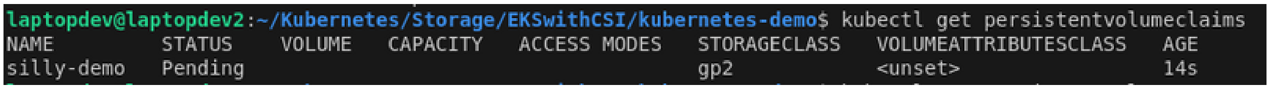
The PersistentVolumeClaim (PVC) requests storage using the CSI driver, specifying the gp2 StorageClass. If storageClassName is omitted, Kubernetes uses the default StorageClass. This PVC requests 1Gi of storage with ReadWriteOnce access mode for both read and write operations.

apply

```
kubectl apply \
   --filename volume/persistent-volume-claim.yaml
```

take a look at the persistentvolumeclaims

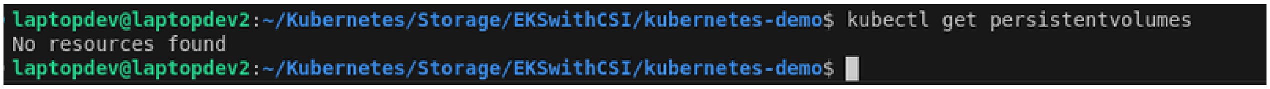
```
kubectl  get persistentvolumeclaims
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl get persistentvolumeclaims
NAME         STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
silly-demo   Pending                                      gp2            <unset>                 14s
```

The claim's status is Pending because the WaitForFirstConsumer mode delays volume creation until it's attached to a Pod. Though storage is claimed, no volume is created yet. We can confirm this by listing all PersistentVolumes.

```
kubectl get persistentvolumes
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl get persistentvolumes
No resources found
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ 
```

There is no volume created yet. If you had chosen a Storage Class with the volume binding mode set to Immediate, the volume would be created even without a Pod consuming it. Next, let’s update the Deployment to use the PVC we just created instead of the current emptyDir volume.

```
cat volume/persistent-volume.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: silly-demo
  ...
spec:
  ...
  template:
    ...
    spec:
          ...
          volumeMounts:
            - mountPath: /cache
              name: silly-cache
      volumes:
        - name: silly-cache
          persistentVolumeClaim:
            claimName: silly-demo
```

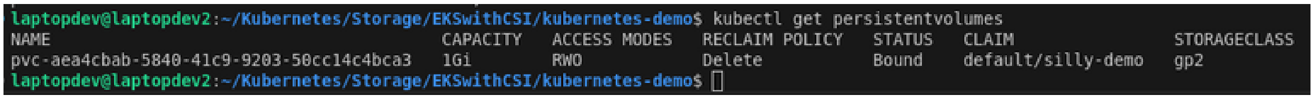
The only change is in the volumes section, where we specify a persistentVolumeClaim with claimName: silly-demo instead of emptyDir. Everything else, including volumeMounts, remains the same.

apply

```
kubectl apply \
   --filename volume/persistent-volume.yaml
```

take look at the persistentvolumes.

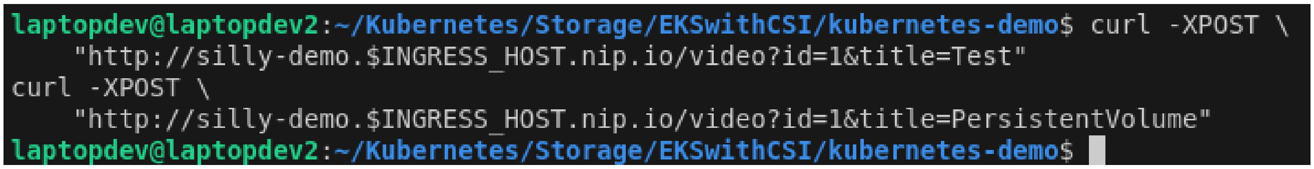
```
kubectl get persistentvolumes
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl get persistentvolumes
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                STORAGECLASS
pvc-aea4cbab-5840-41c9-9203-50cc14c4bca3   1Gi        RWO            Delete           Bound    default/silly-demo   gp2
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ 
```

The volume is now created since the Storage Class was waiting for a consumer (the Pod). To confirm the storage is persistent, we'll send a POST request to the app, which should store data in the volume, and verify using the same commands as before.

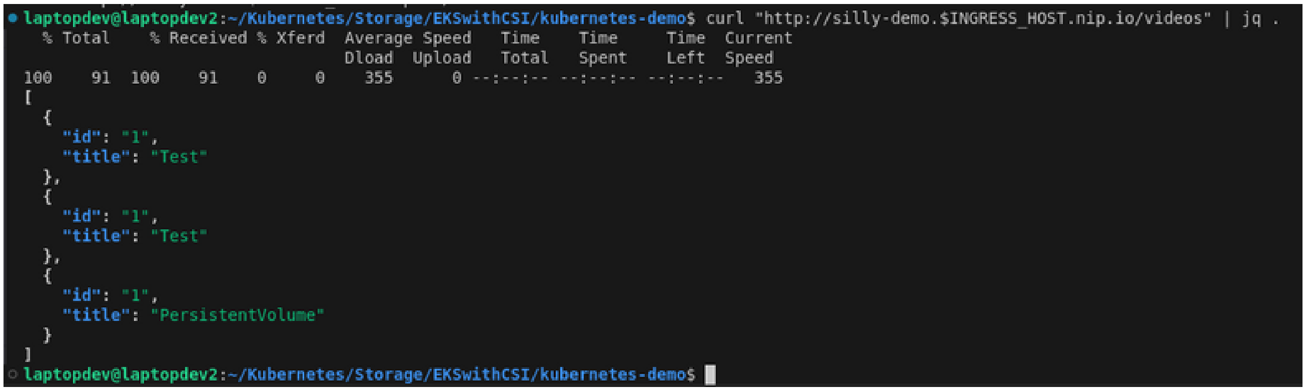
```
curl -XPOST \
   "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=Test"
curl -XPOST \
   "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=PersistentVolume"
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ curl -XPOST \
    "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=Test"
curl -XPOST \
    "http://silly-demo.$INGRESS_HOST.nip.io/video?id=1&title=PersistentVolume"
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ 
```

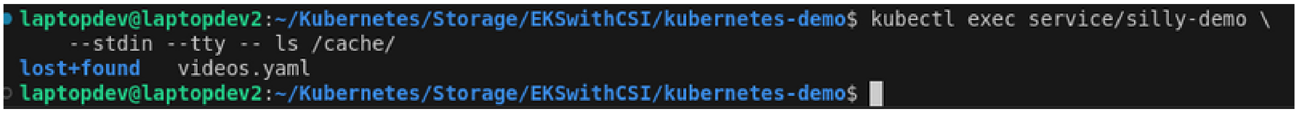
GET request to the application

```
curl "http://silly-demo.$INGRESS_HOST.nip.io/videos" | jq .
```



```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ curl "http://silly-demo.$INGRESS_HOST.nip.io/videos" | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    91  100    91    0     0    355      0 --:--:-- --:--:-- --:--:--   355
[
  {
    "id": "1",
    "title": "Test"
  },
  {
    "id": "1",
    "title": "Test"
  },
  {
    "id": "1",
    "title": "PersistentVolume"
  }
]
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$
```

exec into the container and list (ls) all the files in the /cache directory.



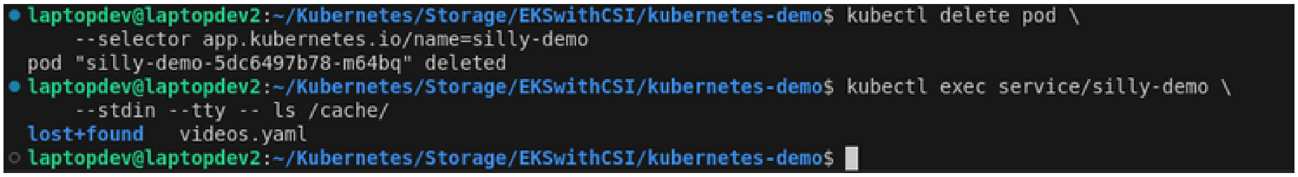
```
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl exec service/silly-demo \
    --stdin --tty -- ls /cache/
lost+found   videos.yaml
laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$
```

Container stores all data in videos.yaml. Now check what will happen if we delete pod and let ReplicaSet create new one:

```
kubectl delete pod \
  --selector app.kubernetes.io/name=silly-demo
```

again exec into container:

```
kubectl exec service/silly-demo \
  --stdin --tty -- ls /cache/
```



```
● laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl delete pod \
      --selector app.kubernetes.io/name=silly-demo
  pod "silly-demo-5dc6497b78-m64bq" deleted
● laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ kubectl exec service/silly-demo \
      --stdin --tty -- ls /cache/
  lost+found   videos.yaml
○ laptopdev@laptopdev2:~/Kubernetes/Storage/EKSwithCSI/kubernetes-demo$ 
```

The videos.yaml file is still there, and its content remains unchanged. Data was persisted on external storage, and the new Pod automatically attached to it.

There is possiblity to create pvc resource withous specifying storage class. If any available storageclass have default flag, pvc claim will automatically chose those if none is specified in yaml:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: silly-demo
  labels:
    app.kubernetes.io/name: silly-demo
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
    - ReadWriteOnce
```

Unfortunately, AWS doesn't set a default StorageClass, so you'll need to modify one with kubectl edit or skip the commands. Alternatively, you can create a PVC without specifying a StorageClass, and Kubernetes will use the default if it's set.

3) Delete resources

```
terraform destroy
```

common troubleshooting

1) Persistent Volume Not Binding to PVC

Cause: The PersistentVolume (PV) is not matching the PersistentVolumeClaim (PVC) due to mismatched storage class, capacity, or access modes.
Solution: Verify the PVC and PV configuration, ensuring that the storage class, capacity, and access modes align. Use kubectl describe pvc <pvc-name> and kubectl describe pv <pv-name> to check for issues.

2) EBS CSI Driver Not Provisioning Volume

Cause: Incorrect configuration of the EBS CSI driver, IAM permissions, or issues with the Terraform provisioning script.
Solution: Review the Terraform configuration to ensure the EBS CSI driver is properly configured. Ensure that the required IAM roles and policies are set correctly. Use kubectl logs <csi-driver-pod-name> for any error messages from the CSI driver.

3) PVC Stuck in Pending State

Cause: The storage class may not be available, or there may be insufficient resources for volume provisioning.
Solution: Check if the correct storage class is applied by running kubectl get pvc <pvc-name> -o yaml and verifying the storage class field. Ensure the underlying storage backend (EBS) has available resources.

4) Check my Kubernetes Troubleshooting series:

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!