**🔐 Kubernetes Secrets – Detailed Lab Guide**

**Objective**

* Create an opaque Kubernetes Secret containing sensitive data like username and password.
* Consume the secret in a pod using:
  1. **Volume Mount** (file-based secret consumption)
  2. **Environment Variables** (env-based secret injection)
* Validate the secret injection in both pods.

**🔧 Prerequisites**

Ensure the following:

* A running Kubernetes cluster (minikube/kubeadm cluster).
* kubectl is configured and pointing to the correct context.
* A namespace called raman exists:

bash

CopyEdit

kubectl create namespace raman

**📄 Step 1: Create the Kubernetes Secret**

**✍️ secret.yaml**

yaml

CopyEdit

apiVersion: v1

kind: Secret

metadata:

name: my-secrets

namespace: raman

type: Opaque

data:

username: cmFtYW5raGFubmEK # base64 encoded: ramankhanna

password: cmFtYW5raGFubmExMjMK # base64 encoded: ramankhanna123

**🔃 Explanation:**

* Opaque: The generic type for arbitrary user-defined data.
* data: Must be base64 encoded. Use the following to encode:

bash

CopyEdit

echo -n 'ramankhanna' | base64

echo -n 'ramankhanna123' | base64

**🚀 Create the secret:**

bash

CopyEdit

kubectl apply -f secret.yaml

**📄 Step 2: Create Pods That Use the Secret**

**📦 Pod 1 – Secret via Volume Mount**

yaml

CopyEdit

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod1

labels:

app: myapp

namespace: raman

spec:

containers:

- name: httpd-container

image: httpd

volumeMounts:

- name: crds

mountPath: /tmp/creds

readOnly: true

volumes:

- name: crds

secret:

secretName: my-secrets

**🔃 Explanation:**

* The secret will be **mounted at /tmp/creds**.
* Two files will be created inside the container:
  + /tmp/creds/username
  + /tmp/creds/password
* File content = decoded base64 values from the secret.

**📦 Pod 2 – Secret via Environment Variables**

yaml

CopyEdit

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod2

labels:

app: myapp

type: front-end

namespace: raman

spec:

containers:

- name: httpd-container

image: httpd

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: my-secrets

key: username

- name: SECRET\_PASSWD

valueFrom:

secretKeyRef:

name: my-secrets

key: password

**🔃 Explanation:**

* The environment variables in the container will have:
  + SECRET\_USERNAME=ramankhanna
  + SECRET\_PASSWD=ramankhanna123
* You can access them using env or echo $SECRET\_USERNAME from within the container.

**✅ Step 3: Apply the Configuration**

Assuming you've saved both pod specs (combined) in secretpod1.yml, run:

bash

CopyEdit

kubectl apply -f secretpod1.yml

Check pod status:

bash

CopyEdit

kubectl get pods -n raman

**🔍 Step 4: Validate the Secret Injection**

**🧪 Check Pod 1 (file mount):**

bash

CopyEdit

kubectl exec -it myapp-pod1 -n raman -- cat /tmp/creds/username

kubectl exec -it myapp-pod1 -n raman -- cat /tmp/creds/password

Expected Output:

nginx

CopyEdit

ramankhanna

ramankhanna123

**🧪 Check Pod 2 (env vars):**

Since the httpd image doesn’t have a shell, let's exec a shell into it:

bash

CopyEdit

kubectl exec -it myapp-pod2 -n raman -- /bin/bash

If bash is not present, use sh.

Inside the pod, check:

bash

CopyEdit

echo $SECRET\_USERNAME

echo $SECRET\_PASSWD

Expected Output:

nginx

CopyEdit

ramankhanna

ramankhanna123

**🧹 Cleanup (Optional)**

bash

CopyEdit

kubectl delete -f secret.yaml

kubectl delete -f secretpod1.yml

**📌 Key Concepts Recap**

| **Concept** | **Volume Mount** | **Env Variable** |
| --- | --- | --- |
| Access Method | Mounted as files | Injected as env vars |
| Flexibility | Better for apps reading from files | Simpler for env-based configs |
| Use Case | Certificates, config files | App configs, credentials |
| Visibility | Visible in file system | Visible via env |
| Security | Slightly better (fs permission control) | Visible in process env (ps risk) |

**🧾 Kubernetes ConfigMap Lab Guide: Serve Dynamic HTML Using NGINX and ConfigMaps**

**🎯 Objective**

* Create two ConfigMaps (prod-cmap and dev-cmap) containing HTML content.
* Mount the ConfigMap to a pod running **NGINX** so that it serves the content over port **80**.
* Swap between different HTML content by switching ConfigMaps.
* Expose the pod using a **NodePort** service.

**🛠️ Pre-requisites**

* Kubernetes cluster up and running (minikube, kubeadm, etc.).
* kubectl CLI configured properly.
* Namespace raman exists:

bash

CopyEdit

kubectl create namespace raman

**🔧 Step 1: Create HTML-based ConfigMaps**

We define two separate HTML pages embedded in a ConfigMap — one for production and another for development.

**📄 cm.yml**

yaml

CopyEdit

apiVersion: v1

kind: ConfigMap

metadata:

name: prod-cmap

namespace: raman

data:

prod.html: |

<!DOCTYPE html>

<html>

<head>

<title>Production Page</title>

</head>

<body>

<h1>Welcome to Production</h1>

</body>

</html>

---

apiVersion: v1

kind: ConfigMap

metadata:

name: dev-cmap

namespace: raman

data:

dev.html: |

<!DOCTYPE html>

<html>

<head>

<title>Dev Page</title>

</head>

<body>

<h1>Welcome to Dev</h1>

</body>

</html>

**🚀 Apply the ConfigMaps:**

bash

CopyEdit

kubectl apply -f cm.yml

**🔍 Verify:**

bash

CopyEdit

kubectl get configmaps -n raman

kubectl describe configmap prod-cmap -n raman

kubectl describe configmap dev-cmap -n raman

**🚀 Step 2: Create an NGINX Pod Using ConfigMap**

This pod uses a ConfigMap to replace NGINX’s default index page.

**📄 cmpod.yml**

yaml

CopyEdit

apiVersion: v1

kind: Pod

metadata:

name: prod-nginx

labels:

app: prod-nginx

namespace: raman

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumeMounts:

- name: config-volume

mountPath: /usr/share/nginx/html

volumes:

- name: config-volume

configMap:

name: dev-cmap # change to prod-cmap for production

items:

- key: dev.html # change to prod.html if using prod-cmap

path: index.html # NGINX looks for index.html by default

**📝 Notes:**

* The key (dev.html/prod.html) is renamed to index.html in the container, so NGINX can serve it by default from /usr/share/nginx/html/index.html.

**🔄 Deploy the Pod:**

bash

CopyEdit

kubectl apply -f cmpod.yml

**✅ Check Pod Status:**

bash

CopyEdit

kubectl get pods -n raman

**🌐 Step 3: Expose NGINX Pod as a Service**

Expose the pod using **NodePort** to access it externally:

bash

CopyEdit

kubectl expose pod prod-nginx \

--type=NodePort \

--name=cmsvc \

--port=80 \

--target-port=80 \

-n raman

**🔍 Get the service and NodePort:**

bash

CopyEdit

kubectl get svc -n raman

Look for output like:

nginx

CopyEdit

cmsvc NodePort 10.99.224.57 <none> 80:30123/TCP

To access:

* http://<NODE-IP>:<NODEPORT>
* Use minikube service cmsvc -n raman --url (for minikube)

**🧪 Step 4: Validate the Setup**

**🔎 Exec into Pod and Check HTML Content:**

bash

CopyEdit

kubectl exec -it prod-nginx -n raman -- cat /usr/share/nginx/html/index.html

Expected Output:

html

CopyEdit

<!DOCTYPE html>

<html>

<head>

<title>Dev Page</title>

</head>

<body>

<h1>Welcome to Dev</h1>

</body>

</html>

If using prod-cmap, it should say "Welcome to Production".

**🔁 Step 5: Switch Between Dev and Prod**

**📌 To switch:**

* Edit cmpod.yml:
  + Change name: dev-cmap → prod-cmap
  + Change key: dev.html → prod.html
* Re-apply:

bash

CopyEdit

kubectl delete pod prod-nginx -n raman

kubectl apply -f cmpod.yml

**🧼 Step 6: Clean Up (Optional)**

bash

CopyEdit

kubectl delete -f cmpod.yml

kubectl delete -f cm.yml

kubectl delete svc cmsvc -n raman

**🧠 Key Concepts Recap**

| **Concept** | **Details** |
| --- | --- |
| **ConfigMap** | Stores non-sensitive config data as key-value pairs (e.g., HTML files) |
| **VolumeMount** | Mounts ConfigMap data as files inside the pod |
| **NGINX** | Serves static content from /usr/share/nginx/html/index.html |
| **items[]** | Maps a specific key in the ConfigMap to a file name inside the pod |
| **index.html** | Default page served by NGINX |
| **NodePort** | Exposes pod on a port accessible externally from the cluster |

**🛠️ Bonus Tips**

* You can **hot-reload** NGINX if you update the ConfigMap by restarting the pod.
* For complex HTML/CSS/JS projects, you can mount entire directories instead of individual files.
* Use **kubectl edit cm** for live config changes.
* You can validate full response using curl or browser.

**🧪 Kubernetes User Authentication & RBAC Lab**

**Objective**:  
Simulate how a new user (Bob) is created, authenticated, and authorized to access Kubernetes resources using client certificates.

**🧠 Key Concepts**

| **Concept** | **Description** |
| --- | --- |
| **Authentication** | Validates identity (e.g., client certificates, bearer tokens) |
| **Authorization (RBAC)** | Determines what the authenticated user can do |
| **CA (Certificate Authority)** | Trusted entity that signs user/client certs |
| **kubeconfig** | Stores cluster, user credentials, and contexts |
| **CSR** | Certificate Signing Request used to request a signed certificate from the CA |

**🧰 Prerequisites**

* A working Kubernetes cluster (e.g., kubeadm-based)
* Access to Kubernetes CA (/etc/kubernetes/pki/ca.crt and ca.key)
* kubectl and openssl installed
* Namespace test created:

bash

CopyEdit

kubectl create namespace test

**🧾 LAB STEPS**

**🔐 Step 1: Generate Bob's Private Key and CSR**

This represents **Bob's ID request**, like he's asking for an ID card.

bash

CopyEdit

# Generate private key

openssl genrsa -out bob.key 2048

# Generate CSR

openssl req -new -key bob.key -out bob.csr -subj "/CN=bob/O=developers"

🔎 Explanation:

* CN=bob: Common Name — this becomes the **username** Kubernetes sees
* O=developers: Organization — this becomes the **group**

**🔏 Step 2: Sign Bob's Certificate with Kubernetes CA**

Now, the **Security Officer** (API server) checks and signs Bob's CSR.

bash

CopyEdit

# Navigate to CA directory

cd /etc/kubernetes/pki/

# Sign the CSR using the Kubernetes CA

openssl x509 -req -in ~/bob.csr -CA ca.crt -CAkey ca.key -CAcreateserial \

-out ~/bob.crt -days 365

🎯 Output: bob.crt — Bob's **trusted certificate**, signed by the Kubernetes CA

**💾 Step 3: Backup Your Kubeconfig**

bash

CopyEdit

cd ~/.kube

cp config config\_bak

**📦 Step 4: Add Bob as a New User in kubeconfig**

**4.1 Add user credentials:**

bash

CopyEdit

kubectl config set-credentials bob \

--client-certificate=~/bob.crt \

--client-key=~/bob.key

**4.2 View current clusters:**

bash

CopyEdit

kubectl config get-clusters

**4.3 Create a context for Bob:**

bash

CopyEdit

kubectl config set-context bob-context \

--cluster=kubernetes \

--user=bob \

--namespace=test

**4.4 Switch to Bob's context:**

bash

CopyEdit

kubectl config use-context bob-context

**4.5 Verify:**

bash

CopyEdit

kubectl config current-context

**🚫 Step 5: Test Access (Without RBAC)**

bash

CopyEdit

kubectl get pods

❌ You should see:

vbnet

CopyEdit

Error from server (Forbidden): User "bob" cannot list resource "pods" in API group "" in the namespace "test"

Kubernetes authenticated Bob successfully, but **he isn’t authorized to do anything** yet.

**✅ Step 6: Configure RBAC Permissions**

We’ll create a **Role** and **RoleBinding** to give Bob permission to list pods.

**6.1 Create a Role (namespace-scoped):**

yaml

CopyEdit

# dev-role.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: test

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

bash

CopyEdit

kubectl apply -f dev-role.yaml

**6.2 Bind the Role to Bob:**

yaml

CopyEdit

# dev-rolebinding.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: bob-read-pods

namespace: test

subjects:

- kind: User

name: bob

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

bash

CopyEdit

kubectl apply -f dev-rolebinding.yaml

**🧪 Step 7: Verify Bob’s Access**

bash

CopyEdit

kubectl get pods

✅ You should now see:

pgsql

CopyEdit

No resources found in test namespace.

Bob is now able to **list pods** in the **test** namespace.

**🧽 Step 8: Cleanup (Remove Bob and Artifacts)**

bash

CopyEdit

# Delete Role and Binding

kubectl delete -f dev-role.yaml

kubectl delete -f dev-rolebinding.yaml

# Delete context and user

kubectl config delete-context bob-context

kubectl config delete-user bob

# Switch back to admin

kubectl config use-context kubernetes-admin@kubernetes

# Delete certs

rm -f ~/bob.key ~/bob.crt ~/bob.csr

**🧠 Conceptual Summary**

| **Component** | **Real-world Analogy** | **Kubernetes** |
| --- | --- | --- |
| bob.key | Bob's private identity | Private key |
| bob.csr | ID card request Bob sends to security | Certificate Signing Request |
| bob.crt | Signed ID card from security officer | Signed certificate |
| /etc/kubernetes/pki/ca.crt | Security officer's signature | Cluster CA |
| kubeconfig | Bob’s access card and cluster directory | kubectl config |
| RBAC RoleBinding | Granting Bob access to specific areas | Role/RoleBinding |

**🧩 Bonus: Add Bob to a ClusterRole**

Want Bob to access **cluster-wide resources**? Use a ClusterRole and ClusterRoleBinding instead:

yaml

CopyEdit

kind: ClusterRole

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: cluster-viewer

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

yaml

CopyEdit

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: bob-cluster-binding

subjects:

- kind: User

name: bob

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: cluster-viewer

apiGroup: rbac.authorization.k8s.io

**🧪 Kubernetes RBAC Lab — Role vs ClusterRole for Bob**

**👤 Goal:**

* Create user **Bob**.
* Restrict his permissions **namespace-wise** with Role.
* Extend his access **cluster-wide** with ClusterRole.

**📌 Pre-Requisites**

* Working Kubernetes cluster
* Kubernetes CA access for user certificate generation
* kubectl, openssl installed
* A namespace raman created

bash

CopyEdit

kubectl create namespace raman

**🔐 STEP 1: (Recap) User Setup for Bob**

If not already done, create user Bob using cert-based auth:

<details> <summary>📜 <strong>Click to expand: Certificate Steps</strong></summary>

bash

CopyEdit

# Generate private key

openssl genrsa -out bob.key 2048

# Create CSR with group developers (optional)

openssl req -new -key bob.key -out bob.csr -subj "/CN=bob/O=developers"

# Sign with Kubernetes CA

openssl x509 -req -in bob.csr -CA /etc/kubernetes/pki/ca.crt \

-CAkey /etc/kubernetes/pki/ca.key -CAcreateserial \

-out bob.crt -days 365

# Add to kubeconfig

kubectl config set-credentials bob \

--client-certificate=bob.crt \

--client-key=bob.key

kubectl config set-context bob-context \

--cluster=kubernetes \

--user=bob \

--namespace=raman

kubectl config use-context bob-context

</details>

**🧪 STEP 2: Test Without Permissions**

bash

CopyEdit

kubectl get pods

kubectl get secrets

Expected output:

pgsql

CopyEdit

Error from server (Forbidden): User "bob" cannot list resource "pods" in API group ""

**🎯 STEP 3: Create Namespace-Scoped Role for Bob**

**🔧 Create role.yml:**

yaml

CopyEdit

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: raman

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods", "secrets"]

verbs: ["get", "watch", "list"]

bash

CopyEdit

kubectl apply -f role.yml

**🔧 Create rolebinding.yml:**

yaml

CopyEdit

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods

namespace: raman

subjects:

- kind: User

name: bob

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

bash

CopyEdit

kubectl apply -f rolebinding.yml

**🔎 Test Again as Bob:**

bash

CopyEdit

kubectl config use-context bob-context

kubectl get pods -n raman

kubectl get secrets -n raman

kubectl get pods # should still fail in default namespace

✅ Bob can now **list pods and secrets only in the raman namespace**.

**🔁 STEP 4: Switch Back and Clean Up Role-based Bindings**

bash

CopyEdit

kubectl config use-context kubernetes-admin@kubernetes

kubectl delete role --all -n raman

kubectl delete rolebinding --all -n raman

**🌐 STEP 5: Create a ClusterRole and Bind Bob to Cluster-Wide Access**

**🔧 clusterrole.yml**

yaml

CopyEdit

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: secret-pod-reader

rules:

- apiGroups: [""]

resources: ["pods", "secrets"]

verbs: ["get", "watch", "list"]

❗ Fixed typo: Kubernetes resource names **must not have commas** like secret,pod-reader.

bash

CopyEdit

kubectl apply -f clusterrole.yml

**🔧 clusterrolebinding.yml**

yaml

CopyEdit

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: read-secrets-pods-global

subjects:

- kind: User

name: bob

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: secret-pod-reader

apiGroup: rbac.authorization.k8s.io

bash

CopyEdit

kubectl apply -f clusterrolebinding.yml

**📊 STEP 6: Validate Cluster-Level Access**

Switch to Bob’s context:

bash

CopyEdit

kubectl config use-context bob-context

Now try:

bash

CopyEdit

kubectl get pods -A

kubectl get secrets -A

✅ Bob can now **access pods and secrets across all namespaces**.

**🧽 STEP 7: Cleanup (Optional)**

bash

CopyEdit

kubectl config use-context kubernetes-admin@kubernetes

kubectl delete clusterrole secret-pod-reader

kubectl delete clusterrolebinding read-secrets-pods-global

kubectl config delete-context bob-context

kubectl config delete-user bob

rm -f bob.key bob.crt bob.csr

**🧠 RBAC Summary Table**

| **Type** | **Scope** | **Permissions Given** | **File** |
| --- | --- | --- | --- |
| Role | Namespaced | Pods & Secrets in raman | role.yml |
| RoleBinding | Namespaced | Grants Bob Role access | rolebinding.yml |
| ClusterRole | Cluster-wide | Pods & Secrets in all namespaces | clusterrole.yml |
| ClusterRoleBinding | Cluster-wide | Grants Bob cluster-wide Role access | clusterrolebinding.yml |

**🧪 Kubernetes Storage Lab Guide**

**🔹 LAB 1: HostPath Volume Without PV/PVC**

**Goal:**

Understand how hostPath volumes work directly with Pods and how data is not portable across nodes.

**Step 1: Create a Pod with HostPath (No PV/PVC)**

yaml

CopyEdit

# pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: redispod

namespace: default

spec:

restartPolicy: Never

containers:

- name: redis

image: redis

volumeMounts:

- name: vol

mountPath: /hostfile

volumes:

- name: vol

hostPath:

path: /home/ubuntu

# Optionally use nodeName to pin the Pod

nodeName: worker2

bash

CopyEdit

kubectl apply -f pod.yaml

**Step 2: Observe the Volume Behavior**

* SSH into the node (worker2) where the pod is running:

bash

CopyEdit

cd /home/ubuntu

ls

* Write something inside the container:

bash

CopyEdit

kubectl exec -it redispod -- sh

echo "hello" > /hostfile/data.txt

* Validate it appears on the host node.

**🔸 LAB 2: Why We Need PV/PVC**

**Use Case:**

If the same pod runs on another node, /home/ubuntu on the second node is *not* automatically populated — this proves hostPath is **node-dependent** and not scalable.

**🔹 LAB 3: Use PV and PVC for HostPath**

**Step 1: Create PersistentVolume**

yaml

CopyEdit

# hostpath-pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: hostpath-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Retain

hostPath:

path: "/data/hostpath-pv"

bash

CopyEdit

kubectl apply -f hostpath-pv.yaml

**Step 2: Developer Creates PVC**

yaml

CopyEdit

# hostpath-pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: hostpath-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 500Mi

bash

CopyEdit

kubectl apply -f hostpath-pvc.yaml

**Step 3: Use PVC in a Pod**

yaml

CopyEdit

# hostpath-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: hostpath-pod

spec:

containers:

- name: app-container

image: busybox

command: ["/bin/sh"]

args: ["-c", "sleep 3600"]

volumeMounts:

- name: host-volume

mountPath: /mnt/volume

volumes:

- name: host-volume

persistentVolumeClaim:

claimName: hostpath-pvc

bash

CopyEdit

kubectl apply -f hostpath-pod.yaml

**Step 4: Validate**

bash

CopyEdit

kubectl get pv

kubectl get pvc

kubectl exec -it hostpath-pod -- sh

echo "hello from pod" > /mnt/volume/hello.txt

**🔸 LAB 4: Local Persistent Volumes (Preferred over hostPath)**

**Step 1: Prepare Local Storage on Node**

bash

CopyEdit

ssh nodeX

sudo mkdir -p /mnt/data

sudo chmod 777 /mnt/data

**Step 2: Create PV**

yaml

CopyEdit

# local-pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: local-pv

spec:

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Retain

storageClassName: local-storage

local:

path: /mnt/data

nodeAffinity:

required:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/hostname

operator: In

values:

- worker2 # Use the exact node name

bash

CopyEdit

kubectl apply -f local-pv.yaml

**Step 3: PVC**

yaml

CopyEdit

# local-pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: local-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

storageClassName: local-storage

bash

CopyEdit

kubectl apply -f local-pvc.yaml

**Step 4: Use PVC in Pod**

yaml

CopyEdit

# app-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: app-pod

spec:

containers:

- name: app-container

image: nginx

volumeMounts:

- mountPath: /usr/share/nginx/html

name: app-storage

volumes:

- name: app-storage

persistentVolumeClaim:

claimName: local-pvc

bash

CopyEdit

kubectl apply -f app-pod.yaml

**Step 5: Validate**

bash

CopyEdit

echo "Hello from backend" | sudo tee /mnt/data/index.html

kubectl exec -it app-pod -- cat /usr/share/nginx/html/index.html

**🔹 LAB 5: Using NFS (or AWS EFS) for Shared Storage**

**Step 1: Create PV with NFS**

yaml

CopyEdit

# nfs-pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: nfs-website

spec:

capacity:

storage: 11Mi

accessModes:

- ReadWriteMany

mountOptions:

- hard

- nfsvers=4.1

nfs:

path: /

server: 172.31.12.142

**Step 2: Create PVC**

yaml

CopyEdit

# nfs-pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nfs-demo

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Mi

volumeName: nfs-website

**Step 3: Create Deployment using PVC**

yaml

CopyEdit

# nfs-deploy.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nfs-raman

spec:

replicas: 10

selector:

matchLabels:

role: nfs-raman

template:

metadata:

labels:

role: nfs-raman

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

volumeMounts:

- name: nfs

mountPath: /usr/share/nginx/deploydata

volumes:

- name: nfs

persistentVolumeClaim:

claimName: nfs-demo

**Step 4: Expose Deployment**

bash

CopyEdit

kubectl expose deployment nfs-raman --name nfs-svc --type NodePort --port 80 --target-port 80

**Step 5: Validate**

bash

CopyEdit

kubectl get pv

kubectl get pvc

kubectl get pods

**Step 6: Mount NFS on Any Node to Inspect**

bash

CopyEdit

apt install -y nfs-common

mkdir /efs

mount -t nfs4 -o nfsvers=4.1 172.31.12.142:/ /efs

cd /efs

ls -l

🔥 Browse: http://<NodeIP>:<NodePort> to see nginx content from NFS

**🧠 Summary Table**

| **Type** | **Reclaimable** | **Shared** | **Node-Affinity** | **Best For** |
| --- | --- | --- | --- | --- |
| hostPath | ❌ | ❌ | ✅ | Testing |
| local PV | ✅ | ❌ | ✅ | On-prem SSDs |
| NFS / EFS | ✅ | ✅ | ❌ | Shared HA workloads |
| cloud block | ✅ | ❌ | ❌ | DB, Stateful apps |