**🧑‍💻 Lab Guide: Kubernetes Persistent Storage with PV/PVC + AWS EFS (NFS)**

**🎯 Lab Objective**

Implement persistent storage for Kubernetes workloads using:

* AWS **EFS** as NFS backend.
* Kubernetes **PersistentVolume (PV)** and **PersistentVolumeClaim (PVC)**.
* Deploy applications that use these persistent volumes.

You will:

✅ Provision EFS and mount targets.  
✅ Validate NFS connectivity from worker nodes.  
✅ Create PV, PVC, and deploy workloads consuming them.  
✅ Test data persistence across pods.

**🛠 Pre-requisites**

* Kubernetes cluster with at least 2 worker nodes.
* kubectl configured to access the cluster.
* AWS EFS already provisioned in the same VPC as the Kubernetes nodes.
* EFS mount target security group allows NFS (TCP/UDP 2049) from worker nodes' security groups.
* Install nfs-common package on worker nodes.

**📝 Lab Steps**

**🔥 Step 1: Validate EFS and NFS Access**

**1.1 Install NFS client utilities (Ubuntu/Debian nodes)**

bash

CopyEdit

sudo apt update

sudo apt install -y nfs-common

**1.2 Mount EFS manually for testing**

* Create a mount directory:

bash

CopyEdit

sudo mkdir /efs

* Mount the EFS:

bash

CopyEdit

sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,noresvport \

<efs-mount-target>:/ /efs

Replace <efs-mount-target> with your EFS DNS or IP (e.g., fs-xxxx.efs.us-east-1.amazonaws.com).

* Verify:

bash

CopyEdit

df -h

cd /efs

touch testfile

ls -l

✅ **If successful**, your Kubernetes nodes can talk to EFS.

**📂 Step 2: Create Kubernetes Namespace**

Create a namespace for isolation.

bash

CopyEdit

kubectl create namespace raman

**📄 Step 3: Create PersistentVolume (PV)**

**3.1 Create pv.yml:**

yaml

CopyEdit

apiVersion: v1

kind: PersistentVolume

metadata:

name: raman-nfs-website

spec:

capacity:

storage: 11Gi

accessModes:

- ReadWriteMany

mountOptions:

- hard

- nfsvers=4.1

persistentVolumeReclaimPolicy: Retain

nfs:

path: /

server: <efs-mount-target>

Replace <efs-mount-target> with your EFS IP or DNS.

**3.2 Apply PV**

bash

CopyEdit

kubectl apply -f pv.yml

**📄 Step 4: Create PersistentVolumeClaim (PVC)**

**4.1 Create pvc.yml:**

yaml

CopyEdit

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: raman-nfs-demo

namespace: raman

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Gi

volumeName: raman-nfs-website

**4.2 Apply PVC**

bash

CopyEdit

kubectl apply -f pvc.yml

**✅ Step 5: Verify PV/PVC Binding**

bash

CopyEdit

kubectl get pv

kubectl get pvc -n raman

✅ Ensure PV is **Bound** and PVC is in **Bound** state.

**📦 Step 6: Deploy Application using PVC**

**6.1 Create deploy.yaml:**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deploy

namespace: raman

spec:

replicas: 3

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: raman

mountPath: /usr/share/nginx/deploydata

volumes:

- name: raman

persistentVolumeClaim:

claimName: raman-nfs-demo

**6.2 Apply Deployment**

bash

CopyEdit

kubectl apply -f deploy.yaml

**🌐 Step 7: Expose the Application**

bash

CopyEdit

kubectl expose deployment raman-deploy \

--name raman-svc \

--namespace raman \

--type NodePort \

--port 80 --target-port 80

**🔍 Step 8: Verify Application and Test Persistence**

**8.1 Check Pods and Service**

bash

CopyEdit

kubectl get pods -n raman -o wide

kubectl get svc -n raman

**8.2 Enter one pod and create a test file**

bash

CopyEdit

kubectl exec -it <pod-name> -n raman -- bash

cd /usr/share/nginx/deploydata

echo "Hello from Pod1" > hello.txt

exit

**8.3 Enter another pod and verify file presence**

bash

CopyEdit

kubectl exec -it <another-pod-name> -n raman -- bash

cd /usr/share/nginx/deploydata

cat hello.txt

✅ If you see Hello from Pod1, persistence is working across pods.

**🚨 Step 9: Security Group Check for EFS**

If mounting fails or pods hang in ContainerCreating state:

* Go to AWS Console → **EFS → Mount Targets → Security Groups**.
* Ensure **inbound rule** for TCP **2049** is open for the Kubernetes nodes’ security group.

**🗑 Step 10: Cleanup**

bash

CopyEdit

kubectl delete ns raman

kubectl delete pv raman-nfs-website

**📝 Key Takeaways**

1. **PV is cluster-wide**, **PVC is namespace scoped**.
2. ReadWriteMany allows **multiple pods on multiple nodes** to share the volume.
3. AWS EFS is a perfect backend for such shared scenarios.
4. Always verify **Security Groups** for NFS traffic.

**🧑‍💻 Lab Guide: Kubernetes Ingress Controller with Path-based Routing**

**🎯 Lab Objective**

Deploy an **NGINX Ingress Controller** using Helm in Kubernetes, create multiple backend applications, and expose them via Ingress resources.

* Route user traffic:
  + /raman → **test-app (nginx)**
  + /raman2 → **test-app2 (httpd)**

**🛠 Pre-requisites**

✔️ A Kubernetes cluster (with at least 1 master and 2 worker nodes).  
✔️ kubectl CLI configured for the cluster.  
✔️ Internet access from nodes (for image pulls and Helm installation).  
✔️ A public IP for any node (or use a cloud LoadBalancer service).

**📝 Lab Steps**

**🔥 Step 1: Install Helm 3**

Helm simplifies application deployment in Kubernetes.

**1.1 Download and install Helm**

bash

CopyEdit

curl -fsSL -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3

chmod 700 get\_helm.sh

./get\_helm.sh

**1.2 Verify Helm installation**

bash

CopyEdit

helm version

✅ You should see Helm client and server versions.

**📂 Step 2: Install NGINX Ingress Controller**

**2.1 Create a namespace for Ingress**

bash

CopyEdit

kubectl create namespace ingress

**2.2 Add the official NGINX Ingress Helm repository**

bash

CopyEdit

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update

**2.3 Install NGINX Ingress Controller with 2 replicas**

bash

CopyEdit

helm install my-release ingress-nginx/ingress-nginx \

--namespace ingress \

--set controller.replicaCount=2 \

--set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \

--set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux

**2.4 Verify Ingress Controller deployment**

bash

CopyEdit

kubectl get all -n ingress

✅ Confirm ingress-nginx-controller pods are running.

**2.5 Check the Service type of the Ingress Controller**

bash

CopyEdit

kubectl get svc -n ingress

* If Service type is LoadBalancer → note the **EXTERNAL-IP**.
* If Service type is NodePort → note the **PORT** to access via public node IP.

**📦 Step 3: Deploy Backend Applications**

We’ll deploy two apps:

* **test-app (nginx)** → /raman
* **test-app2 (httpd)** → /raman2

**3.1 Create secdeploy.yml:**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-app

labels:

app: test-app

spec:

replicas: 3

selector:

matchLabels:

app: test-app

template:

metadata:

labels:

app: test-app

spec:

containers:

- name: test-app

image: nginx:latest

ports:

- containerPort: 80

resources:

limits:

cpu: 100m

memory: 128Mi

requests:

cpu: 50m

memory: 64Mi

---

apiVersion: v1

kind: Service

metadata:

name: raman-service

spec:

type: NodePort

ports:

- port: 80

selector:

app: test-app

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-app2

labels:

app: test-app2

spec:

replicas: 3

selector:

matchLabels:

app: test-app2

template:

metadata:

labels:

app: test-app2

spec:

containers:

- name: test-app2

image: httpd

ports:

- containerPort: 80

resources:

limits:

cpu: 100m

memory: 128Mi

requests:

cpu: 50m

memory: 64Mi

---

apiVersion: v1

kind: Service

metadata:

name: raman-service2

spec:

type: NodePort

ports:

- port: 80

selector:

app: test-app2

**3.2 Apply the deployment and services**

bash

CopyEdit

kubectl create namespace raman

kubectl apply -f secdeploy.yml -n raman

**3.3 Verify**

bash

CopyEdit

kubectl get pods -n raman

kubectl get svc -n raman

✅ All pods should be running and services created.

**🌐 Step 4: Create Ingress Resource**

**4.1 Create ingresource.yml:**

yaml

CopyEdit

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: test-app-ingress

namespace: raman

annotations:

nginx.ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/use-regex: "true"

nginx.ingress.kubernetes.io/rewrite-target: /$2

spec:

ingressClassName: nginx

rules:

- http:

paths:

- path: /raman(/|$)(.\*)

pathType: ImplementationSpecific

backend:

service:

name: raman-service

port:

number: 80

- http:

paths:

- path: /raman2(/|$)(.\*)

pathType: ImplementationSpecific

backend:

service:

name: raman-service2

port:

number: 80

**4.2 Apply Ingress Resource**

bash

CopyEdit

kubectl apply -f ingresource.yml

**4.3 Verify**

bash

CopyEdit

kubectl get ingress -n raman

kubectl describe ingress test-app-ingress -n raman

✅ Note the **Address** (public IP) of the Ingress.

**🖥 Step 5: Test Access**

**Get Node Public IP (or Ingress LoadBalancer IP):**

bash

CopyEdit

kubectl get nodes -o wide

If Ingress Controller Service is NodePort:

* Test access:

php-template

CopyEdit

http://<NodePublicIP>:<NodePort>/raman

http://<NodePublicIP>:<NodePort>/raman2

If Ingress Controller Service is LoadBalancer:

* Test access:

perl

CopyEdit

http://<LoadBalancerIP>/raman

http://<LoadBalancerIP>/raman2

✅ You should see:

* /raman → Nginx default welcome page.
* /raman2 → Apache HTTPD default page.

**📝 Key Takeaways**

1. **Ingress Controller (NGINX)** acts as the gateway for all external traffic.
2. **Ingress Resource** defines routing rules (host/path → backend service).
3. Helm simplifies installing and managing complex apps like Ingress Controller.

**🧑‍💻 Lab Guide: Kubernetes User Management & RBAC (Bob’s Access)**

**🎯 Lab Objective**

In this lab, we will:

✅ Create a Kubernetes user **Bob** (authenticated via client certificates).  
✅ Add **Bob** to the developers group.  
✅ Restrict Bob’s access to only **read pods in a specific namespace** using RBAC.  
✅ Test Bob’s access before and after applying RBAC.

**🛠 Pre-requisites**

* Kubernetes cluster (kubeadm-managed cluster preferred).
* kubectl configured on the admin machine.
* Access to Kubernetes CA certificate and key (usually at /etc/kubernetes/pki/).
* openssl installed on admin workstation.

**📝 Lab Steps**

**🧑‍🏫 Background Concepts**

| **Concept** | **Analogy** |
| --- | --- |
| **Bob’s ID card** | Bob’s private key + signed certificate |
| **Security guard** | Kubernetes API Server (auth check) |
| **Reception desk** | Certificate Authority (CA) |
| **Access Rules** | Kubernetes RBAC (Roles, Bindings) |
| **Company floors** | Kubernetes Namespaces |

**🔥 Step 1: Generate Bob’s Private Key and CSR**

Bob generates his private key (ID card request).

**1.1 Bob creates his private key (bob.key):**

bash

CopyEdit

openssl genrsa -out bob.key 2048

**1.2 Bob creates a Certificate Signing Request (CSR):**

* If Bob belongs to a group (developers), include it in CSR.

bash

CopyEdit

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

| **Field** | **Meaning** |
| --- | --- |
| CN | Common Name = Username (bob) |
| O | Organization = Group (developers) |

**🖋 Step 2: Sign Bob’s Certificate with Kubernetes CA**

The Kubernetes admin acts as the “Reception Desk” to verify and issue Bob’s ID card.

**2.1 Locate Kubernetes CA cert and key:**

bash

CopyEdit

ls /etc/kubernetes/pki/ca.\*

You should see:

swift

CopyEdit

/etc/kubernetes/pki/ca.crt

/etc/kubernetes/pki/ca.key

**2.2 Sign Bob’s CSR:**

bash

CopyEdit

openssl x509 -req -in bob.csr \

-CA /etc/kubernetes/pki/ca.crt \

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

-CAcreateserial \

-out bob.crt -days 365

✅ This produces bob.crt – Bob’s signed certificate.

**📂 Step 3: Backup kubeconfig**

**Backup admin kubeconfig:**

bash

CopyEdit

cp ~/.kube/config ~/.kube/config\_bak

**🗝️ Step 4: Add Bob to kubeconfig**

**4.1 Add Bob as a user:**

bash

CopyEdit

kubectl config set-credentials bob \

--client-certificate=bob.crt \

--client-key=bob.key

✅ Adds a **bob** user entry in kubeconfig.

**4.2 Add a context for Bob:**

Find your cluster name:

bash

CopyEdit

kubectl config get-clusters

Example output: kubernetes

Create a context:

bash

CopyEdit

kubectl config set-context bob-context \

--cluster=kubernetes \

--user=bob \

--namespace=test-namespace

**4.3 Switch to Bob’s context:**

bash

CopyEdit

kubectl config use-context bob-context

✅ Check current context:

bash

CopyEdit

kubectl config current-context

**❌ Step 5: Test Bob’s Access (Before RBAC)**

bash

CopyEdit

kubectl get pods

**Expected Result:**

vbnet

CopyEdit

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

✅ Bob is authenticated, but **not authorized** to access anything yet.

**🛡️ Step 6: Create Namespace for Bob**

As admin, create a namespace:

bash

CopyEdit

kubectl config use-context kubernetes-admin@kubernetes

kubectl create namespace test-namespace

**📝 Step 7: Create RBAC Role and RoleBinding**

**7.1 Create Role (read-only access to pods):**

📄 pod-reader-role.yaml

yaml

CopyEdit

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: test-namespace

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

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

Apply the Role:

bash

CopyEdit

kubectl apply -f pod-reader-role.yaml

**7.2 Create RoleBinding (assign Role to Bob):**

📄 pod-reader-rolebinding.yaml

yaml

CopyEdit

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: pod-reader-binding

namespace: test-namespace

subjects:

- kind: User

name: bob

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

Apply the RoleBinding:

bash

CopyEdit

kubectl apply -f pod-reader-rolebinding.yaml

**✅ Step 8: Test Bob’s Access (After RBAC)**

Switch to Bob’s context:

bash

CopyEdit

kubectl config use-context bob-context

**8.1 List pods in test-namespace:**

bash

CopyEdit

kubectl get pods

✅ Bob can now read pods in test-namespace.

**8.2 Try to create a pod (should fail):**

bash

CopyEdit

kubectl run busybox --image=busybox -- sleep 3600

**Expected Result:**

pgsql

CopyEdit

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

**🧹 Step 9: Clean-Up**

Switch to admin context:

bash

CopyEdit

kubectl config use-context kubernetes-admin@kubernetes

Delete Bob’s user and context:

bash

CopyEdit

kubectl config delete-context bob-context

kubectl config delete-user bob

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

Delete RBAC resources:

bash

CopyEdit

kubectl delete rolebinding pod-reader-binding -n test-namespace

kubectl delete role pod-reader -n test-namespace

kubectl delete namespace test-namespace

**📝 Key Takeaways**

| **🧠 Concept** | **📖 Details** |
| --- | --- |
| **Authentication** | Verified via signed certificates |
| **Authorization** | Controlled via Roles and Bindings |
| **Role vs ClusterRole** | Role → Namespace scoped |
| **RBAC Principle** | Least privilege |