**🚀 Kubernetes Persistent Storage with NFS/EFS – Lab Guide**

This lab demonstrates how to make storage persistent in Kubernetes (since Pods are ephemeral by default) using **NFS/EFS as a backend**, **Persistent Volumes (PV)**, **Persistent Volume Claims (PVC)**, and a simple **nginx Deployment**.

We will cover:

1. Why persistence is needed in Kubernetes
2. Understanding volume types
3. Admin role: Provisioning Persistent Volumes (PV)
4. Developer role: Claiming Persistent Volumes with PVC
5. Deploying an application with persistent storage
6. Verification

**1. Ephemeral vs Persistent Storage in Kubernetes**

* **Ephemeral volumes**: Disappear when Pods restart (e.g., emptyDir, hostPath).
* **Persistent volumes**: Survive Pod restarts and rescheduling (e.g., NFS, EBS, Azure Disk, GCP Persistent Disk).

Examples:

* emptyDir: temporary scratch space tied to Pod lifetime
* configMap & secrets: inject configuration or sensitive data into Pods
* downwardAPI: expose Pod metadata (labels, annotations)
* projected: combine multiple sources into one volume
* PersistentVolume: independent storage resource, reusable across Pods

**2. Roles in Persistent Storage**

* **Kubernetes Admin**: Provisions PersistentVolumes (PV). Example: an NFS-backed volume.
* **Developers**: Request storage using PersistentVolumeClaims (PVC).

Example workloads:

* ML App → needs 100 GB storage
* Web App → 50 GB storage
* Gaming App → 70 GB storage

Admin defines available pools; developers consume via PVC.

**3. Lab Prerequisites**

* A Kubernetes cluster (minikube, kind, or EKS/GKE/AKS).
* An accessible NFS/EFS server.

Optional: To simulate EFS/NFS on Ubuntu:

sudo apt-get update -y

sudo apt-get install -y nfs-common

sudo mkdir /efs

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

df -h

Check mounted directory:

cd /efs && ls -l

**4. Create a PersistentVolume (Admin Task)**

File: pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: raman-nfs-website

spec:

capacity:

storage: 11Mi

accessModes:

- ReadWriteMany

mountOptions:

- hard

- nfsvers=4.1

nfs:

path: /

server: 172.31.14.29

Explanation:

* **capacity**: Max size 11Mi (enforced logically, not physically).
* **accessModes**: ReadWriteMany → multiple Pods can read/write.
* **nfs**: backend path / hosted on server 172.31.14.29.

Apply:

kubectl create -f pv.yaml

kubectl get pv

**5. Create a PersistentVolumeClaim (Developer Task)**

First, create a namespace for isolation:

kubectl create ns raman

File: pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: raman-nfs-demo

namespace: raman

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Mi

volumeName: raman-nfs-website

Explanation:

* PVC requests **5Mi** (less than PV capacity).
* Binds specifically to PV raman-nfs-website.

Apply:

kubectl create -f pvc.yaml

kubectl get pvc -n raman

kubectl get pv

Check status → PVC should be **Bound** to PV.

**6. Deploy Application Using PVC**

File: deploy.yml

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-nfs

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: nfsrk

mountPath: /usr/share/nginx/deploydata

volumes:

- name: nfsrk

persistentVolumeClaim:

claimName: raman-nfs-demo

Explanation:

* **replicas: 3** → three nginx Pods.
* **volumeMounts**: PVC mounted inside container at /usr/share/nginx/deploydata.
* Multiple Pods can read/write since PV uses ReadWriteMany.

Apply:

kubectl create -f deploy.yml

kubectl get pods -n raman

**7. Verification**

1. Check PV/PVC:

kubectl get pv

kubectl get pvc -n raman

1. Exec into a Pod:

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

cd /usr/share/nginx/deploydata

echo "Hello from Raman PV test" > test.txt

1. Open another Pod in same Deployment:

kubectl exec -it -n raman <another-pod> -- /bin/bash

cd /usr/share/nginx/deploydata

cat test.txt

If the file is visible → storage is **persistent + shared**.

**8. Clean-Up**

kubectl delete -f deploy.yml -n raman

kubectl delete -f pvc.yaml -n raman

kubectl delete -f pv.yaml

kubectl delete ns raman

**📝 Key Takeaways**

* Pods are ephemeral; storage persistence requires PV + PVC.
* **Admin** provisions PVs (NFS, EFS, disks).
* **Developers** request PVCs (without worrying about backend).
* ReadWriteMany is powerful for shared storage across Pods.
* This pattern is the foundation for **stateful apps** like databases, ML workloads, gaming apps.

**🌐 Kubernetes Ingress & Ingress Controller – Lab Guide**

This lab demonstrates how to centrally manage external access to multiple Kubernetes services using an **Ingress Controller** (NGINX via Helm) and an **Ingress resource**.

We will cover:

1. Why Ingress is needed
2. Ingress vs Ingress Controller
3. Central administration of cluster traffic
4. Helm-based NGINX Ingress Controller installation
5. Deploying sample apps and services
6. Defining and testing Ingress rules
7. Verification

**1. Why Ingress?**

* **Without Ingress**: To expose each app, you’d typically create a NodePort or LoadBalancer service. This is clunky — every app eats up new ports or external load balancers.
* **With Ingress**: A single entry point (the Ingress Controller) handles HTTP/HTTPS routing into your cluster. Then, fine-grained rules direct traffic to the right Service/Pod.

Think of Ingress as the **traffic cop** at the entrance of your cluster, using rules to route requests like:

* /raman → App1
* /raman2 → App2

**2. Ingress vs Ingress Controller**

* **Ingress resource**: A Kubernetes object that defines *routing rules* (like YAML with paths, hostnames, TLS, etc.).
* **Ingress Controller**: The actual software that implements those rules. Examples:
  + **NGINX Ingress Controller** (most common)
  + HAProxy, Traefik, Istio Gateways

Kubernetes ships with the Ingress API, but not with a controller. You need to deploy one (e.g., NGINX).

**3. Central Administration**

* With an Ingress Controller, the **cluster admin** provides a *single external endpoint* (like an ELB, or NodePort).
* Developers then only need to create Ingress resources with their routing rules.
* This keeps networking centralized and standardized, instead of scattering random NodePorts across nodes.

**4. Install NGINX Ingress Controller with Helm**

First, install Helm if not present:

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

Create namespace:

kubectl create ns ingress

Add repo and update:

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

helm repo update

Install NGINX Ingress Controller:

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

Check:

helm list -n ingress

kubectl get all -n ingress

At this point, you have a running **Ingress Controller**, listening for Ingress resources.

**5. Deploy Sample Applications & Services**

File: inglabdeploy.yaml

# Deployment 1: NGINX

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-app

namespace: raman

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

---

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

ports:

- port: 80

selector:

app: test-app

---

# Deployment 2: Apache HTTPD

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-app2

namespace: raman

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

---

apiVersion: v1

kind: Service

metadata:

name: raman-service2

namespace: raman

spec:

type: NodePort

ports:

- port: 80

selector:

app: test-app2

Apply:

kubectl create ns raman

kubectl create -f inglabdeploy.yaml

kubectl get deploy -n raman

kubectl get svc -n raman

You now have:

* **test-app** (nginx) → raman-service
* **test-app2** (httpd) → raman-service2

**6. Create Ingress Resource**

File: ingress.yml

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

Explanation:

* Ingress class nginx → handled by NGINX Ingress Controller.
* Paths:
  + /raman routes to nginx (raman-service)
  + /raman2 routes to apache (raman-service2)
* rewrite-target ensures the path after /raman or /raman2 is forwarded to backend service properly.

Apply:

kubectl create -f ingress.yml

kubectl get ingress -n raman

kubectl describe ingress test-app-ingress -n raman

**7. Verification**

1. Get ingress controller service:

kubectl get svc -n ingress

Usually my-release-ingress-nginx-controller → has EXTERNAL-IP (LoadBalancer) or NodePort.

1. Test access (assuming EXTERNAL-IP = 1.2.3.4):

curl http://1.2.3.4/raman

curl http://1.2.3.4/raman2

You should see NGINX welcome page and Apache welcome page respectively.

**8. Clean-Up**

kubectl delete -f ingress.yml -n raman

kubectl delete -f inglabdeploy.yaml -n raman

helm uninstall my-release -n ingress

kubectl delete ns ingress raman

**📝 Key Takeaways**

* **Ingress Controller** = the software router (NGINX, Traefik, etc.)
* **Ingress resource** = rules for routing traffic to services
* Benefits: central administration of external access, cleaner than NodePorts/LoadBalancers per app
* Helm makes it easy to install/manage controllers
* You can scale this model to host dozens of apps under one IP/hostname, separated by paths or subdomains

**🏢 Kubernetes as a Secure Building**

Imagine Kubernetes as a high-security **corporate building**.

**Step 1: Authentication = Getting Past the Security Guards**

* **Bob arrives at the building.**  
  He can’t just walk in; he needs a **valid ID card** that the security guards recognize.
* In Kubernetes, this ID card = **bob.crt + bob.key**, signed by the **building’s Certificate Authority (CA)**.
  + The CA is like the **government’s ID office** (trusted issuer).
  + If Bob shows a fake ID (unsigned certificate), the guards (API Server) will say ❌ “not valid.”

So authentication answers:  
➡️ *“Is this person really Bob, and do I trust his ID?”*

**Step 2: Authorization = Getting to the Right Floor**

* Now that Bob is in the lobby, he wants to take the elevator.
* Not everyone can go everywhere. For example:
  + **15th floor = production floor** (very restricted).
  + **2nd floor = training lab** (open to interns).
* Kubernetes RBAC = **access rules for floors**.
  + **Role** = which rooms/floors exist and what actions are allowed (read logs, start machines, etc.).
  + **RoleBinding** = attaches that Role to a specific user like Bob.

So authorization answers:  
➡️ *“Bob is real, but is he allowed on the 15th floor, or only the 2nd?”*

**Step 3: National Identity vs. Smart Chip Badge**

* **Government-issued national ID** = proves Bob exists (like a certificate signed by a trusted CA).
* **Company smart badge** = lets Bob into specific rooms/floors (like an RBAC RoleBinding).

Together:

* **Authentication (ID)** gets him *into the building*.
* **Authorization (badge/RBAC)** decides *which doors open for him*.

**🔑 Kubernetes Mapping**

* **Security guards at lobby** → Kubernetes API Server
* **Government issuing ID** → Certificate Authority (CA)
* **Bob’s ID card** → bob.crt signed by CA
* **Bob’s private key** → proves he owns that cert (not a stolen copy)
* **Access badge for certain floors** → RBAC Roles & RoleBindings
* **Prod floor (15th)** → restricted namespace/cluster resources

So, Bob’s Day 1 in Kubernetes looks like this:

1. He gets his **certificate + key** (ID card).
2. He adds them to his **kubeconfig** (wears the badge around his neck).
3. The API Server lets him in if the cert is valid.
4. RBAC then either:
   * Opens the elevator to prod (if he’s allowed).
   * Or stops him at the lobby saying “Access denied.”

This split — **Authentication first, then Authorization** — is what makes Kubernetes security clean and modular.

**🧪 Lab Guide: Creating and Managing Users in Kubernetes with Certificates + RBAC**

**🎯 Lab Objectives**

By the end of this lab, you will:

1. Understand how **authentication** (certificates) and **authorization** (RBAC) work in Kubernetes.
2. Create a **new user (bob)** in Kubernetes using certificates.
3. Configure kubeconfig for bob.
4. Test access before and after applying RBAC roles.
5. Learn how to clean up (delete user/certs/config).

**Step 1: 🔐 Generate a Certificate for the New User**

**1.1 Generate a private key and CSR (Certificate Signing Request)**

# Generate private key

openssl genrsa -out bob.key 2048

# Generate CSR with username

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

👉 Explanation:

* CN=bob → **Common Name** = username (bob).
* You can also add a **group** with /O=developers:
* openssl req -new -key bob.key -out bob.csr -subj "/CN=bob/O=developers"
  + CN → user
  + O → group (useful for group-based RBAC).

**Step 2: 🖋️ Sign the CSR with Kubernetes CA**

Your cluster already has a **Certificate Authority (CA)**, located (for kubeadm-based clusters) at:

* /etc/kubernetes/pki/ca.crt
* /etc/kubernetes/pki/ca.key

**2.1 Sign bob’s CSR:**

openssl x509 -req -in bob.csr \

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

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

-CAcreateserial \

-out bob.crt -days 365

👉 Result: You now have bob.crt (certificate) signed by Kubernetes CA.

**Step 3: 📂 Backup Current kubeconfig**

Before editing kubeconfig, make a backup:

cd ~/.kube

cp config config\_bak

ls

# cache config config\_bak

**Step 4: 🛠️ Add User and Context to kubeconfig**

**4.1 Add the user**

kubectl config set-credentials bob \

--client-certificate=bob.crt \

--client-key=bob.key

**4.2 Add a context for bob**

Check current cluster name:

kubectl config get-clusters

Add context for bob:

kubectl config set-context bob-context \

--cluster=<your-cluster-name> \

--user=bob \

--namespace=raman

👉 Context = (Cluster + User + Namespace).

**4.3 Switch context (optional)**

kubectl config use-context bob-context

kubectl config current-context

**Step 5: 🔎 Test Authentication (No RBAC Yet)**

Try listing pods as bob:

kubectl get pods --as bob

👉 Expected:

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

✅ This means **authentication worked** (API server recognized bob), but **authorization failed** (bob has no RBAC permissions yet).

**Step 6: 🎟️ Create RBAC Role and RoleBinding**

**6.1 Create a Role (permissions)**

role.yml

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: raman

name: raman-pod-reader

rules:

- apiGroups: [""]

resources: ["pods", "secrets"]

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

👉 This allows reading pods & secrets in the raman namespace.

**6.2 Create a RoleBinding (assign role to user)**

rolebinding.yml

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: ramanpod-reader-binding

namespace: raman

subjects:

- kind: User

name: bob

roleRef:

kind: Role

name: raman-pod-reader

apiGroup: rbac.authorization.k8s.io

Apply:

kubectl create -f role.yml

kubectl create -f rolebinding.yml

**Step 7: ✅ Test Access with RBAC**

**7.1 List pods as bob**

kubectl get pods -n raman --as bob

👉 Now bob should succeed in listing pods.

**7.2 Try creating a pod (should fail)**

kubectl run ramanapp4 --image=nginx -n raman --as bob

👉 Expected: Error from server (Forbidden)

**7.3 List secrets (should succeed, as per Role)**

kubectl get secrets -n raman --as bob

**Step 8: 🧹 Cleanup**

**8.1 Remove user and context from kubeconfig**

kubectl config delete-context bob-context

kubectl config delete-user bob

**8.2 Delete certs**

rm -rf bob.key bob.crt bob.csr

**8.3 Switch back to admin context**

kubectl config use-context kubernetes-admin@kubernetes

kubectl config view

**🎓 Key Learnings**

1. **Authentication** = Proving who you are (certs signed by trusted CA).
2. **Authorization (RBAC)** = What you can do (Roles/RoleBindings).
3. You can test access using --as <username> flag (impersonation).
4. Best practice: Assign **groups (/O=developers)** and bind RBAC to groups, not individual users.