**🔐 Lab Guide: Using Kubernetes Secrets with Deployments (Environment Variable Injection)**

**🎯 Lab Objectives**

By the end of this lab, participants will:

* Understand what Kubernetes Secrets are and why they are used.
* Learn how to create a Secret using a YAML manifest.
* Deploy a Pod that reads values from the Secret as environment variables.
* Access the Pod to verify the environment variable injection.

**🧠 Concepts Recap**

**What is a Kubernetes Secret?**

A **Secret** in Kubernetes is an object that holds a small amount of sensitive data such as passwords, tokens, or keys.

* **Benefits**:
  + Prevents sensitive data from being stored in plaintext in container specs.
  + Can be mounted as files or injected as environment variables.
  + Automatically base64-encoded and stored securely in etcd (with encryption at rest if configured).

**Secret Types**

* Opaque (default): For arbitrary key-value pairs.
* kubernetes.io/dockerconfigjson: For Docker registry auth.
* kubernetes.io/tls: For TLS cert/key pairs.

**🧪 Lab Prerequisites**

* A running Kubernetes cluster (in your case, kbe-cluster)
* Namespace raman exists (kubectl create ns raman if not)
* kubectl configured and access to the cluster
* Pod exec access

**📝 Step 1: Encode the Secret Values**

**Example Credentials**

bash

CopyEdit

echo -n 'ramankhanna' | base64

# Output: cmFtYW5raGFubmE=

echo -n 'ramankhanna123' | base64

# Output: cmFtYW5raGFubmExMjM=

**Note:** Avoid the newline with -n for base64.

**📄 Step 2: Create the Secret Manifest**

**secret.yml**

yaml

CopyEdit

apiVersion: v1

kind: Secret

metadata:

name: my-secrets

namespace: raman

type: Opaque

data:

username: cmFtYW5raGFubmE=

password: cmFtYW5raGFubmExMjM=

**Apply the Secret**

bash

CopyEdit

kubectl apply -f secret.yml

kubectl get secrets -n raman

kubectl describe secret my-secrets -n raman

**📦 Step 3: Create the Deployment Using the Secret**

**deploy.yml or secdeploy.yml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp-deployment

namespace: raman

labels:

app: myapp

spec:

replicas: 1

selector:

matchLabels:

app: myapp

type: front-end

template:

metadata:

labels:

app: myapp

type: front-end

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

**Deploy It**

bash

CopyEdit

kubectl apply -f secdeploy.yml

**📌 Step 4: Validate the Deployment**

**Check Pod Status**

bash

CopyEdit

kubectl get pods -n raman

Find your Pod name, e.g. myapp-deployment-58c6b5c57c-tkw7b

**🧾 Step 5: Access the Pod and Verify Environment Variables**

bash

CopyEdit

kubectl exec -it <POD\_NAME> -n raman -- /bin/bash

Example:

bash

CopyEdit

kubectl exec -it myapp-deployment-58c6b5c57c-tkw7b -n raman -- /bin/bash

Once inside:

bash

CopyEdit

echo $SECRET\_USERNAME

# Output: ramankhanna

echo $SECRET\_PASSWD

# Output: ramankhanna123

**🧼 Cleanup (Optional)**

bash

CopyEdit

kubectl delete -f secdeploy.yml

kubectl delete -f secret.yml

**🔍 Extra Validations**

**1. View Secret Content (base64 encoded)**

bash

CopyEdit

kubectl get secret my-secrets -n raman -o yaml

**2. Decode the Secret:**

bash

CopyEdit

kubectl get secret my-secrets -n raman -o jsonpath='{.data.username}' | base64 -d

**📚 Lab Notes and Tips**

* **Security Tip**: Avoid decoding secrets in plaintext where possible. Use RBAC to limit access.
* **Troubleshooting**: If the secret is not found, verify namespace and key names.
* **Base64 Caution**: The base64 command must be used carefully:
  + Always use echo -n (avoids newline).
  + Check if base64 -w 0 is required (on some systems).
* **Namespace Awareness**: Secrets are **namespace-scoped**. Ensure the Pod and the Secret exist in the same namespace.

**✅ Expected Outcomes**

| **Check** | **Description** | **Expected** |
| --- | --- | --- |
| ✅ | Secret Created | my-secrets appears in kubectl get secrets -n raman |
| ✅ | Pod Running | myapp-deployment Pod in Running state |
| ✅ | Env Vars Injected | echo $SECRET\_USERNAME returns ramankhanna |

**📘 Lab Guide: Serving HTML Using Kubernetes ConfigMaps and NGINX**

**🎯 Lab Objectives**

By the end of this lab, participants will be able to:

* Understand Kubernetes ConfigMaps and their use cases.
* Create ConfigMaps from files.
* Mount ConfigMaps as volumes inside containers.
* Serve a static HTML page through an NGINX container using a mounted ConfigMap.
* Swap environments (e.g., prod/dev) by changing the mounted ConfigMap.
* Expose the Pod using a NodePort service and test via curl.

**🧠 Concepts Recap**

**🔸 What is a ConfigMap?**

A **ConfigMap** is an API object used to store **non-confidential configuration data** in key-value pairs. ConfigMaps can be used to:

* Set environment variables in containers.
* Mount configuration files or single keys as files into containers.
* Configure command-line arguments.

🔐 ConfigMaps are **not meant for sensitive data**. For secrets, use Secret.

**🧪 Lab Prerequisites**

* A running Kubernetes cluster (kbe-cluster)
* Namespace raman already created:

bash

CopyEdit

kubectl create ns raman

* kubectl access
* NGINX image accessible by the cluster nodes
* Basic understanding of Pod and Service objects

**📝 Step-by-Step Instructions**

**✅ Step 1: Create HTML Files**

bash

CopyEdit

echo "hell from prod" > prod.html

echo "hell from dev" > dev.html

These files will be used to simulate different environment configurations.

**✅ Step 2: Create ConfigMaps from Files**

**Create prod.cmap:**

bash

CopyEdit

kubectl create configmap prod.cmap --from-file=prod.html -n raman

**Create dev.cmap:**

bash

CopyEdit

kubectl create configmap dev.cmap --from-file=dev.html -n raman

**🔍 Step 3: Verify the ConfigMaps**

bash

CopyEdit

kubectl get configmaps -n raman

kubectl describe configmap prod.cmap -n raman

kubectl describe configmap dev.cmap -n raman

You can also export to YAML for inspection:

bash

CopyEdit

kubectl get configmap prod.cmap -n raman -o yaml > prod-cmap.yml

**✅ Step 4: Create Pod Manifest Using ConfigMap**

**podcm.yml (Serving prod.html as index.html)**

yaml

CopyEdit

apiVersion: v1

kind: Pod

metadata:

name: nginx

namespace: raman

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumeMounts:

- name: rk

mountPath: /usr/share/nginx/html

volumes:

- name: rk

configMap:

name: prod.cmap # <-- You can change this to dev.cmap as needed

items:

- key: prod.html

path: index.html # <-- This is important! nginx looks for index.html

📌 You can switch between environments by changing name: prod.cmap to name: dev.cmap and updating key accordingly.

**🚀 Step 5: Deploy the Pod**

bash

CopyEdit

kubectl apply -f podcm.yml

If you're reapplying with changes:

bash

CopyEdit

kubectl replace --force -f podcm.yml

**✅ Step 6: Validate the Pod**

bash

CopyEdit

kubectl get pods -n raman

kubectl describe pod nginx -n raman

kubectl get pod nginx -n raman -o wide

**🌐 Step 7: Expose the Pod via a NodePort Service**

bash

CopyEdit

kubectl expose pod nginx -n raman --type=NodePort --port=80 --target-port=80 --name=cmsvc

Check service info:

bash

CopyEdit

kubectl get svc -n raman

Look for something like:

nginx

CopyEdit

cmsvc NodePort 10.103.x.x <none> 80:32xxx/TCP

**🔍 Step 8: Test the Output via curl**

bash

CopyEdit

curl <NodeIP>:<NodePort>

Example:

bash

CopyEdit

curl 192.168.190.105:32xxx

# Should output: hell from prod

**🔁 Step 9: Switch from prod to dev**

To switch environments:

1. Edit podcm.yml:
   * Change:

yaml

CopyEdit

configMap:

name: dev.cmap

items:

- key: dev.html

path: index.html

1. Re-deploy:

bash

CopyEdit

kubectl replace --force -f podcm.yml

1. Verify:

bash

CopyEdit

curl <NodeIP>:<NodePort>

# Should output: hell from dev

**🧼 Optional Cleanup**

bash

CopyEdit

kubectl delete pod nginx -n raman

kubectl delete svc cmsvc -n raman

kubectl delete configmap prod.cmap -n raman

kubectl delete configmap dev.cmap -n raman

**📚 Lab Notes**

| **Resource** | **Type** | **Purpose** |
| --- | --- | --- |
| prod.html | File | Static file served in prod env |
| dev.html | File | Static file served in dev env |
| ConfigMap | API object | Mounts the file into NGINX container |
| Pod | API object | Runs NGINX and uses the ConfigMap |
| Service | API object | Exposes the pod to the network |

**✅ Expected Outcome**

| **Task** | **Verification** |
| --- | --- |
| ConfigMaps created | kubectl get cm -n raman |
| Pod uses correct HTML file | curl <NodePort> returns content |
| Can switch prod/dev easily | Modify podcm.yml + redeploy |

**Kubernetes Lab Guide: NFS-Based Persistent Storage with Deployment**

This lab will walk you through setting up an NFS-based PersistentVolume (PV) and PersistentVolumeClaim (PVC) in Kubernetes. It uses an external NFS server to store data and demonstrates how to mount the storage into an NGINX deployment running in a namespace raman.

**🧰 Prerequisites**

* Kubernetes cluster up and running
* kubectl configured with appropriate context
* nfs-common package installed on nodes that will mount the volume
* Access to an NFS server (e.g., 172.31.17.79)

**1️⃣ NFS Setup (on client nodes)**

sudo apt-get update

sudo apt-get install -y nfs-common

Create a mount point and mount the NFS export:

sudo mkdir -p /efs

sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,noresvport 172.31.17.79:/ /efs

🔍 Verify the mount:

df -h

ls -l /efs

**2️⃣ Kubernetes PV and PVC Configuration**

**Create PersistentVolume (pv.yml)**

apiVersion: v1

kind: PersistentVolume

metadata:

name: raman-website

spec:

capacity:

storage: 11Mi

accessModes:

- ReadWriteMany

mountOptions:

- hard

- nfsvers=4.1

nfs:

path: /

server: 172.31.17.79

Apply:

kubectl create -f pv.yml

kubectl get pv

**Create PersistentVolumeClaim (pvc.yml)**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: raman-demo

namespace: raman

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Mi

volumeName: raman-website

Apply:

kubectl create -f pvc.yml

kubectl get pvc -n raman

**3️⃣ Create NGINX Deployment with Volume Mount (pvdeploy.yml)**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nfs-raman

namespace: raman

spec:

replicas: 5

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: raman-demo

Apply:

kubectl create -f pvdeploy.yml

kubectl get pods -n raman

**4️⃣ Verify Pod Access to NFS**

kubectl get pods -n raman

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

cd /usr/share/nginx/deploydata

ls -l

🧪 You can test read/write if NFS permissions are correct.

**5️⃣ Cleanup (Optional)**

kubectl delete deploy --all -n raman

kubectl delete pvc --all -n raman

kubectl delete pv raman-website

**✅ Summary**

* You mounted an external NFS share on your Kubernetes worker
* You created a PV backed by that NFS share
* A PVC requested storage and was bound to the PV
* You deployed an NGINX app that used that volume for file serving

This setup is ideal for:

* Web apps with shared read-write storage
* CI/CD pipelines requiring intermediate shared storage
* Stateful apps across pods

**📁 Files Reference**

* pv.yml: NFS-backed PersistentVolume
* pvc.yml: PersistentVolumeClaim
* pvdeploy.yml: NGINX Deployment mounting the PVC

**🔐 Kubernetes Client Certificate-Based Authentication – Explained with Analogy**

**📌 Real World Analogy: Badge and ID System**

| **Real World Concept** | **Kubernetes Equivalent** |
| --- | --- |
| Bob’s badge (personal info) | bob.key (private key) |
| Reception/security | Kubernetes administrator (who handles CSRs) |
| Chip-attached ID card | bob.crt (signed certificate by Kubernetes CA) |
| Central building authority | Kubernetes API server with ca.crt |
| Raman with fake badge | raman.key + fake.crt (not signed by API server) |

**🧪 Lab Flow: How Certificate-Based Authentication Works**

**Step 1: User Key Generation (Badge)**

Each user first generates their **private key** (.key). This is like a **badge with basic identity**.

bash

CopyEdit

openssl genrsa -out bob.key 2048

**Step 2: Certificate Signing Request (CSR)**

Bob uses his private key to generate a **CSR** (just like requesting reception to validate and approve his badge to get an ID card):

bash

CopyEdit

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

* CN=bob → Common Name; becomes the username in Kubernetes.
* O=dev-team → Organization; becomes the Kubernetes group.

**Step 3: Create Kubernetes CSR Object**

Now the Kubernetes admin takes Bob’s .csr and creates a Kubernetes CertificateSigningRequest resource.

yaml

CopyEdit

apiVersion: certificates.k8s.io/v1

kind: CertificateSigningRequest

metadata:

name: bob-csr

spec:

groups:

- system:authenticated

request: <base64\_encoded\_bob.csr>

signerName: kubernetes.io/kube-apiserver-client

usages:

- client auth

👉 To convert .csr to base64:

bash

CopyEdit

cat bob.csr | base64 | tr -d '\n'

Then replace <base64\_encoded\_bob.csr> in the YAML.

**Step 4: Submit the CSR**

bash

CopyEdit

kubectl apply -f bob-csr.yaml

**Step 5: Approve the CSR (Admin)**

An admin **must approve** the CSR so that the **API server can sign it with its ca.crt**.

bash

CopyEdit

kubectl certificate approve bob-csr

This mimics **attaching a chip and issuing a valid ID card**.

**Step 6: Download the Signed Certificate**

bash

CopyEdit

kubectl get csr bob-csr -o jsonpath='{.status.certificate}' | base64 --decode > bob.crt

Bob now has:

* bob.key → his private key (badge)
* bob.crt → the signed certificate (valid ID)

**Step 7: Configure kubeconfig for Bob**

bash

CopyEdit

kubectl config set-credentials bob --client-certificate=bob.crt --client-key=bob.key

kubectl config set-context bob-context --cluster=kubernetes --namespace=default --user=bob

kubectl config use-context bob-context

**❌ What if someone fakes the badge? (e.g., Raman)**

If **Raman** generates a private key and **signs his own certificate** or uses a certificate **not signed by Kubernetes CA (ca.crt)**, **API Server will reject** his request because:

1. His certificate signature doesn’t match Kubernetes ca.crt.
2. He doesn't have a valid registration.

That’s like showing up to the office with a fake badge and chip: **access denied**.

**✅ Final Summary Table**

| **User** | **Key** | **Certificate** | **Signed by** | **Access Granted?** |
| --- | --- | --- | --- | --- |
| Bob | bob.key | bob.crt | Kubernetes CA | ✅ Yes |
| Reshma | reshma.key | reshma.crt | Kubernetes CA | ✅ Yes |
| Raman | raman.key | fake.crt | ❌ Not Kubernetes CA | ❌ No |

**🛡️ Bonus: View Kubernetes CA Public Key**

bash

CopyEdit

kubectl config view --raw -o jsonpath='{.clusters[0].cluster.certificate-authority-data}' | base64 --decode

**📘 Suggested Hands-On Lab Activities**

1. ✅ Generate a valid CSR and get it signed.
2. ✅ Try using a signed cert in a custom kubeconfig.
3. ❌ Simulate a fake cert and observe API server rejection.
4. 🔒 Explore RBAC rules after successful authentication.