**🧑‍💻 Lab Guide: Deploy Fluentd as a DaemonSet on Kubernetes**

**🎯 Objective**

By the end of this lab, you will:

✅ Understand **DaemonSets** and their use-cases in Kubernetes.  
✅ Deploy a **Fluentd-Elasticsearch** logging agent on all nodes using a DaemonSet.  
✅ Handle node taints for control plane scheduling.  
✅ Validate that the DaemonSet runs one Pod per node.

**📖 Background**

* A **DaemonSet** ensures a **copy of a pod runs on all (or selected) nodes** in the cluster. Common use-cases:
  + Log collection (e.g., Fluentd, Filebeat).
  + Monitoring agents (e.g., Prometheus Node Exporter).
  + Network plugins (e.g., Cilium, Calico).
* **Fluentd** here collects logs from /var/log and ships them to Elasticsearch.
* The control-plane nodes are tainted to **prevent workloads** from scheduling. We’ll configure tolerations to allow DaemonSet pods on control-plane nodes too.

**🛠 Lab Setup**

**🖥 Prerequisites**

* A running Kubernetes cluster (tested on v1.24+).
* kubectl configured and pointing to your cluster.
* Namespace raman created:

bash

CopyEdit

kubectl create namespace raman

**🚀 Step 1: Inspect Node Taints**

Kubernetes taints control-plane nodes to avoid workloads on them:

bash

CopyEdit

kubectl describe nodes | grep -i taint

📄 Example Output:

bash

CopyEdit

Taints: node-role.kubernetes.io/master:NoSchedule

node-role.kubernetes.io/control-plane:NoSchedule

💡 **Why?**  
This prevents pods from scheduling on the master node unless you explicitly allow it.

**✍️ Step 2: Create the DaemonSet Manifest**

In the directory ~/raman, create a file called daemon.yml:

yaml

CopyEdit

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-elasticsearch

namespace: raman

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: fluentd-elasticsearch

template:

metadata:

labels:

name: fluentd-elasticsearch

spec:

tolerations:

# Allow pods to run on master/control-plane nodes

- key: node-role.kubernetes.io/control-plane

operator: Exists

effect: NoSchedule

- key: node-role.kubernetes.io/master

operator: Exists

effect: NoSchedule

containers:

- name: fluentd-elasticsearch

image: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

volumeMounts:

- name: rkvol

mountPath: /var/log

volumes:

- name: rkvol

hostPath:

path: /var/log

📖 **Key Concepts**

* tolerations: Lets pod tolerate taints on master/control-plane nodes.
* hostPath: Mounts the node’s /var/log directory into the container.
* DaemonSet: Ensures a pod on **every node**.

**📝 Step 3: Deploy the DaemonSet**

Run the following command:

bash

CopyEdit

kubectl apply -f daemon.yml

✅ Check deployment:

bash

CopyEdit

kubectl get daemonset -n raman

📄 Example Output:

pgsql

CopyEdit

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE

fluentd-elasticsearch 3 3 3 3 3

**🔍 Step 4: Validate the DaemonSet Pods**

Check Pods created by the DaemonSet:

bash

CopyEdit

kubectl get pods -n raman -o wide

📄 Example Output:

nginx

CopyEdit

NAME READY STATUS NODE IP AGE

fluentd-elasticsearch-abcde 1/1 Running node1 10.42.0.11 1m

fluentd-elasticsearch-fghij 1/1 Running node2 10.42.0.12 1m

fluentd-elasticsearch-klmno 1/1 Running master 10.42.0.10 1m

💡 **What Happened?**

* One pod runs **per node** (including control-plane because of tolerations).

**⚙️ Step 5: Fixing Taint Issues (Optional)**

If your DaemonSet pods are **not scheduling** on a node, check taints again:

bash

CopyEdit

kubectl describe node <node-name> | grep -i taint

Remove unwanted taints if necessary:

bash

CopyEdit

kubectl taint nodes <node-name> app-

Reapply the manifest if needed:

bash

CopyEdit

kubectl replace --force -f daemon.yml

**🧹 Step 6: Clean Up**

When done testing, delete the DaemonSet and namespace:

bash

CopyEdit

kubectl delete -f daemon.yml

kubectl delete namespace raman

**🧠 Key Takeaways**

* **DaemonSet** schedules **one Pod per node**, ideal for node-level agents.
* **Taints & Tolerations** allow scheduling on tainted nodes (e.g., master).
* **hostPath** exposes node directories to Pods (use with caution).

**🧑‍💻 Lab Guide: Deploy and Expose a Containerized Application on Kubernetes**

**🎯 Lab Objective**

By the end of this lab, you will:

✅ Deploy a multi-replica application (rk-deployment) into a Kubernetes cluster.  
✅ Expose the application using:

* **NodePort**
* **ClusterIP**
* **LoadBalancer**  
  ✅ Access the deployed app from outside the cluster.

**🛠 Lab Setup**

**🖥 Prerequisites**

* A running Kubernetes cluster (v1.20+).
* kubectl configured.
* Docker installed locally for initial container run.
* Namespace raman created:

bash

CopyEdit

kubectl create namespace raman

**📖 Step 1: Run a Docker Container Locally**

Before deploying to Kubernetes, run the app as a standalone container:

bash

CopyEdit

docker run --name c1 -v /var/log:/var/log -p 81:80 httpd

✔ **What happens here?**

* -v /var/log:/var/log: Mounts the host's log directory.
* -p 81:80: Maps host port 81 to container port 80.
* httpd: Runs an Apache web server.

💡 Open a browser at http://<docker-host>:81 to verify the app.

**🚀 Step 2: Deploy the App in Kubernetes**

Create deploy.yaml to define the deployment:

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: rk-deployment

namespace: raman

labels:

purpose: trainingdeployment

spec:

replicas: 3

selector:

matchLabels:

rk: rep

template:

metadata:

labels:

rk: rep

spec:

containers:

- name: customcon

image: ramann123/hotstar:latest

ports:

- containerPort: 3000

**✅ Apply Deployment**

bash

CopyEdit

kubectl apply -f deploy.yaml

Verify Pods:

bash

CopyEdit

kubectl get pods -n raman -o wide

✔ **Expected Output**  
3 replicas of your containerized app running, each on potentially different nodes.

**🌐 Step 3: Expose the App with Services**

**🟢 A. NodePort Service**

Create service.yml:

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-np-svc

namespace: raman

spec:

type: NodePort

selector:

rk: rep

ports:

- port: 3000 # Service port (inside cluster)

targetPort: 3000 # Container port

nodePort: 30007 # Exposed on host nodes

📥 **Apply the Service**

bash

CopyEdit

kubectl create -f service.yml

✅ **Verify Service**

bash

CopyEdit

kubectl get svc -n raman

✔ **Expected Output**

pgsql

CopyEdit

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

raman-np-svc NodePort 10.96.142.219 <none> 3000:30007/TCP 1m

🖥 **Access the App**

* http://<NodeIP>:30007
* Replace <NodeIP> with any node’s IP (kubectl get nodes -o wide).

**🟢 B. ClusterIP Service**

Create clusterip-svc.yml:

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-cip-svc

namespace: raman

spec:

type: ClusterIP

selector:

rk: rep

ports:

- port: 3000

targetPort: 3000

📥 **Apply the Service**

bash

CopyEdit

kubectl create -f clusterip-svc.yml

✅ **Verify ClusterIP**

bash

CopyEdit

kubectl get svc -n raman

✔ **Expected Output**

pgsql

CopyEdit

NAME TYPE CLUSTER-IP PORT(S) AGE

raman-cip-svc ClusterIP 10.96.100.124 3000/TCP 1m

💡 **Access:**

* This service is **internal only**. Use kubectl port-forward or a pod inside the cluster to access.

bash

CopyEdit

kubectl port-forward svc/raman-cip-svc 8080:3000 -n raman

Open browser: http://localhost:8080

**🟢 C. LoadBalancer Service**

Create lb-srvc.yml:

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-lb-svc

namespace: raman

spec:

type: LoadBalancer

selector:

rk: rep

ports:

- port: 3000

targetPort: 3000

📥 **Apply the Service**

bash

CopyEdit

kubectl create -f lb-srvc.yml

✅ **Verify LoadBalancer**

bash

CopyEdit

kubectl get svc -n raman

✔ **Expected Output**

pgsql

CopyEdit

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

raman-lb-svc LoadBalancer 10.96.95.144 <pending> 3000/TCP 1m

💡 **Note:**

* On Minikube or bare metal, external IP might stay <pending>. Use minikube tunnel or MetalLB for bare-metal LoadBalancers.

**🕵️‍♂️ Step 4: Validation**

* 🖥 **NodePort:** Test via http://<NodeIP>:30007
* 🖥 **ClusterIP:** Test with port-forward or from a pod.
* 🌍 **LoadBalancer:** Test via External IP (if provisioned).

**🧹 Step 5: Clean Up**

Delete resources after testing:

bash

CopyEdit

kubectl delete -f deploy.yaml

kubectl delete -f service.yml

kubectl delete -f clusterip-svc.yml

kubectl delete -f lb-srvc.yml

kubectl delete namespace raman

**📚 Key Concepts Recap**

| **Service Type** | **Scope** | **Access** |
| --- | --- | --- |
| ClusterIP | Internal cluster only | Pods in the cluster |
| NodePort | Cluster-wide, external | NodeIP:NodePort |
| LoadBalancer | External load-balanced | Provisioned external IP |

**🧑‍💻 Lab Guide: Blue-Green Deployment in Kubernetes**

**🎯 Lab Objective**

By the end of this lab, you will:

✅ Understand **Blue-Green Deployment** strategy in Kubernetes.  
✅ Deploy **two versions** of an application (nginx as *blue*, httpd as *green*).  
✅ Expose the app using a **NodePort Service**.  
✅ Switch traffic from *blue* to *green* with **zero downtime**.

**📖 What is Blue-Green Deployment?**

Blue-Green is a deployment strategy that keeps **two environments (Blue and Green)** running simultaneously.

* **Blue (Active)**: Current production deployment serving traffic.
* **Green (Idle)**: Next version of the application deployed alongside Blue.

👉 **Switch traffic** from Blue → Green by changing the Service selector.

🖼 **Visual Flow**

lua

CopyEdit

Client

|

| +---------------------------+

+-------> Service (raman-np-svc) |

+---------------------------+

|

-------------------------------

| |

Blue Pods Green Pods

(rk-blue-nginx) (rk-green-httpd)

**🛠 Lab Setup**

**🖥 Prerequisites**

* Kubernetes cluster (v1.20+)
* kubectl configured
* Namespace raman created:

bash

CopyEdit

kubectl create namespace raman

**🚀 Step 1: Deploy Blue and Green Applications**

**1️⃣ Create deploy.yaml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: rk-blue-nginx

namespace: raman

labels:

purpose: trainingdeployment

spec:

replicas: 2

selector:

matchLabels:

rk: rep1

template:

metadata:

labels:

rk: rep1

spec:

containers:

- name: customcon

image: nginx

ports:

- containerPort: 80

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: rk-green-httpd

namespace: raman

labels:

purpose: trainingdeployment

spec:

replicas: 2

selector:

matchLabels:

rk: rep2

template:

metadata:

labels:

rk: rep2

spec:

containers:

- name: customcon

image: httpd

ports:

- containerPort: 80

📥 **Apply the Deployments**

bash

CopyEdit

kubectl apply -f deploy.yaml

✅ Verify Pods:

bash

CopyEdit

kubectl get pods -n raman -o wide

✔ Expected Output:

sql

CopyEdit

rk-blue-nginx-xxxxx 1/1 Running

rk-blue-nginx-yyyyy 1/1 Running

rk-green-httpd-zzzzz 1/1 Running

rk-green-httpd-aaaaa 1/1 Running

**🌐 Step 2: Expose with NodePort Service**

Create service.yml:

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-np-svc

namespace: raman

spec:

type: NodePort

selector:

rk: rep1 # Initially point to Blue

ports:

- port: 80 # Service port

targetPort: 80 # Container port

nodePort: 30007 # Host node port

📥 **Apply the Service**

bash

CopyEdit

kubectl apply -f service.yml

✅ Check Service:

bash

CopyEdit

kubectl get svc -n raman

✔ Expected Output:

pgsql

CopyEdit

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

raman-np-svc NodePort 10.96.15.42 <none> 80:30007/TCP 1m

**🖥 Step 3: Access the App**

* Open browser:  
  http://<NodeIP>:30007
* Or use curl:

bash

CopyEdit

curl http://<NodeIP>:30007

✅ You should see **Nginx welcome page** (Blue).

**🔄 Step 4: Switch Traffic to Green**

Modify service.yml to point to Green by changing the selector:

yaml

CopyEdit

selector:

rk: rep2 # Switch from rep1 to rep2

📥 Re-apply the Service:

bash

CopyEdit

kubectl apply -f service.yml

✅ Confirm Service points to Green:

bash

CopyEdit

kubectl describe svc raman-np-svc -n raman | grep Selector

✔ Expected Output:

makefile

CopyEdit

Selector: rk=rep2

**🖥 Step 5: Verify Green App**

* Open browser again:  
  http://<NodeIP>:30007
* Or use curl:

bash

CopyEdit

curl http://<NodeIP>:30007

✅ You should see **Apache HTTP Server page** (Green).

**🧹 Step 6: Clean Up**

When done testing:

bash

CopyEdit

kubectl delete -f deploy.yaml

kubectl delete -f service.yml

kubectl delete namespace raman

**🧠 Key Concepts Recap**

| **Component** | **Description** |
| --- | --- |
| Blue Deployment | Nginx pods serving production traffic. |
| Green Deployment | Httpd pods deployed for new version/testing. |
| Service Selector Switch | Dynamically shifts traffic from Blue to Green. |
| NodePort Service | Exposes app on <NodeIP>:30007 for external access. |

**🧑‍💻 Lab Guide: Canary Deployment in Kubernetes**

**🎯 Lab Objective**

By the end of this lab, you will:

✅ Understand **Canary Deployment** in Kubernetes.  
✅ Deploy two application versions simultaneously.  
✅ Gradually shift traffic between old and new versions.  
✅ Use **replica adjustments** to achieve traffic split.

**📖 What is Canary Deployment?**

* Canary deployments roll out a **new application version (canary)** to **a small percentage of users** while the rest continue using the **stable version**.
* If no issues arise, traffic is **gradually shifted** from the old version to the new one.

🖼 **Visual Traffic Split Example:**

lua

CopyEdit

Client

|

|---> Service (raman-np-svc)

|

+---> Old App (nginx) - 90%

|

+---> New App (httpd) - 10%

**🛠 Lab Setup**

**🖥 Prerequisites**

* Kubernetes cluster (v1.20+)
* kubectl configured.
* Namespace raman created:

bash

CopyEdit

kubectl create namespace raman

**🚀 Step 1: Deploy Old and New Applications**

**📄 deploy.yaml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: rk-nginx-old

namespace: raman

labels:

purpose: trainingdeployment

spec:

replicas: 4 # Old version replicas

selector:

matchLabels:

rk: rep

template:

metadata:

labels:

rk: rep

spec:

containers:

- name: customcon

image: nginx

ports:

- containerPort: 80

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: rk-httpd-new

namespace: raman

labels:

purpose: trainingdeployment

spec:

replicas: 1 # Canary version replicas

selector:

matchLabels:

rk: rep

template:

metadata:

labels:

rk: rep

spec:

containers:

- name: customcon

image: httpd

ports:

- containerPort: 80

📥 **Apply the Deployments**

bash

CopyEdit

kubectl apply -f deploy.yaml

✅ **Verify Pods:**

bash

CopyEdit

kubectl get pods -n raman -o wide

✔ Expected Output:

sql

CopyEdit

rk-nginx-old-xxxx Running

rk-nginx-old-yyyy Running

rk-nginx-old-zzzz Running

rk-nginx-old-aaaa Running

rk-httpd-new-bbbb Running

**🌐 Step 2: Expose the Applications**

**📄 service.yml**

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-np-svc

namespace: raman

spec:

type: NodePort

selector:

rk: rep # Service sends traffic to Pods with rk=rep

ports:

- port: 80

targetPort: 80

nodePort: 30007

📥 **Apply the Service**

bash

CopyEdit

kubectl apply -f service.yml

✅ **Check Service:**

bash

CopyEdit

kubectl get svc -n raman

✔ Example Output:

pgsql

CopyEdit

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

raman-np-svc NodePort 10.96.100.123 <none> 80:30007/TCP 1m

**🖥 Step 3: Test Initial Deployment**

* Open browser:  
  http://<NodeIP>:30007
* Or use curl:

bash

CopyEdit

curl http://<NodeIP>:30007

🔄 Repeat curl multiple times:  
✅ You should see **nginx** responses **most of the time** and **httpd** occasionally.

**📊 Step 4: Traffic Split Adjustments**

**🎯 Scenario 1: 90% Old, 10% New**

Adjust replica counts:

bash

CopyEdit

kubectl scale deployment/rk-nginx-old --replicas=9 -n raman

kubectl scale deployment/rk-httpd-new --replicas=1 -n raman

✅ **Verify Pods**

bash

CopyEdit

kubectl get pods -n raman

✔ Expected: 9 nginx pods, 1 httpd pod.  
💡 Service distributes traffic roughly **90:10**.

**🎯 Scenario 2: 80% Old, 20% New**

bash

CopyEdit

kubectl scale deployment/rk-nginx-old --replicas=8 -n raman

kubectl scale deployment/rk-httpd-new --replicas=2 -n raman

✅ **Verify Pods**

bash

CopyEdit

kubectl get pods -n raman

✔ Expected: 8 nginx pods, 2 httpd pods.  
💡 Traffic shifts to **80:20**.

**🎯 Scenario 3: 50:50 Split**

bash

CopyEdit

kubectl scale deployment/rk-nginx-old --replicas=5 -n raman

kubectl scale deployment/rk-httpd-new --replicas=5 -n raman

✅ **Test**

bash

CopyEdit

curl http://<NodeIP>:30007

Repeat multiple times to verify equal distribution.

**🎯 Scenario 4: Full Rollout (100% New)**

bash

CopyEdit

kubectl scale deployment/rk-nginx-old --replicas=0 -n raman

kubectl scale deployment/rk-httpd-new --replicas=10 -n raman

✅ All traffic now flows to **httpd** pods.

**🧹 Step 5: Clean Up**

Delete resources after testing:

bash

CopyEdit

kubectl delete -f deploy.yaml

kubectl delete -f service.yml

kubectl delete namespace raman

**🧠 Key Concepts Recap**

| **Stage** | **Old App (nginx)** | **New App (httpd)** | **Traffic Split** |
| --- | --- | --- | --- |
| Initial | 100% | 0% | 100:0 |
| Canary Phase 1 | 90% | 10% | 90:10 |
| Canary Phase 2 | 80% | 20% | 80:20 |
| Halfway | 50% | 50% | 50:50 |
| Full Rollout | 0% | 100% | 0:100 |

**🧑‍💻 Lab Guide: Managing Kubernetes Secrets**

**🎯 Lab Objective**

By the end of this lab, you will:

✅ Understand **Kubernetes Secrets** and how they secure sensitive data.  
✅ Create and apply secrets.  
✅ Access secrets in Pods via:

* **Mounted volumes (files)**.
* **Environment variables**.  
  ✅ Verify and use secrets inside running containers.

**📖 What are Kubernetes Secrets?**

A **Secret** is an object that contains small amounts of sensitive data such as passwords, tokens, or SSH keys.

Kubernetes encodes secrets **(base64)** and stores them securely in etcd.

**🛠 Lab Setup**

**🖥 Prerequisites**

* Kubernetes cluster (v1.20+)
* kubectl configured
* Namespace (optional):

bash

CopyEdit

kubectl create namespace secrets-lab

**🚀 Step 1: Encode the Data**

Kubernetes requires secrets to be base64-encoded.

**✅ Encode username**

bash

CopyEdit

echo 'ramankhanna' | base64

✔ Output:

makefile

CopyEdit

cmFtYW5raGFubmE=

**✅ Encode password**

bash

CopyEdit

echo 'ramankhanna123' | base64

✔ Output:

makefile

CopyEdit

cmFtYW5raGFubmExMjM=

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

**secret.yaml**

yaml

CopyEdit

apiVersion: v1

kind: Secret

metadata:

name: my-secrets

type: Opaque

data:

username: cmFtYW5raGFubmE=

password: cmFtYW5raGFubmExMjM=

📥 Apply the secret:

bash

CopyEdit

kubectl apply -f secret.yaml

✅ Verify:

bash

CopyEdit

kubectl get secrets

✔ Example Output:

pgsql

CopyEdit

NAME TYPE DATA AGE

my-secrets Opaque 2 5s

✅ Describe the secret:

bash

CopyEdit

kubectl describe secret my-secrets

✔ Note: You’ll see **base64-encoded data** in kubectl get secret -o yaml.

**🚀 Step 3: Access Secret as a Volume in Pod**

**📄 secret-pod.yaml**

yaml

CopyEdit

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod1

labels:

app: myapp

spec:

containers:

- name: httpd-container

image: httpd

volumeMounts:

- name: credentials

mountPath: /tmp/creds

readOnly: true

volumes:

- name: credentials

secret:

secretName: my-secrets

📥 Apply Pod:

bash

CopyEdit

kubectl apply -f secret-pod.yaml

✅ Verify Pod:

bash

CopyEdit

kubectl get pods

✅ Inspect Volume:

bash

CopyEdit

kubectl exec -it myapp-pod1 -- /bin/bash

Inside Pod:

bash

CopyEdit

ls /tmp/creds

cat /tmp/creds/username

cat /tmp/creds/password

✔ Output:

nginx

CopyEdit

ramankhanna

ramankhanna123

**🚀 Step 4: Access Secret as Environment Variables**

**📄 Append to secret-pod.yaml**

yaml

CopyEdit

---

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod2

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

📥 Apply Pod:

bash

CopyEdit

kubectl apply -f secret-pod.yaml

✅ Verify Pod:

bash

CopyEdit

kubectl get pods

✅ Check Environment Variables:

bash

CopyEdit

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

Inside Pod:

bash

CopyEdit

echo $SECRET\_USERNAME

echo $SECRET\_PASSWD

✔ Output:

nginx

CopyEdit

ramankhanna

ramankhanna123

**🧹 Step 5: Clean Up**

When done:

bash

CopyEdit

kubectl delete -f secret.yaml

kubectl delete -f secret-pod.yaml

**🧠 Key Concepts Recap**

| **Method** | **How Secrets Appear in Pod** |
| --- | --- |
| **Volume** | Files in a directory (/tmp/creds/username) |
| **Environment Var** | Variables ($SECRET\_USERNAME, $SECRET\_PASSWD) |

✔ Secrets in Kubernetes are **base64-encoded** by default for transport and storage but **not encrypted**. For encryption at rest, enable **KMS (Key Management System)** in Kubernetes.

**🧑‍💻 Lab Guide: Using Kubernetes ConfigMaps**

**🎯 Lab Objective**

By the end of this lab, you will:

✅ Understand **ConfigMaps** in Kubernetes and how they differ from Secrets.  
✅ Create ConfigMaps for **prod** and **dev** environments.  
✅ Mount ConfigMaps into a Pod as files.  
✅ Switch configurations dynamically by modifying the Pod manifest.

**📖 What is a ConfigMap?**

A **ConfigMap** is a Kubernetes API object used to store **non-sensitive, key-value pairs** like configuration data, environment-specific values, or files.

Unlike Secrets (which are base64-encoded), ConfigMaps store **plain-text data** for runtime configuration.

📝 Example Use Cases:

* Application settings (app.conf, prod.html, dev.html).
* Environment-specific files injected into containers.
* Command-line arguments for apps.

**🛠 Lab Setup**

**🖥 Prerequisites**

* Kubernetes cluster (v1.20+)
* kubectl configured
* Namespace raman created:

bash

CopyEdit

kubectl create namespace raman

**🚀 Step 1: Create Configuration Files**

✅ **Create prod.html**

bash

CopyEdit

echo "hello from prod" > prod.html

✅ **Create dev.html**

bash

CopyEdit

echo "hello from dev" > dev.html

**🚀 Step 2: Create ConfigMaps**

**📄 Create prod ConfigMap**

bash

CopyEdit

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

✅ Verify:

bash

CopyEdit

kubectl get cm -n raman

kubectl describe cm prod.cmap -n raman

✔ Output:

markdown

CopyEdit

Name: prod.cmap

Namespace: raman

Data

====

prod.html:

----

hello from prod

**📄 Create dev ConfigMap**

bash

CopyEdit

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

✅ Verify:

bash

CopyEdit

kubectl get cm -n raman

kubectl describe cm dev.cmap -n raman

**🚀 Step 3: Create a Pod with ConfigMap Mounted**

**📄 podcm.yml**

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 # <-- Mount prod ConfigMap

items:

- key: prod.html

path: index.html # Serve as index.html

📥 Apply Pod:

bash

CopyEdit

kubectl apply -f podcm.yml

✅ Verify Pod:

bash

CopyEdit

kubectl get pods -n raman

**🌐 Step 4: Expose the Pod**

Expose the Pod as a NodePort service:

bash

CopyEdit

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

✅ Check Service:

bash

CopyEdit

kubectl get svc -n raman

✔ Example Output:

pgsql

CopyEdit

NAME TYPE CLUSTER-IP PORT(S) AGE

cmsvc NodePort 10.96.35.120 80:30036/TCP 1m

**🖥 Step 5: Test the Configuration**

Find Node IP:

bash

CopyEdit

kubectl get nodes -o wide

Access the app in browser or curl:

bash

CopyEdit

curl http://<NodeIP>:<NodePort>

✔ Expected Output:

csharp

CopyEdit

hello from prod

**🔄 Step 6: Switch to Dev Config**

Update podcm.yml to use dev.cmap:

yaml

CopyEdit

configMap:

name: dev.cmap # Switch to dev ConfigMap

items:

- key: dev.html

path: index.html

📥 Apply Changes:

bash

CopyEdit

kubectl replace --force -f podcm.yml

✅ Test:

bash

CopyEdit

curl http://<NodeIP>:<NodePort>

✔ Expected Output:

csharp

CopyEdit

hello from dev

**🧹 Step 7: Clean Up**

bash

CopyEdit

kubectl delete -f podcm.yml

kubectl delete svc cmsvc -n raman

kubectl delete configmap prod.cmap dev.cmap -n raman

**🧠 Key Concepts Recap**

| **Feature** | **ConfigMap** |
| --- | --- |
| Data Type | Non-sensitive plain-text |
| Mount as File | Yes |
| Use as Env Var | Yes |
| Use Case | Application configs, environment switches. |

**🛡 Bonus Challenge**

✅ Mount multiple ConfigMaps into a Pod (prod.cmap + dev.cmap).  
✅ Expose configuration keys as environment variables.  
✅ Hot-reload configuration in Nginx using a **sidecar container**.

**🔥 Lab Flow Diagram**

lua

CopyEdit

ConfigMap(prod.cmap) ----+

|

v

Mounted Volume

|

v

Nginx Container --> Serves prod.html as index.html