**🔧 LAB GUIDE: Exposing Containerized Applications Using Docker & Kubernetes Services**

**🧩 PART 1: Docker - Run a Web Application in a Container**

**✅ Objective:**

Run an httpd (Apache) web server container and access it through the host machine’s IP and port.

**🧪 Steps:**

bash

CopyEdit

docker run --name c1 -p 8082:80 httpd

* --name c1: Names the container c1.
* -p 8082:80: Maps port 80 of the container (default httpd port) to port 8082 on the host.
* httpd: The image used (Apache HTTP Server).

**🎯 Outcome:**

You can now access the Apache web server on http://<HOST-IP>:8082.

Use curl http://localhost:8082 or a browser to verify the response.

**🧩 PART 2: Kubernetes - Run & Expose Application Pods**

**🔹 Pre-checks**

bash

CopyEdit

k get nodes

k get pods -A

k get deploy -A

These commands verify the health of your cluster and workloads.

**🔹 Find all Service-related resources:**

bash

CopyEdit

k api-resources | grep -i service

This filters API resources related to services like services, endpoints, serviceaccounts, etc.

**🚀 Step-by-Step Application Deployment**

**1️⃣ Run a Pod using httpd image**

bash

CopyEdit

k run ramanapp -n raman --image=httpd

* k run: Creates a pod from the image.
* -n raman: Target namespace raman.
* --image=httpd: Uses Apache image.

Validate:

bash

CopyEdit

k get pods -n raman -o wide

**2️⃣ Curl into Pod IP (Internal Test)**

Use pod IP (from -o wide) and test:

bash

CopyEdit

curl <POD-IP>:80

If it succeeds, pod is running and serving HTTP.

**🌐 PART 3: Exposing the Application with Kubernetes Services**

**🔸 Option 1: NodePort Service**

bash

CopyEdit

k expose pod ramanapp -n raman \

--type=NodePort \

--target-port=80 \

--port=80 \

--name=extpodsvc

* **NodePort** exposes service on a port on each node.
* Accessible at: http://<Node-IP>:<NodePort>

bash

CopyEdit

k get svc -n raman

Look under PORT(S) column like 80:32xxx/TCP → use 32xxx.

Delete service:

bash

CopyEdit

k delete svc extpodsvc -n raman

**🔸 Option 2: LoadBalancer Service**

bash

CopyEdit

k expose pod ramanapp -n raman \

--type=LoadBalancer \

--target-port=80 \

--port=80 \

--name=extpodsvc

* **LoadBalancer** is ideal in cloud platforms; uses external load balancer.
* May need MetalLB or cloud integration for local clusters.

bash

CopyEdit

k get svc -n raman

Again, test with EXTERNAL-IP if supported.

Delete service:

bash

CopyEdit

k delete svc extpodsvc -n raman

**🔄 Repeat Using ClusterIP (Internal Only)**

bash

CopyEdit

k expose pod ramanapp -n raman \

--type=ClusterIP \

--port=80 \

--target-port=80 \

--name=intpodsvc

* ClusterIP is **only accessible from inside the cluster**.

bash

CopyEdit

k get svc -n raman

Test with:

bash

CopyEdit

k run busybox --image=busybox -it --rm --restart=Never -- sh

# Inside pod shell:

wget -qO- http://intpodsvc.raman.svc.cluster.local

**🧬 SUMMARY OF SERVICE TYPES:**

| **Service Type** | **Scope** | **Access URL** | **Use Case** |
| --- | --- | --- | --- |
| ClusterIP | Internal | svc-name.namespace.svc.cluster.local | Microservice communication |
| NodePort | External | http://<Node-IP>:<NodePort> | Basic external access |
| LoadBalancer | External | http://<External-IP>:<Port> | Production cloud deployments |

**📂 Reference Repository:** [**GitHub - E2E Front-Backend Kubernetes**](https://github.com/ramannkhanna2/kubernetes_end2end_front-backend)

If the goal is to go deeper into multi-component apps (frontend/backend), the repo can be used to build:

* Multi-tier deployments.
* Ingress for domain-based routing.
* ConfigMaps & Secrets for environment variables.
* Deployment strategies (RollingUpdate, Blue-Green).

**🧪 LAB GUIDE: Deploy and Expose an NGINX Demo Application Using Kubernetes Deployment + NodePort Service**

**🧩 Lab Objective**

You will:

* Deploy a web application using the nginxdemos/hello image.
* Expose the application externally using a NodePort service.
* Validate service accessibility via Node IP and port.
* Understand how deployments and services link via labels and selectors.

**⚙️ Prerequisites**

Ensure:

* A working Kubernetes cluster (kubectl configured).
* A namespace raman already exists, or create it:

bash

CopyEdit

kubectl create ns raman

**📁 YAML Definition: depservice.yml**

**👇 File Breakdown**

yaml

CopyEdit

# --- DEPLOYMENT DEFINITION ---

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

namespace: raman

labels:

app: nginx # Optional high-level label

spec:

replicas: 4

selector:

matchLabels:

app: tr # SELECTS PODS WITH app=tr

template:

metadata:

labels:

app: tr # LABEL THAT MATCHES THE SELECTOR ABOVE

spec:

containers:

- name: nginx

image: nginxdemos/hello

ports:

- containerPort: 80

---

# --- SERVICE DEFINITION ---

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

selector:

app: tr # THIS LINKS TO PODS LABELED app=tr

ports:

- port: 80

targetPort: 80

nodePort: 30007 # CUSTOM NODEPORT (within 30000–32767 range)

**📌 Key Concepts**

| **Component** | **Purpose** |
| --- | --- |
| Deployment | Manages replica sets of pods, self-healing, declarative management |
| Service | Stable network access to Pods, load-balancing, discovery |
| NodePort | Exposes service on <NodeIP>:<NodePort> |
| Selector | Binds services to pods with matching labels |

**🚀 Step-by-Step Lab Execution**

**✅ Step 1: Clean Up Previous Deployments (if applicable)**

bash

CopyEdit

kubectl delete pods --all -n raman

kubectl delete svc --all -n raman

**✅ Step 2: Apply Deployment and Service YAML**

bash

CopyEdit

kubectl apply -f depservice.yml

Or for initial creation:

bash

CopyEdit

kubectl create -f depservice.yml

**✅ Step 3: Verify Deployment and Pods**

bash

CopyEdit

kubectl get pods -n raman

kubectl get pods -o wide -n raman

Look for:

* READY status showing all replicas are up.
* Each pod running the nginxdemos/hello image.

**✅ Step 4: Verify Service Exposure**

bash

CopyEdit

kubectl get svc -n raman

kubectl describe svc raman-service -n raman

Check for:

* Type: NodePort
* NodePort: 30007
* Endpoints: should list pod IPs

**✅ Step 5: Validate via Endpoints**

bash

CopyEdit

kubectl get ep -n raman

Expected: List of IPs corresponding to the nginx pods. This confirms correct label matching and endpoint registration.

**🌐 Access the Application**

Find the **Node IP**:

bash

CopyEdit

kubectl get nodes -o wide

Then test using curl or browser:

bash

CopyEdit

curl http://<NODE-IP>:30007

You should see the demo response from the nginxdemos/hello container.

**🔍 Troubleshooting Tips**

| **Issue** | **Check/Resolution** |
| --- | --- |
| No response on curl/browser | Ensure the Node IP is reachable, and port 30007 is open |
| Service has no endpoints | Make sure pod labels match service selector (app: tr) |
| Pods not running | Use kubectl describe pod or kubectl logs to debug container issues |
| NodePort not accessible | On cloud VMs, check firewall rules or security groups allowing NodePort traffic |

**🧪 LAB VARIANTS & EXPANSION**

* **Use ClusterIP and test with busybox pod inside the cluster**.
* **Switch to LoadBalancer** (requires MetalLB in local clusters).
* **Use readiness and liveness probes for better lifecycle management**.
* **Expose the service via Ingress** with hostname-based routing.

**📘 OPTIONAL: Clean-up**

bash

CopyEdit

kubectl delete -f depservice.yml

**🧪 LAB GUIDE: Blue-Green Deployment with Label-Based Traffic Switching**

**🎯 Lab Objective**

You will:

* Deploy two versions of a web application: **Blue (nginx)** and **Green (httpd)**.
* Expose them via a shared NodePort service.
* Control traffic routing between versions using label-based selectors.
* Test and verify the deployment strategy.

**⚙️ Prerequisites**

* Kubernetes cluster ready and accessible (kubectl configured)
* Namespace raman must exist:

bash

CopyEdit

kubectl create ns raman

**📂 Files Overview**

**deploy.yml**

This file defines **two deployments**:

* **Blue (nginx:latest)** labeled ap: ng
* **Green (httpd)** labeled ap: ht

yaml

CopyEdit

---

# BLUE DEPLOYMENT

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-blue-original-dep

namespace: raman

labels:

app: web2-prod-app

spec:

replicas: 5

selector:

matchLabels:

ap: ng

template:

metadata:

labels:

ap: ng

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

---

# GREEN DEPLOYMENT

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-green-testing-dep

namespace: raman

labels:

app: web2-prod-app2

spec:

replicas: 5

selector:

matchLabels:

ap: ht

template:

metadata:

labels:

ap: ht

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

**service.yml**

A NodePort service pointing to label ap: ng, which initially routes all traffic to the **Blue Deployment**.

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

selector:

ap: ng # This routes to the BLUE (nginx) deployment

ports:

- port: 80

targetPort: 80

nodePort: 30007

**🧑‍🔬 Step-by-Step Instructions**

**✅ Step 1: Clean Existing Deployments (Optional)**

bash

CopyEdit

kubectl delete deploy --all -n raman

kubectl delete svc --all -n raman

**✅ Step 2: Apply Deployments**

bash

CopyEdit

kubectl apply -f deploy.yml

Confirm deployments and pod statuses:

bash

CopyEdit

kubectl get deploy -n raman

kubectl get pods -n raman -o wide

**✅ Step 3: Apply Service (Initially routes to BLUE deployment)**

bash

CopyEdit

kubectl apply -f service.yml

Check service status:

bash

CopyEdit

kubectl get svc -n raman

kubectl describe svc raman-service -n raman

**✅ Step 4: Access the Service**

1. Get a node IP:

bash

CopyEdit

kubectl get nodes -o wide

1. Test externally via:

bash

CopyEdit

curl http://<NODE-IP>:30007

1. You should see the **nginx default welcome page**, confirming the BLUE version is active.

**🔁 Traffic Switch to GREEN Deployment**

To switch traffic from BLUE (nginx) to GREEN (httpd):

**🔄 Option A: Patch the Service**

bash

CopyEdit

kubectl patch svc raman-service -n raman \

-p '{"spec": {"selector": {"ap": "ht"}}}'

Now the service will route traffic to the GREEN pods.

Test again:

bash

CopyEdit

curl http://<NODE-IP>:30007

You should now see the **httpd welcome page**.

**🧼 Option B: Edit YAML and Reapply**

Modify service.yml:

yaml

CopyEdit

selector:

ap: ht # instead of ap: ng

Then reapply:

bash

CopyEdit

kubectl apply -f service.yml

**🔍 Validation Steps**

bash

CopyEdit

kubectl get ep -n raman

This shows which pod IPs are currently backing the service.

Check logs from a pod:

bash

CopyEdit

kubectl logs <pod-name> -n raman

**🚫 Clean-Up (Optional)**

bash

CopyEdit

kubectl delete -f deploy.yml

kubectl delete -f service.yml

**📘 Concepts Summary**

| **Component** | **Detail** |
| --- | --- |
| Blue Deployment | nginx, label: ap=ng |
| Green Deployment | httpd, label: ap=ht |
| NodePort SVC | Exposes pods on a static port; selector controls which pods are used |
| Traffic Shift | Done by changing service selector (either patch or YAML update) |
| Endpoint | Shows actual backend pods tied to the service |

**🧪 LAB GUIDE: Simulating Canary Deployment in Kubernetes**

**🎯 Objective**

* Deploy **two versions** of your app simultaneously.
* Simulate **Canary Deployment** using **label-based selectors**.
* Route traffic to both versions **proportionally** using **replica distribution**.
* Test the rollout using a looping curl to observe response variation.

**📂 Files & Configuration**

**🔸 deploy.yml**

yaml

CopyEdit

---

# Stable Version - nginx (v1)

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-v1-dep

namespace: raman

labels:

app: web2-prod-app

spec:

replicas: 2 # Less replicas → less traffic

selector:

matchLabels:

app: canary

template:

metadata:

labels:

app: canary

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

---

# Canary Version - httpd (v2)

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-v2-dep

namespace: raman

labels:

app: web2-prod-app2

spec:

replicas: 3 # More replicas → more traffic

selector:

matchLabels:

app: canary

template:

metadata:

labels:

app: canary

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

---

# Common Service for both deployments

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

selector:

app: canary

ports:

- port: 80

targetPort: 80

nodePort: 30008

**Key Concept:** Both nginx and httpd pods share the same label app: canary. The service selector binds to this label, balancing traffic across **both deployments**.

**🧑‍🔬 Lab Execution Steps**

**✅ Step 1: Create Namespace (if not already present)**

bash

CopyEdit

kubectl create ns raman

**✅ Step 2: Deploy Applications & Service**

bash

CopyEdit

kubectl apply -f deploy.yml

**✅ Step 3: Validate Deployment**

bash

CopyEdit

kubectl get pods -n raman -o wide

kubectl get deploy -n raman

kubectl get svc -n raman

Check if the raman-service is correctly exposing port 30008.

**✅ Step 4: Access the Application via NodePort**

Get your **Node IP**:

bash

CopyEdit

kubectl get nodes -o wide

Use curl or browser:

bash

CopyEdit

curl http://<NODE-IP>:30008

Depending on load balancing (round robin), you'll see **responses from both nginx and httpd** backends.

**🔁 Looped Traffic Testing (Simulating User Hits)**

Run the following to continuously test service load balancing:

bash

CopyEdit

while true; do

clear

curl http://<NODE-IP>:30008/

sleep 1

done

This loop will:

* Send repeated requests to the service.
* Output alternating content from nginx and httpd (based on replica weight).

Expectation:

* Since httpd has 3 replicas and nginx has 2, you’ll roughly get 60% of responses from httpd and 40% from nginx.

**🛠️ Troubleshooting**

| **Command** | **Purpose** |
| --- | --- |
| kubectl describe svc raman-service -n raman | View routing info |
| kubectl get ep -n raman | Check backend pod IPs bound to service |
| kubectl logs <pod-name> -n raman | Inspect request logs |
| kubectl get pods -l app=canary -n raman -o wide | Check running pods selected by the service |

**🚦 Rollout Strategy**

You can simulate real Canary rollout scenarios:

**✅ Gradually Increase Canary Traffic**

1. Start with:
   * nginx → 5 replicas
   * httpd → 1 replica
2. Observe low response count for httpd.
3. Gradually increase httpd replicas:

bash

CopyEdit

kubectl scale deploy raman-v2-dep --replicas=3 -n raman

1. Observe traffic weight shifting.

**❌ Rollback if Issues Found**

bash

CopyEdit

kubectl scale deploy raman-v2-dep --replicas=0 -n raman

This removes the Canary version from the traffic pool instantly.

**📘 Conceptual Summary**

| **Component** | **Description** |
| --- | --- |
| **Canary Deployment** | Releases a new version to a **subset of users** to test in production. |
| **NodePort Service** | Makes app accessible outside the cluster via a static port. |
| **Label Selector** | Both versions share a common label app: canary, so the service can distribute traffic. |
| **Proportional Traffic** | Achieved by adjusting pod replica counts. |

**🔚 Cleanup (Optional)**

bash

CopyEdit

kubectl delete -f deploy.yml

kubectl delete ns raman

**🧪 Kubernetes Lab Guide: Node Affinity, Taints & Tolerations**

**🎯 Objective**

1. Deploy a multi-replica application.
2. Explore how pods are distributed across nodes by default.
3. Use **taints** to repel scheduling from specific nodes.
4. Understand how Kubernetes honors taints via **NoSchedule**, **NoExecute**, and how we can override them using **tolerations**.
5. Learn basic pod rebalancing behavior.

**🧱 Lab Environment Setup**

* You have a namespace: raman
* Nodes: master, w1, w2
* File: dep2.yml — contains the deployment spec
* App: nginx with 10 replicas

**🧾 YAML: dep2.yml**

yaml

CopyEdit

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-v1-dep

namespace: raman

labels:

app: web2-prod-app

spec:

replicas: 10

selector:

matchLabels:

app: canary

template:

metadata:

labels:

app: canary

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

**Note:** This is a basic deployment **without affinity or tolerations**. The scheduler will place pods based on available resources.

**🧑‍🔬 Step-by-Step Lab Activities**

**✅ Step 1: Apply the Deployment**

bash

CopyEdit

kubectl apply -f dep2.yml

**✅ Step 2: Observe Pod Distribution Across Nodes**

bash

CopyEdit

kubectl get pods -n raman -o wide

Then:

bash

CopyEdit

kubectl get pods -n raman -o wide | grep w1

kubectl get pods -n raman -o wide | grep w2

🎯 You'll observe which nodes received the pods. Kubernetes spreads them **by default** using best-effort bin-packing.

**✅ Step 3: View Existing Node Taints**

bash

CopyEdit

kubectl describe nodes | grep -i taint

You may see something like:

bash

CopyEdit

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

This **prevents regular pods** from scheduling on master.

**✅ Step 4: Apply Custom Taints to Repel Pods**

bash

CopyEdit

kubectl taint nodes w1 env=prod:NoSchedule

Verify:

bash

CopyEdit

kubectl describe node w1 | grep -i taint

Expected output:

bash

CopyEdit

Taints: env=prod:NoSchedule

🧠 **Explanation:** No new pods will schedule on w1 unless they explicitly **tolerate** this taint.

**✅ Step 5: Observe Rescheduling Behavior**

Delete a pod:

bash

CopyEdit

kubectl delete pod -n raman <pod-name-running-on-w1>

Check where the replacement pod lands:

bash

CopyEdit

kubectl get pods -n raman -o wide | grep w1

kubectl get pods -n raman -o wide | grep w2

🧪 You should observe **replacement pods avoid w1**.

**✅ Step 6: Add a Stronger Taint (NoExecute) on Another Node**

bash

CopyEdit

kubectl taint nodes w2 env=dev:NoExecute

Verify:

bash

CopyEdit

kubectl describe node w2 | grep -i taint

⛔ **NoExecute** affects both **new** and **existing** pods. Existing pods are **evicted** unless they tolerate the taint.

**✅ Step 7: Observe Eviction**

Watch pod migration:

bash

CopyEdit

kubectl get pods -n raman -o wide | grep w2

🚨 All pods on w2 should be **terminated** and rescheduled to master or another untained node (if possible).

**✅ Step 8: Remove the Taint (if needed)**

bash

CopyEdit

kubectl taint nodes w2 env-

This **removes** the taint from node w2.

**✅ Step 9: Final Checks**

bash

CopyEdit

kubectl get pods -n raman -o wide

kubectl describe node w1 | grep -i taint

kubectl describe node w2 | grep -i taint

kubectl describe node master | grep -i taint

Ensure the state matches your expectations.

**🔍 Key Concepts Explained**

**📌 Taints**

Taints allow a node to **repel pods**.

bash

CopyEdit

kubectl taint nodes <node> key=value:Effect

* **NoSchedule** – Pod will *not* be scheduled on the node unless it tolerates the taint.
* **PreferNoSchedule** – Tries to avoid scheduling but doesn't strictly prevent.
* **NoExecute** – Pod will be *evicted* if it’s already running and doesn’t tolerate it.

**📌 Tolerations**

To schedule pods on tainted nodes, add tolerations:

yaml

CopyEdit

tolerations:

- key: "env"

operator: "Equal"

value: "prod"

effect: "NoSchedule"

This is added inside the spec of the pod template.

**📌 Use Cases**

| **Taints Use Case** | **Real-world Scenario** |
| --- | --- |
| env=prod:NoSchedule | Isolate production-only workloads |
| env=dev:NoExecute | Evict non-tolerant apps from dev-only nodes |
| Master taints | Keep system pods exclusive to master nodes |

**🧼 Cleanup**

bash

CopyEdit

kubectl delete deployment raman-v1-dep -n raman

kubectl taint nodes w1 env-

kubectl taint nodes w2 env-

**📘 Lab Recap Summary**

| **Activity** | **Outcome** |
| --- | --- |
| Deploy app with 10 replicas | See default scheduler behavior |
| Taint w1 with NoSchedule | New pods avoid it |
| Taint w2 with NoExecute | Pods get evicted |
| Observe rescheduling | Kubernetes tries to rebalance based on node availability |

**🧪 Kubernetes Lab Guide: Scheduling Pods on Master Nodes Using nodeSelector and Tolerations**

**🎯 Objective**

By default, Kubernetes **taints master/control-plane nodes** to prevent scheduling non-system pods. In this lab, we will:

1. Deploy a workload using nodeSelector to explicitly target a master node.
2. Add tolerations to allow scheduling despite taints.
3. Observe scheduling behavior.
4. Understand taints, tolerations, and node selectors in-depth.

**🧱 Preconditions**

* A Kubernetes cluster with:
  + Master/Control Plane node (master)
  + Worker nodes (e.g., w1, w2)
* Namespace raman created.
* File: dep2.yml

**📦 Deployment YAML**

yaml

CopyEdit

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-v1-dep

namespace: raman

labels:

app: web2-prod-app

spec:

replicas: 10

selector:

matchLabels:

app: canary

template:

metadata:

labels:

app: canary

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

nodeSelector:

node-role.kubernetes.io/control-plane: "" # Or use "master" based on your setup

tolerations:

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

operator: Exists

effect: NoSchedule

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

operator: Exists

effect: NoSchedule

**🧑‍🔬 Step-by-Step Lab Instructions**

**✅ Step 1: Inspect Node Taints**

bash

CopyEdit

kubectl describe nodes | grep -i taint

Output example:

bash

CopyEdit

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

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

These taints **prevent pods from being scheduled** on master/control-plane nodes **unless** they have **matching tolerations**.

**✅ Step 2: Apply the Deployment**

bash

CopyEdit

kubectl create -f dep2.yml

This creates 10 replicas of an nginx deployment intended to run **only on the master/control-plane node** using nodeSelector and tolerations.

**✅ Step 3: Check Pod Placement**

bash

CopyEdit

kubectl get pods -n raman -o wide

Look under the NODE column. All pods should be scheduled on the master node (control plane).

You can filter just the master node pods:

bash

CopyEdit

kubectl get pods -A -o wide | grep -i master

**✅ Step 4: Review All Deployments and Pods (Optional Cleanup Before Redeploy)**

bash

CopyEdit

kubectl delete deploy -n raman --all

This command clears out the previous deployments before re-creating:

bash

CopyEdit

kubectl create -f dep2.yml

Then verify again:

bash

CopyEdit

kubectl get pods -n raman -o wide

**✅ Step 5: Observe Node-Specific Pod Count**

bash

CopyEdit

kubectl get pods -A -o wide | grep -i master

You should see your nginx pods deployed **only on the master node**.

**🔍 Conceptual Deep Dive**

**📌 nodeSelector**

This tells the scheduler **only to schedule pods on nodes with a specific label**.

yaml

CopyEdit

nodeSelector:

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

To view node labels:

bash

CopyEdit

kubectl get nodes --show-labels

Look for:

bash

CopyEdit

node-role.kubernetes.io/master

node-role.kubernetes.io/control-plane

**📌 Taints on Master Nodes**

By default, master/control-plane nodes have these taints:

bash

CopyEdit

Taints:

node-role.kubernetes.io/master:NoSchedule

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

This prevents **all non-tolerant pods** from being scheduled there.

**📌 Tolerations**

These explicitly allow a pod to be scheduled on a tainted node:

yaml

CopyEdit

tolerations:

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

operator: Exists

effect: NoSchedule

**💡 Pro Tip**

You **must use both**:

* nodeSelector → tells Kubernetes **where** you *want* the pod
* tolerations → tells Kubernetes you're okay running on a tainted node

Without tolerations, your pod will **never** land on a tainted node (even if your nodeSelector matches).

**✅ Cleanup (Optional)**

bash

CopyEdit

kubectl delete -f dep2.yml

To clear deployments and pods.

**📘 Recap Summary**

| **Concept** | **Explanation** |
| --- | --- |
| nodeSelector | Used to direct scheduling to a node with specific label |
| Taints | Repel pods from node unless tolerated |
| Tolerations | Allow pods to tolerate and schedule on tainted nodes |
| Master Node Pods | Normally prohibited, allowed only with toleration |

**🧪 Kubernetes Lab Guide: DaemonSet Deployment (Fluentd/Logging Example)**

**🎯 Objective**

Learn how to:

* Deploy a DaemonSet (e.g., fluentd) across **all nodes**.
* Use **tolerations** to run DaemonSet pods on tainted **master/control-plane** nodes.
* Mount host directories using **hostPath volumes**.
* Compare behavior with Deployment-based scheduling.

**📦 DaemonSet YAML: daemon.yml**

yaml

CopyEdit

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-elasticsearch

namespace: raman

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: raman

template:

metadata:

labels:

name: raman

spec:

tolerations:

# Tolerate taints on master/control-plane to schedule pods there

- 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: nginx # Placeholder for real logging agent image

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

volumeMounts:

- name: rkhanna

mountPath: /var/log

terminationGracePeriodSeconds: 30

volumes:

- name: rkhanna

hostPath:

path: /var/log

**Note:** Here, nginx is used as a placeholder. In production, replace with actual Fluentd/Logstash image.

**🔁 Step-by-Step Lab Instructions**

**✅ Step 1: Prepare Namespace**

Ensure namespace raman exists:

bash

CopyEdit

kubectl create ns raman # if not already created

**✅ Step 2: Inspect Node Taints (Confirm Master Is Tainted)**

bash

CopyEdit

kubectl describe nodes | grep -i taint

Expected taints on master/control-plane:

bash

CopyEdit

Taints:

node-role.kubernetes.io/master:NoSchedule

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

**✅ Step 3: Apply the DaemonSet**

bash

CopyEdit

kubectl create -f daemon.yml

This schedules one fluentd-elasticsearch pod **per node**, including on master, due to tolerations.

**✅ Step 4: Observe DaemonSet Pods**

bash

CopyEdit

kubectl get pods -n raman

kubectl get pods -n raman -o wide

Expected:

* One pod per node (master + all workers).
* Each pod running the nginx image (stand-in for fluentd).

**✅ Step 5: Pod Details & Node Placement**

Inspect an individual pod:

bash

CopyEdit

kubectl describe pod fluentd-elasticsearch-<pod-suffix> -n raman

Look for:

* Assigned Node
* Volume mounts
* Tolerations section
* Resource requests/limits

**✅ Step 6: Check Host Volume Mounts**

On the **node where pod is running**, check if /var/log is mounted inside the pod:

bash

CopyEdit

kubectl exec -it -n raman fluentd-elasticsearch-<pod> -- ls /var/log

This validates that hostPath from the node’s /var/log is visible inside the container.

**✅ Step 7: Delete a DaemonSet Pod & Observe Reconciliation**

bash

CopyEdit

kubectl delete pod -n raman fluentd-elasticsearch-<pod>

Then watch Kubernetes recreate it:

bash

CopyEdit

kubectl get pods -n raman -w

DaemonSet controller ensures **one pod per node**. Deletion triggers immediate recreation.

**✅ Step 8: Cleanup**

bash

CopyEdit

kubectl delete -f daemon.yml

Verify all pods are removed:

bash

CopyEdit

kubectl get pods -n raman

**🧠 Conceptual Breakdown**

**📌 What is a DaemonSet?**

* A **DaemonSet** ensures **one copy of a pod runs on every node** (or specific subset using nodeSelector/affinity).
* Common use cases:
  + Log collectors (fluentd, logstash)
  + Node monitoring agents (prometheus-node-exporter)
  + CNI/CNI plugins (calico, weave)

**📌 Difference Between Deployment vs DaemonSet**

| **Feature** | **Deployment** | **DaemonSet** |
| --- | --- | --- |
| Pod Distribution | Arbitrary nodes (based on scheduler) | One pod per node (or matched subset) |
| Common Use Case | Web apps, APIs, microservices | Log collectors, monitoring agents |
| Scaling | Manual via replicas: | Automatically handled per node |
| Pod Replacement | By controller | Recreated per node automatically |

**📌 Tolerations in DaemonSets**

By default, master/control-plane nodes are **tainted**. Without tolerations, DaemonSet pods **won’t be scheduled** on them.

Your DaemonSet uses:

yaml

CopyEdit

tolerations:

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

operator: Exists

effect: NoSchedule

to **explicitly allow** scheduling there.

**📌 hostPath Volume**

This maps **host machine directories** into the pod:

yaml

CopyEdit

hostPath:

path: /var/log

⚠️ Used with care — grants container access to host filesystem.

**✅ Summary Table**

| **Component** | **Value** |
| --- | --- |
| Kind | DaemonSet |
| Namespace | raman |
| Image | nginx (placeholder) |
| Volume | /var/log mounted from host |
| Tolerations | Master taint toleration |
| Behavior | 1 pod/node across cluster |