**🧪 Kubernetes Deployment Lab Guide – Deploying NGINX in a Custom Namespace (raman)**

**🎯 Objective**

Deploy an NGINX application in a custom namespace (raman) using a Kubernetes Deployment manifest (deploy.yml) and manage its lifecycle using kubectl.

**🧱 Prerequisites**

* A working Kubernetes cluster (e.g., created using kubeadm, Minikube, Kind, etc.)
* kubectl CLI configured to access your cluster
* Basic familiarity with YAML and Kubernetes objects

**📂 Step-by-Step Lab Instructions**

**✅ Step 1: Create Namespace raman**

**Why?** Your Deployment targets the raman namespace. It must exist beforehand.

bash

CopyEdit

kubectl create namespace raman

🔍 **Verify:**

bash

CopyEdit

kubectl get namespaces

**📝 Step 2: Create the Deployment YAML**

Create a file named deploy.yml with the following content:

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: ramandep1

namespace: raman

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

name: rk

template:

metadata:

labels:

name: rk

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

🧠 **Explanation of Important Fields:**

| **Field** | **Description** |
| --- | --- |
| apiVersion: apps/v1 | Uses the stable apps/v1 API for Deployments |
| kind: Deployment | Indicates you're deploying a Deployment controller |
| metadata.namespace | Targets the custom namespace raman |
| replicas: 3 | Requests 3 pod replicas for high availability |
| selector.matchLabels & template.metadata.labels | Must match exactly for the Deployment to manage Pods |
| containers.image: nginx:1.14.2 | Specifies the NGINX version to run |
| containerPort: 80 | Exposes port 80 inside the container |

**🧪 Step 3: Apply the Deployment**

bash

CopyEdit

kubectl apply -f deploy.yml

🧠 apply is preferred over create for idempotency—it updates existing resources.

**🔎 Step 4: Inspect Deployment and Pods**

bash

CopyEdit

kubectl get deploy -n raman

kubectl get pods -n raman

Check rollout status:

bash

CopyEdit

kubectl rollout status deploy ramandep1 -n raman

**🔍 Step 5: Deep Dive into Objects**

**📌 Deployment Details**

bash

CopyEdit

kubectl describe deployment ramandep1 -n raman

Check that:

* Selector and Template labels match
* 3 replicas are running

**📌 ReplicaSet (RS) Info**

bash

CopyEdit

kubectl get rs -n raman

kubectl describe rs -n raman

The RS is created by the Deployment and manages the actual Pods.

**📌 Pod Details**

bash

CopyEdit

kubectl get pods -n raman

kubectl describe pod <pod-name> -n raman

To get the image used in Pods:

bash

CopyEdit

kubectl describe pods -n raman | grep -i image

**🔁 Step 6: Update Deployment (Optional)**

To demonstrate a rolling update, change the image:

bash

CopyEdit

kubectl set image deployment/ramandep1 nginx=nginx:1.16.1 -n raman

Verify the update:

bash

CopyEdit

kubectl rollout status deployment/ramandep1 -n raman

kubectl describe deployment ramandep1 -n raman | grep Image

**🧹 Step 7: Cleanup**

bash

CopyEdit

kubectl delete -f deploy.yml

kubectl delete namespace raman

**📚 Summary of Key Concepts**

| **Concept** | **Description** |
| --- | --- |
| **Deployment** | Manages ReplicaSets to ensure a defined number of Pods are running |
| **ReplicaSet** | Ensures the right number of pod replicas are running |
| **Namespace** | Provides a way to divide cluster resources among multiple users or projects |
| **Rolling Update** | Handled automatically by the Deployment controller |
| **Label Selectors** | Crucial for linking Deployments to their Pods |

**🧠 Pro Tips**

* Always validate YAML before applying:

bash

CopyEdit

kubectl apply -f deploy.yml --dry-run=client -o yaml

* View live logs:

bash

CopyEdit

kubectl logs -f <pod-name> -n raman

* Scale replicas dynamically:

bash

CopyEdit

kubectl scale deploy ramandep1 --replicas=5 -n raman

**🧪 Lab Guide: Managing Deployment Rollouts and ReplicaSets in Kubernetes**

**🎯 Objective**

Understand how a Deployment manages rolling updates, creates multiple ReplicaSets over time, and how to inspect, track, and roll back changes using rollout history and revision management.

**🧱 Prerequisites**

* Kubernetes cluster setup (e.g., kubeadm, Minikube)
* kubectl configured
* Namespace raman already created:

bash

CopyEdit

kubectl create namespace raman

**📁 Step 1: Initial Deployment of App (nginx:1.14.1)**

**1.1 Prepare initial deploy.yml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: ramandep1

namespace: raman

labels:

app: nginx

spec:

replicas: 5

selector:

matchLabels:

name: rk

template:

metadata:

labels:

name: rk

spec:

containers:

- name: nginx

image: nginx:1.14.1 # First version of the app

ports:

- containerPort: 80

**1.2 Create Deployment and record the change:**

bash

CopyEdit

kubectl create -f deploy.yml --record=true

💡 The --record=true flag adds the command to the deployment's annotation (kubernetes.io/change-cause) for history tracking.

**1.3 Check deployment:**

bash

CopyEdit

kubectl get deploy -n raman

kubectl get pods -n raman

kubectl get rs -n raman

**🔍 Step 2: Observe ReplicaSets and Revisions**

**2.1 View created ReplicaSet:**

bash

CopyEdit

kubectl get rs -n raman

You'll see something like:

CopyEdit

ramandep1-58bfb65d5d 5 5 5 ...

This RS controls the initial 5 pods running nginx:1.14.1.

**2.2 Check rollout history:**

bash

CopyEdit

kubectl rollout history deploy ramandep1 -n raman

kubectl rollout history deploy ramandep1 -n raman --revision=1

**🆕 Step 3: Perform an Update (to nginx:1.14.2)**

**3.1 Modify deploy.yml image:**

yaml

CopyEdit

image: nginx:1.14.2

**3.2 Apply the update with recorded history:**

bash

CopyEdit

kubectl apply -f deploy.yml --record=true

**3.3 Watch the update:**

bash

CopyEdit

kubectl rollout status deploy ramandep1 -n raman

**3.4 Check new ReplicaSet:**

bash

CopyEdit

kubectl get rs -n raman

Now you should see two ReplicaSets:

CopyEdit

ramandep1-58bfb65d5d 0 0 0

ramandep1-7f85cb66f7 5 5 5

* Old RS (1.14.1) scaled down
* New RS (1.14.2) scaled up

**3.5 Confirm pods and image:**

bash

CopyEdit

kubectl get pods -n raman

kubectl describe pods -n raman | grep -i image

**📜 Step 4: Review Rollout History**

bash

CopyEdit

kubectl rollout history deploy ramandep1 -n raman

Example output:

csharp

CopyEdit

REVISION CHANGE-CAUSE

1 kubectl create -f deploy.yml --record=true

2 kubectl apply -f deploy.yml --record=true

Get details:

bash

CopyEdit

kubectl rollout history deploy ramandep1 -n raman --revision=1

kubectl rollout history deploy ramandep1 -n raman --revision=2

**↩️ Step 5: Perform a Rollback**

**5.1 Rollback to revision 1:**

bash

CopyEdit

kubectl rollout undo deploy ramandep1 -n raman --to-revision=1

🧠 This will bring the deployment back to using nginx:1.14.1.

**5.2 Confirm pods:**

bash

CopyEdit

kubectl get pods -n raman

kubectl describe pods -n raman | grep -i image

**5.3 Check updated RS status:**

bash

CopyEdit

kubectl get rs -n raman

You'll now have **3 ReplicaSets**:

* RS for nginx:1.14.1 (current)
* RS for nginx:1.14.2 (inactive)
* RS for nginx:1.14.1 (previous, reused or re-created depending on hash)

**🧹 Step 6: Cleanup**

bash

CopyEdit

kubectl delete -f deploy.yml

kubectl delete namespace raman

**🔬 Deep Dive: Why Multiple ReplicaSets?**

When a Deployment is updated (e.g., image changed), Kubernetes doesn't modify the existing RS. Instead:

1. A new RS is created with the new Pod template (e.g., different image).
2. Old RS is scaled down gradually (zero pods).
3. New RS is scaled up.
4. You now have:
   * One active RS (current revision)
   * One or more inactive RS (past revisions)

This behavior enables:

* **Zero-downtime rolling updates**
* **Granular rollbacks**
* **Revision tracking via rollout history**

**📚 Summary Table**

| **Command** | **Purpose** |
| --- | --- |
| kubectl create/apply -f ... --record=true | Create/update Deployment and store command in history |
| kubectl rollout history | View change history |
| kubectl rollout undo | Roll back to previous or specific revision |
| kubectl get rs | View active/inactive ReplicaSets |
| `kubectl describe pods | grep image` |
| kubectl rollout status | Track progress of rollout in real-time |

**💡 Bonus Tips**

* Use kubectl diff -f deploy.yml before applying changes to preview differences.
* Enable revisionHistoryLimit in your Deployment to control how many old ReplicaSets are kept:

yaml

CopyEdit

spec:

revisionHistoryLimit: 3

* Use kubectl get events -n raman to troubleshoot rollout issues (e.g., image pull errors).

**🧪 Lab Guide: Kubernetes Services – ClusterIP, NodePort, and LoadBalancer**

**🎯 Objectives**

* Deploy an NGINX app using a Deployment
* Expose the Deployment via all 3 major Kubernetes service types
  + **ClusterIP** (internal access)
  + **NodePort** (external access via node IP & high port)
  + **LoadBalancer** (external access via cloud load balancer or simulated in Minikube/MetalLB)
* Understand how services route traffic to Pods using selectors and labels
* Validate service accessibility

**🧱 Prerequisites**

* Kubernetes cluster (kubeadm/Minikube/kind)
* kubectl CLI
* Namespace raman exists:

bash

CopyEdit

kubectl create namespace raman

**📦 Step 1: Deploy the Application**

**1.1 Create a Deployment file deploy.yml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: ramandep1

namespace: raman

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

name: rk

template:

metadata:

labels:

name: rk

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**1.2 Apply the Deployment**

bash

CopyEdit

kubectl apply -f deploy.yml

**🔍 Step 2: Observe Pods and Deployment**

bash

CopyEdit

kubectl get pods -n raman -o wide

kubectl get deploy -n raman

**🌐 Step 3: Create a NodePort Service**

**3.1 service.yml – NodePort Service**

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-service

namespace: raman

spec:

type: NodePort

selector:

name: rk

ports:

- port: 80

targetPort: 80

nodePort: 30007

💡 **Selector name: rk** matches pod label to send traffic to correct pods.

**3.2 Apply the Service**

bash

CopyEdit

kubectl apply -f service.yml

**3.3 Validate Service**

bash

CopyEdit

kubectl get svc -n raman

Example output:

pgsql

CopyEdit

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

raman-service NodePort 10.96.232.12 <none> 80:30007/TCP 1m

**3.4 Test Access from Node (outside cluster)**

bash

CopyEdit

curl http://<NodeIP>:30007

Replace <NodeIP> with the IP of any Kubernetes node. Use:

bash

CopyEdit

kubectl get nodes -o wide

**🧪 Step 4: Test ClusterIP (Default Service)**

Let's now simulate a **ClusterIP service**.

**4.1 Expose using ClusterIP (default)**

bash

CopyEdit

kubectl expose deploy ramandep1 -n raman --port 80 --target-port 80 --name=clusterip-svc

**4.2 Check service:**

bash

CopyEdit

kubectl get svc -n raman

Output:

css

CopyEdit

clusterip-svc ClusterIP 10.100.0.50 <none> 80/TCP 10s

**4.3 Test from within cluster**

bash

CopyEdit

kubectl run tester --rm -i -t --image=busybox --restart=Never -n raman -- sh

# Inside the pod:

wget -qO- http://clusterip-svc

**☁️ Step 5: Create a LoadBalancer Service**

**5.1 Expose Deployment via LoadBalancer type**

bash

CopyEdit

kubectl expose deploy ramandep1 -n raman --type=LoadBalancer --port=80 --target-port=80 --name=lbsvc

**5.2 View Service**

bash

CopyEdit

kubectl get svc -n raman

Example:

pgsql

CopyEdit

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

lbsvc LoadBalancer 10.100.20.5 192.168.1.100 80:30319/TCP 1m

In **cloud environments** (EKS, GKE, AKS), EXTERNAL-IP will be public.  
In **bare-metal/minikube**, it may stay <pending> unless you're using **MetalLB** or a proxy tunnel.

**Optional: Minikube Users**

bash

CopyEdit

minikube service lbsvc -n raman

**🔄 Step 6: Scaling the Deployment**

bash

CopyEdit

kubectl scale deploy ramandep1 -n raman --replicas=1

kubectl get pods -n raman

Confirm traffic is still routed properly to the remaining pod.

**🧹 Step 7: Cleanup**

bash

CopyEdit

kubectl delete svc raman-service clusterip-svc lbsvc -n raman

kubectl delete -f deploy.yml

kubectl delete ns raman

**📚 Summary Table**

| **Service Type** | **Description** | **Accessible From** | **Port Type** |
| --- | --- | --- | --- |
| ClusterIP | Default | Internal cluster only | ClusterIP:80 |
| NodePort | Exposes service on node IP + high port | External to cluster | NodeIP:30007 |
| LoadBalancer | Exposes externally via cloud LB or simulated LB | External/public | EXTERNAL-IP:80 |

**🧠 Key Concepts Recap**

* **Service Selectors** map to Pod labels.
* **ClusterIP** is default, useful for internal-only traffic.
* **NodePort** is a simple way to expose service externally for on-prem/self-hosted clusters.
* **LoadBalancer** delegates external IP allocation to a cloud provider or load-balancer add-on.
* **kubectl expose** is a quick way to generate a Service from existing Deployment.