**Lab Guide: Kubernetes Pod Deployment with kubectl Commands**

**Objective**

This lab will guide you through creating, managing, and troubleshooting a Kubernetes Pod named ramanapp2 in a custom namespace, raman. The Pod will use the httpd image and expose port 80.

**Prerequisites**

* **Kubernetes CLI (kubectl)**: Ensure kubectl is installed and configured.
* **Kubernetes Cluster Access**: You should have access to a running Kubernetes cluster.
* **Namespace**: A custom namespace (raman) for deployment.

**Step 1: Setup Kubernetes Alias**

For convenience, set an alias for the kubectl command.

bash

Copy code

alias k=kubectl

**Step 2: Verify Cluster and Pod Status**

Start by checking all running Pods and their statuses across namespaces.

bash

Copy code

k get pods -A -o wide

**Step 3: Create the Namespace raman**

Create a new namespace raman to host your Pod.

bash

Copy code

k create namespace raman

Verify that the namespace was created successfully:

bash

Copy code

k get namespaces

**Step 4: Run a Test Pod in raman Namespace**

To validate that the namespace is operational, deploy a basic httpd Pod in raman.

bash

Copy code

k run ramanapp --image=httpd -n raman

List the Pods in the raman namespace to confirm the deployment:

bash

Copy code

k get pods -n raman

**Step 5: Delete the Test Pod**

To keep the namespace clear, delete the test Pod before creating the main Pod from pod.yml.

bash

Copy code

k delete pod ramanapp -n raman

**Step 6: Create a Pod Definition File (pod.yml)**

Using a text editor, create a file named pod.yml with the following contents:

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: ramanapp2

namespace: raman

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

**Step 7: Deploy the Pod Using the Configuration File**

Deploy the Pod defined in pod.yml with the following command:

bash

Copy code

k create -f pod.yml

**Step 8: Verify the Deployment**

Check the status of ramanapp2 to confirm that the Pod is running correctly.

bash

Copy code

k get pods -A -o wide

If ramanapp2 is not running, review the events and status for troubleshooting.

bash

Copy code

k describe pod ramanapp2 -n raman

**Step 9: Update or Troubleshoot the Pod**

If any changes are required in pod.yml, make them, then redeploy by deleting and re-creating the Pod:

1. **Edit** the pod.yml file as needed.
2. **Delete** the existing Pod:

bash

Copy code

k delete pod ramanapp2 -n raman

1. **Recreate** the Pod:

bash

Copy code

k create -f pod.yml

**Step 10: Cleanup**

Once you're done, clean up all resources to reset the environment.

bash

Copy code

k delete pod ramanapp2 -n raman

k delete namespace raman

**Summary**

This guide provided a detailed approach to create and manage a Kubernetes Pod in a custom namespace using kubectl commands and YAML configuration files. Through this exercise, you practiced common Kubernetes operations such as Pod creation, namespace management, and troubleshooting deployment issues.

**Lab Guide: Creating and Managing Kubernetes Pods with Labels and Selectors**

**Objective**

This lab will walk you through deploying a Kubernetes Pod (ramanapp) in a custom namespace (raman) with labels for organization and management. You’ll also learn to use kubectl commands to filter Pods by labels.

**Prerequisites**

* **Kubernetes CLI (kubectl)**: Ensure kubectl is installed and configured.
* **Kubernetes Cluster Access**: You should have access to a running Kubernetes cluster.
* **Namespace**: A custom namespace (raman) for deployment.

**Step 1: Set Up Alias for kubectl Command**

For ease of use, set up an alias for the kubectl command:

bash

Copy code

alias k=kubectl

**Step 2: Verify Cluster Status**

Check the current Pods in all namespaces to get an overview of the cluster.

bash

Copy code

k get pods -A

**Step 3: Create the Namespace raman**

If you haven't created the raman namespace, create it now:

bash

Copy code

k create namespace raman

Verify the namespace creation:

bash

Copy code

k get namespaces

**Step 4: Create the Pod Definition File with Labels (pod.yml)**

Using a text editor, create a file named pod.yml with the following contents to define the ramanapp Pod with a front-end label.

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: ramanapp

namespace: raman

labels:

tier: front-end

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

In this YAML configuration:

* **Pod Name**: ramanapp
* **Namespace**: raman
* **Label**: tier: front-end
* **Container Image**: httpd
* **Container Port**: 80

**Step 5: Deploy the Pod Using pod.yml**

Deploy the Pod by applying the configuration file:

bash

Copy code

k create -f pod.yml

**Step 6: Verify the Pod Creation**

List all Pods in the raman namespace to confirm that ramanapp is running.

bash

Copy code

k get pods -n raman

To see more detailed information, use:

bash

Copy code

k get pods -n raman -o wide

**Step 7: Verify Labels on the Pod**

Check if the ramanapp Pod has the tier: front-end label.

bash

Copy code

k get pods -n raman --show-labels

This command will display the labels applied to each Pod in the namespace.

**Step 8: Use Label Selectors to Filter Pods**

You can filter Pods by label using the --selector option. This is particularly useful for managing large clusters.

**Example: List Pods with the Label tier=front-end**

bash

Copy code

k get pods -n raman --selector tier=front-end

**Example: List Pods with the Label Across All Namespaces**

bash

Copy code

k get pods -A --selector tier=front-end

**Step 9: Describe the Pod for Detailed Information**

To get more information on the ramanapp Pod, including label details and events, use:

bash

Copy code

k describe pod ramanapp -n raman

**Step 10: Update the Pod’s Labels**

If you need to update labels after deployment, you can patch the Pod. For example, to change the label from tier=front-end to tier=backend, run:

bash

Copy code

k label pod ramanapp -n raman tier=backend --overwrite

Verify the updated labels:

bash

Copy code

k get pods -n raman --show-labels

**Step 11: Clean Up**

After completing the lab, delete the ramanapp Pod and the raman namespace to clean up resources:

bash

Copy code

k delete pod ramanapp -n raman

k delete namespace raman

**Summary**

This lab covered creating a Pod with labels, using selectors to filter and manage Pods, and modifying labels dynamically. Labels and selectors are essential for grouping and managing Kubernetes resources efficiently, especially in complex environments.

**Lab Guide: Creating and Managing Kubernetes Deployments with Labels and Selectors**

**Objective**

This lab covers:

* Deploying a multi-replica Deployment in Kubernetes.
* Managing labels and selectors to target specific Pods.
* Understanding how selectors work in controlling Pod creation under Deployments.

**Prerequisites**

* **Kubernetes CLI (kubectl)**: Ensure kubectl is installed and configured.
* **Access to Kubernetes Cluster**: A running Kubernetes cluster.
* **Namespace**: Use a custom namespace (raman) for deploying the resources.

**Step 1: Set Up Alias for kubectl Command (Optional)**

For ease of use, set an alias for the kubectl command:

bash

Copy code

alias k=kubectl

**Step 2: Verify Cluster and Namespace**

Ensure you have the necessary access by listing Pods across all namespaces and creating the raman namespace if it does not already exist.

bash

Copy code

k get pods -A

k create namespace raman

k get namespaces

**Step 3: Create the Deployment YAML File**

Using a text editor, create a file named deploy.yaml with the following content:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-deployment2

namespace: raman

labels:

app: nginx2

spec:

replicas: 5

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

**Explanation:**

* **Deployment Name**: raman-deployment2.
* **Namespace**: raman.
* **Labels**: The Deployment itself has a label app: nginx2.
* **Replicas**: 5 replicas of the httpd container.
* **Selector**: matchLabels: app: nginx ensures that only Pods with this label are managed by this Deployment.
* **Template Metadata Labels**: The Pod template includes app: nginx to match the selector, ensuring these Pods are managed by this Deployment.

**Step 4: Apply the Deployment Configuration**

Deploy the raman-deployment2 Deployment by applying the YAML file:

bash

Copy code

k apply -f deploy.yaml

**Step 5: Verify the Deployment and Pods**

List Deployments in the raman namespace to ensure raman-deployment2 is running:

bash

Copy code

k get deploy -n raman

Check for all created Pods and ensure five replicas are running as specified:

bash

Copy code

k get pods -n raman

For additional information on labels assigned to each Pod, run:

bash

Copy code

k get pods -n raman --show-labels

**Step 6: Use Label Selectors to Filter Pods**

Use label selectors to filter and manage specific resources. For example, you can filter Pods by the app=nginx label as specified in the template:

bash

Copy code

k get pods -n raman --selector app=nginx

You can also view the app: nginx2 label on the Deployment level with:

bash

Copy code

k get deploy -n raman --show-labels

**Step 7: Describe the Deployment and ReplicaSet**

Get detailed information on the Deployment to understand how it controls Pod replicas:

bash

Copy code

k describe deploy raman-deployment2 -n raman

Describe the ReplicaSet created by this Deployment (automatically generated with a unique name based on the Deployment name):

bash

Copy code

k get rs -n raman

k describe rs <replicaset-name> -n raman

**Note**: Replace <replicaset-name> with the name of the ReplicaSet from the output of the get rs command.

**Step 8: Scale the Deployment**

Scale the raman-deployment2 Deployment to 3 replicas to see how Kubernetes dynamically manages Pods based on the specified replica count.

bash

Copy code

k scale deploy raman-deployment2 -n raman --replicas=3

Verify the change:

bash

Copy code

k get pods -n raman

**Step 9: Update the Deployment’s Container Image**

To demonstrate updating a Deployment, modify the image used by the Deployment. Update deploy.yaml to specify a new image, such as nginx:latest:

yaml

Copy code

containers:

- name: nginx

image: nginx:latest

Then apply the updated configuration to trigger a rolling update:

bash

Copy code

k apply -f deploy.yaml

Verify that the Pods are being updated by watching the rollout status:

bash

Copy code

k rollout status deploy raman-deployment2 -n raman

**Step 10: View Pods and ReplicaSets After Update**

List all Pods and ReplicaSets in the namespace to verify the update:

bash

Copy code

k get pods -n raman --show-labels

k get rs -n raman

**Step 11: Clean Up Resources**

After completing the lab, delete the raman-deployment2 Deployment and the namespace to clean up resources:

bash

Copy code

k delete deploy raman-deployment2 -n raman

k delete namespace raman

**Summary**

This lab demonstrated creating a Kubernetes Deployment with label-based selectors, scaling and updating a Deployment, and utilizing labels and selectors to manage resources effectively. Understanding these concepts is critical for deploying scalable, manageable workloads in Kubernetes.

**Lab Guide: Deploying and Exposing Applications in Kubernetes**

**Objective**

In this lab, you will:

* Deploy an application using a Deployment in Kubernetes.
* Expose the application with both NodePort and LoadBalancer services.
* Use various kubectl commands to manage Pods, services, and get detailed information.

**Prerequisites**

* **Kubernetes CLI (kubectl)**: Ensure kubectl is installed and configured.
* **Access to Kubernetes Cluster**: A running Kubernetes cluster.
* **Namespace**: Default namespace or custom namespaces (as needed).

**Step 1: Setting Up a Custom Namespace**

If desired, create a new namespace for this lab to keep resources organized. For example, create a namespace named raman:

bash

Copy code

kubectl create namespace raman

To confirm, list all namespaces:

bash

Copy code

kubectl get namespaces

**Step 2: Deploy an Application Using kubectl run**

Start by running a basic httpd container in the raman namespace:

bash

Copy code

kubectl run ramanapp --image=httpd -n raman

This command creates a Pod named ramanapp running the httpd server. Verify the Pod’s status:

bash

Copy code

kubectl get pods -n raman -o wide

**Step 3: Expose the Pod Using a NodePort Service**

Expose the ramanapp Pod on port 80 using a NodePort service to make it accessible on any node’s IP and the assigned NodePort:

bash

Copy code

kubectl expose pod ramanapp --type=NodePort --port=80 --target-port=80 -n raman

To verify the service:

bash

Copy code

kubectl get svc -n raman

kubectl describe svc ramanapp -n raman

The output will show the NodePort assigned by Kubernetes (a port in the range 30000-32767). Use this to access the application.

**Step 4: Access the Application Using curl**

To test connectivity, use the curl command with the node's IP address and assigned NodePort:

bash

Copy code

curl <Node\_IP>:<NodePort>

Replace <Node\_IP> with the actual IP of any node in your cluster, and <NodePort> with the port shown in the service description.

**Step 5: Clean Up the Pod and Service**

Delete the Pod and the associated NodePort service to prepare for the Deployment-based setup:

bash

Copy code

kubectl delete pod ramanapp -n raman

kubectl delete svc ramanapp -n raman

**Step 6: Create a Deployment for the Application**

Now, create a Deployment for the application with multiple replicas for high availability. This time, use nginx as the container image and set the replica count to 5:

bash

Copy code

kubectl create deploy ramandep --image=nginx --replicas=5 -n raman

Verify the Deployment and check that 5 replicas are running:

bash

Copy code

kubectl get deploy -n raman

kubectl get pods -n raman -o wide

**Step 7: Expose the Deployment Using a NodePort Service**

Expose the ramandep Deployment on port 80 using a NodePort service, targeting port 80 on the container:

bash

Copy code

kubectl expose deploy ramandep --type=NodePort --port=80 --target-port=80 -n raman

Confirm that the service is created and check the assigned NodePort:

bash

Copy code

kubectl get svc -n raman

kubectl describe svc ramandep -n raman

**Step 8: Access the Nginx Application Using NodePort**

Using curl or a browser, access the Nginx application through any node’s IP and the assigned NodePort.

bash

Copy code

curl <Node\_IP>:<NodePort>

**Step 9: Clean Up the NodePort Service**

If desired, clean up the NodePort service for ramandep before proceeding to create a LoadBalancer service:

bash

Copy code

kubectl delete svc ramandep -n raman

**Step 10: Expose the Deployment Using a LoadBalancer Service**

In environments that support LoadBalancer services (e.g., cloud platforms like GCP or AWS), expose ramandep as a LoadBalancer service:

bash

Copy code

kubectl expose deploy ramandep --type=LoadBalancer --port=80 --target-port=80 -n raman

Verify that the service is created and check for an external IP:

bash

Copy code

kubectl get svc -n raman

kubectl describe svc ramandep -n raman

Depending on your environment, this may assign a public IP address, allowing access from outside the cluster.

**Step 11: Verify the LoadBalancer Service Access**

Once the external IP address is available, use curl or a browser to access the application:

bash

Copy code

curl <External\_IP>:80

**Step 12: View and Describe Pods, Services, and Endpoints**

Use the following commands to gather detailed information about the resources in the namespace:

* List Pods with labels:

bash

Copy code

kubectl get pods -n raman --show-labels

* Get endpoint information:

bash

Copy code

kubectl get endpoints -n raman

* Describe the ramandep Deployment to check configurations and events:

bash

Copy code

kubectl describe deploy ramandep -n raman

**Step 13: Clean Up Resources**

To finish, clean up all resources created during this lab:

bash

Copy code

kubectl delete deploy ramandep -n raman

kubectl delete svc ramandep -n raman

kubectl delete namespace raman

**Summary**

This lab demonstrated creating and exposing applications in Kubernetes with different service types, scaling applications using Deployments, and checking the status and details of various resources. The NodePort and LoadBalancer services provide flexibility in accessing applications from inside or outside the cluster, making Kubernetes a powerful choice for scalable web applications.

4o

Top of Form

Bottom of Form

**Lab Guide: Exposing an Application with a ClusterIP Service**

**Objective**

* Deploy an application in Kubernetes.
* Expose it with a ClusterIP service.
* Test access within the cluster.

**Prerequisites**

* **Kubernetes CLI (kubectl)**: Ensure kubectl is installed and configured.
* **Access to Kubernetes Cluster**: A running Kubernetes cluster.
* **Namespace**: Use the default namespace or create a custom one for this lab.

**Step 1: Setting Up a Custom Namespace (Optional)**

If you want to keep resources organized, create a namespace named raman:

bash

Copy code

kubectl create namespace raman

To confirm, list all namespaces:

bash

Copy code

kubectl get namespaces

**Step 2: Create a Deployment for the Application**

Create a Deployment using the nginx image, with 3 replicas for redundancy. Place it in the raman namespace (or your chosen namespace):

bash

Copy code

kubectl create deploy nginx-deployment --image=nginx --replicas=3 -n raman

Verify that the Deployment and Pods are running:

bash

Copy code

kubectl get deploy -n raman

kubectl get pods -n raman -o wide

**Step 3: Expose the Deployment with a ClusterIP Service**

Expose the nginx-deployment Deployment with a ClusterIP service. This makes the application accessible only within the cluster, using an internal IP:

bash

Copy code

kubectl expose deploy nginx-deployment --type=ClusterIP --port=80 --target-port=80 -n raman

To verify the service, list all services in the namespace:

bash

Copy code

kubectl get svc -n raman

**Step 4: Check the ClusterIP Assigned to the Service**

To see the details of the service and its internal ClusterIP, use:

bash

Copy code

kubectl describe svc nginx-deployment -n raman

The output will show the ClusterIP, which is the internal IP for the service. This IP is accessible only within the Kubernetes cluster.

**Step 5: Access the Service from Within the Cluster**

1. **Using a BusyBox Pod**:
   * Start a temporary BusyBox Pod in the same namespace, which you can use to access the service:

bash

Copy code

kubectl run testbox --rm -it --image=busybox -- /bin/sh -n raman

* + Inside the BusyBox shell, use wget or curl to test access to the nginx-deployment service using the ClusterIP:

bash

Copy code

wget -qO- <ClusterIP>:80

* + If successful, you’ll see the Nginx default HTML content. This confirms that the service is reachable within the cluster.

1. **Using the Service Name (DNS) Instead of IP**:
   * Kubernetes automatically creates DNS records for services, so you can also access the service by its name (e.g., nginx-deployment) and namespace.

bash

Copy code

wget -qO- nginx-deployment.raman.svc.cluster.local:80

**Step 6: Clean Up Resources**

Once done, clean up the resources:

bash

Copy code

kubectl delete deploy nginx-deployment -n raman

kubectl delete svc nginx-deployment -n raman

kubectl delete pod testbox -n raman

kubectl delete namespace raman

**Summary**

This activity demonstrated how to expose an application internally within a Kubernetes cluster using a ClusterIP service, as well as accessing it via the ClusterIP or service DNS name. ClusterIP services are useful for internal communication between microservices or backend systems that don’t require external exposure.