**Lab Guide for Deploying a Kubernetes Rolling Update Deployment with Nginx**

This lab will guide you through the deployment, rollout, and version management of a Kubernetes application using a Deployment resource with the RollingUpdate strategy. By the end of this lab, you’ll understand how to deploy multiple versions of an application, check rollout status, rollback to previous versions, and view rollout history.

**Prerequisites:**

* A running Kubernetes cluster
* kubectl installed and configured to communicate with your cluster
* Basic knowledge of Kubernetes concepts such as pods, deployments, and replica sets

**Step 1: Set Up the Deployment YAML**

1. **Create the Deployment Configuration File**:
   * Open your terminal and navigate to the directory where you want to save your deployment YAML file.
   * Use a text editor to create a file named deploy.yaml:

bash

Copy code

vi deploy.yaml

1. **Define the Deployment in deploy.yaml**:
   * Copy and paste the following configuration into deploy.yaml:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-app

labels:

app: web2-prod-app

spec:

replicas: 10

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

selector:

matchLabels:

app: web2-prod-app

template:

metadata:

labels:

app: web2-prod-app

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

* + This configuration specifies:
    - **10 replicas** of the nginx container.
    - A **RollingUpdate** strategy with maxSurge and maxUnavailable set to 25%, allowing the deployment to maintain availability while updating.

1. **Save and Close the File**:
   * Exit and save changes by pressing :wq in vi.

**Step 2: Deploy the First Version of Your Application**

1. **Apply the Deployment Configuration**:
   * Run the following command to apply deploy.yaml to the Kubernetes cluster, creating the initial deployment with nginx:latest:

bash

Copy code

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

* + The --record=true option ensures that the change is recorded in the deployment’s revision history.

1. **Check Deployment Status**:
   * Verify the status of the rollout with:

bash

Copy code

kubectl rollout status deployment raman-app

* + This command will inform you once the deployment has been successfully rolled out.

1. **Verify Pod Status**:
   * List the pods to confirm that the deployment has created 10 running instances of the application:

bash

Copy code

kubectl get pods

**Step 3: Update the Application to a New Version**

1. **Edit the Deployment File to Use nginx:1.14.1**:
   * Open deploy.yaml and change the image version under the containers section:

yaml

Copy code

image: nginx:1.14.1

1. **Apply the Updated Deployment**:
   * Run the following command to update the deployment to nginx:1.14.1:

bash

Copy code

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

1. **Check Rollout Status**:
   * Track the rollout to ensure that the update is progressing correctly:

bash

Copy code

kubectl rollout status deployment raman-app

1. **Verify the Update**:
   * Run the following command to list pods and see if the new replica set (raman-app-6444496947) has been created and deployed:

bash

Copy code

kubectl get pods

**Step 4: Perform Another Update to nginx:1.14.2**

1. **Edit the Deployment to Use nginx:1.14.2**:
   * In deploy.yaml, change the image version to nginx:1.14.2:

yaml

Copy code

image: nginx:1.14.2

1. **Apply the Deployment Update**:
   * Apply the changes with:

bash

Copy code

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

1. **Monitor the Rollout**:
   * Check the rollout status to ensure the update to nginx:1.14.2 is progressing:

bash

Copy code

kubectl rollout status deployment raman-app

1. **Confirm the New Version is Running**:
   * List the pods to verify that the new replica set (raman-app-c955dc6d) is running:

bash

Copy code

kubectl get pods

**Step 5: Deploy the Latest Nginx Version**

1. **Update the Deployment to Use nginx:latest**:
   * Open deploy.yaml and modify the image field to use nginx:latest:

yaml

Copy code

image: nginx:latest

1. **Apply the Update**:
   * Run the following command to deploy the latest image:

bash

Copy code

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

1. **Check Rollout Progress**:
   * Monitor the status of this rollout to ensure completion:

bash

Copy code

kubectl rollout status deployment raman-app

1. **Validate the Update**:
   * Use kubectl get pods to see if the new replica set (raman-app-7bb9777b67) is running:

bash

Copy code

kubectl get pods

**Step 6: Rollback to a Previous Version**

1. **Check Deployment History**:
   * Use the following command to view all previous versions of the deployment:

bash

Copy code

kubectl rollout history deployment raman-app

* + This will show a list of revisions with details about each deployment.

1. **Rollback to the Second Version**:
   * Rollback to nginx:1.14.1 by specifying the revision number (replace <revision-number> with the actual revision):

bash

Copy code

kubectl rollout undo deployment raman-app --to-revision=<revision-number>

1. **Verify the Rollback**:
   * Check the rollout status again:

bash

Copy code

kubectl rollout status deployment raman-app

1. **Confirm Pod Version**:
   * Use the describe command to confirm that the image has reverted to nginx:1.14.1:

bash

Copy code

kubectl describe pods | grep -i image

**Step 7: Cleanup**

1. **Delete the Deployment**:
   * Once you’ve finished, delete the deployment to clean up the resources:

bash

Copy code

kubectl delete deployment raman-app

1. **Verify Deletion**:
   * Run the following command to ensure all resources are deleted:

bash

Copy code

kubectl get pods

kubectl get rs

**Additional Commands and Observations**

* **View Specific Deployment Details**:

bash

Copy code

kubectl describe deployment raman-app

* **Get a Snapshot of All Replica Sets**:

bash

Copy code

kubectl get rs

* **Detailed History for a Specific Revision**:

bash

Copy code

kubectl rollout history deployment raman-app --revision=<revision-number>

This lab covered:

* Creating a Kubernetes deployment with a rolling update strategy
* Updating and rolling back application versions
* Viewing rollout history and status

step-by-step lab guide for setting up and testing a **Blue-Green Deployment** with Nginx and HTTPD using Kubernetes, based on the files provided.

**Lab Overview**

In this lab, you will deploy two versions of an application (one with **Nginx** and one with **HTTPD**) using **Blue-Green Deployment** in Kubernetes. You’ll configure services to switch between deployments and validate connectivity. By the end of this lab, you’ll understand how to deploy and test a Blue-Green setup using Kubernetes deployments, services, and labels.

**Prerequisites**

* A Kubernetes cluster with a master and worker nodes.
* kubectl installed and configured to interact with your cluster.
* Basic understanding of Kubernetes Deployments, Services, and Blue-Green Deployment strategy.

**Step 1: Setup Deployment YAML Files**

1. **Create Deployment File** (deploy.yaml):
   * Open a new file named deploy.yaml.
   * Paste the following content to define the **Green Deployment** with nginx and the **Blue Deployment** with httpd.

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-green

labels:

app: web2-prod-app

spec:

replicas: 5

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: httpd-blue

labels:

app: web2-prod-app2

spec:

replicas: 5

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

selector:

matchLabels:

app: httpd

template:

metadata:

labels:

app: httpd

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

1. **Save and exit** the file.

**Step 2: Deploy the Blue-Green Applications**

1. Apply the deployment file to create both **nginx-green** and **httpd-blue** deployments:

bash

Copy code

kubectl apply -f deploy.yaml

1. Verify that both deployments are created and running:

bash

Copy code

kubectl get deployments

kubectl get pods --show-labels

* + You should see two sets of pods, one with the label app=nginx and another with app=httpd.

**Step 3: Configure Service for Blue-Green Switching**

1. **Create a Service File** (service.yaml):
   * Open a new file named service.yaml.
   * Add the following configuration to expose the **Blue Deployment** (httpd-blue) via a NodePort service:

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: bluegreensvc

spec:

type: NodePort

selector:

app: httpd

ports:

- port: 80

targetPort: 80

nodePort: 30007

1. **Save and exit** the file.
2. Apply the service configuration:

bash

Copy code

kubectl apply -f service.yaml

1. Check that the service is created:

bash

Copy code

kubectl get svc bluegreensvc

1. Note the NodePort assigned to bluegreensvc. You will use this port to test the connection.

**Step 4: Test the Blue Deployment (HTTPD)**

1. Identify the IP address of any Kubernetes node (worker node) and combine it with the NodePort (30007).
2. Access the service through the NodePort to test the **Blue Deployment**:

bash

Copy code

curl <Node\_IP>:30007

* + You should receive a response from the **HTTPD server**.

1. Verify that the current version served by the application is running HTTPD:

bash

Copy code

kubectl get pods -l app=httpd

1. Review the Service details:

bash

Copy code

kubectl describe svc bluegreensvc

**Step 5: Switch to the Green Deployment (Nginx)**

1. **Update the Service Selector**:
   * Edit service.yaml to change the selector from app: httpd to app: nginx:

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: bluegreensvc

spec:

type: NodePort

selector:

app: nginx

ports:

- port: 80

targetPort: 80

nodePort: 30007

1. **Apply the updated service configuration**:

bash

Copy code

kubectl apply -f service.yaml

1. Test the service again using curl on the same <Node\_IP>:30007 to confirm it is now pointing to the **Nginx** deployment:

bash

Copy code

curl <Node\_IP>:30007

1. Verify that the current version served by the application is now running **Nginx**:

bash

Copy code

kubectl get pods -l app=nginx

**Step 6: Roll Back to the Blue Deployment**

1. If you want to revert to the **Blue Deployment**, edit the service.yaml again, changing the selector back to app: httpd:

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: bluegreensvc

spec:

type: NodePort

selector:

app: httpd

ports:

- port: 80

targetPort: 80

nodePort: 30007

1. **Apply the configuration** to switch back:

bash

Copy code

kubectl apply -f service.yaml

1. Test the service again to confirm it has rolled back to the HTTPD deployment.

**Step 7: View Deployment History and Rollbacks**

1. View the history of changes made to the deployment:

bash

Copy code

kubectl rollout history deployment nginx-green

kubectl rollout history deployment httpd-blue

1. If needed, you can rollback to a previous version using:

bash

Copy code

kubectl rollout undo deployment <deployment\_name> --to-revision=<revision\_number>

1. Confirm the rollback status:

bash

Copy code

kubectl rollout status deployment <deployment\_name>

**Cleanup**

1. Delete the service:

bash

Copy code

kubectl delete -f service.yaml

1. Delete the deployments:

bash

Copy code

kubectl delete -f deploy.yaml

**Summary**

In this lab, you created a Blue-Green deployment using Nginx (Green) and HTTPD (Blue). You switched between deployments by updating the Service selector and confirmed the service points to the correct deployment with each change. This approach allows for seamless updates and rollbacks with minimal downtime.

**Canary Deployment** in Kubernetes, using nginx and httpd as two test services. In this guide, you will deploy both applications, configure the service to switch between versions, and manage the deployment using scaling and testing.

**Lab Overview**

In this lab, you will:

1. Deploy two applications (nginx and httpd) under the same label (app: canary) to simulate a canary release environment.
2. Configure a service to target these deployments.
3. Gradually scale up/down the replicas to monitor traffic distribution, mimicking canary deployment behavior.

By the end of this lab, you will understand how to create, test, and monitor a canary deployment in Kubernetes.

**Prerequisites**

* A Kubernetes cluster set up and accessible.
* kubectl installed and configured.
* Basic understanding of Kubernetes Deployments, Services, and canary deployment concepts.

**Step 1: Create Deployment YAML Files**

1. **Create the Deployment File** (deploy.yaml):
   * In your working directory, create a new file named deploy.yaml.
   * Copy and paste the following content into the file to define **two deployments**:
     + nginx-green for the primary application version.
     + httpd-blue for the canary version.

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-green

labels:

app: web2-prod-app

spec:

replicas: 5

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

selector:

matchLabels:

app: canary

template:

metadata:

labels:

app: canary

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: httpd-blue

labels:

app: web2-prod-app2

spec:

replicas: 5

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

selector:

matchLabels:

app: canary

template:

metadata:

labels:

app: canary

spec:

containers:

- name: httpd

image: httpd

ports:

- containerPort: 80

1. **Save and exit** the file.

**Step 2: Deploy the Canary Applications**

1. Apply the deployment file to create both **nginx-green** and **httpd-blue** deployments:

bash

Copy code

kubectl apply -f deploy.yaml

1. Verify that both deployments have been created and their pods are running:

bash

Copy code

kubectl get deployments

kubectl get pods --show-labels

* + You should see two sets of pods, both labeled with app=canary.

**Step 3: Configure Service for Canary Testing**

1. **Create the Service File** (service.yaml):
   * In your working directory, create a file named service.yaml.
   * Add the following configuration, which creates a NodePort service that targets any pod labeled app: canary.

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: bluegreensvc

spec:

type: NodePort

selector:

app: canary

ports:

- port: 80

targetPort: 80

nodePort: 30007

1. **Save and exit** the file.
2. Apply the service configuration:

bash

Copy code

kubectl apply -f service.yaml

1. Check the service status:

bash

Copy code

kubectl get svc bluegreensvc

1. **Note** the NodePort (30007 in this case), which you’ll use to access the canary deployment.

**Step 4: Test the Canary Deployment**

1. Identify the IP address of any worker node in your cluster.
2. Test the service by sending requests to <Node\_IP>:30007:

bash

Copy code

curl <Node\_IP>:30007

* + Since both nginx and httpd are configured under the app: canary label, traffic may be distributed between them. You might see responses from either the nginx or httpd server.

1. **Monitor the Pods**:

bash

Copy code

kubectl get pods -l app=canary -o wide

1. **Describe the Service** to confirm the selected pods and the current configuration:

bash

Copy code

kubectl describe svc bluegreensvc

**Step 5: Gradual Traffic Shift (Scaling)**

1. **Reduce Traffic to HTTPD (Blue) Canary**:
   * Scale down httpd-blue deployment to 0 replicas:

bash

Copy code

kubectl scale deploy httpd-blue --replicas=0

* + Verify the status:

bash

Copy code

kubectl get pods -l app=canary

* + **Test the Service** again:

bash

Copy code

curl <Node\_IP>:30007

* + - At this point, you should only receive responses from nginx.

1. **Gradually Reintroduce HTTPD (Canary)**:
   * Scale up httpd-blue to 1 replica to introduce it gradually:

bash

Copy code

kubectl scale deploy httpd-blue --replicas=1

* + Observe the distribution by running the curl command multiple times:

bash

Copy code

for i in {1..10}; do curl <Node\_IP>:30007; done

* + If you see responses from both nginx and httpd, the traffic is being split between them, indicating a canary-style deployment.

1. **Further Increase Canary Traffic**:
   * Scale httpd-blue deployment to 3 replicas and observe the changes:

bash

Copy code

kubectl scale deploy httpd-blue --replicas=3

* + Monitor the pods:

bash

Copy code

kubectl get pods -l app=canary

* + Test the service by sending multiple requests:

bash

Copy code

for i in {1..10}; do curl <Node\_IP>:30007; done

* + - You should see more responses from httpd as its replica count increases relative to nginx.

1. **Fully Rollout Canary**:
   * Scale httpd-blue to 5 replicas (matching nginx-green), effectively rolling out the canary version:

bash

Copy code

kubectl scale deploy httpd-blue --replicas=5

* + Confirm that the canary release is fully deployed and check the pods:

bash

Copy code

kubectl get pods -l app=canary

* + Test the service again:

bash

Copy code

curl <Node\_IP>:30007

* + Since nginx and httpd are evenly scaled, traffic should now be balanced equally.

**Step 6: Rollback the Canary Deployment**

1. If issues arise with httpd, you can scale it down, effectively rolling back to the original version.

bash

Copy code

kubectl scale deploy httpd-blue --replicas=0

1. Test to confirm that traffic only routes to nginx:

bash

Copy code

curl <Node\_IP>:30007

**Step 7: Cleanup**

1. **Delete the Service**:

bash

Copy code

kubectl delete -f service.yaml

1. **Delete the Deployments**:

bash

Copy code

kubectl delete -f deploy.yaml

**Summary**

In this lab, you successfully implemented a **Canary Deployment** in Kubernetes using two applications, nginx and httpd, under a shared label. By scaling deployments up and down, you controlled the traffic distribution to introduce a new version gradually. Canary deployment allows for gradual rollouts, so you can monitor the stability and performance of new versions before a full rollout.

**Lab Guide: Configuring Kubernetes Horizontal Pod Autoscaler (HPA) for Ubuntu Deployment**

**Objective**

In this lab, you will learn how to:

1. Deploy a sample Ubuntu container.
2. Install the Kubernetes Metrics Server.
3. Set up a Horizontal Pod Autoscaler (HPA) to scale the deployment based on CPU usage.

**Prerequisites**

* Kubernetes cluster with at least one worker node.
* kubectl command-line tool configured to connect to your Kubernetes cluster.
* **Metrics Server** must be deployed in the cluster to allow HPA to get CPU metrics.

**Step 1: Prepare the Ubuntu Deployment YAML**

Create a deployment YAML file named deploy2.yaml that defines an Ubuntu deployment with minimal resource requirements.

1. In your terminal, create the file:

bash

Copy code

vi deploy2.yaml

1. Add the following configuration for deploy2.yaml:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: ubuntu-deployment

spec:

replicas: 1

selector:

matchLabels:

app: ubuntu

template:

metadata:

labels:

app: ubuntu

spec:

containers:

- name: ubuntu-container

image: ubuntu:20.04

command: ["sleep", "infinity"]

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

1. Save and close the file.

**Step 2: Deploy the Ubuntu Application**

1. Deploy the application using the kubectl command:

bash

Copy code

kubectl create -f deploy2.yaml

1. Confirm the deployment and the pod status:

bash

Copy code

kubectl get deploy

kubectl get pods

You should see the ubuntu-deployment with 1 replica running in the cluster.

**Step 3: Install the Kubernetes Metrics Server**

If your cluster doesn’t already have the **Metrics Server** installed, follow these steps:

1. Clone the official Kubernetes Metrics Server repository:

bash

Copy code

git clone https://github.com/ramannkhanna2/k8s\_metrics\_server.git

cd k8s\_metrics\_server/deploy/1.8+/

1. Deploy the Metrics Server:

bash

Copy code

kubectl create -f .

1. Verify the Metrics Server pods are running:

bash

Copy code

kubectl get pods -n kube-system

**Step 4: Confirm Metrics Collection**

Once the Metrics Server is up and running, confirm that CPU and memory metrics are being collected:

1. Check node metrics:

bash

Copy code

kubectl top nodes

1. Check pod metrics (ensure your Ubuntu pod is listed):

bash

Copy code

kubectl top pods

If metrics are displayed, the Metrics Server is functioning correctly.

**Step 5: Configure the Horizontal Pod Autoscaler (HPA)**

Now, configure the HPA for the ubuntu-deployment.

1. Set up HPA with CPU target utilization at 70%, with minimum replicas set to 1 and maximum replicas set to 5:

bash

Copy code

kubectl autoscale deployment ubuntu-deployment --cpu-percent=70 --min=1 --max=5

1. Confirm that the HPA is created:

bash

Copy code

kubectl get hpa

You should see output indicating that the HPA is monitoring ubuntu-deployment with a target CPU utilization of 70%.

**Step 6: Generate CPU Load to Test Autoscaling**

Since the Ubuntu container has minimal activity, generate load to observe the HPA in action.

1. Enter a shell session inside the running Ubuntu container:

bash

Copy code

kubectl exec -it <ubuntu-pod-name> -- /bin/bash

Replace <ubuntu-pod-name> with the name of your Ubuntu pod.

1. Install stress in the container:

bash

Copy code

apt update

apt install -y stress

1. Run stress to consume CPU:

bash

Copy code

stress --cpu 1 --timeout 300

This command will keep the CPU at high utilization for 5 minutes.

**Step 7: Monitor the HPA Scaling Behavior**

1. Watch the HPA to observe any changes in replica count as CPU utilization increases:

bash

Copy code

kubectl get hpa -w

You should see the HPA adjusting the replica count based on the CPU usage of the ubuntu-deployment.

1. To view the status of the pods and verify that additional replicas are created:

bash

Copy code

kubectl get pods -w

The HPA should increase the number of replicas as CPU utilization rises, and eventually, reduce the count back to the minimum when load decreases.

**Step 8: Cleanup**

After testing, clean up the resources to avoid unnecessary charges and resource usage.

1. Delete the HPA:

bash

Copy code

kubectl delete hpa ubuntu-deployment

1. Delete the deployment:

bash

Copy code

kubectl delete -f deploy2.yaml

1. Optionally, uninstall the Metrics Server if it’s not required for other tasks:

bash

Copy code

kubectl delete -f k8s\_metrics\_server/deploy/1.8+/

**Summary**

In this lab, you:

* Deployed a sample Ubuntu application.
* Installed and verified the Kubernetes Metrics Server.
* Configured an HPA to automatically scale the deployment based on CPU load.
* Observed autoscaling behavior by generating CPU load within the container.

**Lab Guide: Managing Secrets in Kubernetes**

**Objective**

In this lab, you will:

1. Create a Kubernetes Secret to securely store sensitive data.
2. Deploy two application pods that access the secrets:
   * myapp-pod1 will access the secret data via a volume mount.
   * myapp-pod2 will access the secret data via environment variables.

**Prerequisites**

* Kubernetes cluster with at least one worker node.
* kubectl command-line tool configured to connect to your Kubernetes cluster.

**Step 1: Encode Sensitive Data in Base64**

To create a Kubernetes Secret, sensitive data (like passwords) must be base64 encoded.

1. **Encode the username**:

bash

Copy code

echo 'ramankhanna' | base64

* + Output: cmFtYW5raGFubmE=

1. **Encode the password**:

bash

Copy code

echo 'ramankhanna123' | base64

* + Output: cmFtYW5raGFubmExMjMK

Take note of these encoded values, as they will be used in the secret.yaml file.

**Step 2: Create the Secret YAML File**

1. In your terminal, create a YAML file named secret.yaml:

bash

Copy code

vi secret.yaml

1. Add the following configuration to define the secret:

yaml

Copy code

apiVersion: v1

kind: Secret

metadata:

name: my-secrets

type: Opaque

data:

username: cmFtYW5raGFubmE=

password: cmFtYW5raGFubmExMjMK

* + username and password values are base64-encoded versions of ramankhanna and ramankhanna123, respectively.

1. Save and close the file.

**Step 3: Apply the Secret to the Cluster**

1. Apply the secret.yaml file to create the secret in the cluster:

bash

Copy code

kubectl apply -f secret.yaml

1. Confirm that the secret has been created:

bash

Copy code

kubectl get secret

1. To view the details of the secret (in a secure environment):

bash

Copy code

kubectl describe secret my-secrets

You should see username and password under the data section, showing their encoded values.

**Step 4: Create Application Pods to Access the Secret**

We will now create two pods:

1. **myapp-pod1** - Accesses the secret data via a volume mount.
2. **myapp-pod2** - Accesses the secret data via environment variables.

**1. Define the Application Pods in a YAML File**

1. In your terminal, create a file named secret-pod.yaml:

bash

Copy code

vi secret-pod.yaml

1. Add the following configuration for myapp-pod1 and myapp-pod2:

yaml

Copy code

# myapp-pod1 - Accessing secrets via volume mount

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

---

# myapp-pod2 - Accessing secrets via environment variables

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

* + myapp-pod1 mounts the secret as a file in /tmp/creds.
  + myapp-pod2 exposes the username and password values as environment variables, SECRET\_USERNAME and SECRET\_PASSWD.

1. Save and close the file.

**Step 5: Deploy the Application Pods**

1. Apply the secret-pod.yaml file to create the pods:

bash

Copy code

kubectl apply -f secret-pod.yaml

1. Confirm that the pods are running:

bash

Copy code

kubectl get pods

You should see both myapp-pod1 and myapp-pod2 listed as running.

**Step 6: Verify Secret Access in myapp-pod1 (via Volume Mount)**

1. Access the shell of myapp-pod1:

bash

Copy code

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

1. Check the /tmp/creds directory where the secret is mounted:

bash

Copy code

ls /tmp/creds

You should see files named username and password.

1. View the contents of the username and password files:

bash

Copy code

cat /tmp/creds/username

cat /tmp/creds/password

These commands will display ramankhanna and ramankhanna123 respectively.

1. Exit the pod shell:

bash

Copy code

exit

**Step 7: Verify Secret Access in myapp-pod2 (via Environment Variables)**

1. Access the shell of myapp-pod2:

bash

Copy code

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

1. Verify that the secret values are set as environment variables:

bash

Copy code

echo $SECRET\_USERNAME

echo $SECRET\_PASSWD

* + These commands should output ramankhanna and ramankhanna123, respectively.

1. Exit the pod shell:

bash

Copy code

exit

**Step 8: Clean Up**

To avoid unnecessary resource usage, delete the resources created in this lab:

1. Delete the pods:

bash

Copy code

kubectl delete -f secret-pod.yaml

1. Delete the secret:

bash

Copy code

kubectl delete -f secret.yaml

**Summary**

In this lab, you:

* Created a Kubernetes Secret to store sensitive data (username and password).
* Configured two application pods to access the secret:
  + myapp-pod1 accessed the secret through a volume mount.
  + myapp-pod2 accessed the secret through environment variables.

By completing this lab, you've gained experience in securely managing sensitive data in Kubernetes and using secrets in different ways to suit application requirements.

**Lab Guide: Understanding Persistent Volumes (PV) and Persistent Volume Claims (PVC) in Kubernetes**

**Objective**

In this lab, you will:

1. Learn the concept of Persistent Volumes (PV) and Persistent Volume Claims (PVC) in Kubernetes.
2. Set up a simple application that stores data persistently across pod restarts.
3. Use a practical example to create a PV, bind it to a PVC, and mount the storage in an application pod.

**Prerequisites**

* Kubernetes cluster with at least one worker node.
* kubectl command-line tool configured to connect to your Kubernetes cluster.

**Key Concepts**

Before diving into the lab, here’s a brief overview of key terms:

* **Persistent Volume (PV)**: A storage resource in Kubernetes that represents physical storage (like a disk). It is provisioned by an administrator and exists independently of any specific pod.
* **Persistent Volume Claim (PVC)**: A request for storage by a pod. PVCs request specific storage size and access modes and are bound to available PVs that satisfy these requirements.
* **Access Modes**:
  + ReadWriteOnce (RWO): The volume can be mounted as read-write by a single node.
  + ReadOnlyMany (ROX): The volume can be mounted as read-only by many nodes.
  + ReadWriteMany (RWX): The volume can be mounted as read-write by many nodes.

**Lab Steps**

**Step 1: Create a Persistent Volume (PV)**

1. Create a YAML file named pv.yaml to define a Persistent Volume:

bash

Copy code

vi pv.yaml

1. Add the following configuration to the file:

yaml

Copy code

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-pv

spec:

capacity:

storage: 1Gi # Specifies 1 GiB of storage

accessModes:

- ReadWriteOnce # This volume can only be attached to a single node for read-write

hostPath:

path: /mnt/data # Local storage path on the worker node where data will be stored

Explanation:

* + capacity: Specifies the storage size.
  + accessModes: Limits how the volume can be accessed. ReadWriteOnce means only one pod at a time can write to this PV.
  + hostPath: Defines a specific path on the host machine. For simplicity, we are using a local path, but in production, this could be a network storage solution.

1. Save and apply the pv.yaml file to create the PV:

bash

Copy code

kubectl apply -f pv.yaml

1. Verify that the PV is created:

bash

Copy code

kubectl get pv

The output should show my-pv with the AVAILABLE status.

**Step 2: Create a Persistent Volume Claim (PVC)**

1. Create a YAML file named pvc.yaml to define a Persistent Volume Claim:

bash

Copy code

vi pvc.yaml

1. Add the following configuration to request a storage volume:

yaml

Copy code

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 500Mi # Requests 500 MiB of storage

Explanation:

* + accessModes: Must match the access mode defined in the PV (in this case, ReadWriteOnce).
  + resources.requests.storage: Specifies the amount of storage requested by the PVC. Since this is less than the total size of my-pv, the request can be fulfilled.

1. Save and apply the pvc.yaml file to create the PVC:

bash

Copy code

kubectl apply -f pvc.yaml

1. Verify that the PVC is created and bound to the PV:

bash

Copy code

kubectl get pvc

The my-pvc should have a Bound status, which indicates that it is successfully bound to my-pv.

**Step 3: Create a Pod to Use the PVC for Persistent Storage**

1. Create a YAML file named pod.yaml to define a pod that uses the PVC:

bash

Copy code

vi pod.yaml

1. Add the following configuration:

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: pvc-test-pod

spec:

containers:

- name: busybox

image: busybox

command: ["sleep", "3600"] # Keeps the pod running for testing

volumeMounts:

- mountPath: "/data" # Directory inside the container where the PVC will be mounted

name: my-storage

volumes:

- name: my-storage

persistentVolumeClaim:

claimName: my-pvc # Binds this volume to the PVC created earlier

Explanation:

* + volumeMounts: Defines where in the container the volume should be mounted. In this example, it’s mounted at /data.
  + volumes.persistentVolumeClaim.claimName: Connects the volume to my-pvc, which in turn binds to my-pv.

1. Save and apply the pod.yaml file to create the pod:

bash

Copy code

kubectl apply -f pod.yaml

1. Confirm that the pod is running:

bash

Copy code

kubectl get pods

The pvc-test-pod should be in the Running state.

**Step 4: Verify Persistent Storage in the Pod**

1. Enter the pod to verify that the volume is correctly mounted:

bash

Copy code

kubectl exec -it pvc-test-pod -- /bin/sh

1. Inside the pod, navigate to the /data directory where the volume is mounted:

bash

Copy code

cd /data

1. Create a test file to confirm data persistence:

bash

Copy code

echo "This is a test file" > testfile.txt

1. Verify that the file was created:

bash

Copy code

cat testfile.txt

You should see the content This is a test file, indicating that data can be written to the mounted storage.

1. Exit the pod:

bash

Copy code

exit

**Step 5: Test Data Persistence**

1. Delete the pod but leave the PVC and PV intact:

bash

Copy code

kubectl delete pod pvc-test-pod

1. Re-create the pod with the same configuration:

bash

Copy code

kubectl apply -f pod.yaml

1. Once the pod is running, enter it again:

bash

Copy code

kubectl exec -it pvc-test-pod -- /bin/sh

1. Go to the /data directory and verify the test file is still present:

bash

Copy code

cd /data

cat testfile.txt

The content This is a test file should still be there, demonstrating that the data persisted even though the pod was deleted.

**Step 6: Clean Up Resources**

To free up resources, delete the PVC, PV, and pod:

1. Delete the pod:

bash

Copy code

kubectl delete pod pvc-test-pod

1. Delete the PVC:

bash

Copy code

kubectl delete pvc my-pvc

1. Delete the PV:

bash

Copy code

kubectl delete pv my-pv

**Summary**

In this lab, you:

* Created a Persistent Volume (PV) as a storage resource in the Kubernetes cluster.
* Created a Persistent Volume Claim (PVC) to request storage for a pod.
* Configured a pod to mount the PVC, allowing it to store data persistently.
* Verified data persistence by deleting and recreating the pod.

By completing this lab, you've gained practical experience with persistent storage in Kubernetes, an essential component for stateful applications.