**Kubernetes Lab Guide**

**Lab Overview:**

This lab will guide you through creating and managing Kubernetes Pods using YAML configuration files. It covers basic commands for interacting with Pods, executing commands within containers, and managing multiple containers in a Pod.

**Pre-requisites:**

* A running Kubernetes cluster.
* kubectl configured to interact with the cluster.
* Basic knowledge of Kubernetes concepts like Pods, containers, and namespaces.

**Step 1: Create a Namespace**

First, create a namespace called raman.

bash

Copy code

kubectl create namespace raman

You can verify the namespace is created with:

bash

Copy code

kubectl get ns

**Step 2: Run an Initial Pod**

Create a simple Pod running the httpd image in the raman namespace.

bash

Copy code

kubectl run app1 --image=httpd -n raman

Verify the Pod is running:

bash

Copy code

kubectl get pods -n raman

**Step 3: Create a Pod with Multiple Containers**

Now, create a YAML file (pod.yaml) to define a Pod with two containers: nginx and redis.

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: app2

namespace: raman

spec:

containers:

- name: con1

image: nginx

ports:

- containerPort: 80

- name: con2

image: redis

ports:

- containerPort: 6379

Save this file in a directory, for example raman/pod.yaml.

**Step 4: Create the Pod**

Use the YAML file to create the Pod:

bash

Copy code

kubectl create -f raman/pod.yaml

Verify the Pod is created:

bash

Copy code

kubectl get pods -n raman

**Step 5: Explore the Pod and Containers**

You can describe the Pod to view its details:

bash

Copy code

kubectl describe pod app2 -n raman

To get more detailed information (e.g., the node it's running on, IP addresses):

bash

Copy code

kubectl get pods -n raman -o wide

**Step 6: Interact with Containers**

You can execute commands inside the containers. For example, to enter the nginx container (con1):

bash

Copy code

kubectl exec -it app2 -c con1 -n raman -- /bin/bash

Similarly, for the redis container (con2):

bash

Copy code

kubectl exec -it app2 -c con2 -n raman -- /bin/bash

You can perform actions inside the containers. For example, to create a directory inside the redis container:

bash

Copy code

kubectl exec -it app2 -c con2 -n raman -- mkdir /ramanfolder

Verify the folder is created:

bash

Copy code

kubectl exec -it app2 -c con2 -n raman -- ls

**Step 7: Updating the Pod**

If you need to update the Pod definition (e.g., changing the image or ports), you can modify the pod.yaml file and then apply the changes:

bash

Copy code

kubectl apply -f raman/pod.yaml

Alternatively, you can replace the Pod configuration:

bash

Copy code

kubectl replace -f raman/pod.yaml

**Step 8: Clean Up**

To delete the Pod:

bash

Copy code

kubectl delete -f raman/pod.yaml

To delete all Pods in the namespace:

bash

Copy code

kubectl delete pods --all -n raman

**Step 9: Additional Commands**

* **List all Pods across all namespaces:**

bash

Copy code

kubectl get pods -A

* **Execute a command in the con1 container to get its hostname:**

bash

Copy code

kubectl exec -it app2 -c con1 -n raman -- hostname

This concludes the basic lab guide for working with Pods in Kubernetes, including managing multi-container Pods and using the kubectl commands effectively.

**Lab Guide: Building and Running a Flask Application in Docker**

**Lab Overview:**

In this lab, you will learn how to containerize a simple Flask web application using Docker. The lab involves building a multi-stage Docker image, which reduces the final image size, and running the Flask app inside a container.

**Prerequisites:**

* A machine with Docker installed.
* Basic knowledge of Flask and Docker concepts.

**Directory Structure:**

Your Flask application directory (myapp) should look like this:

bash

Copy code

myapp/

├── .dockerignore

├── app.py

├── dockerfile

└── requirements.txt

**Step 1: Create the Application Code**

The application code is in the app.py file, which is a simple Flask application that allows you to view posts, add new posts, and navigate through different pages.

Here is the app.py code:

python

Copy code

from flask import Flask, request, redirect, url\_for, render\_template\_string

app = Flask(\_\_name\_\_)

# In-memory database for demonstration purposes

posts = [

{"id": 1, "title": "First Post", "content": "This is the content of the first post."},

{"id": 2, "title": "Second Post", "content": "This is the content of the second post."}

]

# HTML templates as strings

layout = '''<!DOCTYPE html>...</html>'''

home\_page = '''<h2>...</h2>'''

about\_page = '''<h2>...</h2>'''

post\_page = '''<h2>...</h2>'''

add\_post\_page = '''<h2>...</h2>'''

@app.route('/')

def index():

return render\_template\_string(layout, content=render\_template\_string(home\_page, posts=posts))

@app.route('/about')

def about():

return render\_template\_string(layout, content=render\_template\_string(about\_page))

@app.route('/post/<int:post\_id>')

def post(post\_id):

post = next((post for post in posts if post["id"] == post\_id), None)

if post is None:

return "Post not found", 404

return render\_template\_string(layout, content=render\_template\_string(post\_page, post=post))

@app.route('/add', methods=['GET', 'POST'])

def add\_post():

if request.method == 'POST':

title = request.form['title']

content = request.form['content']

posts.append({"id": len(posts) + 1, "title": title, "content": content})

return redirect(url\_for('index'))

return render\_template\_string(layout, content=render\_template\_string(add\_post\_page))

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000, debug=True)

**Step 2: Create a .dockerignore File**

Create a .dockerignore file to ignore unnecessary files from being copied into the Docker image:

bash

Copy code

\_\_pycache\_\_

\*.pyc

\*.pyo

\*.pyd

.env

**Step 3: Create the requirements.txt File**

The requirements.txt file includes the dependencies required by your Flask application:

bash

Copy code

Flask==2.2.3

Werkzeug==2.2.3

**Step 4: Write the dockerfile**

Create the dockerfile to define how the Docker image for the Flask app will be built. This is a multi-stage Dockerfile that reduces the image size by copying only the necessary dependencies and application code to the final image.

dockerfile

Copy code

# Stage 1: Build Stage

FROM python:3.11-slim AS build

# Build arguments

ARG DEBIAN\_FRONTEND=noninteractive

# Set the working directory

WORKDIR /app

# Install curl

RUN apt-get update && \

apt-get install -y curl && \

apt-get clean && \

rm -rf /var/lib/apt/lists/\*

# Copy and install dependencies

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Final Stage

FROM python:3.11-slim

# Set the working directory

WORKDIR /app

# Copy installed dependencies from the build stage

COPY --from=build /usr/local/lib/python3.11 /usr/local/lib/python3.11

COPY --from=build /usr/local/bin /usr/local/bin

# Copy the application code

COPY app.py .

# Create a non-root user and switch to it

RUN useradd -ms /bin/sh -u 1001 appuser

USER appuser

# Expose port for the Flask app

EXPOSE 5000

# Set environment variables

ENV FLASK\_APP=app.py

ENV FLASK\_RUN\_HOST=0.0.0.0

# Health check to ensure the application is running

HEALTHCHECK --interval=30s --timeout=10s --retries=3 \

CMD curl -f http://localhost:5000/ || exit 1

# Add metadata as described above

LABEL org.opencontainers.image.title="My Flask App" \

org.opencontainers.image.version="1.0" \

org.opencontainers.image.description="A simple Flask application" \

org.opencontainers.image.authors="Your Name <your.email@example.com>"

# Run the Flask application

CMD ["flask", "run"]

**Step 5: Build the Docker Image**

Now that all the required files are in place, you can build the Docker image for the Flask app.

Run the following command in the root of the myapp directory:

bash

Copy code

docker build -t my-flask-app .

This will create a Docker image tagged as my-flask-app.

**Step 6: Run the Docker Container**

Once the image is built, run the Flask application inside a container:

bash

Copy code

docker run -d -p 5000:5000 --name my-flask-app-container my-flask-app

This will run the Flask app and expose it on port 5000 on your machine.

**Step 7: Verify the Application**

Open a browser and navigate to http://localhost:5000. You should see the home page of the Flask app with a list of posts.

* Visit http://localhost:5000/about to see the About page.
* Visit http://localhost:5000/add to add a new post.

**Step 8: Health Check**

The dockerfile includes a health check that periodically checks if the Flask app is running. You can verify the health status of the container with:

bash

Copy code

docker inspect --format='{{json .State.Health}}' my-flask-app-container

**Step 9: Cleanup**

To stop and remove the running container:

bash

Copy code

docker stop my-flask-app-container

docker rm my-flask-app-container

To remove the Docker image:

bash

Copy code

docker rmi my-flask-app

**Conclusion:**

You have successfully built and run a Flask application using Docker. This lab guide demonstrated how to create a multi-stage Dockerfile, optimize image size, and run a containerized Python web application.

**Lab Guide: Kubernetes Deployment for a Flask Application**

**Lab Overview:**

This lab guide demonstrates how to deploy a Flask application in a Kubernetes environment. You will create a deployment, manage pods, scale the application, and troubleshoot using Kubernetes commands. This guide assumes basic familiarity with Kubernetes.

**Prerequisites:**

* A working Kubernetes cluster.
* kubectl command-line tool configured to interact with the cluster.
* An image of the Flask app pushed to a container registry (in this case, ramann123/natwest:my-flask-appV1).

**Step 1: Create a Deployment**

In this step, you will create a deployment for your Flask application.

1. Use the kubectl create command to create a deployment with 3 replicas, specifying the Flask app's Docker image:

bash

Copy code

kubectl create deploy dep1 --image=ramann123/natwest:my-flask-appV1 --replicas=3 -n raman

**Explanation:**

* + dep1: The name of the deployment.
  + ramann123/natwest:my-flask-appV1: The Docker image of your Flask app.
  + --replicas=3: Creates 3 replicas (pods) for high availability.
  + -n raman: Creates the deployment in the raman namespace.

**Step 2: Verify the Deployment**

To verify the status of your deployment and pods:

1. List all resources in the raman namespace:

bash

Copy code

kubectl get all -n raman

1. Get detailed information about the pods:

bash

Copy code

kubectl get pods -n raman -o wide

**Explanation:**

* + -o wide: Provides additional information about the pods, including their IP addresses and node assignments.

1. Check the deployment details:

bash

Copy code

kubectl describe deploy dep1 -n raman

This command gives an overview of the deployment's configuration and its status, including replica sets and pod creation status.

**Step 3: Test the Application**

Now, access the application using the pod IP addresses:

1. Use kubectl get pods -o wide to get the IP address of one of the pods:

bash

Copy code

kubectl get pods -n raman -o wide

1. Use curl to access the application (assuming it is exposed on port 5000):

bash

Copy code

curl <POD\_IP>:5000

Replace <POD\_IP> with the actual IP of one of the running pods.

**Step 4: Delete a Pod and Observe Self-Healing**

Kubernetes ensures high availability by automatically recreating failed pods. Let’s simulate a pod failure and observe how Kubernetes recovers the deployment:

1. Delete a pod manually:

bash

Copy code

kubectl delete pod dep1-<pod\_name> -n raman

For example:

bash

Copy code

kubectl delete pod dep1-8469884bc8-d4fvs -n raman

1. Verify that a new pod is created to replace the deleted one:

bash

Copy code

kubectl get pods -n raman

**Expected Output:** You should see a new pod with a different name being created to maintain the desired number of replicas (3 in this case).

**Step 5: Scaling the Deployment**

You can scale the number of replicas in the deployment dynamically:

1. Scale the deployment to 5 replicas:

bash

Copy code

kubectl scale deploy dep1 --replicas=5 -n raman

1. Check the updated number of pods:

bash

Copy code

kubectl get pods -n raman

You should now see 5 pods running for the dep1 deployment.

**Step 6: Explore ReplicaSets and Labels**

ReplicaSets are responsible for maintaining the correct number of pod replicas. In this step, you will explore the ReplicaSets and the label-based selection mechanism.

1. List the ReplicaSets in the raman namespace:

bash

Copy code

kubectl get rs -n raman

1. Describe a specific ReplicaSet to see its label selector:

bash

Copy code

kubectl describe rs dep1-8469884bc -n raman | grep -i selector

**Explanation:** This will display the label selectors used to match pods to the ReplicaSet.

1. Describe the deployment and check its selector:

bash

Copy code

kubectl describe deploy dep1 -n raman | grep -i selector

This shows the label selector that the deployment uses to manage its pods.

**Step 7: Cleanup**

To clean up all resources created during this lab:

1. Delete the deployment:

bash

Copy code

kubectl delete deploy dep1 -n raman

1. Optionally, delete all resources in the namespace:

bash

Copy code

kubectl delete all --all -n raman

**Conclusion:**

In this lab, you learned how to:

* Create a deployment in Kubernetes.
* Manage pods and scale a deployment.
* Use labels and ReplicaSets to manage and troubleshoot resources.
* Ensure high availability and self-healing by deleting pods and observing recovery.

**Lab Guide: Kubernetes Deployment with Flask App and Label Management**

**Lab Overview:**

This lab guide will walk you through creating and managing a Kubernetes deployment for a Flask-based web application. It will also cover how to interact with labels, scale the deployment, and perform label-based operations on pods.

**Prerequisites:**

* A working Kubernetes cluster.
* kubectl installed and configured.
* Flask app Docker image available (in this case, ramann123/natwest:my-flask-appV1).

**Step 1: Create a Deployment with deploy.yaml**

In this step, you'll create a Kubernetes Deployment using a YAML file that deploys a Flask application in the raman namespace.

1. **Create the Deployment YAML file**:

Ensure the following content is in the deploy.yaml file:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: flask-deployment

namespace: raman

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: flask

template:

metadata:

labels:

app: flask

spec:

containers:

- name: flaskcon

image: ramann123/natwest:my-flask-appV1

ports:

- containerPort: 5000

1. **Apply the Deployment**:

Use the following command to apply the deploy.yaml file and create the deployment:

bash

Copy code

kubectl apply -f deploy.yaml

1. **Verify the Deployment**:

Check the status of the deployment and pods:

bash

Copy code

kubectl get pods -n raman

You should see three pods running under the flask-deployment deployment.

**Step 2: Examine Deployment Selectors and Labels**

Kubernetes uses labels and selectors to organize and manage objects. In this step, you will examine the labels and selectors.

1. **Get the selector of the deployment**:

bash

Copy code

kubectl describe deploy flask-deployment -n raman | grep -i selector

This command shows the label selectors used to match pods managed by the deployment.

1. **Describe the ReplicaSet and examine its selector**:

bash

Copy code

kubectl describe rs flask-deployment-<replicaset-id> -n raman | grep -i selector

You can find the replicaset ID by using the following command:

bash

Copy code

kubectl get rs -n raman

1. **Examine the labels applied to pods**:

bash

Copy code

kubectl describe pods -n raman | grep -i label

This displays the labels attached to the running pods.

**Step 3: Test Access to the Flask Application**

You can access the Flask application by using curl to test its availability via the pod IP.

1. **Get the IP addresses of the pods**:

bash

Copy code

kubectl get pods -o wide -n raman

1. **Use curl to access the Flask application**:

bash

Copy code

curl <POD\_IP>:5000

Replace <POD\_IP> with the actual IP address of one of the running pods. If the Flask application is running successfully, you should receive a response from the application.

**Step 4: Working with Labels**

Labels are key-value pairs that can be used to identify and organize Kubernetes objects. In this step, you will add labels to a pod and use them to filter objects.

1. **Create a test pod**:

Create a simple test pod using the httpd image:

bash

Copy code

kubectl run myapptest --image=httpd

1. **Check the pod’s labels**:

bash

Copy code

kubectl describe pod myapptest | grep -i label

Initially, the pod will not have any custom labels.

1. **Add a label to the pod**:

Add a label name=raman to the myapptest pod:

bash

Copy code

kubectl label pod myapptest "name=raman"

1. **Verify the label**:

Check that the label has been successfully applied:

bash

Copy code

kubectl describe pod myapptest | grep -i label

1. **Filter pods by label**:

List the pods using the name=raman label:

bash

Copy code

kubectl get pods --selector "name=raman"

1. **Add another label to the pod**:

Now, label the myapptest pod with app=flask:

bash

Copy code

kubectl label pod myapptest "app=flask"

1. **Verify the label again**:

Check that both labels are applied:

bash

Copy code

kubectl describe pod myapptest | grep -i label

1. **Filter pods with app=flask label**:

List the pods using the app=flask label:

bash

Copy code

kubectl get pods --selector "app=flask" -A

This command lists all pods across namespaces with the app=flask label.

**Step 5: Scaling the Deployment**

You can dynamically scale the number of replicas in your deployment.

1. **Scale the deployment to 4 replicas**:

bash

Copy code

kubectl scale deploy flask-deployment --replicas=4 -n raman

1. **Verify the scaling**:

Check the number of running pods:

bash

Copy code

kubectl get pods -n raman

You should now see 4 pods running for the flask-deployment.

**Conclusion:**

In this lab, you learned how to:

* Create a deployment for a Flask app using a YAML configuration.
* Inspect and manage label selectors for deployments, ReplicaSets, and pods.
* Test application availability via pod IPs.
* Work with Kubernetes labels and use them to filter and organize resources.
* Scale a deployment dynamically.

**Lab Guide: Exposing Kubernetes Services**

**Lab Overview:**

In this lab guide, you will learn how to expose Kubernetes pods and deployments as services using different service types such as NodePort and LoadBalancer. You will expose services for a simple HTTPD pod and a Flask-based deployment and understand how to interact with the services using the Cluster IP and external IPs.

**Prerequisites:**

* A running Kubernetes cluster with kubectl installed and configured.
* Pods and Deployments created (e.g., myapptest, dep1).

**Step 1: Create a Service for a Pod Using NodePort**

In this step, we will expose a pod as a service using the NodePort type, which will allow external access to the application.

1. **Create an HTTPD pod (if not already running)**:

bash

Copy code

kubectl run myapptest --image=httpd

Verify that the pod is running:

bash

Copy code

kubectl get pods -o wide

1. **Expose the Pod as a Service using NodePort**:

Use the kubectl expose command to expose the myapptest pod as a service on port 80:

bash

Copy code

kubectl expose pod myapptest --name=httpdsvc --type=NodePort --port=80 --target-port=80

1. **Verify the service**:

Check the details of the service to see the assigned NodePort:

bash

Copy code

kubectl get svc

kubectl describe svc httpdsvc

The output will show the NodePort (a port in the range 30000-32767) that the service is exposed on.

1. **Access the Service**:

You can access the service using the IP address of any node in the cluster and the assigned NodePort. If you’re using minikube, you can get the IP of the node with:

bash

Copy code

minikube ip

If you're using a multi-node cluster, use the following command to get the node IP:

bash

Copy code

kubectl get nodes -o wide

Then access the service using curl:

bash

Copy code

curl <Node\_IP>:<NodePort>

**Step 2: Expose a Deployment as a Service Using NodePort**

Now, we will expose a deployment (dep1) as a service using the NodePort type.

1. **Expose the deployment as a service**:

Use the kubectl expose command to expose the dep1 deployment as a service on port 80:

bash

Copy code

kubectl expose deploy dep1 --name=httpdsvc2 --type=NodePort --port=80 --target-port=80

1. **Verify the service**:

List the services and check details of the new service (httpdsvc2):

bash

Copy code

kubectl get svc

kubectl describe svc httpdsvc2

1. **Access the Service**:

Similar to the pod, you can access the service via the NodePort assigned to it. Use curl to test:

bash

Copy code

curl <Node\_IP>:<NodePort>

**Step 3: Expose a Deployment as a LoadBalancer Service**

In this step, we will expose a deployment as a LoadBalancer service, which provides an external IP to access the service directly.

1. **Expose the deployment as a LoadBalancer service**:

Use the kubectl expose command to expose the myapptest deployment (or any deployment) as a service:

bash

Copy code

kubectl expose deploy myapptest --name=httpdsvc-lb --type=LoadBalancer --port=80 --target-port=80

1. **Verify the service**:

Check the status of the LoadBalancer service:

bash

Copy code

kubectl get svc

kubectl describe svc httpdsvc-lb

Initially, the EXTERNAL-IP field may show <pending> as it takes time for the cloud provider (if applicable) to provision the external IP. Once the service is available, the EXTERNAL-IP field will show the assigned IP.

1. **Access the Service**:

Once the service has an external IP, you can access the service directly using curl:

bash

Copy code

curl <External\_IP>:80

**Step 4: Working with ClusterIP Service**

Kubernetes also supports internal services that are only accessible within the cluster using the ClusterIP type (default). Let's inspect the internal service.

1. **List the services**:

The default type of service is ClusterIP, which allows communication within the cluster. To see all services running, use:

bash

Copy code

kubectl get svc

1. **Access the ClusterIP Service (within cluster)**:

If you want to interact with a ClusterIP service, you need to access it from within the cluster (e.g., from another pod).

First, check the ClusterIP address of the service:

bash

Copy code

kubectl describe svc <service-name>

To test the service internally, you can run a temporary pod inside the cluster and use curl from there:

bash

Copy code

kubectl run -it --rm --image=busybox testbox --restart=Never -- sh

Inside the busybox pod, use curl to access the service:

bash

Copy code

curl <ClusterIP>:<port>

**Conclusion:**

In this lab, you learned how to:

* Expose a pod and a deployment as a NodePort service.
* Expose a deployment as a LoadBalancer service.
* Access services using external and internal methods in Kubernetes.
* Work with services in different scopes (NodePort, LoadBalancer, ClusterIP).

This concludes the lab guide for exposing Kubernetes services.