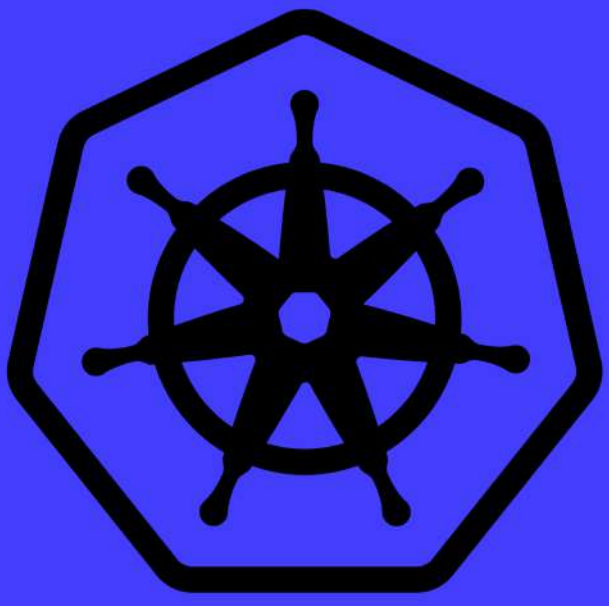
KOENIG
step forward

K8S

CKA LAB GUIDE



List of labs

1. Installation of Kubernetes 1-Master and 2-Nodes Cluster
2. Choose a Network Solution and Configure
3. Deploying Applications as a Pod
4. Managing Labels & Selector
5. Deploying Replica on Controller & Replica Set
6. Creating & Managing Service
7. Deploying Applications as Daemon Sets
8. Using Manual Scheduling or Taints and Tolerations
9. Deploying Applications as Deployment
10. Using Node Affinity to Deploy Pods
11. Deploying Applications as Deployment
12. Implementing Deployment Strategies on Deployments
13. Using Plain Keys, Config Map & Generic Secret as Environment Variables
14. Mount Environment Variable as Volumes
15. Using PV & PVC to a ach Persistent Volume to a Pod as HostPath
16. Creating and Managing Users in Kubernetes
17. Creating Service Accounts
18. Managing Cluster Role and Cluster Role Binding
19. Adding Security Context to Pod to enable ping
20. Upgrade a Kubernetes Cluster Version
21. Deploying Pods as Static Pod
22. ETCD Backup
23. Deploying Pod as Cron Job
24. Understand how to Read Applica on & Cluster Component Logs
25. Deploying Prometheus & Grafana to Monitor K8s Cluster
26. Configure and Manage Ingress Rule
27. Creating Namespace & Deploying K8s resources in Different Namespaces
28. Deploying Metal Load Balancer

Lab Details

1. This lab walks you through the steps to install the Kubernetes cluster.
2. You will use terminal.

Introduction

What is Kubernetes?

- Kubernetes is a container orchestration tool majorly used to control and manage container-based environments.
- Kubernetes works on a master-slave node setup where one central node will control the underlying worker nodes.
- Here we are using kubeadm for the installation of the kubernetes cluster and in the current lab, the cluster is a 1- master node and 2 worker node cluster.

  Kubernetes is required to manage and control container-based applications as it streamlines the process of performing container operations even in a large-scale environment.
- We require container images to run the container and all the container-related commands can be run using the command line utility kubectl. Kubectl utility provides an abstraction over the underlying container engine. This means, that whatever the container runtime is, the operations commands would be the same. It is applicable for the runtimes which are part of the CRI-O initiative.
- With Kubernetes, we can implement various deployment strategies and plan the release timeframes and in most cases, without any downtime. Also, it is a cost-effective solution as containers require minimal resources and environment based on that would cut out the overhead costs.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Install Kubernetes.

Lab Steps

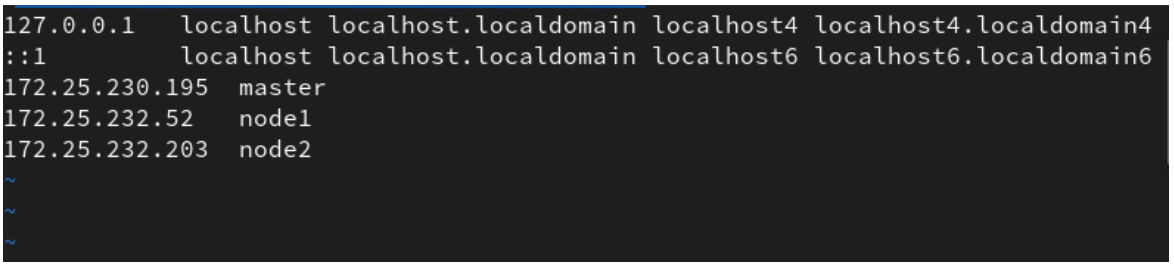
Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Mappings of hostnames to IP addresses

1. Run the below command on master machine to configure a local DNS resolver that maps hostnames to IP addresses.

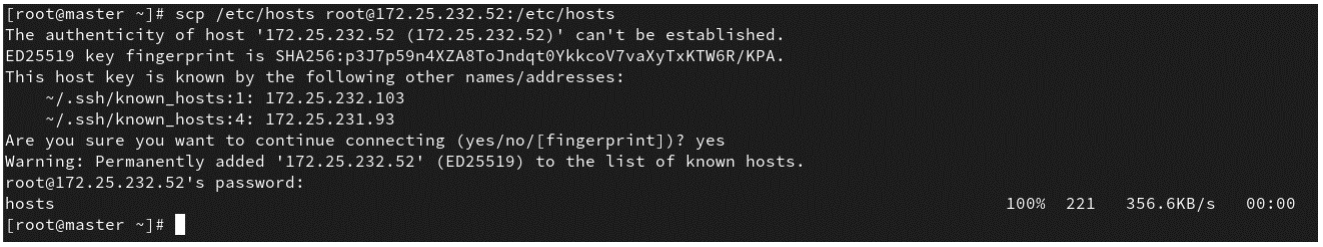
   vim /etc/hosts



2. Now, we will use scp to copy the /etc/hosts file from master to worker node.

   scp /etc/hosts root@<node1-IP>:/etc/hosts

   scp /etc/hosts root@<node2-IP>:/etc/hosts



Task 3: Packages on the system

1. In this step, we will run the YUM package manager to update all installed packages on all the machines. The -y flag automatically confirms all prompts, allowing the update process to proceed without user intervention.

   yum update -y

```
[root@master ~]# yum update -y
CentOS Stream 9 - BaseOS                                                    5.9 kB/s | 4.8 kB     00:00
CentOS Stream 9 - BaseOS                                                    3.1 MB/s | 8.0 MB     00:02
CentOS Stream 9 - AppStream                                                 5.7 kB/s | 4.9 kB     00:00
CentOS Stream 9 - AppStream                                                 2.5 MB/s |  19 MB     00:07
CentOS Stream 9 - Extras packages                                           8.5 kB/s | 6.9 kB     00:00
devel:kubic:libcontainers:stable:cri-o:1.28 (CentOS_8)                      3.3 kB/s | 1.7 kB     00:00
Docker CE Stable - x86_64                                                    42 kB/s | 3.5 kB     00:00
Docker CE Stable - x86_64                                                   250 kB/s |  38 kB     00:00
Extra Packages for Enterprise Linux 9 - x86_64                              8.5 kB/s | 7.2 kB     00:00
Extra Packages for Enterprise Linux 9 - x86_64                              382 kB/s |  20 MB     00:54
Extra Packages for Enterprise Linux 9 openh264 (From Cisco) - x86_64        3.9 kB/s | 993  B     00:00
Extra Packages for Enterprise Linux 9 - Next - x86_64                        36 kB/s | 9.3 kB     00:00
Extra Packages for Enterprise Linux 9 - Next - x86_64                       283 kB/s | 1.4 MB     00:04
Hashicorp Stable - x86_64                                                    22 kB/s | 1.4 kB     00:00
Hashicorp Stable - x86_64                                                   5.3 MB/s | 1.3 MB     00:00
Jenkins-stable                                                               41 kB/s | 2.9 kB     00:00
Jenkins-stable                                                              150 kB/s |  28 kB     00:00
Kubernetes                                                                  4.4 kB/s | 1.7 kB     00:00
Stable Releases of Upstream github.com/containers packages (CentOS_8)       3.1 kB/s | 1.7 kB     00:00
Visual Studio Code                                                          4.3 kB/s | 1.5 kB     00:00
Visual Studio Code                                                          2.9 MB/s | 3.5 MB     00:01
Dependencies resolved.
==========================================================================================================
 Package                          Architecture    Version                Repository                  Size
==========================================================================================================
Installing:
 kernel                           x86_64          5.14.0-412.el9         baseos                     6.0 M
Upgrading:
 NetworkManager                   x86_64          1:1.45.10-1.el9        baseos                     2.3 M
 NetworkManager-adsl              x86_64          1:1.45.10-1.el9        baseos                      38 k
 NetworkManager-bluetooth         x86_64          1:1.45.10-1.el9        baseos                      64 k
 NetworkManager-config-server     noarch          1:1.45.10-1.el9        baseos                      24 k
 NetworkManager-libnm             x86_64          1:1.45.10-1.el9        baseos                     1.8 M
 NetworkManager-team              x86_64          1:1.45.10-1.el9        baseos                      43 k
```

2. This command installs the specified packages (crio, kubectl, kubeadm, and kubelet) using the YUM package manager. The -y flag automatically confirms the installation of packages without requiring user confirmation.(on all machines)

yum install crio kubectl kubeadm kubelet -y

```
[root@master ~]# yum install crio kubectl kubeadm kubelet -y
Last metadata expiration check: 1:00:18 ago on Mon 12 Feb 2024 03:37:06 PM IST.
Dependencies resolved.
==========================================================================================================
 Package                   Architecture   Version              Repository                              Size
==========================================================================================================
Installing:
 cri-o                     x86_64         1.28.2-2.3.el8       devel_kubic_libcontainers_stable_cri-o_1.28  37 M
 kubeadm                   x86_64         1.29.1-150500.1.1    kubernetes                             9.7 M
 kubectl                   x86_64         1.29.1-150500.1.1    kubernetes                              10 M
 kubelet                   x86_64         1.29.1-150500.1.1    kubernetes                              19 M
Installing dependencies:
 conntrack-tools           x86_64         1.4.7-2.el9          appstream                              235 k
 cri-tools                 x86_64         1.29.0-150500.1.1    kubernetes                             8.2 M
 kubernetes-cni            x86_64         1.3.0-150500.1.1     kubernetes                             6.7 M
 libnetfilter_cthelper     x86_64         1.0.0-22.el9         appstream                               24 k
 libnetfilter_cttimeout    x86_64         1.0.0-19.el9         appstream                               24 k
 libnetfilter_queue        x86_64         1.0.5-1.el9          appstream                               29 k
 socat                     x86_64         1.7.4.1-5.el9        appstream                              305 k
Installing weak dependencies:
 runc                      x86_64         4:1.1.11-1.el9       appstream                              3.1 M

Transaction Summary
==========================================================================================================
Install  12 Packages

Total download size: 94 M
Installed size: 444 M
Downloading Packages:
(1/12): libnetfilter_cthelper-1.0.0-22.el9.x86_64.rpm                   28 kB/s |  24 kB     00:00
(2/12): libnetfilter_cttimeout-1.0.0-19.el9.x86_64.rpm                  26 kB/s |  24 kB     00:00
```

3. Now let's start and enable kubelet and crio services. (all machines)

systemctl enable --now crio kubelet

```
[root@master ~]# systemctl enable --now crio kubelet
Created symlink /etc/systemd/system/cri-o.service → /usr/lib/systemd/system/crio.service.
Created symlink /etc/systemd/system/multi-user.target.wants/crio.service → /usr/lib/systemd/system/crio.service.
Created symlink /etc/systemd/system/multi-user.target.wants/kubelet.service → /usr/lib/systemd/system/kubelet.service.
[root@master ~]# █
```

Task 4 : Initialize the cluster

1. We will run this command which initializes a Kubernetes control-plane node. The --pod-network-cidr=10.244.0.0/16 flag specifies the range of IP addresses to assign to pods in the cluster. (master only)

```
kubeadm init --pod-network-cidr=10.244.0.0/16
```

```
[root@master ~]# kubeadm init --pod-network-cidr=10.244.0.0/16
[init] Using Kubernetes version: v1.29.1
[preflight] Running pre-flight checks
        [WARNING Hostname]: hostname "master" could not be reached
        [WARNING Hostname]: hostname "master": lookup master on 172.25.250.254:53: no such host
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.clu
ster.local master] and IPs [10.96.0.1 172.25.230.195]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost master] and IPs [172.25.230.195 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost master] and IPs [172.25.230.195 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "super-admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
```

2. This command creates a directory named .kube in the user's home directory if it doesn't already exist. This directory is typically used to store Kubernetes configuration files. (master only)

```
mkdir -p $HOME/.kube
```

3. This command copies the Kubernetes cluster configuration file (admin.conf) to the .kube directory in the user's home directory, naming it config. (master only)

```
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

4. This command changes the ownership of the copied Kubernetes configuration file to match the user's ownership, ensuring that the user has the necessary permissions to access the cluster configuration. (master only)

```
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

5. Now, to join the worker nodes to the cluster you can run the command printed in master machines. (on worker nodes only)

kubeadm join 172.25.230.195:6443 --token rb1j.v7ugj --discovery-token-ca-cert-hash sha256:6c56e

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 172.25.230.195:6443 --token rb1cxj.v79uu6k35flyw7gj \
        --discovery-token-ca-cert-hash sha256:6c56ee29b03b37ae035aa98a1325642770903f4548259fb71938d6bd2412e6a2
[root@master ~]#
```

```
[root@node1 ~]# kubeadm join 172.25.230.195:6443 --token rb1cxj.v79uu6k35flyw7gj        --discovery-token-ca-cert-hash sha256:6c56ee29b
03b37ae035aa98a1325642770903f4548259fb71938d6bd2412e6a2
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

[root@node1 ~]#
```

Completion and Conclusion

1. You have successfully installed packages required.
2. You have successfully initiated K8s cluster.

End Lab

1. You have successfully completed the lab.

Lab Details

1. This lab walks you through the steps to Network Solution on and Configure in the Kubernetes cluster.
2. You will use terminal.

Introduction

What is CNI?

Container Network Interface (CNI):

Container Network Interface (CNI) is a standardized interface between container runtimes and network plugins. It enables configuring networking for containers by allowing container runtimes like Docker, Kubernetes, and others to call plugins to handle networking functionality. CNI plugins are responsible for tasks such as assigning IP addresses, setting up network namespaces, and creating network interfaces for containers.

Calico:

Calico is an open-source networking and network security solution designed for Kubernetes and other containerized environments. It provides highly scalable networking capabilities for connecting containers, virtual machines, and bare-metal workloads. Here's an explanation of Calico in key points:

Network Policy Enforcement: Calico provides robust network policy enforcement capabilities, allowing administrators to define fine-grained network policies based on labels, selectors, and IP addresses. These policies enable segmentation and isolation of network traffic, enhancing security within the cluster.

Integration with Kubernetes: Calico seamlessly integrates with Kubernetes, leveraging its network policy API to enforce security policies and manage network resources. It operates as a CNI plugin, allowing Kubernetes to leverage Calico's advanced networking features.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Install calico as network solution.
4. Validation Test

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Verify the cluster configuration

1. Run the below command to check the version of `kubectl` package. Kubectl allows us to control the cluster via a command-line interface. One can run the kubectl commands followed by the resource arguments to send API requests to the master node and run the intended task over the worker node.

   kubectl version

```
[root@master mylab]# kubectl version
Client Version: v1.28.2
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: v1.28.5
[root@master mylab]#
```

2. Now, verify the kubelet service is running by the following command. Kubelet service pods are running on both the master and worker nodes and this service is responsible for sending and receiving API requests within the cluster.

   systemctl status kubelet

```
[root@master mylab]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2024-01-03 21:24:17 IST; 6 days ago
     Docs: https://kubernetes.io/docs/
 Main PID: 4993 (kubelet)
    Tasks: 11
   Memory: 46.1M
   CGroup: /system.slice/kubelet.service
           └─4993 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --confi...
```

Task 3: Install Calico as network solution and configure

1. In this step, we will download the Calico YAML manifest file from the specified URL. The manifest file contains the definitions of the various Kubernetes resources required to deploy Calico, such as pods, services, and configuration..

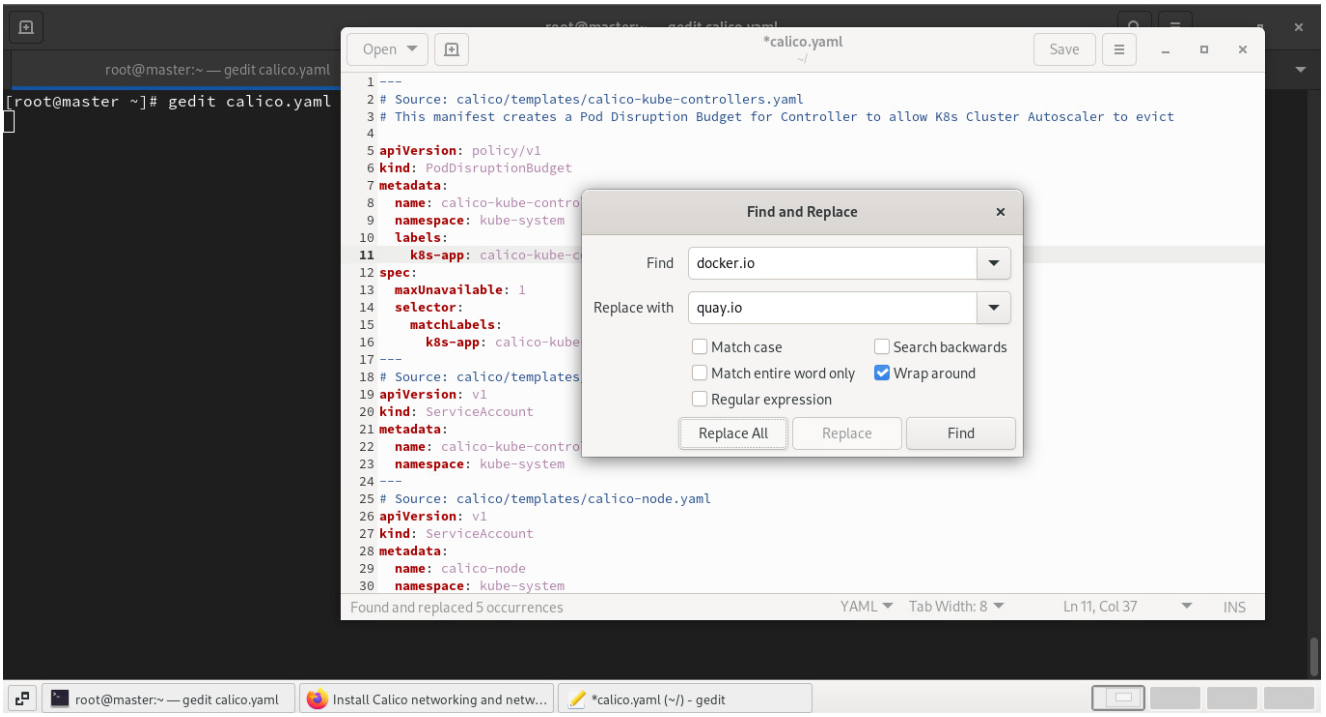
```
curl https://raw.githubusercontent.com/projectcalico/calico/v3.27.0/manifests/calico.yaml -O
```

```
[root@master ~]# curl https://raw.githubusercontent.com/projectcalico/calico/v3.27.0/manifests/calico.yaml -O
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  246k  100  246k    0     0   300k      0 --:--:-- --:--:-- --:--:--  300k
[root@master ~]#
```

2. This command opens the calico.yaml file in the Gedit text editor. Gedit is a simple text editor commonly used in Linux distributions with graphical user interfaces.

```
gedit calico.yaml
```

The user need to find and replace feature in the text editor to replace occurrences of "docker.io" with "quay.io" in the calico.yaml file. After making the replacements, the user should save the file.



3. Now let's deploy Calico components such as pods, services, and configuration

```
kubectl apply -f calico.yaml
```

```
[root@master ~]# kubectl apply -f calico.yaml
poddisruptionbudget.policy/calico-kube-controllers created
serviceaccount/calico-kube-controllers created
serviceaccount/calico-node created
serviceaccount/calico-cni-plugin created
configmap/calico-config created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgpfilters.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
clusterrole.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrole.rbac.authorization.k8s.io/calico-node created
clusterrole.rbac.authorization.k8s.io/calico-cni-plugin created
clusterrolebinding.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrolebinding.rbac.authorization.k8s.io/calico-node created
clusterrolebinding.rbac.authorization.k8s.io/calico-cni-plugin created
daemonset.apps/calico-node created
deployment.apps/calico-kube-controllers created
[root@master ~]#
```

Task 4: Validation Test

1. Last step is to verify all the prerequisite pods are up and running. These pods are the helping hand in performing cluster operations like scheduling, network policy and route management, API request handling, proxy service and so on. Run the following command to fetch the pods running in all the namespaces.

```
kubectl get pods  --all-namespaces
```

```
[root@master ~]# kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-86778b9f8c-z9k99   1/1     Running   0          109s
kube-system   calico-node-bb6jf                          1/1     Running   0          109s
kube-system   calico-node-jcp8v                          1/1     Running   0          109s
kube-system   calico-node-zc95s                          1/1     Running   0          109s
kube-system   coredns-76f75df574-22qn5                   1/1     Running   0          4h52m
kube-system   coredns-76f75df574-bg6hr                   1/1     Running   0          4h52m
kube-system   etcd-master                                1/1     Running   0          4h52m
kube-system   kube-apiserver-master                      1/1     Running   0          4h53m
kube-system   kube-controller-manager-master             1/1     Running   0          4h53m
kube-system   kube-proxy-q52fn                           1/1     Running   0          4h52m
kube-system   kube-proxy-xgfzz                           1/1     Running   0          3h41m
kube-system   kube-proxy-xssh9                           1/1     Running   0          3h55m
kube-system   kube-scheduler-master                      1/1     Running   0          4h52m
[root@master ~]#
```

Make sure all pods display status **RUNNING** which implies all the required pods are upand running

Completion and Conclusion

1. You have successfully downloaded and modified calico yaml.
2. You have successfully deployed calico into K8s cluster.

End Lab

1. You have successfully completed the lab.

Lab Details

1. This lab walks you through the steps to launch and configure a web server deployment in the Kubernetes cluster.
2. You will use terminal.

Introduction

What is Kubernetes?

- Kubernetes is a container orchestration tool majorly used to control and manage container-based environments.
- The smallest unit in a Kubernetes cluster is a pod. Containers run inside the pods. In some use cases, a single pod can also contain multiple containers. By using Kubernetes operators, we communicate to the pods and further, the pods communicate with the underlying containers to perform the tasks. The actual application always runs on the container but the pod is the abstraction unit from which we can interact with the containers.

  Kubernetes is required to manage and control container-based applications as it streamlines the process of performing container operations even in a large-scale environment.
- In this lab, we will create a pod that will include a container serving the Apache web server inside it.
- We require container images to run the container and all the container-related commands can be run using the command line utility kubectl. Kubectl utility provides an abstraction over the underlying container engine. This means, that whatever the container runtime is, the operations commands would be the same. It is applicable for the runtimes which are part of the CRI-O initiative.
- With Kubernetes, we can implement various deployment strategies and plan the release timeframes and in most cases, without any downtime. Also, it is a cost-effective solution as containers require minimal resources and environment based on that would cut out the overhead costs.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Install an Apache Server
4. Create a Pod with Apache server
5. Create and publish the page
6. Validation Test

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Verify the cluster configuration

1. Run the below command to check the version of `kubectl` package. Kubectl allows us to control the cluster via a command-line interface. One can run the kubectl commands followed by the resource arguments to send API requests to the master node and run the intended task over the worker node.

   kubectl version

   ```
   [root@master mylab]# kubectl version
   Client Version: v1.28.2
   Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
   Server Version: v1.28.5
   [root@master mylab]#
   ```

2. Now, verify the kubelet service is running by the following command. Kubelet service pods are running on both the master and worker nodes and this service is responsible for sending and receiving API requests within the cluster.

systemctl status kubelet

```
[root@master mylab]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2024-01-03 21:24:17 IST; 6 days ago
     Docs: https://kubernetes.io/docs/
 Main PID: 4993 (kubelet)
    Tasks: 11
   Memory: 46.1M
   CGroup: /system.slice/kubelet.service
           └─4993 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --confi...
```

3. Last step is to verify all the prerequisite pods are up and running. These pods are the helping hand in performing cluster operations like scheduling, network policy and route management, API request handling, proxy service and so on. Run the following command to fetch the pods running in all the namespaces.

kubectl get pods --all-namespaces

```
[root@master mylab]# kubectl get pods --all-namespaces
NAMESPACE          NAME                                       READY   STATUS    RESTARTS   AGE
kube-system        calico-kube-controllers-5f58fdf897-kjnvg   1/1     Running   0          5d22h
kube-system        calico-node-4b9tl                          1/1     Running   0          5d22h
kube-system        calico-node-mft59                          1/1     Running   0          5d22h
kube-system        calico-node-r2b7s                          1/1     Running   0          5d22h
kube-system        coredns-5dd5756b68-bwhr4                   1/1     Running   0          6d
kube-system        coredns-5dd5756b68-smb6d                   1/1     Running   0          6d
kube-system        etcd-master                                1/1     Running   0          6d
kube-system        kube-apiserver-master                      1/1     Running   0          6d
kube-system        kube-controller-manager-master             1/1     Running   0          6d
kube-system        kube-proxy-4ghbl                           1/1     Running   0          5d23h
kube-system        kube-proxy-bjxp6                           1/1     Running   0          6d
kube-system        kube-proxy-wtzhn                           1/1     Running   0          5d23h
kube-system        kube-scheduler-master                      1/1     Running   0          6d
metallb-system     controller-67d9f4b5bc-65jf7                1/1     Running   0          4d23h
metallb-system     speaker-8v8nk                              1/1     Running   0          4d23h
metallb-system     speaker-dk6zv                              1/1     Running   0          4d23h
metallb-system     speaker-xgm4k                              1/1     Running   0          4d23h
[root@master mylab]#
```

Make sure all pods display status **RUNNING** which implies all the required pods are up and running and we can perform the further steps.

Task 3: Create a Pod with apache server

1. In this step, we will launch a pod running the Apache web server inside a container. We will use the official container image of the web server which is httpd. Run the following command to deploy a pod with httpd container image.

kubectl run server --image=httpd

```
[root@master mylab]# kubectl run server --image=httpd
pod/server created
[root@master mylab]#
```

2. Verify the pod is running or not by the following command. Initially, the node will

download the container image from the image registry and hence it will take a few seconds to spin up the container. The following command will stream the phases in which the pod gets deployed.

```
kubectl get pods
```

```
[root@master mylab]# kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
server   1/1     Running   0          62s
[root@master mylab]#
```

3. Observe the behind the scene process and terminate the command after the pod status is running. Press **ctrl+c** to terminate the command execution.
4. Now let's go inside the container and configure web page on top of the container image downloaded from the registry

```
kubectl exec server -it  -- bash
```

5. You will see a change in the user and directory of the current terminal which implies that we are now inside the container.

Task 4 : Create and publish the page

1. First, navigate to the document root folder of the web server. The document root is the directory from which the content is being served. This simply means that whatever web pages are available inside this folder location, will be served to the end user.

```
cd /usr/local/apache2/htdocs
```

2. To add the contents into the index.html file using echo, copy and paste the below command to the shell of the container.

```
echo "<html>Hi, We are testing web server</html>" > index.html
```

3. Now, exit the container and return to the base Terminal

```
exit
```

```
[root@master mylab]# k exec server -it -- bash
root@server:/usr/local/apache2# ls
bin  build  cgi-bin  conf  error  htdocs  icons  include  logs  modules
root@server:/usr/local/apache2# cd htdocs/
root@server:/usr/local/apache2/htdocs# ls
index.html
root@server:/usr/local/apache2/htdocs# pwd
/usr/local/apache2/htdocs
root@server:/usr/local/apache2/htdocs# echo "<html> Hi, We are testing web server</html>" > index.html
root@server:/usr/local/apache2/htdocs# cat index.html
<html> Hi, We are testing web server</html>
root@server:/usr/local/apache2/htdocs# exit
exit
[root@master mylab]#
```

Till here, we have launched and configured a pod and Apache web server is running inside it. And we configured a sample index page and pasted the same into the document root of the server.

Task 5: Validation Test

1. Let's fetch the IP address of the container by the following command:

```
kubectl describe pods server | grep IP
```

2. We will now put a curl request within the terminal to check whether the container is hosting the webserver or not

```
curl <ip-of-container>/index.html
```

```
[root@master mylab]# k describe pods server | grep IP
                    cni.projectcalico.org/podIP: 10.244.189.114/32
                    cni.projectcalico.org/podIPs: 10.244.189.114/32
IP:                 10.244.189.114
IPs:
  IP:  10.244.189.114
[root@master mylab]# curl 10.244.189.114/index.html
<html> Hi, We are testing web server</html>
[root@master mylab]#
```

3. You will see the content from the index.html file on the command line which indicates that the webserver is successfully hosted on a container. The server is only accessible locally as we have not configured the networking and routing for the pod.

Completion and Conclusion

1. You have successfully verified Kubernetes cluster components.
2. You have successfully deployed a sample web server pod inside K8s cluster.
3. You have successfully created a webpage and published it backed by a container

image named httpd.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details:

1. This Lab Walks You Through Understanding Labels and Selectors in Kubernetes
2. Cluster Configuration: **1 Master And 1 Worker Node**

What are Labels and Selectors?

In Kubernetes (K8s), labels and selectors are key concepts used to organize and manage resources within the cluster. They are metadata attributes attached to K8s objects that enable grouping, filtering, and selection of resources based on specific criteria. This helps in organizing and managing resources more effectively.

Labels:

1. Labels are key-value pairs attached to K8s objects (such as pods, services, nodes, etc.) to provide additional metadata.
2. Labels are used for identifying and categorizing resources based on different characteristics, such as environment (e.g., production, development), role (e.g., frontend, backend), version, etc.

Selectors:

1. Selectors are used to filter and select resources based on their labels. They allow you to define criteria for identifying a set of resources.
2. Selectors are essential for operations like deploying applications to specific environments, grouping resources for scaling, and defining access policies.

Lab Tasks:

1. Define Kubernetes Pod
2. List and Retrieve the Pod

Launching Lab Environment:

1. Launch The Lab Environment By opening the **Master** machine.

2. Open the terminal.

Lab Steps:

Define Kubernetes Pod:

In this step, we will create the manifest file that defines a Kubernetes Pod configuration. We are going to create two pods in this step one with environment as production and the other one with production environment.

Create manifest file using YAML approach

1. Create a YAML file using the command below:

```
vim labelpod.yml
```

2. Now pass the below sample yml code. This will attempt to create the specified Pod based on the provided details. Once done press **CTRL+X** and the **Y** and **Enter** to exit the terminal.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: quay.io/gauravkumar9130/nginx
```

3. The next step is to apply this yml file to create the deployment as declared.

```
kubectl apply -f labelpod.yml
```

```
[root@master mylab]# vim labelpod.yml
[root@master mylab]# kubectl apply -f labelpod.yml
pod/pod-demo created
[root@master mylab]#
```

4. Now let's create the other pod with **development** environment. To do so make use of the following command:

```
nano labelpod2.yml
```

5. Paste the following **yaml manifest** in the editor and save and return.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-demo2
  labels:
    environment: development
    app: nginx
spec:
  containers:
  - name: nginx
    image: quay.io/gauravkumar9130/nginx
```

6. Now apply the yml file to the deployment.

```
kubectl apply -f whiz2.yml
```

```
[root@master mylab]# vim labelpod2.yml
[root@master mylab]# kubectl apply -f labelpod2.yml
pod/pod-demo2 created
[root@master mylab]#
```

7. Now, let's list both the created pods with the help of the following command to ensure successful creation:

```
kubectl get pods --show-labels
```

```
[root@master mylab]# kubectl get pods --show-labels
NAME                                READY   STATUS    RESTARTS   AGE   LABELS
nginx-deployment-5995975c44-dxdlc   1/1     Running   0          28m   app=nginx,pod-template-hash=5995975c44
nginx-deployment-5995975c44-f6fng   1/1     Running   0          28m   app=nginx,pod-template-hash=5995975c44
pod-demo                            1/1     Running   0          13s   app=nginx,environment=production
pod-demo2                           1/1     Running   0          49s   app=nginx,environment=development
[root@master mylab]#
```

List and Retrieve the Pod:

In this step, we will list pods based on label selectors. In our case, we are retrieving pods that have labels **environment=production**. The command would be:

```
kubectl get pods -l environment=production
```

```
[root@master mylab]# kubectl get pods -l environment=production
NAME       READY   STATUS    RESTARTS   AGE
pod-demo   1/1     Running   0          81s
[root@master mylab]#
```

End Lab:

1. You Have Successfully Completed The Lab.

2. Once You Have Completed The Steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and configure replication controller resources in the Kubernetes.
2. You will practice using the terminal.

Introduction

What is Replication?

- Replication in Kubernetes is an important resource that maintains identical copies of the pod. It is responsible for managing the lifecycle of the pods. Containerized applications generally have a large number of containers and managing the lifecycle of those pods manually isn't a good option as containers are volatile.
- The replication concept ensures the uptime of the pods required at one point in time. The administrator defines the ideal number of pods supposed to be running while declaring the configuration and the replication service keeps track of the mentioned number. Replication service works in maintaining the desired state for the application.
- So, if some pods crash or fail, replication takes note of this and launches child replicas having the same environment to ensure the application doesn't observe downtime due to the unavailability of the service.
- Replication is a concept and there are 2 services responsible for governing replication.

  ReplicaSet
  Replication Controller

What is ReplicationController?

- As we already know replication takes care of the desired state in containerized applications. ReplicationContoller ensures a specified number of pods are up and running for a given point in time. This program keeps track of the underlying pods with the labels attached to them and matches the number of running pods with the desired number of pods.
  This program always keeps on monitoring the underlying pods and maintains the exact number of replicas. The use of ReplicationController is to run an identical number of pods and ensure uptime.
- The short abbreviation of ReplicationController is "RC" which is widely used in the command line.
  RC works with the label selector concept. More specifically, equality-based selectors.
- We will see this concept in detail in the later half.

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Create the YML declarative file for RC
4. Describe the resource
5. Delete the replication controller

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Create RC using YAML approach

We can create RC resources using both the command line and the YAML approach. But, as the application grows, it is recommended practice to go with the YAML approach to maintain all the application configurations in a declarative manner.

The keyword **replicas** in the YAML file are referred to as the desired number of pods and add labels to the resource so that the program can monitor and take care of the uptime. labels are linked with the pods and used to keep track of the replication program.

Using YAML approach

1. Create a YAML file using the command below:

```
vim replication_controller.yml
```

2. Now pass the below sample yml code. Here, apiVersion is v1 which means this resource is included and maintained under the mentioned version of the Kubernetes API. Also, the keyword kind suggests the type of the resource which is ReplicationController in this scenario.

```
apiVersion: v1
kind: ReplicationController
metadata:
 name: nginx
spec:
 replicas: 2
 selector:
  app: nginx
 template:
  metadata:
   name: nginx
    labels:
     app: nginx
  spec:
   containers:
   - name: nginx
     image: quay.io/gauravkumar9130/nginx
```

3. The next step is to apply this yml file to create the RC as declared.

```
kubectl apply -f replication_controller.yml
```

```
[root@master mylab]# vim replication_controller.yml
[root@master mylab]# k apply -f replication_controller.yml
replicationcontroller/nginx created
[root@master mylab]# kubectl get pods -w
NAME                          READY   STATUS    RESTARTS   AGE
cm-deploy-55b5ddf678-jxzm7    1/1     Running   0          78m
nginx-p2cnz                   1/1     Running   0          12s
nginx-rmvph                   1/1     Running   0          13s
^C[root@master mylab]#
```

With this command, we can successfully create the RC from the yml file created above. Verify the same by running the **kubectl get pods** command. It may take a few minutes first time as the nginx container images are being pulled from the registry.

Task 3: Describe the replication controller

1. As we launched the ReplicationController in the previous step, let's now perform a describe operation to get the detailed data about the replication controller resource.

```
kubectl describe rc/nginx
```

2. Run the above command to get a detailed description of the launched resource.
   From the whole output, "pod status" and "replicas" are important as they tell the desired state pod status and the actual desired state. From these two fields, we can determine whether the controller is working or not.

```
[root@master mylab]# kubectl describe rc/nginx
Name:         nginx
Namespace:    default
Selector:     app=nginx
Labels:       app=nginx
Annotations:  <none>
Replicas:     2 current / 2 desired
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=nginx
  Containers:
   c1:
    Image:        quay.io/pandeysp/nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  79s   replication-controller  Created pod: nginx-rmvph
  Normal  SuccessfulCreate  79s   replication-controller  Created pod: nginx-p2cnz
[root@master mylab]#
```

Task 4: How to delete RC?

1. Delete RC including the pods
   This will delete the RC resource and all underlying pods by scaling down them to 0. It is suggested to take a backup before performing this as it will delete the replication controller as well as all the running pods including the temporary data.

```
kubectl delete rc nginx
```

```
[root@master mylab]# k delete rc nginx
replicationcontroller "nginx" deleted
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created a replication controller resource in the Kubernetes.
2. You have successfully described the created resources and observed the detailed information of the created RC
3. You have deleted the configured RC.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to configure ReplicaSet resources using the kubectl command-line utility.
2. You will use terminal and create a sample declarative file to launch the pod with the desired state configuration.

Introduction

What is ReplicaSet?

- Replicaset works on the concept of replication. The primary responsibility of ReplicaSet is to maintain the desired number of pods up and running at one point in time. It is often used to guarantee that specified pods are available at all times. This program monitors underlying pods with specific label and keeps track of the uptime of those pods.
- It works on the label selector method. But ReplicaSet uses a "set-based" selector which provides more customization than ReplicationController. The query method of the labels provides wider options which makes this service widely used nowadays instead of ReplicationController.
- The short form of ReplicaSet is "rs". We mostly use the abbreviated term in the command line.
- It creates new pods with the same labels and templates if the desired state is not matched and destroys extra pods if running pods are more than the desired state.

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Create the YML declarative file for RS
4. Describe the resource
5. Delete the ReplicaSet

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Configure ReplicaSet

We can use the command-line and YAML approach to create ReplicaSet and both approaches give the same results. However, the YAML approach is more used and recommended as the declarative approach gives more flexibility and we can maintain the whole configuration of the resource in a single file and maintain versioning which can be helpful at the time of rollback to previous versions.

The main attributes in the replica set are labels because the whole program works to match the labels on the underlying resources. Take note of the kind and labels keywords in the declarative file below. Here, we've attached multiple tags to the pods based on the environment and provide a unique identity to the resource.

ReplicaSet using YAML approach:

1. Create a declarative YAML file using the below command.

```
vim replica_set.yml
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: frontend
 labels:
  app: guestbook
  tier: frontend
spec:
 replicas: 2
 selector:
  matchLabels:
   tier: frontend
 template:
  metadata:
   labels:
    tier: frontend
  spec:
   containers:
   - name: webapp
     image: quay.io/gauravkumar9130/mywebapp
```

2. Now paste the YAML file below in the editor. The pod label as a specification to indicate the rs to watch the pods matching the set-based rule and maintain uptime for the same. For creating the YAML code, kind would be "ReplicaSet" and apiVersion would be "apps/v1" that indicates ``rs`` has been released in a later part than RC

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: frontend
 labels:
  app: guestbook
  tier:frontend
spec:
 replicas: 2
 selector:
  matchLabels:
   tier: frontend
  template:
   metadata:
    labels:
     tier: frontend
   spec:
    containers:
    - name: webapp
      image: quay.io/gauravkumar9130/mywebapp
```

3. Having this file completed, run the below command to apply and create the ReplicaSet as mentioned:

```
kubectl apply -f replica_set.yml
```

```
[root@master mylab]# kubectl apply -f replica_set.yml
replicaset.apps/frontend created
[root@master mylab]# kubectl get pod -w
NAME             READY   STATUS              RESTARTS   AGE
frontend-hc22f   0/1     ContainerCreating   0          2s
frontend-rz8qv   0/1     ContainerCreating   0          2s
frontend-hc22f   1/1     Running             0          7s
frontend-rz8qv   1/1     Running             0          7s
^C[root@master mylab]#
```

So, this is one of the ways to create an independent ReplicaSet and ensure the uptime for the application pods. The watch command will stream the whole container deployment process and we can see the desired number of pods up and running after a few seconds. First-time execution may take a little more time as the container images are being pulled from the hub..

Task 3: Describe the ReplicaSet:

The task of creating the ReplicaSet is done. Now is the time to fetch the details about it and understand it in a better format. Use the describe command to get all the details about ``rs``.

```
kubectl describes replicaset/frontend
```

```
[root@master mylab]# kubectl describe rs/frontend
Name:         frontend
Namespace:    default
Selector:     tier=frontend
Labels:       app=guestbook
              tier=frontend
Annotations:  <none>
Replicas:     2 current / 2 desired
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  tier=frontend
  Containers:
   webapp:
    Image:        quay.io/gauravkumar9130/mywebapp
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age    From                   Message
  ----    ------            ----   ----                   -------
  Normal  SuccessfulCreate  9m55s  replicaset-controller  Created pod: frontend-rz8qv
  Normal  SuccessfulCreate  9m55s  replicaset-controller  Created pod: frontend-hc22f
[root@master mylab]#
```

In ReplicaSet, pod status and desired number of pods are the key outputs to watch out for. From these details, we can get a whole idea about the process. Also, the events section in the bottom states all the details about the pod and throws the error in case of deployment failure. With the describe command, we can see the actual running pods and desired number of pods and take needful actions if there is some fault.
Bare pods are not a recommended practice if you're thinking of an environment at scale. Pods being watched by a program like ``rs`` gives assurance of minimal downtime.

Task 4: Delete the resources

We learned how to create and describe rs resources and now we'll see how to delete the created rs using the command line.

1. **Delete the whole ReplicaSet:**
   This approach is used when we want to destroy the rs environment completely. Here, the rs and underlying pods will be deleted and further uptime won't be maintained as the watching program rs is not on the head. It is recommended to take backup of the container data before running this command as it will erase the whole temporary data.

```
kubectl delete replicaset frontend
```

```
[root@master mylab]# kubectl delete rs frontend
replicaset.apps "frontend" deleted
[root@master mylab]# kubectl get rs
No resources found in default namespace.
[root@master mylab]# kubectl get pods
No resources found in default namespace.
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created a ReplicaSet resource.
2. You have successfully described and deleted the created resources through it.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and configure service networking with Cluster IP type.
2. You will use the terminal.

Introduction

What is Cluster IP service?

- As we already know about service networking in Kubernetes, Cluster IP is yet another type of defining networking rules within the cluster.
- This service allows us to expose the pods internally for particular addresses and the mentioned resources will be only accessible inside the cluster.
- By default, any provisioned resource will have this policy applied and the resources are accessible within the clutter by default.
- Using this service, we can restrict access to some applications from external access and only allow other services within the cluster.
- For example, in a multi-tier application, rather than exposing the backend or database pods to the public endpoint, we only access the frontend to the external access and for inter-connection of database and backend, we implement Cluster IP service.
- Also, with this service, we can allow inter-communication for a specific static IP address defined in the manifest file.

Task Details

1. Launching Lab Environment.
2. Using the terminal
3. Create the YML declarative file for the sample application.
4. Verify the service using Kubectl command
5. Delete the service

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Create the YML declarative file for the application

In this step, we will create the manifest file for a sample nginx application and we will launch multiple replicas for the same. Then, we will create a cluster IP service and verify the endpoints of the replicas are included in the service

Create a manifest file using YAML approach

1. Create a YAML file using the command below

```
vim ecom.yml
```

2. Now pass the below sample yml code. Here, we are only creating replica set of the web application.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: ecom
spec:
 replicas: 3
 selector:
  matchLabels:
   app: ecommerce
 template:
  metadata:
   labels:
    app: ecommerce
  spec:
   containers:
   - name: c1
     image: quay.io/gauravkumar9130/mywebapp
```

3. The next step is to apply this yml file to create the resource as declared.

```
kubectl apply -f ecom.yml
```

4. Afterward, by running the **kubectl get pods -o wide** command, we can verify the nginx pods are up and running with 3 identical replicas.

5. Now, let's define the YAML file for the Cluster IP service where we will use the mapped port number to expose within the cluster. Create a new file with the command below:

```
[root@master mylab]# kubectl create -f ecom.yml
replicaset.apps/ecommerce created
[root@master mylab]# kubectl get pods -o wide
NAME              READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
ecommerce-54fjb   1/1     Running   0          44s   10.244.189.113   worker2   <none>           <none>
ecommerce-bbj79   1/1     Running   0          44s   10.244.189.112   worker2   <none>           <none>
ecommerce-l67kr   1/1     Running   0          44s   10.244.235.181   worker1   <none>           <none>
[root@master mylab]#
```

```
vim clusterip.yml
```

6. Now paste the following code in the file and save the same.

Here, we have used the label selector method to match the labels for the application pods and corresponding replicas to perform network communication and load balancing. The kind Service refers to the networking resource in the Kubernetes. Also, we are using TCP protocol for this particular service. The argument of mentioning protocol is optional.

```
apiVersion: v1
kind: Service
metadata:
 name: my-app
spec:
 type: ClusterIP
 ports:
 - targetPort: 80
   port: 80
 selector:
  app: ecommerce
```

7. Now run the below command to apply the service resource in the cluster.

```
kubectl apply -f clusterip.yml
```

```
[root@master mylab]# vim clusterip.yml
[root@master mylab]# kubectl apply -f clusterip.yml
service/my-app created
[root@master mylab]# kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   5d23h
my-app       ClusterIP   10.102.21.41   <none>        80/TCP    15s
[root@master mylab]#
```

We have deployed the YAML file for both the replica set and the service resource.

Task 3: Verify the service using Kubectl command

1. Run the command below to verify the service resource is applied and working properly.

```
kubectl get svc
```

2. You will see a service named “my-app” where the type is mentioned as “ClusterIP” and the port protocol is mapped as declared. In order to get more details about the deployed service, run the describe command and notice the output.

```
kubectl describe service my-app
```

```
[root@master mylab]# kubectl describe service my-app
Name:              my-app
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=ecommerce
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.102.21.41
IPs:               10.102.21.41
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.189.112:80,10.244.189.113:80,10.244.235.181:80
Session Affinity:  None
Events:            <none>
[root@master mylab]#
```

3. Here, we can see that the labels mentioned in the deployment are mapped with the service which means the service is independent of the pod lifecycle and only works with respect to the deployment labels and replicas.

   Also, by checking the endpoints field, we can confirm that the endpoints are nothing but the private ip addresses of the deployment pods which we launched earlier.
   Apart from this, ClusterIP can also be used for assigning user-defined IP addresses to the pods and performing networking. Also, by using DNS services we can eliminate usage of IP addresses and call the services by domain names itself.
4. You can verify the application
   `kubectl get svc`
   `curl <clusterIP>:80`

```
[root@master mylab]# kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   5d23h
my-app       ClusterIP   10.102.21.41   <none>        80/TCP    27m
[root@master mylab]# curl 10.102.21.41:80
<!doctype html>
<title> ecommerce-l67kr Application </title>
<body style="background: #30336b;"></body>
<div style="color: #e4e4e4;
    text-align:  center;
    height: 90px;
    vertical-align:  middle;">

  <h1>Welcome To ecommerce-l67kr</h1>



</div>[root@master mylab]#
```

Task 4 : How to delete the service?

By running the command below, we can delete the ClusterIP service. After performing this action, the service will no longer serve as a networking component and no load balancing will take place for the mentioned endpoints.

```
kubectl delete svc my-app
```

```
[root@master mylab]# kubectl delete svc my-app
service "my-app" deleted
[root@master mylab]# kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   5d23h
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created a ClusterIP service in Kubernetes.
2. You have successfully described the created resources and verified the applied resources are working.
3. You have deleted the configured service and come across a few use cases of the same service.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and configure service networking with NodePort type
2. You will use terminal.

Introduction

What are services in Kubernetes?

- Services in Kubernetes allow us to implement a logical set of networking rules. Kubernetes internally allocates unique IP addresses to each pod and by defining services, we can control the communication with the pods.
- As pods are volatile and pods are created and destroyed multiple times to match the desired state, a pod becomes a non-permanent resource.
- Kubernetes services provide abstraction over the underlying pods and set networking policies from which pods can communicate with each other and the application can be exposed to the end users.
- The set of pods to route the traffic is managed through unique selectors. For example, pods having label frontend and backend respectively will have internal network communication by the service resource.
- Generally, there are 3 types of service resources in Kubernetes:
    - Cluster IP
    - NodePort
    - Load Balancer
- In this lab, we will look into the NodePort service with hands-on demo examples.

What is NodePort?

- As we already know pod networking is majorly defined by the service resource, NodePort provides the capability to expose the cluster to a port or set of ports for external access.
- This service allows us to open specific ports of the mentioned nodes from where the external user can access the underlying application running on the pod.
- So, rather than connecting to the ephemeral pods on a specific IP address, the NodePort

service created an abstraction and provided a single point of access to the end user. On the other hand, it equally distributes traffic on the running pods by using the selector to detect the label and route the request.

- So, after exposing the port from the host node, one can share the combination of the host IP address and the exposed port as a proxy to access the service running inside a pod of that node.
- The default port range to expose from the node is 30000 to 32767. If the port is not defined in the manifest while creating the service, Kubernetes will automatically assign unused and available ports from the mentioned range and bind the same to the NodePort.
- Now let's implement the NodePort service and understand the concept in detail.

Task Details

1. Launching Lab Environment.
2. SSH through the Whiz terminal
3. Create the YML declarative file for the NodePort resource
4. Verify the service using Kubectl command
5. Delete the service

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening the **Master** machine.
2. Open the terminal.

Task 2: Create the YML declarative file for NodePort resource

In this step, we will create the manifest file for NodePort service

Here, we have to mention 2 types of ports to route the traffic to the application running inside the pod. One is the port to be exposed from the node(can be picked from the default range mentioned above). The other port is the actual application port. For example: for a web server running inside the pod, the target port would be 80.

Using YAML approach

1. Create a YAML file for the deployment of sample nginx pod using the command below:

```
vim deploy.yml
```

2. Copy the below code into the YML file which will create a deployment of nginx server and by default we will serve the content from port 80.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
  app: nginx
 name: nginx-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
     image: quay.io/gauravkumar9130/mywebapp
```

3. The next step is to apply the YML file to launch the deployment using the command below:

```
kubectl apply -f deploy.yml
```

```
[root@master mylab]# vim deploy.yml
[root@master mylab]# kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2/2     2            2           71s
[root@master mylab]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-5995975c44-dxdlc   1/1     Running   0          74s
nginx-deployment-5995975c44-f6fng   1/1     Running   0          74s
[root@master mylab]#
```

4. Now comes the part to create the service and define the networking rule using which we will expose the nginx pod to the public world. Create a YML file using the command below:

```
vim nodeport.yml
```

5. Now pass the below sample yml code. Here, apiVersion is v1 which means this resource is included and maintained under the mentioned version of the Kubernetes API. Also, the keyword kind suggests the type of resource which is service in this scenario. We will use label selectors for the service to detect the apps running on the same hierarchy and eventually, it will distribute traffic equally based on the nodes.

```
apiVersion: v1
kind: Service
metadata:
 name: my-service
 labels:
  app: nginx
spec:
 selector:
  app: nginx
 type: NodePort
 ports:
 - nodePort: 31000
   port: 80
   targetPort: 80
```

6. The next step is to apply this yml file to create the service as declared.

```
kubectl apply -f nodeport.yml
```

Till now, we have launched nginx server and created NodePort service which will expose the port number 31000 from the host node and bind the same to the target port 80 which is the default port for nginx.

Task 3: Verify the service using Kubectl command

1. Let's verify the deployed resources with kubectl commands. First, we will check the deployments and verify the running status of the pods.

```
kubectl get pods
```

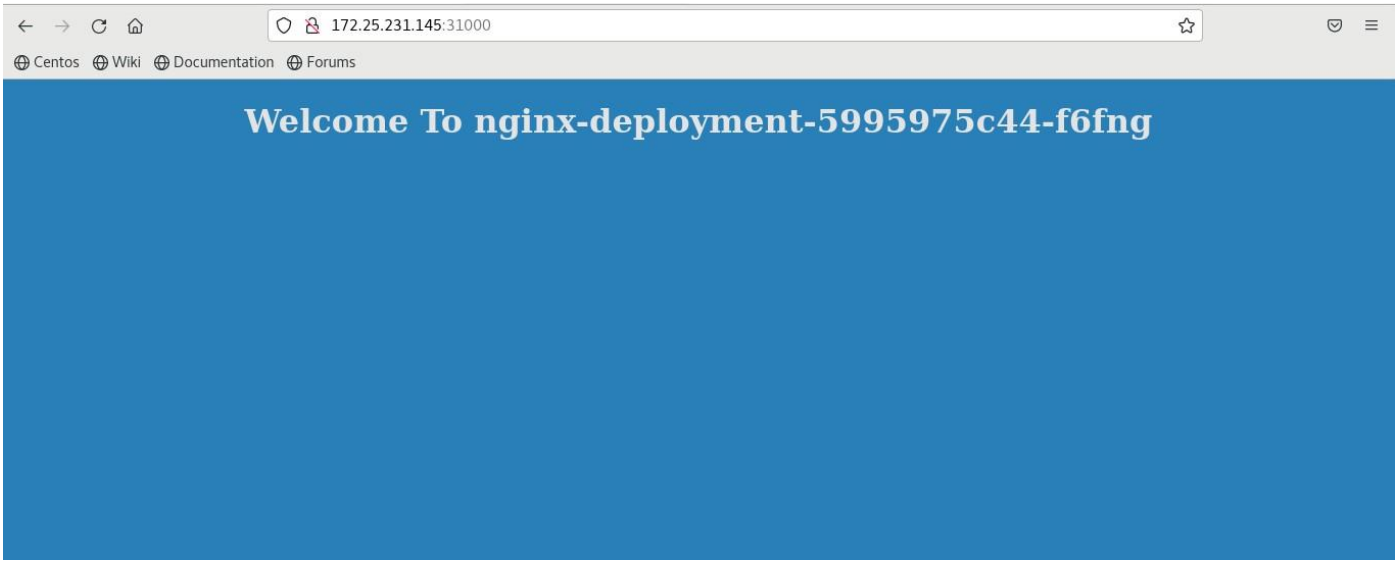
2. After verifying 2 replicas of nginx are running, run the below command to check whether the NodePort service is applied or not:

```
kubectl get svc
```

```
[root@master mylab]# kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        12d
my-service   NodePort    10.105.33.163   <none>        80:31000/TCP   61s
[root@master mylab]# kubectl describe svc my-service
Name:                     my-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.105.33.163
IPs:                      10.105.33.163
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  31000/TCP
Endpoints:                10.244.189.111:80,10.244.235.164:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
[root@master mylab]#
```

Here the port mapping states that port number 31000 from the host is linked with the port 80 of the pods having the label mentioned in the declarative file.

Now pick the node and copy the same in ip_address:31000 format and run it in the new browser window. The webpage will be displayed.



Hence, we can verify the cluster is exposed to the port and it is serving the content from the underlying pods. Besides, the service will provide load-balancing features and equally divert the requests to the replicas of the pods.

Task 5: How to delete the service?

1. This will delete the service resource and the exposed service will be stopped. Afterward, the request hit to the host from the node port will not be routed to the nginx pod.

```
kubectl delete svc my-service
```

```
[root@master mylab]# kubectl delete svc my-service
service "my-service" deleted
[root@master mylab]# kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   12d
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created a NodePort service resource in the Kubernetes.
2. You have successfully exposed the Nginx-app service to public requests.
3. You have deleted the configured service and detached the exposed application.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and configure service networking with LoadBalancer type
2. You will using terminal.

Introduction

What is a Load Balancer service in Kubernetes?

- Load balancer is yet another service resource in Kubernetes which allows us to perform networking on the Cluster.
- Consider a general scenario of a multi-node cluster where each node is serving the same application inside the labeled pod. To expose the application to the end users, we can use the NodePort service. But each node will have a separate IP address and we have to explicitly share the IP port combination to the users for them to use the application. With this, we disclose the original IP of the node that is serving the application and it increases potential threat to the application as the endpoint is directly shared.
- Also, in a situation of node failure, the end clients won't be able to access the application as the given node is no longer reachable.
- Hence, we need a service that runs on top of all underlying nodes and works as a proxy service between the clients and the application servers. Here we use the LoadBalancer service in Kubernetes.
- We can attach external, cloud provider LoadBalancers to the existing clusters too. Also, this proxy service will equally distribute traffic to the underlying nodes and provide a LoadBalancing service.
- In case the application is running on the cloud, by mentioning the LoadBalancer type service, it automatically provisions a load balancer and returns a public ip to share with the users.

Task Details

1. Launching Lab Environment.
2. Using the terminal.
3. Create the YML declarative file for the LoadBalancer resource

4. Verify the service using Kubectl command
5. Delete the service

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Create the YML declarative file for LoadBalancer resource

In this step, we will create the manifest file for the LoadBalancer service

Here, we have to mention 2 types of ports to route the traffic to the application running inside the pod. One is the port to be exposed from the node. The other port is the actual application port. For example: for a web server running inside thepod, the target port would be 80.

Using YAML approach

1. Create a YAML file for the deployment of a sample nginx pod using the command below:

```
vim deploy.yml
```

2. Copy the below code into the YML file which will create a deployment of the nginx server and by default we will serve the content from port 80.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
  app: nginx
 name: app-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - image: quay.io/gauravkumar9130/mywebapp
     name: nginx
```

3. The next step is to apply the YML file to launch the deployment using the command below:

```
kubectl apply -f deploy.yml
```

4. Now comes the part to create the service and define the networking rule using which we will expose the nginx pod to the public world. Create a YML file using the command below:

```
vim loadbalancer.yml
```

5. Now pass the below sample yml code. Here, apiVersion is v1 which means this resource is included and maintained under the mentioned version of the Kubernetes API. Also, the keyword kind suggests the type of resource which is service in this scenario. We will use label selectors for the service to detect the apps running on the same hierarchy and eventually, it will distribute traffic equally based on the nodes.

```
apiVersion: v1
kind: Service
metadata:
 name: my-app-service
 labels:
  app: nginx
spec:
 selector:
  app: nginx
 type: LoadBalancer
 ports:
 - port: 80
   targetPort: 80
```

6. The next step is to apply this yml file to create the service as declared.

```
kubectl apply -f loadbalancer.yml
```

Till now, we have launched the nginx server and created a LoadBalancer service, and configured Kubernetes to expose the Nginx service to the external world. This allows external users to access the Nginx application through the assigned external IP.

Task 3: Verify the service using Kubectl command

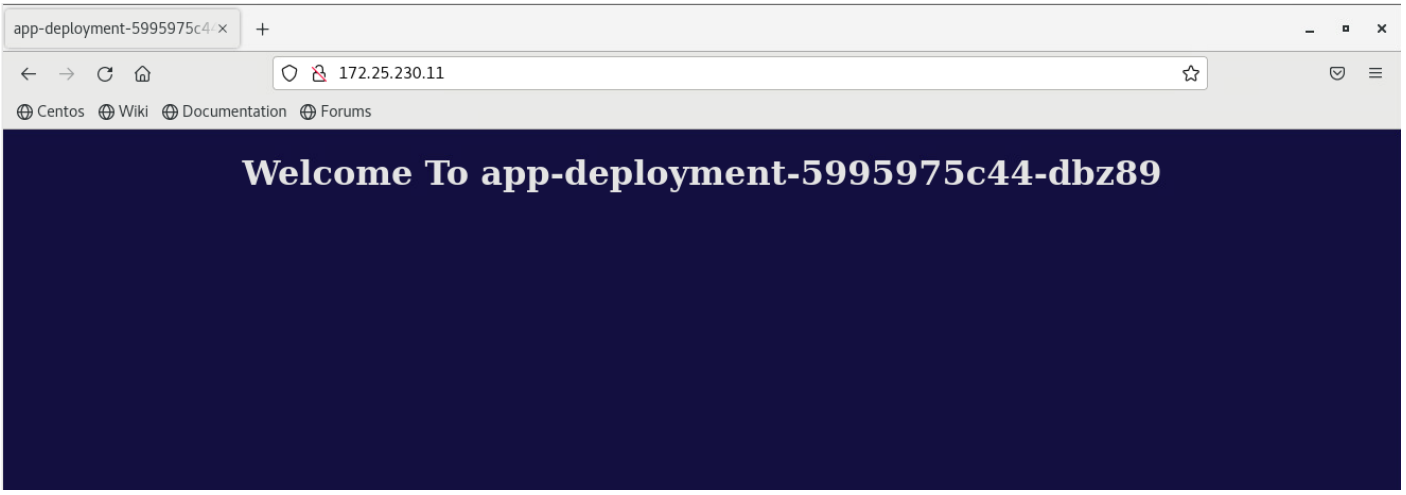
1. Let's verify the deployed resources with kubectl commands. First, we will check the deployments and verify the running status of the pods.

```
kubectl get pods
```

2. After verifying 2 replicas of nginx are running, run the below command to check whether the LoadBalancer service is applied or not:

```
kubectl get svc
```

```
[root@master mylab]# k get pods,svc
NAME                                    READY   STATUS    RESTARTS   AGE
pod/app-deployment-5995975c44-4dfs8     1/1     Running   0          27s
pod/app-deployment-5995975c44-dbz89     1/1     Running   0          27s

NAME                     TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
service/kubernetes       ClusterIP      10.96.0.1       <none>          443/TCP        10d
service/my-app-service   LoadBalancer   10.107.213.76   172.25.230.11   80:32178/TCP   62s
[root@master mylab]#
```



Here the port mapping states that port number 32178 from the host is linked with the port 80 of the pods having the label mentioned in the declarative file.

Now pick the external ip and run it in the new browser window. The nginx-app default webpage will be displayed.

Hence, we can verify the cluster is exposed to the port and it is serving the content from the underlying pods. Besides, the service will provide load-balancing features and equally divert the requests to the replicas of the pods.

Task 4: How to delete the service?

1. This will delete the service resource and the exposed service will be stopped. Afterward, the request hit to the host from the port will not be routed to the nginx pod.

```
kubectl delete svc my-service
```

```
kubectl delete deploy app-deployment
```

Completion and Conclusion

1. You have successfully created a LoadBalancer service resource.
2. You have successfully exposed the nginx-based app to external-IP.
3. You have deleted the configured service and the exposed application.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand Daemon Set.

Introduction

What is DaemonSet and Why use it?

- DaemonSet resource ensures identical copies of the declared pods in all the nodes(or specific) added in the cluster.
- Typically, DaemonSet is used in scenarios where the same type of daemon (abbreviated as service) is meant to run on all the nodes to serve a specific use case. For example, a monitoring solution pod is needed to be deployed in all the nodes (maybe in a multi-container pod setup) where the monitoring solution will work as a background daemon to the business logic application running on the front.
- DaemonSet is dynamic. That means, in case of node addition or deletion, the mentioned daemon pods will be automatically launched and destroyed respectively. A program named garbage collector will clean up the deleted pods and corresponding configuration.
- The major difference between ReplicaSet and DaemonSet is that the former is meant for managing the lifecycle of the pod and ensuring the uptime of the desired number of pods at one point in time. Whereas, the latter is used where we want to inject identical copies of the pod to all the underlying nodes which need not have a monitoring process like ReplicaSet to ensure the multiple replicas of the same pod.
- To remove the pod launched using DaemonSet, just delete the DaemonSet resource from the master node and it will remove all the underlying pods. Manually deleting pods won't remove them as the DaemonSet resource will schedule other pods in the same node.

Task Details

1. Create a manifest file for sample Daemonset
2. Verify Daemonset resource using kubectl command
3. Delete the Daemonset

Lab Steps:

Task 1: Create a manifest file for sample Daemonset

- In this step, we will create a manifest file for the sample Prometheus application(monitoring tool) and launch it as a DaemonSet.
- In this lab, we are implementing the concept over a multi-node Kubernetes cluster. You can verify the same by running kubectl get nodes command.
- In the YAML declarative file, we need to mention daemons as the kind of resource.

Also, in the template block, we specify the labels and selector for internal management of the program to make sure one copy of the pod having the mentioned label is running in each node (or mentioned nodes)

- Here we will use the sample Prometheus node exporter container image and launch it as a single pod. Make sure that the labels and selectors are the same otherwise the selector won't be able to keep track of the daemon

Using yaml approach

1. Create a YAML file for the deployment of sample Prometheus image daemonset
   - using the command below:

```
vim daemonset.yml
```

2. Copy the below code into the YML file which contains the declaration of the daemon set resource with the required blocks explained above.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: prometheus-daemonset
spec:
  selector:
    matchLabels:
      tier: monitoring
      name: prometheus-exporter
  template:
    metadata:
      labels:
        tier: monitoring
        name: prometheus-exporter
    spec:
      containers:
      - name: prometheus
        image: quay.io/gauravkumar9130/prometheus
        ports:
        - containerPort: 80
```

3. The next step is to apply the YML file to launch the deployment using the command below:

```
kubectl apply -f daemonset.yml
```

```
[root@master mylab]# vim daemonset.yml
[root@master mylab]# kubectl apply -f daemonset.yml
daemonset.apps/prometheus-daemonset created
[root@master mylab]#
```

Task 2: Verify Daemonset resource using kubectl command

Let's verify the deployed DaemonSet with the below command.

```
kubectl get daemonset
```

```
[root@master mylab]# kubectl get daemonset
NAME                 DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
prometheus-daemonset   2         2         2       2            2           <none>          59s
[root@master mylab]#
```

1. Here we can see the running and desired count with the health status of the launched pods from DaemonSet.
2. To get more insights from the resource, run the describe command and note a few more details about the launched resource.

```
kubectl describe daemonset
```

```
[root@master mylab]# kubectl describe daemonset
Name:           prometheus-daemonset
Selector:       name=prometheus-exporter,tier=monitoring
Node-Selector:  <none>
Labels:         <none>
Annotations:    deprecated.daemonset.template.generation: 1
Desired Number of Nodes Scheduled: 2
Current Number of Nodes Scheduled: 2
Number of Nodes Scheduled with Up-to-date Pods: 2
Number of Nodes Scheduled with Available Pods: 2
Number of Nodes Misscheduled: 0
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  name=prometheus-exporter
           tier=monitoring
  Containers:
   prometheus:
    Image:        quay.io/gauravkumar9130/prometheus
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age    From                  Message
  ----    ------            ----   ----                  -------
  Normal  SuccessfulCreate  3m30s  daemonset-controller  Created pod: prometheus-daemonset-6dc4p
  Normal  SuccessfulCreate  3m30s  daemonset-controller  Created pod: prometheus-daemonset-nwd8v
[root@master mylab]#
```

In the output, we can see the labels and selectors are matching with the labels of the pods.
With DaemonSet, we can control the scheduling of the daemon pod and restrict the provisioning of the pods to selective nodes only. In the output of the above command, we can see Node Selector is set to none which means all the nodes will get one copy of the Daemon. By mentioning specific nodes with this feature, we can limit the deployment of the daemon pods.

Task 4: How to delete the Daemonset

1. This will delete the daemonSet configuration and clean up all the provisioned daemon pods in all the nodes.

```
kubectl delete daemonset prometheus-daemonset
```

One thing to note is that we can restrict the deletion of the daemon pods by setting up taint and allowing tolerance for specific nodes. Also, directly deleting the pods will only result in the deployment of the same pods again from the DaemonSet program as it is tracking the pods with the label selectors, and on not matching the desired pod count, it will compensate for new pods in the nodes.

Completion and conclusion

1. You have successfully created a DaemonSet resource in a multi-node Kubernetes cluster.
2. You have successfully observed the resource configuration.
3. You have deleted the configured DaemonSet and removed the daemon pods from all the nodes.

End Lab:

1. You Have successfully performed the lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details:

1. This Lab will walk you through the step-by-step guide to help you deploy Pods with manual scheduling.

What is Manual Scheduling in Kubernetes?

In Kubernetes, manual scheduling refers to the process of explicitly specifying the node where a particular Pod should be scheduled. By default, Kubernetes uses a scheduler that automatically determines the best node for a Pod based on resource requirements, constraints, and other factors. However, in some scenarios, you may want to take control over the scheduling process and dictate on which node a Pod should run. This is where manual scheduling comes into play.

Lab Tasks:

1. Open the VM lab
2. Create a pod with random scheduling
3. View the Pods
4. Manually Schedule Pods
5. Verify Manual Scheduling

Launching Lab Environment:

1. Launch the Lab Environment by opening **Master** machine
2. Open the terminal.

Lab Steps:

Task 1: Create a Pod with Manual Scheduling

1. Enter the below command to create a file named random-pod.yaml.

```
vim random-pod.yaml
```

2. After you enter the above command, copy and paste the following Pod Configuration in that new window.

Note: This YAML defines a basic Kubernetes Pod running a Nginx container.

```
apiVersion: v1
kind: Pod
metadata:
 labels:
  run: nginx-random-scheduling
 name: nginx-random-scheduling
spec:
 containers:
 - image: quay.io/gauravkumar9130/nginx
   name: nginx
```

3. Now press '**ESC**' and then press '**:wq**' to save and exit.
4. Apply the Pod using the following command.

```
kubectl create -f random-pod.yaml
```

```
[root@master mylab]# kubectl create -f random-pod.yaml
pod/nginx-random-scheduling created
[root@master mylab]#
```

Task 2: View the Pods

In this step, we will confirm if the pods have been created or not.

1. Execute the command below to verify that the Pods have been successfully created and are running.

```
kubectl get pods -o wide
```

2. The output will look like this. We can see that the pod was randomly assigned to any of the available nodes.

```
[root@master mylab]# kubectl get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE     IP               NODE      NOMINATED NODE   READINESS GATES
nginx-random-scheduling   1/1     Running   0          3m15s   10.244.189.122   worker2   <none>           <none>
[root@master mylab]#
```

Task 3: Manually Schedule Pods

In this step, we will create the **pod-manual-scheduled.yaml** file to include node name. Node name is used to specify rules that constrain the scheduling of Pods based on node name.

1. Execute the command below to get the node details. Make a note of any of the desired node name.

```
kubectl get node
```

```
[root@master mylab]# kubectl get nodes
NAME       STATUS    ROLES           AGE     VERSION
master     Ready     control-plane   7d1h    v1.28.2
worker1    Ready     <none>          7d      v1.28.2
worker2    Ready     <none>          7d      v1.28.2
[root@master mylab]#
```

2. To create a new pod configuration file. Execute the following command.

```
vim pod-manual-scheduled.yaml
```

3. Use the below updated pod-manual-scheduled.yaml. Replace **<node-name>** with the name of the node where you want the Pod to be scheduled.

```
apiVersion: v1
kind: Pod
metadata:
 labels:
  run: nginx-manual-scheduling
 name: nginx-manual-scheduling
spec:
 nodeName: worker2
 containers:
 - name: nginx
   image: quay.io/gauravkumar9130/nginxdemo
```

4. Apply the Pod to the Kubernetes cluster. This configures the Pods to be scheduled on the specified node.

```
kubectl apply -f pod-manual-scheduled.yaml
```

```
[root@master mylab]# kubectl apply -f pod-manual-scheduled.yaml
pod/nginx-manual-scheduling created
[root@master mylab]#
```

Task 4: Verify Manual Scheduling

In this step, we are going to Confirm that the Pods are running on the specified node.

1. Execute the command below to verify that the Pods have been successfully scheduled on the specified node by executing the following command. This provides details about the Pods, including the node on which they are running.

```
kubectl get pods -o wide
```

2. Confirm that the Pods have been successfully scheduled on the node specified in nodeName inside the spec block. This ensures that manual scheduling has been applied as intended.

```
[root@master mylab]# kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE    IP              NODE      NOMINATED NODE   READINESS GATES
nginx-manual-scheduling   1/1     Running   0          112s   10.244.189.123   worker2   <none>           <none>
[root@master mylab]#
```

End Lab:

1. You Have Successfully Completed the Lab.
2. Once You Have Completed the Steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to implement Taint and Tolerance in a Kubernetes Cluster
2. You will use terminal.

Introduction

Taint and Tolerance in Kubernetes:

Taint and tolerance is a mechanism in Kubernetes that allows you to specify that certain nodes should only be used for certain pods. This can be useful for a variety of purposes, such as ensuring that the containers with Java environment run only on a node specifically configured to run Java workloads or run GPU-optimized workloads on a highly GPU-equipped node.

Taints are applied to nodes, and they represent a property or characteristic of the node that should be considered when scheduling pods. Tolerations are applied to pods, and they specify which taints the pod can tolerate. If a pod has a toleration for a taint applied to a node, it can be scheduled onto that node. Taints and tolerations are specified in the `spec` field of the node and pod resource definitions, respectively.

Taints can be applied using the `kubectl taint` command. Taints can have different effects, such as, preventing pods without a matching toleration from being scheduled onto the node, or, preferring not to schedule pods without a matching toleration onto the node but still allows it.

Tolerations can have a `key`, `value`, and `effect` field. The `key` and `value` fields must match the taint's `key` and `value` fields, respectively, and the `effect` field must match the taint's effect.

Taints and tolerations can be used to isolate pods that may be behaving poorly, by tainting the nodes they are running on and creating a new pod with tolerations for those taints. This allows the pod to be rescheduled onto a different node without impacting other pods.

Lab Tasks:

1. Open Kubernetes cluster
2. Apply taint to the available node
3. Apply tolerance to pod definition
4. Run the manifest file for the pod

Launching Lab Environment:

1. Launch the Lab environment by opening **Master** machine.
2. Open the terminal.

Lab Steps:

Task 1: Apply taint to available nodes

1. Once you have a Kubernetes cluster set up and running, decide on the taint and toleration that you want to apply. A taint represents a property or characteristic of a node that should be considered when scheduling pods. A toleration specifies which taints a pod can tolerate.
2. For this example, let's say that you want to apply a taint with the key "color" and the value "red", and you want to apply the "NoSchedule" effect, which means that pods without a matching toleration will not be scheduled onto the node.

```
kubectl taint nodes <node-name> color=red:NoSchedule
```

3. Use the kubectl taint command to apply the taint to the node. The kubectl taint command allows you to apply, modify, or delete a taint on a node.
4. In this case, we want to apply a taint with the key "color" and the value "red" and the "NoSchedule" effect. The kubectl taint command takes the following form:
5. Replace **<node-name>** with the actual name of the node that you want to apply the taint to. You can find the name of your node by running the

```
kubectl get nodes
```

```
[root@master mylab]# k get nodes
NAME      STATUS   ROLES           AGE     VERSION
master    Ready    control-plane   6d20h   v1.28.2
worker1   Ready    <none>          6d20h   v1.28.2
worker2   Ready    <none>          6d20h   v1.28.2
[root@master mylab]# kubectl taint nodes worker1 color=red:NoSchedule
node/worker1 tainted
[root@master mylab]#
```

Task 2: Apply tolerance to pod definition

Next, create a pod resource definition that includes a toleration for the taint you just applied.

A pod resource definition is a YAML file that describes the desired state of a pod. It includes information such as the pod's name, the containers it should run, and any tolerations or other constraints that should be applied to the pod.

- Create YAML file using the command below:

```
vim pod_taint.yml
```

1. Here is an example of a pod resource definition that includes a toleration for the taint we applied.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  tolerations:
  - key: color
    value: red
    effect: NoSchedule
  containers:
  - name: my-container
  - image: quay.io/gauravkumar9130/nginxdemo
```

2. Here, In the Pod definition file, we are explicitly telling the schedular to tolerate the key value pairs applied with the pod which will eventually allow the pod to tolerate the taint applied over the node. So, Only the resources having the matching tolerance block in the declaration will be allowed to be deployed over the corresponding tainted node.
3. This pod resource definition creates a pod named "my-pod" that runs a single container based on the nginx image. The pod also includes a toleration for the taint with the key "color" and the value "red" and the "NoSchedule" effect.

Task 3: Run the manifest file for the pod

Run the below command to deploy the pod on the node:

```
kubectl apply -f pod_taint.yml
```

You can verify that the pod is scheduled on the node by running

```
kubectl get pods
```

```
[root@master mylab]# vim pod_taint.yml
[root@master mylab]# kubectl apply -f pod_taint.yml
pod/my-pod created
[root@master mylab]# kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
my-pod   1/1     Running   0          27s
[root@master mylab]#
```

Task 4: Removing the taint and deleting the pod

Run the below command to remove the taint on the node:

```
kubectl taint nodes <nodename> color-
```

You can verify that the taint is removed from the node by running

```
kubectl describe node <nodename> | grep Taints
```

You can verify that the taint is removed from the node by running

```
kubectl delete -f pod_taint.yml
```

Completion and conclusion

1. You have successfully tainted the node.
2. You have successfully applied tolerance to the pod definition and scheduled a pod.

End lab

1. You Have successfully performed the lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details:

1. This Lab we'll walk through the process of deploying a simple Nginx application using a Kubernetes Deployment. Additionally, we'll explore how to control the placement of the Pods using Node Selector and Node Affinity.

Introduction

Node Selector:

Definition:

- Node Selector is a simple way to constrain a Pod to run on a particular node or set of nodes.

How it Works:

- You label nodes with key-value pairs.
- In the Pod specification, you define a nodeSelector field with a set of key-value pairs. The Pod will only be scheduled on nodes that match all of the key-value pairs.

Node Affinity:

Definition:

- Node Affinity is a more expressive way to constrain Pod placement based on node affinity rules.

How it Works:

- Node Affinity allows you to specify rules that influence scheduling decisions at a more granular level.
- You can use requiredDuringSchedulingIgnoredDuringExecution or preferredDuringSchedulingIgnoredDuringExecution to define node affinity.

Lab Tasks:

1. Create a Kubernetes Deployment
2. Verify Deployment & Pod creation
3. Implement Node Selector
4. Verify Node Selector for Pods
5. Implement Node Affinity
6. Verify Node Affinity for Pods

Launching Lab Environment:

1. Launch The Lab Environment by opening **Master** machine.

2. Open the terminal.

Lab Steps:

Task 1: Create a Kubernetes Deployment

In this step, Create a Deployment YAML file (e.g., nginx-deployment.yaml) with a basic Nginx Deployment configuration

1. Enter the below command to create a kubernetes deployment.

```
vim nginx-deployment.yaml
```

2. After you enter the above command, copy and paste the following content.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: quay.io/gauravkumar9130/nginx
```

3. Apply the deployment configuration using the following command.

```
kubectl apply -f nginx-deployment.yaml
```

```
[root@master mylab]# kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
[root@master mylab]#
```

Task 2: Verify Deployment & Pods

In this step, we will Check the status of the Deployment and Pods

1. Execute the command below to check the status of deployment.

```
kubectl get deployments
```

```
[root@master mylab]# kubectl get deployments
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           55s
[root@master mylab]#
```

2. Execute the command below to check the status of pods.

```
kubectl get pods
```

```
[root@master mylab]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-56b4c6fd4b-6kpsp   1/1     Running   0          119s
nginx-deployment-56b4c6fd4b-dp5p5   1/1     Running   0          119s
nginx-deployment-56b4c6fd4b-rztvl   1/1     Running   0          119s
[root@master mylab]#
```

Task 3: Implement Node Selector

In this step, let's enhance our Deployment to use Node Selector. Create the YAML (e.g., nginx-deployment-node-selector.yaml) to include a Node Selector for nodes labeled as "high-performance"

1. Enter the below command to create nginx-deployment-node-selector.yaml file.

```
vim nginx-deployment-node-selector.yaml
```

2. After you enter the above command, copy and paste the following content in that new window.

Note: This YAML will define Node selector rules.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-selector
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        performance: high
      containers:
      - name: nginx-container
        image: quay.io/gauravkumar9130/nginx
```

3. Execute the command below to get node details. Take a note of the node name as it will be used in the next step.

```
kubectl get nodes
```

```
[root@master mylab]# kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
master    Ready    control-plane   12d   v1.28.2
worker1   Ready    <none>          12d   v1.28.2
worker2   Ready    <none>          12d   v1.28.2
[root@master mylab]#
```

4. Now run the following command to label the selector node as performance=high.

```
kubectl label nodes <node-name> performance=high
```

```
[root@master mylab]# kubectl label node worker2 performance=high
node/worker2 labeled
[root@master mylab]#
```

5. Apply the node selector rules using the following command.

```
kubectl apply -f nginx-deployment-node-selector.yaml
```

```
[root@master mylab]# kubectl apply -f nginx-deployment-node-selector.yaml
deployment.apps/nginx-selector created
[root@master mylab]#
```

Task 4: Verify Node Selector for Pods

1. Execute the command below to Check that the Pods are scheduled on nodes with the specified label.

```
kubectl get pods -o wide
```

```
[root@master mylab]# kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
nginx-deployment-56b4c6fd4b-6kpsp   1/1     Running   0          10m   10.244.235.163   worker1   <none>           <none>
nginx-deployment-56b4c6fd4b-dp5p5   1/1     Running   0          10m   10.244.235.164   worker1   <none>           <none>
nginx-deployment-56b4c6fd4b-rztvl   1/1     Running   0          10m   10.244.189.98    worker2   <none>           <none>
nginx-selector-9688c4466-fchmk      1/1     Running   0          56s   10.244.189.100   worker2   <none>           <none>
nginx-selector-9688c4466-l6mp9      1/1     Running   0          56s   10.244.189.101   worker2   <none>           <none>
nginx-selector-9688c4466-pjshx      1/1     Running   0          56s   10.244.189.99    worker2   <none>           <none>
[root@master mylab]#
```

2. Run the command below to verify that the pods are scheduled on the node with specified label.

```
kubectl describe deploy nginx-selector | grep Selector
```

Task 5: Implement Node Affinity

1. Enter the below command to create nginx-deployment-node-affinity.yaml file.

```
vim nginx-deployment-node-affinity.yaml
```

2. After you enter the above command, copy and paste the following content in that new window. Note: This YAML will define Node Affinity rules.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-affinity
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:nodeSelectorTerms:
            - matchExpressions:
              - key: disk
                operator: In
                values:
                - ssd
      containers:
      - name: nginx-container
        image: quay.io/gauravkumar9130/nginx
```

3. Now run the following command to label the node as disk=ssd.
   Note: Use same node name you noted earlier in this lab.

```
kubectl label nodes <node-name> disk=ssd
```

4. Apply the node affinity rules using the following command.

```
kubectl apply -f nginx-deployment-node-affinity.yaml
```

Task 6: Verify Node Affinity for Pods

1. Execute the command below to Check that the Pods are scheduled on nodes based on the specified affinity.

```
kubectl get pods -o wide
```

```
[root@master mylab]# k get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
nginx-affinity-7df95d54b4-jbrt6   1/1     Running   0          19s   10.244.189.102   worker2   <none>           <none>
nginx-affinity-7df95d54b4-mkn2c   1/1     Running   0          19s   10.244.189.103   worker2   <none>           <none>
nginx-affinity-7df95d54b4-mkph4   1/1     Running   0          19s   10.244.189.104   worker2   <none>           <none>
[root@master mylab]#
```

2. Run the command below to verify that the pods are scheduled on the node with specified label.

```
kubectl get pod <pod-name> -ojson | jq '.spec.affinity.nodeAffinity'
```

```
[root@master mylab]# kubectl get pod nginx-affinity-7df95d54b4-jbrt6 -ojson | jq '.spec.affinity.nodeAffinity'
{
  "requiredDuringSchedulingIgnoredDuringExecution": {
    "nodeSelectorTerms": [
      {
        "matchExpressions": [
          {
            "key": "disk",
            "operator": "In",
            "values": [
              "ssd"
            ]
          }
        ]
      }
    ]
  }
}
[root@master mylab]#
```

End Lab:

1. You Have Successfully Completed the Lab.
2. Once You Have Completed the Steps, **delete** the resources.

Lab Details:

1. In this lab, we will explore different deployment strategies in Kubernetes using a simple application. We'll focus on Rollout Deployment and Canary Deployment as two common strategies for managing updates and releases.

2. Cluster Configuration: **1 Master And 2 Worker Node**

What are Deployment Strategies?

In Kubernetes, deployment strategies refer to the methods used to release and update applications running on a cluster. Two common deployment strategies are rollout updates and canary releases.

1. **Rollout Deployment**:

   Rolling updates involve gradually replacing instances of the old application version with instances of the new version. This process ensures that the application remains available throughout the update.

2. **Canary Deployment**:

   Canary deployment involve introducing the new version of the application to a subset of users or instances before rolling it out to the entire cluster. This allows for testing the new version in a real-world environment with limited impact.

Lab Tasks:

1. Deploy Kubernetes pod
2. RollOut Deployment
3. Canary Deployment

Launching Lab Environment:

1. Launch The Lab Environment by opening **Master** machine.

2. Open the terminal.

Lab Steps:

Deploy Kubernetes Pod:

In this step, we will create the manifest file that defines a Kubernetes Pod configuration.

1. Create manifest file using YAML approach

   - Create a YAML file using the command below:

     ```
     vim app-deployment.yaml
     ```

2. Now pass the below sample yml code. This will attempt to create the specified Pod based on the provided details.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rollout-deployment
  labels:
    app: prodx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: prodx
  template:
    metadata:
      labels:
        app: prodx
    spec:
      containers:
      - name: prod
        image: quay.io/gauravkumar9130/production:v1
        ports:
        - containerPort: 80
```

3. The next step is to apply this yml file to create the deployment as declared.

```
kubectl apply -f app-deployment.yaml
```

```
[root@master mylab]# vim app-deployment.yaml
[root@master mylab]# kubectl apply -f app-deployment.yaml
deployment.apps/rollout-deployment created
[root@master mylab]#
```

4. Run below command to verify the deployment

```
kubectl get deployments
```

```
[root@master mylab]# kubectl get deployments
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
rollout-deployment   3/3     3            3           14s
[root@master mylab]#
```

5. Now run the following command to know more about the deployed nginx image.

```
kubectl describe deployment rollout-deployment | grep Image
```

```
[root@master mylab]# kubectl describe deployments rollout-deployment | grep Image
    Image:        quay.io/gauravkumar9130/production:v1
[root@master mylab]#
```

Rollout Deployment:

In this step, we will roll out deployment to a new version.

1. Enter below command to Update the image version in file to roll out a new version of the application.

```
kubectl set image deployment/rollout-deployment prod=quay.io/gauravkumar9130/production:v2
```

```
[root@master mylab]# kubectl set image deployment/rollout-deployment prod=quay.io/gauravkumar9130/production:v2
deployment.apps/rollout-deployment image updated
[root@master mylab]#
```

2. Run the following command to confirm whether the deployment image has been successfully set to new image ie. production:v2.

```
kubectl describe deployment rollout-deployment | grep Image
```

```
[root@master mylab]# kubectl describe deployments rollout-deployment | grep Image
    Image:        quay.io/gauravkumar9130/production:v2
[root@master mylab]#
```

3. Now, enter below command to check the rollout status

```
kubectl rollout status deployment/rollout-deployment
```

```
[root@master mylab]# kubectl rollout status deployment/rollout-deployment
deployment "rollout-deployment" successfully rolled out
[root@master mylab]#
```

3. Enter the command below to perform a rollback to the previous version.

```
kubectl rollout undo deployment/rollout-deployment
```

```
[root@master mylab]# kubectl rollout undo deployment/rollout-deployment
deployment.apps/rollout-deployment rolled back
[root@master mylab]#
```

4. Now run the following command whether the image has been successfully rolled back to the previous version:

```
kubectl describe deployment rollout-deployment | grep Image
```

```
[root@master mylab]# kubectl describe deployments rollout-deployment | grep Image
    Image:        quay.io/gauravkumar9130/production:v1
[root@master mylab]#
```

Canary Deployment:

Now, to perform a canary deployment first we have to create another manifest file.

1. First create a YAML File which will create a service and two deployments—one for the main application and another for the canary deployment.

```
vim nginx-app.yaml
```

2. Now pass the below code to create the file.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: quay.io/gauravkumar9130/production:v1
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-canary
  labels:
    app: app-canary
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: quay.io/gauravkumar9130/production:v2
        ports:
        - containerPort: 80
```

3. Apply the file to create the desired resources.

```
kubectl apply -f nginx-app.yaml
```

```
[root@master mylab]# vim nginx-app.yaml
[root@master mylab]# kubectl apply -f nginx-app.yaml
service/my-app-svc created
deployment.apps/my-app created
deployment.apps/my-app-canary created
[root@master mylab]#
```

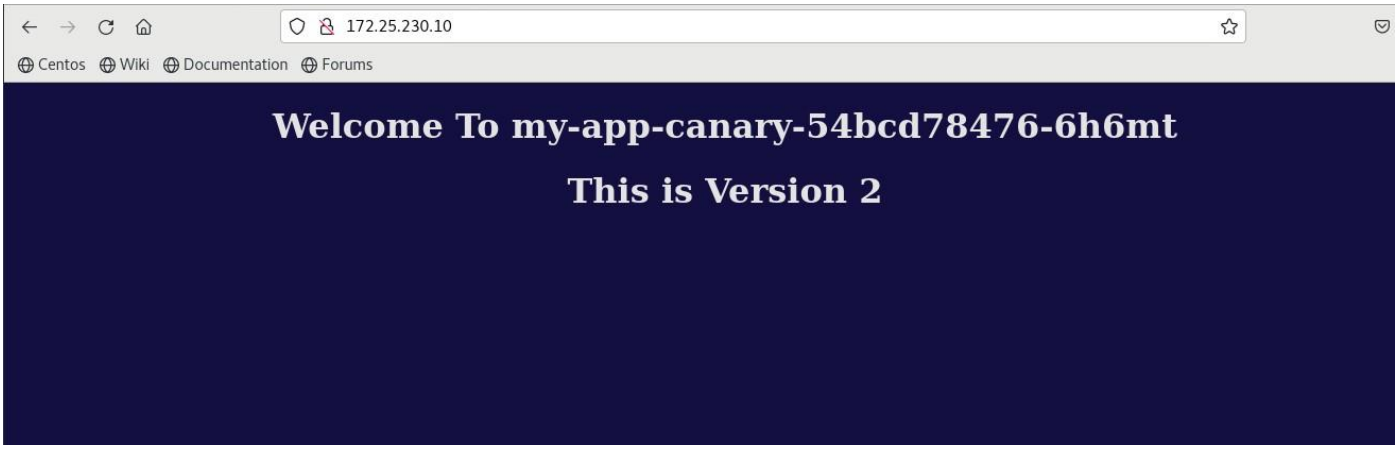
4. Now check Load Balancer IP: After applying the configuration, wait for the service to be assigned an external IP address:

```
kubectl get services
```

```
[root@master mylab]# kubectl get services
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes    ClusterIP      10.96.0.1       <none>          443/TCP        8d
my-app-svc    LoadBalancer   10.99.117.188   172.25.230.10   80:32210/TCP   98s
[root@master mylab]#
```

6. Now let's assume that our canary version of the application is working as expected.

7. You can check by opening the external IP in browser and you will land to the application.



8. To complete our deployment strategy, we will delete the main deployment (my-nginx) and scale up the canary deployment (my-nginx-canary) to 3 replicas.

```
kubectl delete deployment my-app
kubectl scale deployment my-app-canary --replicas=3
```

```
[root@master mylab]# kubectl delete deployment my-app
deployment.apps "my-app" deleted
[root@master mylab]# kubectl scale deployment my-app-canary --replicas=5
deployment.apps/my-app-canary scaled
[root@master mylab]#
```

End Lab:

1. You Have Successfully Completed The Lab.

2. Once You Have Completed The Steps, **delete** the resources.

Lab Details:

1. This hands-on-lab will teach you how to define environment variables from a Plain key Kubernetes.

Key Considerations: Environment Variables from a Plain Key in Kubernetes

- Key-Value Pairs:

Kubernetes allows setting environment variables directly from plain key-value pairs, eliminating the need for ConfigMaps or Secrets for simple configuration.

- Pod Spec Definition:

Environment variables are defined directly in the Pod specification using the env field, specifying the key and its corresponding value.

- Configurations without ConfigMaps:

For straightforward configuration needs, individual key-value pairs can be injected into pods without the overhead of ConfigMaps or Secrets.

- Static Configuration:

Values set directly as environment variables are static and do not dynamically update when the configuration changes, making this approach suitable for relatively fixed configurations.

- Simplicity and Readability:

Using plain keys in environment variables simplifies the configuration process, enhancing readability and ease of understanding in the Pod specification.

- Limited to Non-sensitive Data:

Best suited for non-sensitive configuration data, as plain keys lack the encryption features provided by Secrets for securing sensitive information.

- Quick Configuration Updates:

Changes to environment variables require pod restarts for the updates to take effect, providing a quick and straightforward way to apply changes to the configuration.

Lab Tasks:

1. Kubernetes cluster
2. Use plain key as a variable in a pod
3. Create a Pod

Launching Lab Environment:

1. Launch The Lab Environment By opening the **Master** machine.
2. Open the terminal.

Task 1: Use env to define variables in a pod

Let's start by creating a simple Pod that uses the env field. This will help illustrate how the env is being utilized as a variable in the Pod.

1. Create a file named **plain.yaml**:

```
vim plain.yaml
```

2. Now copy and paste the following YAML configurations and to save the file. This plain.yaml file describes a Kubernetes Pod named "plain" running a mysql. It incorporates an environment variable. This Pod is meant solely for illustrating how the env is used in the Pod.

```
apiVersion: v1
kind: Pod
metadata:
 name: plain
spec:
 containers:
 - name: c1
   image: quay.io/gauravkumar9130/mysql
   env:
   - name: MYSQL_ROOT_PASSWORD
     value: myroot
   - name: MYSQL_USER
     value: sam
   - name: MYSQL_PASSWORD
     value: sam12345
```

3. Apply the Pod to the cluster with the following command:

```
kubectl apply -f plain.yaml
```

```
[root@master mylab]# vim plain.yml
[root@master mylab]# kubectl apply -f  plain.yml
pod/plain created
[root@master mylab]#
```

Task 2: Verify the variables

1. Now, let's access the Pod and verify that the environment variable is successfully mounted. To do so make use of the command mentioned below

```
kubectl exec -it plain -- bash
```

2. Now paste the below command and you should get all the env used in container.

```
env
```

```
[root@master mylab]# kubectl exec -it plain -- bash
root@plain:/# env
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_PORT=443
MYSQL_MAJOR=8.0
HOSTNAME=plain
PWD=/
MYSQL_ROOT_PASSWORD=myroot
MYSQL_PASSWORD=pan12345
MYSQL_USER=pandey
HOME=/root
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
MYSQL_VERSION=8.0.22-1debian10
GOSU_VERSION=1.12
TERM=xterm
SHLVL=1
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=/usr/bin/env
root@plain:/#
```

3. Enter the **exit** command to exit this terminal.

```
exit
```

End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and configure ConfigMap resources in a kubernetes cluster.
2. Cluster Configuration: 1 Master And 2 Worker Node

Introduction

What is the ConfigMap resource in Kubernetes?

- ConfigMap is a Kubernetes object which allows you to separate the custom configuration files from your pods and components.
- As a result, it keeps your container portable and makes the configuration easier to change and manage and prevents the hardcoding configuration data into pod spec.
- So ConfigMap stores the configuration data as a key value pair. Example: A configuration file for a web application can be passed as a unique key-value pair and stored in a single place where it becomes easier to manage.
- You can also pass key value pairs straight from the command line or as an environmental variable.
- So with the help of ConfigMap in Kubernetes one can manage configuration of containers irrespective of the container lifecycle.
- One thing to note is that, in case of storing confidential information, one should opt for secret as a resource rather than ConfigMap
- ConfigMap is meant for storing the configuration files and data at one place and efficiently maintaining it.
- ConfigMap is a local ephemeral storage When a configMap is actively consumed as a volume, the kubelet service performs frequent sync processes and automatically updates the configuration files. One exception to this is the configMaps used via environmental variables require pod restart to perform sync and update the content.

Task Details

1. Launching Lab Environment.
2. Configure YAML file for the configMap resource
3. Edit the Config File and verify synchronization
4. Delete the service

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Create a ConfigMap as a YML file

- First, let's create a YML file using vim command:

```
vim cm-creds.yml
```

- Now paste the following code to create a secret resource where we will store a username and a password which will eventually be stored inside a secret in encoded form.

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: cm-credentials
data:
 MYSQL_ROOT_PASSWORD: Koenigl@b$
 MYSQL_PASSWORD: koeniglabs
 MYSQL_USER: koenig
```

- Now let's create the resource using the kubectl command

```
kubectl apply -f cm-creds.yml
```

- Verify the resource is created using the get query with the kubectl command

```
kubectl get cm
```

Task 3: Create a Deployment

1. In this step, we will create a deployment wherein we will specify the volume configuration and mention the secret resource details in it so that the deployment and the underlying pod has access to the secret and the data stored in it. Create a file using vim command

```
vim db.yml
```

2. Paste the following code in the created file and save it. Here, we are taking a busybox container image that is capable of performing tiny executables. With the keyword *volume*, we are going to specify the secret created in the previous stage.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: cm-deploy
spec:
 replicas: 1
 selector:
  matchLabels:
   name: db
 template:
  metadata:
   labels:
    name: db
  spec:
   containers:
   - name: my-container
     image: quay.io/gauravkumar9130/mysql
     envFrom:
     - configMapRef:
         name: cm-credentials
```

3. Now, apply the changes and provision the deployment using the kubctl command

```
kubectl apply -f db.yml
```

```
[root@master mylab]# k apply -f db.yml
deployment.apps/cm-deploy created
[root@master mylab]#
```

4. Cross-verify that the pod is up and running by the following command. Make sure the pod state is RUNNING before proceeding to the next step.

```
kubectl get pods
```

```
[root@master mylab]# k get pods
NAME                         READY   STATUS    RESTARTS   AGE
cm-deploy-55b5ddf678-jxzm7   1/1     Running   0          2m29s
[root@master mylab]#
```

Task 4: Verify the access of the cm within the pod.

1. Now, run the command below to list the env data inside the container. Here, the data can be observed which is mounted from config map.

```
kubectl exec -it <pod-name> -- env
```

2. First, we get inside the pod by using the command below so that we can test the data.

```
kubectl exec -it <pod-name> -- bash
```

```
[root@master mylab]# k get pods
NAME                          READY   STATUS    RESTARTS   AGE
cm-deploy-55b5ddf678-jxzm7    1/1     Running   0          2m29s
[root@master mylab]# k exec -it cm-deploy-55b5ddf678-jxzm7 -- bash
root@cm-deploy-55b5ddf678-jxzm7:/#
```

3. Now, run the command for mysql database and check if you can login with credentials.

```
mysql -u root -p
```

we can enter the password value which we have defined and then we exit from the pod.

```
root@cm-deploy-55b5ddf678-jxzm7:/# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 8.0.22 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
mysql>
mysql> exit
Bye
root@cm-deploy-55b5ddf678-jxzm7:/# exit
[root@master mylab]#
```

This is how we can manage data within the Kubernetes scope and independent from the pod lifecycle.

End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details:

1. This hands-on-lab will teach you how to mount environment variables from a ConfigMap as a volume in a Kubernetes Pod.

Key Considerations: Mounting Environment Variables as Volumes

1. Mounting environment variables as volumes enables dynamic configuration in Kubernetes Pods, allowing you to update settings without redeploying the entire application.
2. By separating environment variables into volumes, you enhance security and isolation. Sensitive information, such as API keys or credentials, can be stored in ConfigMaps or Secrets, providing an additional layer of protection.
3. Mounting environment variables through volumes promotes application portability across different environments. It allows for easy adjustment of configurations without modifying the application code.
4. When configuration changes are required, updating the ConfigMap or Secret automatically reflects the changes in the mounted volumes, ensuring efficient handling of configuration modifications without disrupting the running Pods.

Lab Tasks:

1. Kubernetes cluster
2. Use configMap as a variable in a pod
3. Create a configMap
4. Create a Pod
5. Verify the Mount

Launching Lab Environment:

1. Launch The Lab Environment By opening the **Master** machine.
2. Open the terminal.

Task 1: Use configMap as a variable in a pod

Let's start by creating a simple Pod that uses the environment variable from the ConfigMap.

This will help illustrate how the ConfigMap is being utilized as a variable in the Pod.

1. Create a file named **pod-configmap-variable.yaml**:

```
vim pod-configmap-variable.yaml
```

2. Now copy and paste the following YAML configurations and to save the file. This pod-configmap-variable.yaml file describes a Kubernetes Pod named "configmap-variable-pod" running an Nginx container. It incorporates an environment variable from a ConfigMap named "my- config". This Pod is meant solely for illustrating how the ConfigMap is used as a variablein the Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-variable-pod
spec:
  containers:
  - name: my-container
    image: quay.io/gauravkumar9130/nginx
    env:
    - name: MY_ENV_VARIABLE
      valueFrom:
        configMapKeyRef:
          name:  my-config
          key: MY_ENV_VARIABLE
```

3. Apply the Pod to the cluster with the following command:

```
kubectl apply -f pod-configmap-variable.yaml
```

```
[root@master mylab]# vim pod-configmap-variable.yaml
[root@master mylab]# kubectl apply -f pod-configmap-variable.yaml
pod/configmap-variable-pod created
[root@master mylab]#
```

Task 2: Create a configMap

In Kubernetes, a ConfigMap is an API resource that allows you to decouple configuration artifacts from the container images, keeping the configuration separate from the application code. It provides a way to store key-value pairs, configuration files, or environment variables that can be consumed by Pods or other Kubernetes objects.

1. Let’s start by creating a ConfigMap that contains the environment variable that we want to mount, for that create a file named as **configmap.yaml.**

```
vim configmap.yaml
```

2. Now paste the below file configuration in the terminal that opens up and after pasting them press **CTRL+X** and then **Y** and **Enter** to save the file.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  MY_ENV_VARIABLE: "Hello from ConfigMap!"
```

The `configmap.yaml` file creates a Kubernetes ConfigMap named "my-config" with a key-value pair containing the environment variable **"MY_ENV_VARIABLE"** set to "Hello from ConfigMap!". This ConfigMap can be referenced and utilized by other Kubernetes resources.

3. Apply the configurations to the cluster with the help of the following command:

```
kubectl apply -f configmap.yaml
```

```
[root@master mylab]# vim configmap.yaml
[root@master mylab]# kubectl apply -f configmap.yaml
configmap/my-config created
[root@master mylab]#
```

Task 3: Create a pod

1. Create a Pod that uses the environment variable from the ConfigMap. Save the following YAML in a file named pod.yaml. To do so, paste the following command in the terminal:

```
vim pod.yaml
```

2. Now copy and paste the below file configurations in the terminal that opens up, and after doing that press **CTRL+X** and the **Y** and **Enter** to save the file.

```
apiVersion: v1
kind: Pod
metadata:
  name: env-var-pod
spec:
  containers:
  - name: my-container
    image: quay.io/gauravkumar9130/nginx
    env:
    - name: MY_ENV_VARIABLE
      valueFrom:
        configMapKeyRef:
          name:  my-config
          key: MY_ENV_VARIABLE
    volumeMounts:
    - name: env-var-volume
      mountPath: "/etc/my-config"
  volumes:
  - name: env-var-volume
    configMap:
      name: my-config
```

The `**pod.yaml**` file describes a Kubernetes Pod named "**env-var-pod**" running an Nginx container. It incorporates an environment variable from a ConfigMap named "my-config" using a volume mount at "**/etc/my-config**". This configuration allows the Pod toaccess and utilize the specified environment variable from the ConfigMap within its file system.

3. Apply the Pod to the cluster with the help of the following command:

```
kubectl apply -f pod.yaml
```

```
[root@master mylab]# vim pod.yaml
[root@master mylab]# kubectl apply -f pod.yaml
pod/env-var-pod created
[root@master mylab]#
```

Task 4: Verify the mount

1. Now, let's access the Pod and verify that the environment variable is successfully mounted. To do so make use of the command mentioned below:

```
kubectl exec -it env-var-pod -- bash
```

2. Now paste the below command and you should get **"Hello from ConfigMap!"** as the output that will determine that our variable from the configMap has been successfully mounted into the volume

```
cat /etc/my-config/MY_ENV_VARIABLE
```

```
[root@master mylab]# kubectl exec -it env-var-pod -- bash
root@env-var-pod:/# cat /etc/my-config/MY_ENV_VARIABLE
Hello from ConfigMap!root@env-var-pod:/#
```

3. Enter the **exit** command to exit this terminal.

```
exit
```

End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to provision and mount a secret to a pod as a volume.
2. You will createa sample pod and attach a secret resource to it as a storage unit.

Introduction

What are the Secrets in Kubernetes?

- The secret is a resource kind in Kubernetes that allows us to natively handle sensitive data like passwords, SSH keys, and similar data that needs to be stored securely.
- This resource eliminates the usage of plain text while inserting passwords or similar sensitive data and replaces the same with a layer of security.
- Secrets perform encoding operations on the text and can be created independently concerning the pods. This means, sensitive data like passwords are not passed into the application or the container directly.
- With this, we can avoid unanticipated exposure of confidential data in plain text format during the whole deployment process and can avoid shoulder surfing.
- There are three ways to create secrets:
  - Via YAML file
  - Via environmental variable
  - Via Kubelet while making an API call
- In this lab, we will use the YAML file approach and bind the created secret to a pod in a modular way.
- The data stored in a secret is still unencrypted and it can be accessed with API access from the privileged user. However, the main use-case of the secret is not to encrypt the data but to encode the data to eliminate plain text usage and add more flexibility to use the data in a plug-and-play method.

Pre-requisites:

-Basic Linux

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Create a secret as YML file
4. Create a Deployment
5. Mount the secret into the deployment using the config file
6. Verify the access of the secret within the pod.

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Create a secret as a YML file

- First, let's create a YML file using vim command:

```
vim creds.yml
```

- Now paste the following code to create a secret resource where we will store a username and a password which will eventually be stored inside a secret in encoded form.

```
apiVersion: v1
kind: Secret
metadata:
 name: credentials
stringData:
 password: Koenig1@b$
 username: koeniglabs
 type: Opaque
```

- Now let's create the resource using the kubectl command

```
kubectl apply -f creds.yml
```

- Verify the resource is created using the get query with the kubectl command

```
kubectl apply -f creds.yml
```

- To retrieve only the username, we can run the below addon command

```
kubectl get secret credentials -o jsonpath="{.data.username}" | base64 --decode
```

Task 3: Create a Deployment

1. In this step, we will create a deployment wherein we will specify the volume configuration and mention the secret resource details in it so that the deployment and the underlying pod has access to the secret and the data stored in it. Create a file using vim command

```
vim busybox.yml
```

2. Paste the following code in the created file and save it. Here, we are taking a busybox container image that is capable of performing tiny executables. With the keyword *volume*, we are going to specify the secret created in the previous stage.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: busybox
spec:
  replicas: 1
  selector:
    matchLabels:
      name: busybox
  template:
    metadata:
      labels:
        name: busybox
    spec:
      containers:
      - name: my-container
        image: quay.io/gauravkumar9130/busybox
        command: ["/bin/sh","-c"]
        args: ["while true; do sleep 9999999; done"]
      volumes:
      - name: credentials-volume
        secret:
          secretName: credentials
```

3. Now, apply the changes and provision the deployment using the kubctl command

```
kubectl apply -f busybox.yml
```

4. Cross-verify that the pod is up and running by the following command. Make sure the pod state is RUNNING before proceeding to the next step.

```
kubectl get pods
```

Task 4: Mount the secret into the deployment using the config file

1. Now append the below configuration in the **busybox.yml** after the volume file which will mount the volume to a directory inside a container. Till now, we have attached the volume to the pod but the container will not be able to access the data until a native directory is mounted with it.
   Using the volumeMounts keyword block, we are attaching the created secret as a volume to the mentioned directory at mountPath of the container. As an outcome, we will be able to retrieve the data stored inside the secret from the container. Also, the keyword readOnly will restrict the user to edit the content of the mounted volume.
   The resultant busybox.yml file will look like this.

```
=== this part is for reference only ===
volumeMounts:
- mountPath: /mnt/credentials
  name: credentials-volume
  readOnly: true
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: busybox
spec:
 replicas: 1
 selector:
  matchLabels:
   name: busybox
 template:
  metadata:
   labels:
    name: busybox
  spec:
   containers:
   - name: my-container
     image: quay.io/gauravkumar9130/busybox
     command: ["/bin/sh","-c"]
     args: ["while true; do sleep 9999999; done"]
     volumeMounts:
     - mountPath: /mnt/credentials
       name: credentials-volume
       readOnly: true
   volumes:
   - name: credentials-volume
     secret:
      secretName: credentials
```

Now, apply the changes of the updated file and add the mount configuration to the existing deployment with the following command:

```
kubectl apply -f busybox.yml
```

Now the deployment container will be able to access the data through /mnt/credentials directory.

Task 5: Verify the access of the secret within the pod.

1. First, set the pod as an environmental variable by using the command below so that we don't have to remember the ID now and then.

```
POD=`kubectl get pod -l name=busybox -o jsonpath="{.items[0].metadata.name}"`
```

2. Now, run the command below to read the data inside the /mnt/credentials directory inside the container. Here, the data will be called through the mounted secret. Since we haven't passed the credentials anywhere while launching the pod, the output data indicates that it is being called from the secret.

```
kubectl exec $POD -it -- cat /mnt/credentials/username
```

In the same way, we can call the password value by replacing the username in the command above.

This is how we can manage sensitive data separately and natively within the Kubernetes scope and independent from the pod lifecycle.

Completion and Conclusion

1. You have successfully provisioned a secret resource.
2. You have successfully launched a busybox deployment inside it.
3. You have successfully mounted the secret as a volume and read the data from outside the container.

End Lab

1. You have successfully completed the lab.

Lab Details

1. This lab walks you through the steps to provision Persistent Volume in the Kubernetes Cluster
2. You will practice to create and attach a persistent volume to a pod.

Introduction

What is Persistent Storage in Kubernetes?

- Data is one of the key elements for any multi-tier application. Data storage is also a distinct challenge. Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod.
- Persistent in simple terms means permanent. Yes, this storage type is used to preserve data for the long term and it should be independent of the container lifecycle. Persistent storage doesn't come by default with container technologies. The PersistentVolume system provides this feature to extend the default capabilities of storing the data. With this subsystem, we can implement permanent storage of data clubbed with the Kubernetes cluster.
- We can consider this as external storage that runs on top of the container lifecycle. It is not limited to runtime container issues also we can get the assurance about preserving the precious data.
- There are a variety of storage options and products available. It becomes the role of the admins to choose the size, access modes, and configuration, and maintain the transparency for the unit.

Why Do We Need Persistent Storage?

- As we have seen this storage type preserves the data for the long term, it best fits for the use cases to store data like databases and similar. Such data must be sustained irrespective of underlying pods because such data are more related to the application than the containers.

- The major difference between ephemeral and persistent storage is pretty clear. Ephemeral can't store data for longer and persistence takes charge of doing that part. The storage plug-ins and addons in K8s allow the admin to plug and play various storage types and select the best-fit option according to the use case.
- We only have an ephemeral storage option by default which is not enough. That's why we need a subsystem like PersistentVolume that provides this feature. The PersistentVolume has two API-based resources under its umbrella.

1. PersistentVolume
2. PersistentVolumeClaim

- Both of these clubs work together and work towards the goal of providing permanent storage to the cluster and application. We'll learn more about the subsystem in the next scenarios.!

Task Details

1. Launching Lab Environment.
2. Open the terminal
3. Create Persistent Volume
4. Create Persistent Volume Claim
5. Mount the Persistent Volume to a pod
6. Verify the data is persistent within the volume

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by by opening **Master** machine.
2. Open the terminal

Task 2: Create a Persistent Volume:

PV is volume and has a lifecycle independent of the pods' lifecycle. This means that even if the connected pods get crashed, the storage is safe and working.

PersistentVolume can be NFS, ISCSI, ceph, or any other provider-specific storage type. There are two ways of provisioning `pv`. One is static and the other one is dynamic. Static

provisioning is handled manually by the admins and dynamic is done by using a dynamic provisioner for specific storage classes.

1. Create an empty directory in the system which will be used to mount with the storage unit provisioned through Kubernetes:

```
mkdir /mnt/labs
```

2. Now let's paste a sample index file inside this directory using the command below:

```
echo "Hello from Koenig labs K8s" > /mnt/labs/index.html
```

3. Next step is to create a YAML declarative file for PersistentVolume Resource. Here, we are using the manual storage provisioning approach and hence we will mention the storage class as manual. Also, We mention the storage capacity of the volume in the specification block which is 2Gi in the example.

   As the storage unit contains sensitive data, managing access to the resource becomes important. Hence, a block named accessmode in volumes is defined where we have mentioned ReadWriteMany which means multiple nodes can perform read and write operations on the storage disk. The other access modes are ReadOnlyMany and ReadWriteOnce. Create a YAML file and paste the following code in it:

```
vim persistent_volume.yaml
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: task-pv-volume
 labels:
  type: local
spec:
 storageClassName: manual
 capacity:
  storage: 2Gi
 accessModes:
 - ReadWriteMany
 hostPath:
  path: "/mnt/labs"
```

4. Let's apply the created file using the kubectl command line utility:

```
kubectl apply -f persistent_volume.yaml
```

5. Verify the volume is available to use by running the command below:

```
kubectl describe pv task-pv-volume
```

```
[root@master mylab]# kubectl describe pv task-pv-volume
Name:            task-pv-volume
Labels:          <none>
Annotations:     <none>
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    manual
Status:          Available
Claim:
Reclaim Policy:  Retain
Access Modes:    RWX
VolumeMode:      Filesystem
Capacity:        2Gi
Node Affinity:   <none>
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /mnt/labs
    HostPathType:
Events:            <none>
[root@master mylab]#
```

Task 3: Create a Persistent Volume Claim:

PersistentVolumeClaim is not a volume. It is just requesting or claiming the created volume. PVC will allow consuming the created storage.

1. We created a PersistentVolumne in the previous scenario. Now let's see how we can link it to a PersistentVolumeClaim and make it usable. Refer to the YAML code below:

```
vim volume_claim.yaml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: task-pv-claim
spec:
 storageClassName: manual
 accessModes:
 - ReadWriteMany
 resources:
  requests:
   storage: 2Gi
```

2. Now let’s apply the YAML file and make the pvc available to be used with the volume

```
kubectl apply -f volume_claim.yaml
```

3. Let's verify the pvc has appeared or not by running the command below

```
kubectl describe pvc task-pv-claim
```

```
[root@master mylab]# kubectl describe pvc task-pv-claim
Name:          task-pv-claim
Namespace:     default
StorageClass:  manual
Status:        Bound
Volume:        task-pv-volume
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      2Gi
Access Modes:  RWX
VolumeMode:    Filesystem
Used By:       <none>
Events:        <none>
[root@master mylab]#
```

Task 4 : Mount the Persistent Volume to a Pod:

In this step, we are going to create a sample pod of Nginx webserver and mount the created volume to the document root of the proxy which is /var/www/html. Hence, the content we put inside the persistent volume in the /root/mnt/data directory will be reflected in the /var/www/html directory inside the container.

1. Create a YAML file for a sample nginx pod using the code below:

```
vim pvc_pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
name: task-pv-pod
spec:
 volumes:
 - name: task-pv-storage
   persistentVolumeClaim:
    claimName: task-pv-claim
 containers:
 - name: task-pv-container
   image: quay.io/gauravkumar9130/nginx
   volumeMounts:
   - mountPath: "/usr/share/nginx/html"
     name: task-pv-storage
```

2. Here, you may have observed a new block of code in the specification block which is volumes. Here we are mentioning the volume claim name which we created in the earlier steps. Also, the volumeMount block contains the destination mount directory of the pod.
   One thing to note is that both the names in volume and volume mount blocks must be the same to map the volume with the corresponding pod.
3. Apply the YAML file to create the pod resource using the command below:

```
kubectl apply -f pvc_pod.yaml
```

4. Now run the command below to verify the pod is running

```
kubectl get pods -o wide
```

5. Kindly note the ip address of the pod which we will use in the next step
6. Now, let's hit a curl request to the ip address on the pod and see if the output is the same content that we pasted in the index.html or not

```
curl <ip-of-the-pod>/index.html
```

7. The output is the same as the content we pasted inside the index.html file in the host directory created in step 3. Hence, without even going inside the container, we managed to serve the website content to the client as we have mounted both directories.

Task 5 : Verify the data is persistent within the volume

1. Let's delete the running container first to verify the persistence of the content stored in the volume

```
kubectl delete pods task-pv-pod
```

2. Now, launch a new container with the same configuration and same YAML file from task 5. Also take note of the ip address of the new pod and hit the curl request same as task 5.

```
kubectl apply -f pvc_pod.yaml
```

```
kubectl get pods -o wide
```

```
curl <ip-of-pod>/index.html
```

This time you will see a different IP address assigned to the pod but still, the curl request will reply to the same output content that we saved in the volume. This means the volume is independent of the pod lifecycle and the data stored within the volume will remain intact even if the container is destroyed.

Completion and Conclusion

1. You have successfully provisioned a Persistent volume and its corresponding persistent volume claim.
2. You have successfully launched a sample nginx pod and mounted the same with the persistent volume.
3. You have successfully mounted the volume to the document root of the container and verified the data is independent of the pod lifecycle.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Overview:

Lab Details:

1. In this hands-on lab, participants will gain practical experience in managing users within a Kubernetes cluster. The lab covers essential tasks, including creating and updating user accounts, assigning roles and permissions through RBAC, and exploring the nuances of ClusterRoles and ClusterRoleBindings.

Key Aspects of User Management in Kubernetes

1. Kubernetes employs Service Accounts to represent identities within the cluster. Users can create and manage Service Accounts to grant Pods specific access rights, controlling interactions with other resources in the cluster.
2. RBAC is integral to managing user access by defining roles, role bindings, and permissions. Administrators use RBAC to granularly control user privileges, ensuring that each user or service account has the necessary permissions for their tasks while avoiding unnecessary access.
3. ClusterRoles extend RBAC to a cluster-wide scope, providing permissions that apply globally.
4. ClusterRoleBindings associate ClusterRoles with users, groups, or service accounts across the entire cluster.

Lab Tasks:

1. Create a user account
2. Assign roles and permissions
3. Verify service account permissions
4. Delete user account

Lab Steps:

Task 1: Launching Lab Environment

1. Launch The Lab Environment By opening the **Master** machine.
2. Open the terminal.

Task 2: Create a service account

In this task, we will learn to create a new service account, named "lab-user," within a Kubernetes cluster using manifests. Here we will apply the configuration and verify the successful creation of the user account using the kubectl command.

1. Let's start by creating a file named **lab-user.yaml**. To do so paste the following command in the terminal.

```
vim lab-user.yaml
```

2. In the terminal that opens up, we will write the following and save the file.

```
apiVersion: v1
kind:
ServiceAccount
metadata:
  name: lab-user
```

3. Now apply the configuration to create the service account with the help of the following command:

```
kubectl apply -f lab-user.yaml
```

```
[root@master mylab]# vim lab-user.yaml
[root@master mylab]# kubectl apply -f lab-user.yaml
serviceaccount/lab-user created
[root@master mylab]#
```

4. Verify the account creation by pasting the following command in the terminal.

```
kubectl get serviceaccount lab-user
```

```
[root@master mylab]# kubectl get sa lab-user
NAME       SECRETS   AGE
lab-user   0         54s
[root@master mylab]#
```

Task 3: Assign roles and permissions

In this task we will explore the intricacies of Role-Based Access Control (RBAC) by creating a ClusterRole with specific permissions and binding it to the previously created "lab-user" account. This task focuses on establishing fine-grained access control for Pods and Deployments and verifying the assigned permissions.

1. Let's start by creating a **ClusterRole**. To do so, create a file named **lab-cluster-role.yaml**. Paste the following command in the terminal.

```
vim lab-cluster-role.yaml
```

2. In the terminal that opens up, write the below code.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: lab-cluster-role
rules:
- apiGroups: [""]
  resources: ["pods", "deployments"]
  verbs: ["get", "list"]
```

3. Now make use of the following command to apply the configurations to the yaml file.

```
kubectl apply -f lab-cluster-role.yaml
```

```
[root@master mylab]# vim lab-cluster-role.yaml
[root@master mylab]# kubectl apply -f lab-cluster-role.yaml
clusterrole.rbac.authorization.k8s.io/lab-cluster-role created
[root@master mylab]#
```

4. Let's move forward by binding ClusterRole to the user. To do so create a file named **lab-role-binding.yaml**.

```
vim lab-role-binding.yaml
```

5. Now paste the content below in the terminal.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: lab-cluster-role-binding
subjects:
- kind: ServiceAccount
  name: lab-user
  namespace: default
roleRef:
  kind: ClusterRole
  name: lab-cluster-role
  apiGroup: rbac.authorization.k8s.io
```

6. Apply the configuration to bind the ClusterRole to the user.

```
kubectl apply -f lab-role-binding.yaml
```

```
[root@master mylab]# vim lab-role-binding.yaml
[root@master mylab]# kubectl apply -f lab-role-binding.yaml
clusterrolebinding.rbac.authorization.k8s.io/lab-cluster-role-binding created
[root@master mylab]#
```

7. Let's confirm whether the user has the necessary permissions by pasting the following commands:

```
kubectl auth can-i list pods
```

```
kubectl auth can-i get deployments
```

```
[root@master mylab]# kubectl auth can-i get deployments
yes
[root@master mylab]# kubectl auth can-i get pods
yes
[root@master mylab]#
```

Task 4: Verify service account permissions

1. Let's create and run a nginx image as a pod without a service account in the kubernetes cluster with the help of the following command.

```
kubectl run defaultsa-pod --image=quay.io/gauravkumar9130/nginx
```

2. Let's go inside the pod with the help of the following command.

```
kubectl exec -it defaultsa-pod -- bash
```

```
[root@master mylab]# k exec -it defaultsa-pod -- bash
root@defaultsa-pod:/#
```

3. Once you are inside the pod run the following command.

```
TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
```

```
[root@master mylab]# k exec -it defaultsa-pod -- bash
root@defaultsa-pod:/# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
root@defaultsa-pod:/#
```

4. Now run the following command and you will see that it will give a message of **failed** as the service account does not have the permission to list the pods in the namespace. Use **exit** command to exit the bash terminal.

```
curl -H "Authorization: Bearer $TOKEN"
https://kubernetes/api/v1/namespaces/default/pods/ --insecure
```

```
[root@master mylab]# k exec -it defaultsa-pod -- bash
root@defaultsa-pod:/# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
root@defaultsa-pod:/# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods/ --insecure
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:default\" cannot list resource \"pods\" in API group
 \"\" in the namespace \"default\"",
  "reason": "Forbidden",
  "details": {
    "kind": "pods"
  },
  "code": 403
}root@defaultsa-pod:/#
```

5. Now, let's create a pod with the **service account** with the help of the following command:

```
vim pod.yaml
```

6. Paste the following file in the terminal and save the file and exit.

```
apiVersion: v1
kind: Pod
metadata:
  name: lab-sa-pod
spec:
  containers:
  - name: abc
    image: nginx
  serviceAccountName: lab-user
```

7. Run the following command to create the pod.

```
kubectl create -f pod.yaml
```

```
[root@master mylab]# vim pod.yml
[root@master mylab]# kubectl create -f pod.yml
pod/lab-sa-pod created
[root@master mylab]#
```

8. Now let's go inside the pod with the help of the following command:

```
kubectl exec -it lab-sa-pod -- bash
```

```
[root@master mylab]# kubectl exec -it lab-sa-pod -- bash
root@lab-sa-pod:/#
```

9. In the bash terminal, run the following command to read the content of the service account token file and to store it in **TOKEN** variable.

```
TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
```

10. Let's try to list the pods in the default namespace with the help of the following command and you will see that the command will now run successfully:

```
curl -H "Authorization: Bearer $TOKEN"
https://kubernetes/api/v1/namespaces/default/pods/ --insecure
```


```
root@lab-sa-pod:/# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/default/pods/ --insecure
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "1477203"
  },
  "items": [
    {
      "metadata": {
        "name": "app-deployment-5995975c44-4dfs8",
        "generateName": "app-deployment-5995975c44-",
        "namespace": "default",
        "uid": "2a2325e2-95f9-4c53-a45f-e4c0969b355e",
        "resourceVersion": "1451297",
        "creationTimestamp": "2024-01-14T15:37:13Z",
        "labels": {
          "app": "nginx",
          "pod-template-hash": "5995975c44"
        },
        "annotations": {
          "cni.projectcalico.org/containerID": "b7cc2921a9ba54453fc99bd8ac1e6843b48114ed8b57f757a269114482fc1731",
          "cni.projectcalico.org/podIP": "10.244.189.91/32",
          "cni.projectcalico.org/podIPs": "10.244.189.91/32"
        },
        "ownerReferences": [
          {
            "apiVersion": "apps/v1",
            "kind": "ReplicaSet",
            "name": "app-deployment-5995975c44",
            "uid": "c56eeb6f-6775-40f6-bfb5-7c7bd02cccc7",
            "controller": true,
            "blockOwnerDeletion": true
          }
        ],
        "managedFields": [
          {
            "manager": "kube-controller-manager",
```


11. Let's try to list the pods in the '**kube-system**' namespace as well with the help of the following command and you will notice that even this command will run successfully. You will get the output as shown in the image below. After that use **exit** command to exit the bash terminal.

```
curl -H "Authorization: Bearer $TOKEN"
https://kubernetes/api/v1/namespaces/kube-system/pods/ --insecure
```

```
root@lab-sa-pod:/# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
root@lab-sa-pod:/# curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/namespaces/kube-system/pods/ --insecure
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "1477666"
  },
  "items": [
    {
      "metadata": {
        "name": "calico-kube-controllers-5f58fdf897-kjnvg",
        "generateName": "calico-kube-controllers-5f58fdf897-",
        "namespace": "kube-system",
        "uid": "6a12ddc1-e693-4a09-a5b1-a7ceda56eef0",
        "resourceVersion": "1451430",
        "creationTimestamp": "2024-01-03T17:10:27Z",
        "labels": {
          "k8s-app": "calico-kube-controllers",
          "pod-template-hash": "5f58fdf897"
        },
        "annotations": {
          "cni.projectcalico.org/containerID": "039b0bcfd14df38c63e923b27e36d18e3efa892192c53f1b41aa3060f6af2f05",
          "cni.projectcalico.org/podIP": "10.244.189.95/32",
          "cni.projectcalico.org/podIPs": "10.244.189.95/32"
        },
        "ownerReferences": [
          {
            "apiVersion": "apps/v1",
            "kind": "ReplicaSet",
            "name": "calico-kube-controllers-5f58fdf897",
            "uid": "462247c0-d02a-4afb-b0a2-fb3f92df4531",
            "controller": true,
```

Task 5: Delete the serivce account

In this task we will gain hands-on experience in user management by deleting the "lab-user" service account. This task covers the proper removal process, ensuring that you will understand how to clean up service accounts and verify the successful deletion within the Kubernetes cluster.

1. Let’s delete the service account with the help of the following command:

```
kubectl delete serviceaccount lab-user
```

```
[root@master mylab]# kubectl delete serviceaccount lab-user
serviceaccount "lab-user" deleted
[root@master mylab]#
```

2. Verify the deletion by pasting the following command:

```
kubectl get serviceaccount lab-user
```

End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to implement Role-based Access Control Over Kubernetes Cluster.
2. You will use terminal.

Introduction

Understand Role-Based Access Control In Kubernetes:

- Role-based access control (RBAC) is a method of regulating access to resources in a Kubernetes cluster. It allows you to specify which users or service accounts are allowed to perform which actions on which resources.
- RBAC is implemented using `Role` and `ClusterRole` resources, which define the permissions that a user or service account has within the cluster.
- `Role` resources are used to grant permissions within a single namespace, while `ClusterRole` resources are used to grant cluster-wide permissions. `RoleBinding` and `ClusterRoleBinding` resources are used to bind a `Role` or `ClusterRole` to a user or service account, granting them the permissions defined in the role.
- RBAC uses `Subjects` to specify the users or service accounts that a role should be granted to. A `Subject` can be a `User`, a `Group`, or a `ServiceAccount`. RBAC uses `Verbs` to specify the actions that a user or service account is allowed to perform. Common verbs include `get`, `list`, `create`, `update`, and `delete`. RBAC uses `Resources` to specify the resources that a user or service account is allowed to access. Resources can include objects such as pods, services, and deployments.
- RBAC can be used to grant fine-grained permissions to users and service accounts, allowing them to only perform the actions that they need to perform. This approach can be used to improve the security of a Kubernetes cluster by only granting permissions to users and service accounts that need them.

Pre-requisites:

- Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Create a User in the cluster
4. Create a role binding and assign it to the user

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Create a user in the Cluster:

1. Create a new namespace for your user. This will allow you to isolate the resources that the user has access to and control their access to those resources.
2. Create a service account for your user in the namespace. A user account is a type of account that is used to authenticate and authorize a real-life user to access resources in the cluster.

To manage Role Based Access Control in the cluster, we have created a new user in a new namespace. Further, we will assign access to this particular user to only the created namespace. Hence the created user will not have any other access than the assigned role within the namespace. With namespaces, we can implement logical resource isolation within the cluster.

```
git clone https://github.com/gauravkumar9130/kube-user.git
cd /kube-user
chmod +x user_script.sh
./user_script.sh
```

```
[root@master kube-user]# ./user_script.sh
please type namespace
newproject
namespace/newproject created
please type username
project01
please type project01 password
project01
Changing password for user project01.
passwd: all authentication tokens updated successfully.
------------------------------Generating Certificates-----------------------------------------
Generating RSA private key, 2048 bit long modulus
.................+++
.....................................................................................+++
e is 65537 (0x10001)
'/etc/kubernetes/pki/ca.crt' -> '/root/mylab/kube-user/ca.crt'
'/etc/kubernetes/pki/ca.key' -> '/root/mylab/kube-user/ca.key'
Signature ok
subject=/CN=project01/O=newproject
Getting CA Private Key
--------------------------------Creating kubeconfig File-------------------------------
Cluster "kubernetes" set.
---------------------------------- Add user in Kube Config File---------------------------------
User "project01" set.
Context "project01-kubernetes" created.
-------------------- Copying Files -------------------------
'/root/mylab/kube-user/ca.crt' -> '/home/project01/.kube/ca.crt'
'/root/mylab/kube-user/ca.key' -> '/home/project01/.kube/ca.key'
'/root/mylab/kube-user/ca.srl' -> '/home/project01/.kube/ca.srl'
'/root/mylab/kube-user/config' -> '/home/project01/.kube/config'
'/root/mylab/kube-user/koenig.crt' -> '/home/project01/.kube/koenig.crt'
'/root/mylab/kube-user/koenig.csr' -> '/home/project01/.kube/koenig.csr'
'/root/mylab/kube-user/koenig.key' -> '/home/project01/.kube/koenig.key'
'/root/mylab/kube-user/project01.crt' -> '/home/project01/.kube/project01.crt'
'/root/mylab/kube-user/project01.csr' -> '/home/project01/.kube/project01.csr'
'/root/mylab/kube-user/project01.key' -> '/home/project01/.kube/project01.key'
'/root/mylab/kube-user/user_script.sh' -> '/home/project01/.kube/user_script.sh'
[root@master kube-user]#
```

Task 3: Create a role binding and assign it to the user:

- In this step, we will create roles and a role-binding resource that will assign particular permissions to the user. For the demo,we will create role for "get and list" for pod and services and create a role binding for the created user in the created namespace.

```
kubectl create role <rolename>--verb=get,list--resource=pods,services
```

```
[root@master mylab]# kubectl create role myrole --verb=get,list --resource=pods,services
role.rbac.authorization.k8s.io/myrole created
[root@master mylab]# kubectl describe role myrole
Name:         myrole
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 []              [get list]
  services   []                 []              [get list]
[root@master mylab]#
```

- Bind the user account to the desired role in the namespace. Roles in Kubernetes define the actions that a user is allowed to perform, such as reading and writing to resources

in the cluster. There are several built-in roles in Kubernetes, such as `view`, `edit`, and `admin`. To bind the user account to a particular role, you can use the `create rolebinding` command. For example, to bind the user account to the `myrole` role:

```
kubectl create rolebinding my-binding --role=myrole --user=project01 -n newproject
```

```
[root@master mylab]# kubectl create rolebinding mybinding --role=myrole --user=project01
rolebinding.rbac.authorization.k8s.io/mybinding created
[root@master mylab]# kubectl describe rolebinding mybinding
Name:         mybinding
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  myrole
Subjects:
  Kind  Name       Namespace
  ----  ----       ---------
  User  project01
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created a user in the cluster.
2. You have successfully created a role binding and assigned it to the user.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and configure security context for network.
2. You will using terminal.

Introduction

Security Context in Kubernetes:

In Kubernetes, a Security Context defines privilege and access control settings for a pod or container. It includes settings such as user IDs, group IDs, and Linux capabilities, influencing the security posture of the pod.

Adding Security Context to Enable Ping in a Pod:

Define Security Context:

Specify a Security Context in the pod's YAML, setting the necessary parameters.

Run as Non-Root User:

Set the "runAsNonRoot" field to true, ensuring the container runs as a non-root user.

Set Capabilities:

Assign specific Linux capabilities using the "capabilities" field, granting the pod the required permissions for executing ping.

Task Details

1. Launching Lab Environment.
2. Using the terminal.
3. Create the YML declarative file for adding security context
4. Verify the resource using Kubectl command
5. Delete the resource

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Create the YML declarative file for security context resource

In this step, we will create the manifest file

Using YAML approach

1. Create a YAML file for the pod1 and pod2 using the command below:

```
vim webpod.yml
```

2. Copy the below code into the YML file which will create a pod.

```
apiVersion: v1
kind: Pod
metadata:
 name: webpod
 labels:
  app: web
spec:
 containers:
 - name: c1
   image: quay.io/gauravkumar9130/nginxdemo
   securityContext:
    capabilities:
     add: ["NET_RAW"]
```

3. The next step is to apply the YML file to launch the pod using the commandbelow:

```
kubectl apply -f webpod.yml
```

4. Now comes the part to create another pod. Create a YML file using the command below:

```
vim testpod.yml
```

5. Copy the below code into the YML file which will create a pod.

```
apiVersion: v1
kind: Pod
metadata:
 name: testpod
 labels:
  app: web
spec:
 containers:
 - name: c1
   image: quay.io/gauravkumar9130/nginxdemo
   securityContext:
    capabilities:
     add: ["NET_RAW"]
```

6. The next step is to apply this yml file to create the resource.

```
kubectl apply -f testpod.yml
```

Till now, we have launched two pods with security context. This allows pod to ping across them.

Task 3: Verify the service using Kubectl command

1. Let's verify the deployed resources with kubectl commands. We will check and verify the running status of the pods.

```
kubectl get pods -o wide
```

```
[root@master mylab]# kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE     IP               NODE      NOMINATED NODE   READINESS GATES
testpod   1/1     Running   0          3m49s   10.244.189.121   worker2   <none>           <none>
webpod    1/1     Running   0          5m23s   10.244.189.120   worker2   <none>           <none>
[root@master mylab]#
```

2. After verifying 2 pod of nginx are running, run the below command to check whether they can ping or not:

```
kubectl exec -it testpod -- sh
ping -c3 <webpod-ip>
```

```
[root@master mylab]# kubectl exec -it testpod -- sh
/opt # ping -c3 10.244.189.120
PING 10.244.189.120 (10.244.189.120): 56 data bytes
64 bytes from 10.244.189.120: seq=0 ttl=63 time=0.095 ms
64 bytes from 10.244.189.120: seq=1 ttl=63 time=0.068 ms
64 bytes from 10.244.189.120: seq=2 ttl=63 time=0.079 ms

--- 10.244.189.120 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.068/0.080/0.095 ms
/opt #
```

3. After verifying 2 pod of nginx are running, run the below command to check whether they can ping or not:

```
kubectl exec -it webpod -- sh
ping -c3 <testpod-ip>
```

```
[root@master mylab]# kubectl exec -it webpod -- sh
/opt # ping -c3 10.244.189.121
PING 10.244.189.121 (10.244.189.121): 56 data bytes
64 bytes from 10.244.189.121: seq=0 ttl=63 time=0.099 ms
64 bytes from 10.244.189.121: seq=1 ttl=63 time=0.075 ms
64 bytes from 10.244.189.121: seq=2 ttl=63 time=0.082 ms

--- 10.244.189.121 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.075/0.085/0.099 ms
/opt #
```

Completion and Conclusion

1. You have successfully created a added security context in pod resource.
2. You have successfully pinged from one pod to other.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to upgrade the Kubernetes cluster.
2. You will use terminal.

Introduction

Upgrading a Kubernetes Cluster

Backup and Snapshot:

Perform a backup or snapshot of important cluster components and etcd data to ensure data integrity during the upgrade.

Check Compatibility:

Verify the compatibility of the Kubernetes version you are upgrading to with your existing cluster components and applications.

Upgrade Control Plane:

Upgrade the control plane components (kube-apiserver, kube-controller-manager, kube-scheduler) incrementally, following the official Kubernetes upgrade guide.

Node Upgrades:

Upgrade worker nodes one at a time to avoid service disruptions, ensuring continuous application availability.

CNI and Add-ons:

Upgrade Container Network Interface (CNI) plugins and other cluster add-ons to versions compatible with the new Kubernetes release.

Monitor and Test:

Monitor cluster health during the upgrade, and conduct thorough testing to validate application functionality and performance.

Rollback Plan:

Have a rollback plan in case issues arise, allowing you to revert to the previous state if necessary.

Communication:

Communicate upgrade plans and potential downtimes with relevant stakeholders to manage expectations.

Documentation Update:

Update cluster documentation to reflect the new Kubernetes version, including any changes in features or configurations.

Post-Upgrade Tasks:

Perform post-upgrade tasks such as updating Kubernetes client tools (kubectl), reviewing and adjusting configurations, and ensuring compatibility with applications and integrations.

Cluster Validation:

Validate the upgraded cluster using tools like kubeadm or kube-upgrade, ensuring all components are functioning as expected.

By following these steps, you can smoothly upgrade your Kubernetes cluster while minimizing disruptions and ensuring compatibility with applications and dependencies. Always refer to the official Kubernetes documentation and release notes for specific guidance on upgrading to a new version.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Upgrade the cluster
4. Validation Test

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Verify the cluster configuration

1. Run the below command to check the version of `kubectl` package. Kubectl allows us to control the cluster via a command-line interface. One can run the kubectl commands followed by the resource arguments to send API requests to the master node and run the intended task over the worker node.

   kubectl version

```
[root@master mylab]# kubectl version
Client Version: v1.28.2
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: v1.28.5
[root@master mylab]#
```

2. Now, verify the kubelet service is running by the following command. Kubelet service pods are running on both the master and worker nodes and this service is responsible for sending and receiving API requests within the cluster.

systemctl status kubelet

```
[root@master mylab]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2024-01-03 21:24:17 IST; 6 days ago
     Docs: https://kubernetes.io/docs/
 Main PID: 4993 (kubelet)
    Tasks: 11
   Memory: 46.1M
   CGroup: /system.slice/kubelet.service
           └─4993 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --confi...
```

3. Last step is to verify all the nodes are up and running. These nodes are running on some version. Run the following command to fetch the k8s version.

kubectl get nodes

```
[root@master mylab]# kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
master    Ready    control-plane   12d   v1.28.2
worker1   Ready    <none>          12d   v1.28.2
worker2   Ready    <none>          12d   v1.28.2
[root@master mylab]#
```

Task 2: To upgrade a node

1. In this step, we will cordon/disable scheduling of pods on the node.

kubectl cordon <nodename>

kubectl get nodes

```
[root@master mylab]# kubectl cordon worker1
node/worker1 cordoned
[root@master mylab]# kubectl get nodes
NAME      STATUS                     ROLES           AGE   VERSION
master    Ready                      control-plane   12d   v1.28.2
worker1   Ready,SchedulingDisabled   <none>          12d   v1.28.2
worker2   Ready                      <none>          12d   v1.28.2
[root@master mylab]#
```

2. Drain the node.

```
kubectl drain <nodename> --force --ignore-daemonsets --delete-emptydir-data
```

```
[root@master mylab]# kubectl drain worker1 --force --ignore-daemonsets --delete-emptydir-data
node/worker1 already cordoned
Warning: deleting Pods that declare no controller: default/configmap-variable-pod, default/plain, default/pod-demo2; ignoring DaemonSet-managed Pods:
kube-system/calico-node-mft59, kube-system/kube-proxy-4ghbl, metallb-system/speaker-xgm4k
evicting pod ns-1/my-dep-846d4746db-5nspn
evicting pod default/configmap-variable-pod
evicting pod default/nginx-deployment-5995975c44-f6fng
evicting pod default/plain
evicting pod ingress-nginx/ingress-nginx-admission-create-m4lgj
evicting pod default/web-1
evicting pod kube-system/kube-state-metrics-777b7bfdb5-k8l84
evicting pod metallb-system/controller-67d9f4b5bc-65jf7
evicting pod default/pod-demo2
pod/controller-67d9f4b5bc-65jf7 evicted
pod/ingress-nginx-admission-create-m4lgj evicted
pod/kube-state-metrics-777b7bfdb5-k8l84 evicted
pod/my-dep-846d4746db-5nspn evicted
pod/configmap-variable-pod evicted
pod/pod-demo2 evicted
pod/web-1 evicted
pod/plain evicted
pod/nginx-deployment-5995975c44-f6fng evicted
node/worker1 drained
[root@master mylab]#
```

3. Check the latest upgrade available.

   ```
   kubeadm upgrade plan
   ```

4. Apply the latest upgrade available

   ```
   kubeadm upgrade apply v1.xx.xx
   ```

Completion and Conclusion

1. You have successfully verified Kubernetes cluster components.
2. You have successfully upgraded K8s cluster.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to understand and deploy static pod in cluster.
2. You will practice using the terminal.

Introduction

What are Static Pods in Kubernetes?

Static pods are pods that are created and managed directly by the kubelet, rather than the API server.

They are defined in a file, rather than through the API server, and are stored in a pre-defined directory within the cluster. The Kubelet service monitors this location and performs container operations accordingly. If a static pod file is present and the pod is not running, the Kubelet service will run the static pod and likewise process in order to destroy the static pod.

Interesting part is that these pods are not under the radar of the Kube API and hence won't allow users to perform generic operations through the cli commands.

Static pods are useful in situations where the API server is not available or reliable, such as in a local development environment or during a disaster recovery scenario.

It is important to note that static pods are not recommended for production use, as they bypass the API server and therefore do not benefit from features such as API server-level authentication, authorization, and resource quotas.

Task Details

1. Launching Lab Environment.
2. Open the terminal.
3. Obtain static Pod Path
4. Create a static Pod
5. Observe the pod behaviour

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: Obtain Static Pod Path

In this task, we are going to access a Kubernetes node via SSH, examine the kubelet configuration file (`/var/lib/kubelet/config.yaml`), and identify the directory path specified by the static Pod Path configuration setting.

1. You have already SSH into one of the nodes in the kubernetes cluster. Now execute the following command to view the contents of the kubelet configuration file. Once you get the output, Look for the static Pod Path configuration setting. Note down the directory path specified.

```
cat /var/lib/kubelet/config.yaml
```

```
[root@worker1 ~]# cat /var/lib/kubelet/config.yaml
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt
authorization:
  mode: Webhook
  webhook:
    cacheAuthorizedTTL: 0s
    cacheUnauthorizedTTL: 0s
cgroupDriver: systemd
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local
containerRuntimeEndpoint: ""
cpuManagerReconcilePeriod: 0s
evictionPressureTransitionPeriod: 0s
fileCheckFrequency: 0s
healthzBindAddress: 127.0.0.1
healthzPort: 10248
httpCheckFrequency: 0s
imageMinimumGCAge: 0s
kind: KubeletConfiguration
logging:
  flushFrequency: 0
  options:
    json:
      infoBufferSize: "0"

kind: KubeletConfiguration
logging:
  flushFrequency: 0
  options:
    json:
      infoBufferSize: "0"
  verbosity: 0
memorySwap: {}
nodeStatusReportFrequency: 0s
nodeStatusUpdateFrequency: 0s
rotateCertificates: true
runtimeRequestTimeout: 0s
shutdownGracePeriod: 0s
shutdownGracePeriodCriticalPods: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s
```

Task 3: Create a static pod

In this task we will Navigate to the directory specified by the static Pod Path (e.g., /etc/kubernetes/manifests), create a static pod manifest file (static-web.yaml) defining an Nginx pod, and verify its deployment status using kubectl get pods.

1. Now, navigate to the directory specified by the **static Pod Path** with the help of the following command.

```
cd /etc/kubernetes/manifests
```

```
[root@worker1 ~]# cd /etc/kubernetes/manifests
[root@worker1 manifests]#
```

2. Create a file named static-web.yaml with the help of the following command

```
vim static-web.yaml
```

3. Paste the following yaml content.

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    type: stat
spec:
  containers:
    - name: web
      image: quay.io/gauravkumar9130/nginx
```

4. Now, let's check the pod status from the master node.

```
kubectl get pods
```

```
static-web-worker1                  1/1     Running   0          2m24s
[root@master mylab]#
```

Task 4: Observe static pod behaviour

Now, let's delete the static pod and then check the pod status after deletion.

1. In order to delete the pod, make use of the following command.

```
Kubectl delete pod <pod-name>
```

2. After successful deletion, check the pod status with the help of the following command.

```
kubectl get pods
```

3. You will observe the pod is recreated, so to dlete the pod we need to delete the manifest file.

```
rm -rf static-web.yaml
```

Completion and Conclusion

1. You have successfully obtained the static Pod Path
2. You have successfully created a static Pod.
3. You have successfully observed the static pod behavior.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to take ETCD backup the Kubernetes cluster.
2. You will use terminal.

Introduction

Etcd Backup in Kubernetes:

Etcd is a distributed key-value store that serves as the persistent datastore for Kubernetes, storing crucial cluster information. Performing regular backups of etcd is vital for ensuring data integrity, disaster recovery, and maintaining the stability of the Kubernetes cluster.

Data Resilience:

Etcd backups safeguard against data loss caused by accidental deletions, corruptions, or cluster failures.

Cluster Recovery:

In the event of a cluster failure or data corruption, etcd backups enable quick recovery, reducing downtime and ensuring business continuity.

Rollback Capabilities:

Etcd backups allow for rollbacks to previous states, facilitating the restoration of a cluster to a known and stable configuration.

Disaster Preparedness:

Backups are essential for disaster preparedness, providing a safety net in case of catastrophic events that might compromise etcd data.

Version Upgrades:

Before performing Kubernetes version upgrades, etcd backups serve as a precautionary measure, allowing safe rollback in case issues arise during the upgrade process.

Consistency Across Nodes:

Etcd backups contribute to maintaining data consistency across etcd nodes, ensuring synchronized and coherent cluster states.

Periodic Backups:

Regular, scheduled backups prevent data staleness and provide up-to-date snapshots of the etcd datastore.

Documentation Compliance:

Etcd backups align with best practices and compliance requirements, ensuring that data management adheres to industry standards and regulations.

By regularly creating and maintaining etcd backups, Kubernetes administrators enhance the reliability and robustness of their clusters, mitigating risks associated with data loss, corruption, or unforeseen issues that might impact the stability of the etcd datastore.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Take the backup

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Verify the cluster configuration

1. Run the below command to check the version of `kubectl` package. Kubectl allows us to control the cluster via a command-line interface. One can run the kubectl commands followed by the resource arguments to send API requests to the master node and run the intended task over the worker node.

   kubectl version

```
[root@master mylab]# kubectl version
Client Version: v1.28.2
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: v1.28.5
[root@master mylab]#
```

2. Now, verify the kubelet service is running by the following command. Kubelet service pods are running on both the master and worker nodes and this service is responsible for sending and receiving API requests within the cluster.

   systemctl status kubelet

```
[root@master mylab]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2024-01-03 21:24:17 IST; 6 days ago
     Docs: https://kubernetes.io/docs/
 Main PID: 4993 (kubelet)
    Tasks: 11
   Memory: 46.1M
   CGroup: /system.slice/kubelet.service
           └─4993 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --confi...
```

3. Last step is to verify all the nodes are up and running. These nodes are running on some version. Run the following command to fetch the k8s version.

```
kubectl get nodes
```

```
[root@master mylab]# kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
master    Ready    control-plane   12d   v1.28.2
worker1   Ready    <none>          12d   v1.28.2
worker2   Ready    <none>          12d   v1.28.2
[root@master mylab]#
```

Task 2: To take the ETCD backup and restore

1. In this step, run the following code on the node to take backup.

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot save /tmp/etcd.db
```

```
[root@master mylab]# ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot save /tmp/etcd.db
Snapshot saved at /tmp/etcd.db
[root@master mylab]#
```

2. To check the backup status.

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot status /tmp/etcd.db
```

```
[root@master mylab]# ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot status /tmp/etcd.db
b5443977, 1554986, 1223, 5.2 MB
[root@master mylab]#
```

3. To Restore the ETCD.

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot restore /tmp/etcd.db
```

```
[root@master mylab]# ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot restore /tmp/etcd.db
2024-01-16 07:48:14.626481 I | mvcc: restore compact to 1554281
2024-01-16 07:48:14.633489 I | etcdserver/membership: added member 8e9e05c52164694d [http://localhost:2380] to cluster cdf818194e3a8c32
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully verified Kubernetes cluster components.
2. You have successfully took backup of ETCD K8s cluster.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details:

1. This hands-on lab will guide you through the process of creating and managing Kubernetes Jobs and CronJobs, empowering you to automate and schedule tasks efficiently in your containerized applications.

What are CronJobs?

- CronJobs are a Kubernetes resource designed for scheduling and automating recurring tasks, allowing users to define cron-like schedules to execute jobs at specified intervals.
- CronJobs use a familiar cron syntax to define schedules, enabling users to specify when and how often a task should run.
- They create Jobs based on the defined schedule, providing a convenient way to automate periodic tasks within Kubernetes clusters.
- Here is the syntax for the CronJobs:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: example-cronjob
spec:
  schedule: "*/1 * * * *" # Cron schedule
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: example-container
            image: your-image
            # Additional container configuration
  # Additional CronJob configuration
```

Lab Tasks:

1. Create a Pod with volume
2. Create a Job for the Volume Snapshot
3. Verify the snapshot

Launching lab Environment:

1. Launch The Lab Environment By opening **Master** machine.
2. Open the terminal.

Lab Steps:

Task 1: Create a volume pod

1. Let's start by creating a file named **pod.yaml**. To do the same paste the below command in the terminal.

```
vim pod.yaml
```

2. In the terminal copy and paste the below **yaml file**. Save the file.

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - name: volume-container
    image: quay.io/gauravkumar3190/nginx
    volumeMounts:
    - name: data-volume
      mountPath: /data
  volumes:
  - name: data-volume
    emptyDir: {}
```

3. Run the following command to apply the configurations to the **yaml file.** Replace **<pod-name>** with the name of the pod. In this case it is **pod.yaml**.

```
kubectl apply -f <Pod-Name>
```

Task 2: Create a Job

Snapshots are used to create a copy of the current state of a file system or storage volume, providing a reliable and consistent point for backup, recovery, or data analysis purposes.

1. Create a **yaml file** named as **cron-job.yaml** with the help of the following command:

```
vim cron-job.yaml
```

2. Now paste the below yaml file in the terminal that opens up in order to create a **cronjob** named as **volume-snapshot-cronjob:**

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: volume-snapshot-cronjob
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
    spec:
      completions: 1
      template:
        metadata:
          labels:
            app: volume-snapshot
        spec:
          containers:
          - name: snapshot-container
            image: quay.io/gauravkumar9130/busybox
            command:
            - "/bin/sh"
            - "-c"
            - |
              date
              ls -la /data # Check contents of /data directory
              echo "Snapshot taken at: $(date)"
          restartPolicy: OnFailure
```

Here, the **volume-snapshot-cronjob** is a cron job that runs a scheduled task every **2 minutes**. This task is defined by a container in the cron job pod that executes a command. The command executed by the **snapshot-container** in the cron job involves interacting with the **volume-pod** that was created in the previous task. It uses **kubectl exec** to execute a command (tar) inside the volume-pod.

3. Now exit and save the file.
4. Now let's apply the configurations with the help of the following command:

```
kubectl apply -f cron-job.yaml
```

```
[root@master mylab]# vim cron-job.yaml
[root@master mylab]# kubectl apply -f cron-job.yaml
cronjob.batch/volume-snapshot-cronjob created
[root@master mylab]#
```

Task 3: Verify Snapshot

In this task, we are going to verify whether the job that we had created in the previous task is running properly and the snapshots of our **volume-pod** are been taken within the time configurations.

1. Let's start by checking the status of **Cron-job** with the help of the following command:

```
kubectl get cronjob volume-snapshot-cronjob
```

```
[root@master mylab]# kubectl get cronjob volume-snapshot-cronjob
NAME                      SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
volume-snapshot-cronjob   */2 * * * *   False     0        52s             3m45s
[root@master mylab]#
```

Now, wait for **2 mins** and then check the created jobs. Here we are waiting for two minutes as in the specifications we have mentioned the schedule for the snapshot as **2 minutes.**

2. Now wait for 2mins to check the created jobs. Here we are waiting for 2 mins as in the specifications we have mentioned the schedule for the snapshot as 2 minutes.

```
kubectl get job
```

```
[root@master mylab]# kubectl get job
NAME                               COMPLETIONS   DURATION   AGE
volume-snapshot-cronjob-28422350   1/1           6s         3m34s
volume-snapshot-cronjob-28422352   1/1           6s         94s
[root@master mylab]#
```

3. Execute the following command to check whether the job has been successfully completed.

```
kubectl describe job volume-snapshot-cronjob
```

```
Events:
  Type    Reason            Age    From            Message
  ----    ------            ----   ----            -------
  Normal  SuccessfulCreate  5m53s  job-controller  Created pod: volume-snapshot-cronjob-28422350-zrktv
  Normal  Completed         5m47s  job-controller  Job completed
```

This part in the above image shows that the job controller has successfully created the pod and the job has been successfully completed.

```
[root@master mylab]# kubectl describe job volume-snapshot-cronjob-28422350
Name:             volume-snapshot-cronjob-28422350
Namespace:        default
Selector:         batch.kubernetes.io/controller-uid=76a87579-9e46-4f76-b1e9-b92d32290c98
Labels:           app=volume-snapshot
                  batch.kubernetes.io/controller-uid=76a87579-9e46-4f76-b1e9-b92d32290c98
                  batch.kubernetes.io/job-name=volume-snapshot-cronjob-28422350
                  controller-uid=76a87579-9e46-4f76-b1e9-b92d32290c98
                  job-name=volume-snapshot-cronjob-28422350
Annotations:      batch.kubernetes.io/cronjob-scheduled-timestamp: 2024-01-15T17:50:00Z
Controlled By:    CronJob/volume-snapshot-cronjob
Parallelism:      1
Completions:      1
Completion Mode:  NonIndexed
Start Time:       Mon, 15 Jan 2024 23:20:00 +0530
Completed At:     Mon, 15 Jan 2024 23:20:06 +0530
Duration:         6s
Pods Statuses:    0 Active (0 Ready) / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  app=volume-snapshot
           batch.kubernetes.io/controller-uid=76a87579-9e46-4f76-b1e9-b92d32290c98
           batch.kubernetes.io/job-name=volume-snapshot-cronjob-28422350
           controller-uid=76a87579-9e46-4f76-b1e9-b92d32290c98
           job-name=volume-snapshot-cronjob-28422350
  Containers:
   snapshot-container:
    Image:      quay.io/gauravkumar9130/busybox
    Port:       <none>
    Host Port:  <none>
    Command:
      /bin/sh
      -c
      date
      ls -la /data # Check contents of /data directory
      echo "Snapshot taken at: $(date)"

    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age    From            Message
  ----    ------            ----   ----            -------
  Normal  SuccessfulCreate  4m52s  job-controller  Created pod: volume-snapshot-cronjob-28422350-zrktv
  Normal  Completed         4m46s  job-controller  Job completed
[root@master mylab]#
```

4. At the end you can also see the snapshots taken by the job with the help of the following command. In the outcome you will be able to see that after every 2 minutes a snapshot has been taken.

```
kubectl logs -l app=volume-snapshot
```

End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details:

1. This Lab Walks You Through Installing Prometheus along with Grafana for Monitoring a Kubernetes Cluster.
2. Cluster Configuration: **1 Master And 2 Worker Node**

What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit originally developed by SoundCloud. It is intended to gather metrics and monitor a range of systems, containers, and applications within a modern and dynamic setting. Prometheus is widely used in the DevOps and cloud-native computing fields. It is a Cloud Native Computing Foundation (CNCF) Graduated project.

What is Grafana?

Grafana is an open-source analytics and monitoring platform that provides real-time analytics, monitoring, and visualization by integrating with several data sources. It is commonly used to build configurable, interactive dashboards that track the functionality of systems, infrastructure, and applications. Grafana has grown in popularity in the DevOps and IT sectors and is a member of the Cloud Native Computing Foundation (CNCF).

Key Features of Prometheus and Grafana:

1. **Multi-Dimensional Data Model:** Prometheus makes use of a versatile key-value pair data model that makes it possible to query metrics effectively and expressively.
2. **Pull-based paradigm:** Prometheus scrapes metrics from preset targets regularly using a pull-based paradigm. This method works best in dynamic situations.
3. **PromQL Query Language:** PromQL is an effective query language that can be used to analyze and manipulate gathered metrics to reveal information about the health and performance of a system.
4. **Dynamic Dashboards:** Provides real-time data for the production of dynamic, interactive dashboards that may be customized with different graphs, charts, and visualizations.
5. **Query Editor:** Offers a versatile query editor with support for source-specific query languages that makes it easy to get and visualize data from connected data sources.

Lab Tasks:

1. Installing Prometheus
2. Installing Grafana
3. Configure Grafana with Prometheus
4. Visualize Metrics Collected from Prometheus

Launching Lab Environment:

1. Launch The Lab Environment By opening **Master** machine.
2. Open the terminal.

Lab Steps:

Installing Prometheus and Grafana:

1. We will be cloning github repo to install these packages.

```
git clone https://github.com/gauravkumar9130/grafana
```

2. Use the below-given command to deploy Prometheus and Grafana. It will create a separate namespace for installing all the monitoring resources . Install Prometheus and grafana using the below command. Get all the resources deployed in the monitoring namespace.

```
cd grafana/
```

```
kubectl create -f 1-prometheus/.
kubectl create -f 2-grafana/.
```

```
[root@master grafana]# kubectl apply -f 1-prometheus/.
clusterrolebinding.rbac.authorization.k8s.io/kube-state-metrics created
clusterrole.rbac.authorization.k8s.io/kube-state-metrics created
deployment.apps/kube-state-metrics created
namespace/monitoring created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
configmap/prometheus-server-conf created
deployment.apps/prometheus-deployment created
service/prometheus-service created
serviceaccount/kube-state-metrics created
service/kube-state-metrics created
[root@master grafana]# kubectl apply -f 2-grafana/.
deployment.apps/grafana created
configmap/grafana-datasources created
service/grafana created
[root@master grafana]#
```

3. Verify Prometheus and Grafana deployment

```
#kubectl get all -n monitoring
```

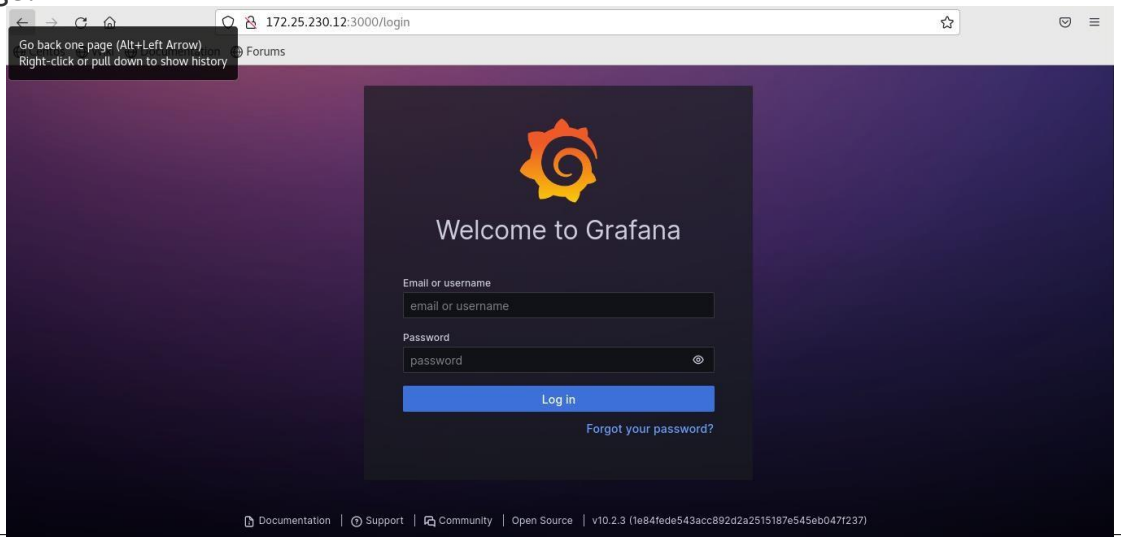
```
[root@master grafana]# kubectl apply -f 1-prometheus/.
clusterrolebinding.rbac.authorization.k8s.io/kube-state-metrics created
clusterrole.rbac.authorization.k8s.io/kube-state-metrics created
deployment.apps/kube-state-metrics created
namespace/monitoring created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
configmap/prometheus-server-conf created
deployment.apps/prometheus-deployment created
service/prometheus-service created
serviceaccount/kube-state-metrics created
service/kube-state-metrics created
[root@master grafana]# kubectl apply -f 2-grafana/.
deployment.apps/grafana created
configmap/grafana-datasources created
service/grafana created
[root@master grafana]# kubectl get all -n monitoring
NAME                                      READY   STATUS    RESTARTS   AGE
pod/grafana-657b7b7c76-rjf4b              1/1     Running   0          3m10s
pod/prometheus-deployment-7966ff495b-jcbml   1/1     Running   0          3m16s

NAME                         TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
service/grafana              LoadBalancer   10.99.46.203    172.25.230.12   3000:31423/TCP   3m10s
service/prometheus-service   ClusterIP      10.100.28.129   <none>          8080/TCP         3m16s

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/grafana               1/1     1            1           3m10s
deployment.apps/prometheus-deployment   1/1     1            1           3m16s

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/grafana-657b7b7c76              1         1         1       3m10s
replicaset.apps/prometheus-deployment-7966ff495b   1         1         1       3m16s
[root@master grafana]#
```

Configure Grafana with Prometheus:

1. Get the LoadBalancer IP for the Grafana Deployment from the previous command.

```
[root@master grafana]# kubectl get svc -n monitoring
NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
grafana              LoadBalancer   10.99.46.203    172.25.230.12   3000:31423/TCP   6m16s
prometheus-service   ClusterIP      10.100.28.129   <none>          8080/TCP         6m22s
[root@master grafana]#
```

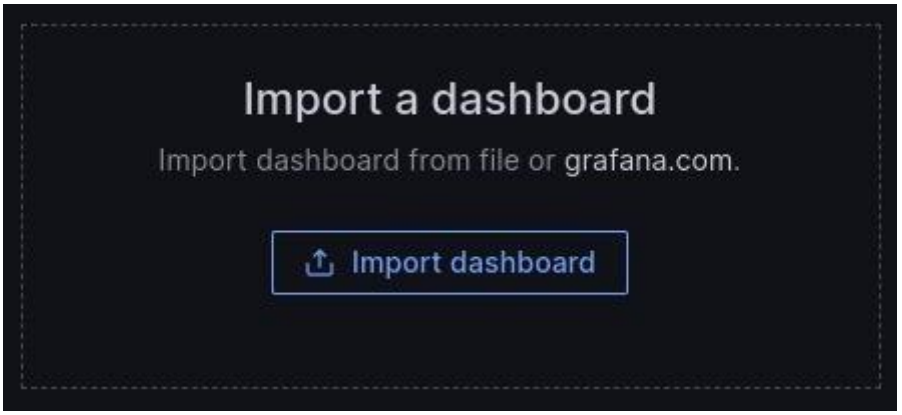
2. Paste the Grafan External IP:3000 in a web browser. It will open up the Grafana Log In Page.



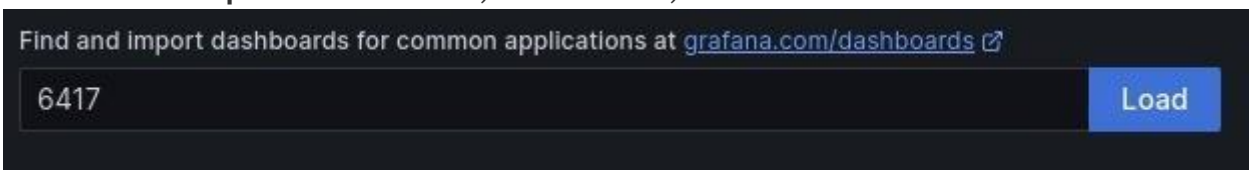
3. Use **admin** as the **Username.**
4. And, the password as **admin**.

Visualize Metrics Collected from Prometheus:

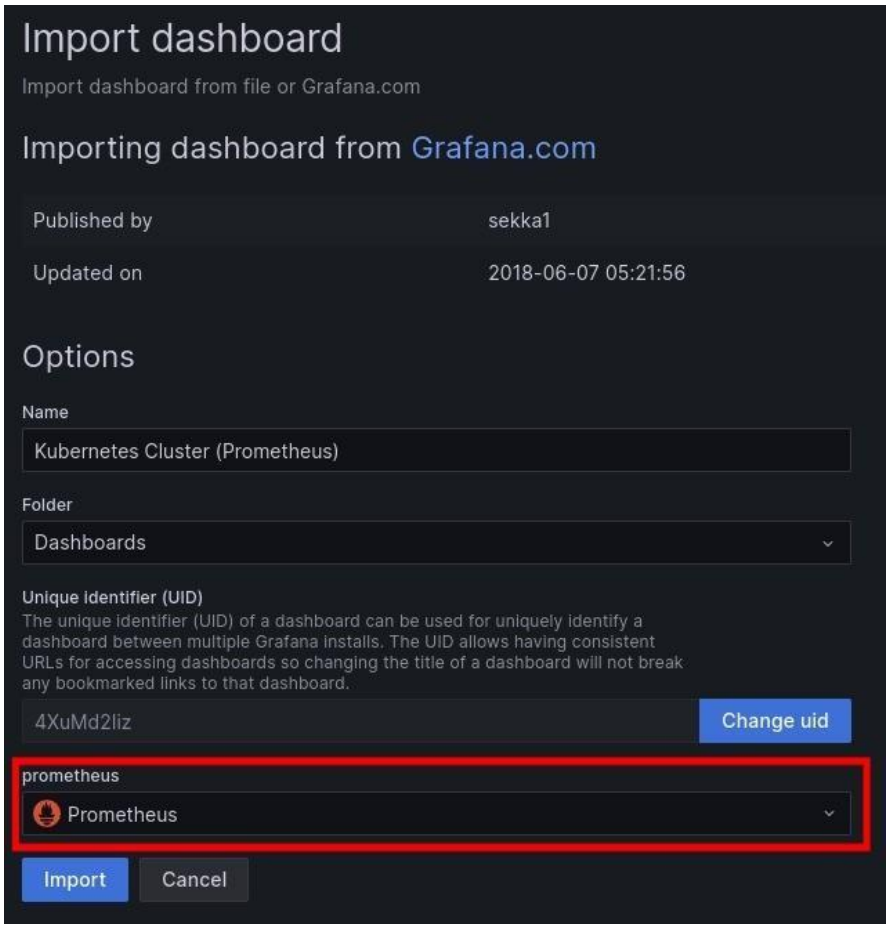
1. Click on **Import Dashboard.**



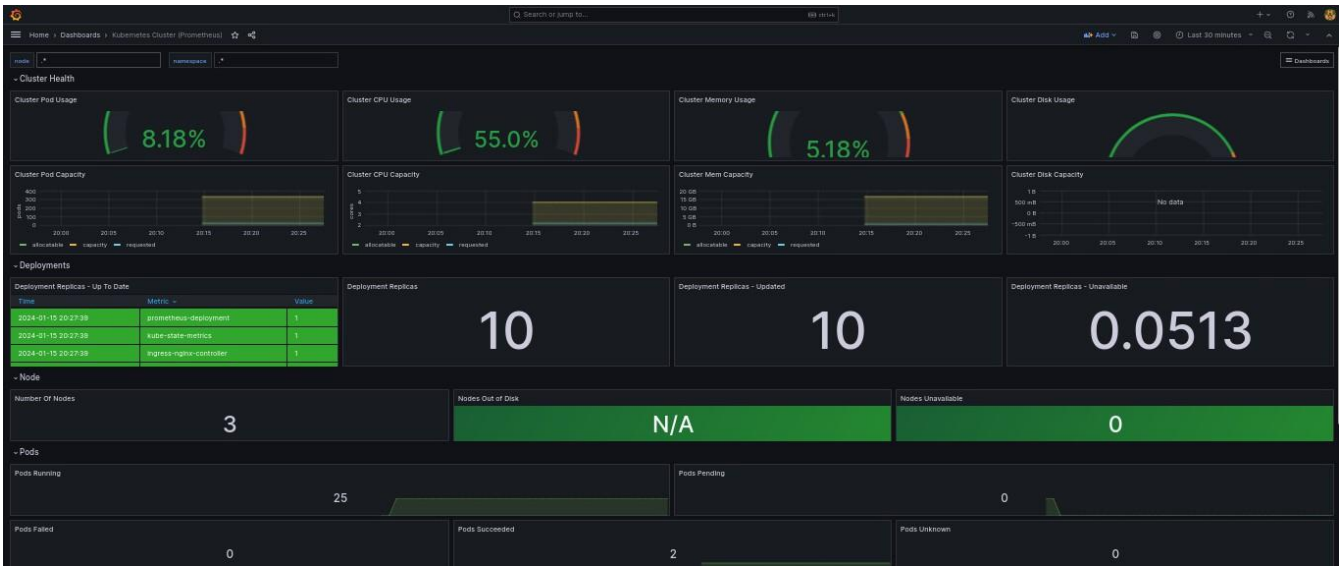
2. For **Find and Import dashboards**, Enter **6417,** and Click on **Load.**



3. Click on **Select a Prometheus data source**, Select **Prometheus**, and Click on **Import.**



4. The Dashboard will be loaded with the predefined metrics for monitoring the cluster.



End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

Lab Details:

1. This Lab Walks You Through adding Elasticsearch and Kibana Integration with Kubernetes and collecting logs using Filebeat.
2. Cluster Configuration: **1 Master And 2 Worker Node**

What is Elasticsearch?

Based on Apache Lucene, Elasticsearch is a distributed, open-source search and analytics engine. Because of its horizontal scalability design, users may effortlessly index and search massive amounts of data across numerous nodes and clusters. Elasticsearch is an essential part of the Elastic Stack (formerly called the ELK Stack), which provides a complete search, logging, and data visualization solution along with Logstash and Kibana.

What is Kibana?

Elasticsearch is used in combination with Kibana, an open-source platform for data analysis and visualization. Together with Elasticsearch and Logstash, it is a component of the Elastic Stack (formerly called the ELK Stack). An intuitive web interface called Kibana makes it easy to explore, analyze, and visualize data kept in Elasticsearch.

What is Filebeat?

Filebeat is an open-source, lightweight log shipper and data shipper that is a member of the Elastic Stack's Beats family. Its purpose is to gather, process, and send log data from several sources to centralized data storage, such as Logstash or Elasticsearch, for additional processing, analysis, and visualization. Filebeat makes it simpler to manage and analyze logs centrally by streamlining the process of importing log data from many files and systems.

Key benefits of using Elasticsearch and Kibana with Filebeat:

- **End-to-End Log Management:** From log collecting to storage, analysis, and visualization, Elasticsearch, Kibana, and Filebeat work together to offer an end-to-end log management solution.
- **Elastic Stack Interoperability:** Elasticsearch handles data storage and search, Beats handles multiple data sources, and Logstash handles data processing. All of these components work together harmoniously with EK.
- Elasticsearch and Kibana include user authentication and authorization protocols that enable administrators to restrict access to log data according to user responsibilities.

Lab Tasks:

1. Cloning EFK yaml from github repo.
2. Installing elastisearch and Kibana.
3. Installing Filebeat.
4. Accessing logs in Kibana Dashboard.

Task 1: Launching Lab Environment

1. Launch the Lab Environment opening **Master** machine.
2. Open the terminal.

Task 2: Setting up EFK

1. Clone the ELK repo from github.

   git clone https://github.com/gauravkumar9130/kube-elk

   cd kube-elk

   cat Instructions

```
[root@master kube-elk]# cat Instructions
kubectl create -f 1-elastic.yml
kubectl create -f 2-kibana.yml
kubectl create -f 3-filebeat.yml
git clone https://github.com/kubernetes/kube-state-metrics.git
kubectl apply -f kube-state-metrics/examples/standard/.
kubectl create -f 4-metricbeat.yml

Create index pattern filebeat-* metricbeat-*

[root@master kube-elk]#
```

2. Now deploying the elastic-seach, kibana and filebeat. Change the directory to kube-elk and run the following commands.
   kubectl create -f 1-elastic.yml
   kubectl create -f 2-kibana.yml
   kubectl create -f 3-filebeat.yml

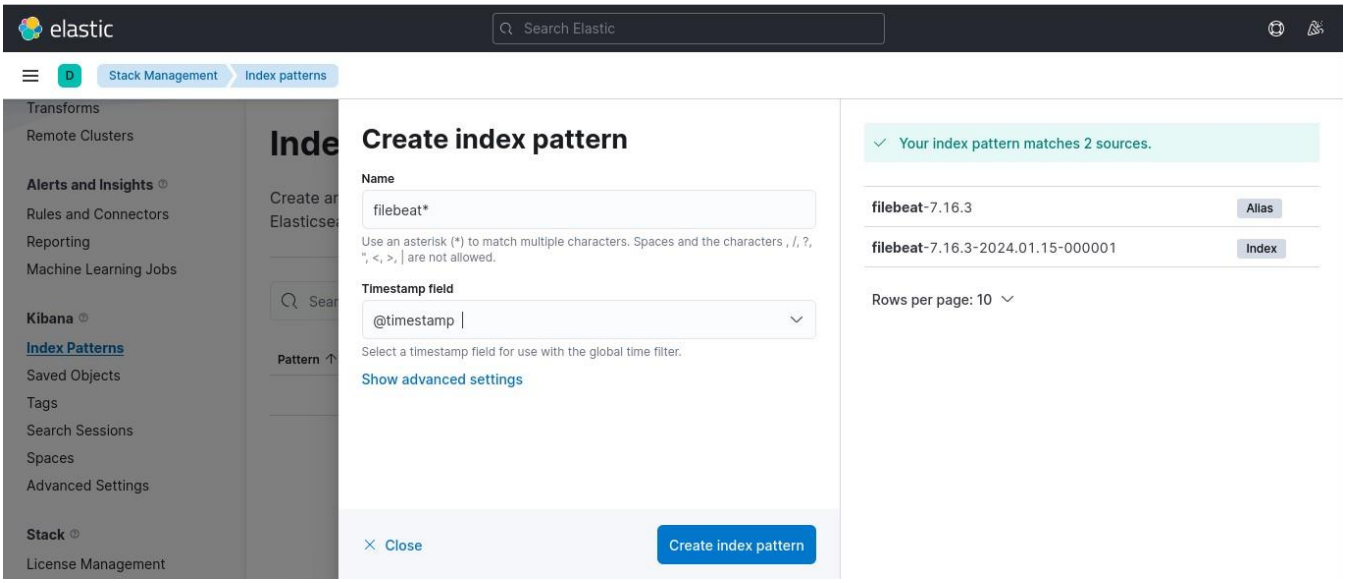
```
[root@master kube-elk]# kubectl create -f 1-elastic.yml
configmap/elasticsearch-config created
statefulset.apps/elasticsearch created
service/elasticsearch-headless created
service/elasticsearch created
[root@master kube-elk]# kubectl create -f 2-kibana.yml
deployment.apps/kibana created
service/kibana created
service/kibana-internal created
[root@master kube-elk]# kubectl create -f 3-filebeat.yml
configmap/filebeat-config created
daemonset.apps/filebeat created
clusterrolebinding.rbac.authorization.k8s.io/filebeat created
rolebinding.rbac.authorization.k8s.io/filebeat created
rolebinding.rbac.authorization.k8s.io/filebeat-kubeadm-config created
clusterrole.rbac.authorization.k8s.io/filebeat created
role.rbac.authorization.k8s.io/filebeat created
role.rbac.authorization.k8s.io/filebeat-kubeadm-config created
serviceaccount/filebeat created
[root@master kube-elk]#
```

3. Now you can go the kibana url and setup the pattern index to view the collected logs.

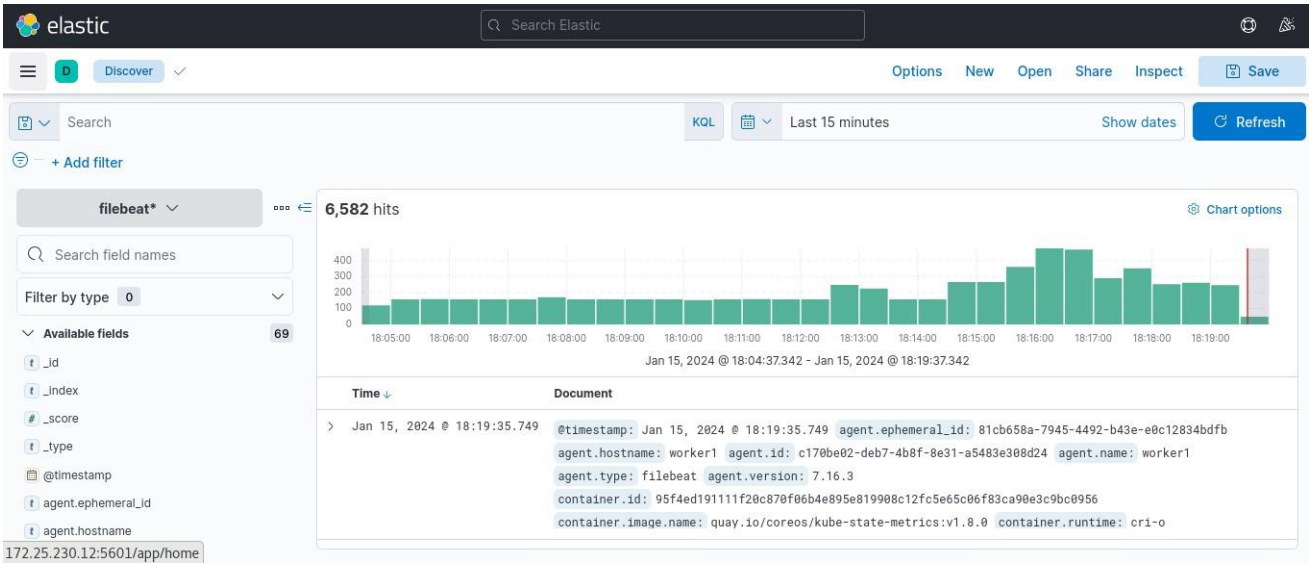
   kubectl get svc

```
[root@master kube-elk]# k get svc
NAME                    TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)             AGE
elasticsearch           ClusterIP      10.105.21.205   <none>          9200/TCP,9300/TCP   2m25s
elasticsearch-headless  ClusterIP      None            <none>          9300/TCP            2m25s
kibana                  LoadBalancer   10.99.70.66     172.25.230.12   5601:30349/TCP      2m18s
kibana-internal         ClusterIP      10.100.28.129   <none>          5601/TCP            2m18s
kubernetes              ClusterIP      10.96.0.1       <none>          443/TCP             11d
```

4. Now open the external-IP of kibana and create a index pattern to view the collected logs.



5. Now to view the logs, go to discover.



End Lab:

1. You Have Successfully Completed The Lab.
2. Once You Have Completed The Steps, **delete** the resources.

KOENIG
step forward

Lab Details

1. This lab walks you through the steps to understand and configure service networking with Ingress Controller.
2. You will use terminal.

Introduction

What are services in Kubernetes?

Services in Kubernetes allow us to implement a logical set of networking rules. Kubernetes internally allocates unique IP addresses to each pod and by defining services, we can control the communication with the pods.

As pods are volatile and pods are created and destroyed multiple times to match the desired state, a pod becomes a non-permanent resource.

Kubernetes services provide abstraction over the underlying pods and set networking policies from which pods can communicate with each other and the application can be exposed to the end users.

The set of pods to route the traffic is managed through unique selectors. For example, pods having label frontend and backend respectively will have internal network communication by the service resource.

Generally, there are 3 types of service resources in Kubernetes:

- Cluster IP
- NodePort
- Load Balancer

In this lab, we will look into the **Nginx-Ingress Controller** with hands-on demo examples.

What is Nginx Ingress Controller?

Nginx Ingress Controller is a Kubernetes component using Nginx for routing external traffic to services. With path-based routing, it directs requests based on URL paths, enabling hosting multiple apps on one domain. This dynamic solution offers SSL termination, load balancing, and flexible configurations, enhancing Kubernetes clusters' external access management.

Now let’s implement the **Nginx-Ingress Controller** and understand the concept in detail.

Task Details

1. Launching Lab Environment.
2. Using the terminal.

Lab Steps:

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.
3. Alongside the EC2 instance, the Whizlabs terminal is also being provisioned.

Task 2: Adding Ingress Controller

1. We will add ingress-controller from Kubernetes github repo to our cluster.
2. Run the command

```
git clone https://github.com/kubernetes/ingress-nginx
```

3. Now we will deploy the controller.

```
kubectl apply -f ingress-nginx/deploy/static/provider/cloud/deploy.yaml
```

```
[root@master mylab]# kubectl create -f ingress-nginx/deploy/static/provider/cloud/deploy.yaml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
serviceaccount/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
service/ingress-nginx-controller-admission created
deployment.apps/ingress-nginx-controller created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
[root@master mylab]#
```

4. Confirm the controller status

```
kubectl get pods,svc -n ingress-nginx
```

```
[root@master mylab]# kubectl get pods,svc -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
pod/ingress-nginx-admission-create-m4lgj    0/1     Completed   0          56s
pod/ingress-nginx-admission-patch-2x522     0/1     Completed   0          56s
pod/ingress-nginx-controller-5d974c544-t2ll5   1/1     Running     0          56s

NAME                                         TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)                      AGE
service/ingress-nginx-controller             LoadBalancer   10.107.108.230   172.25.230.10   80:31408/TCP,443:32187/TCP   57s
service/ingress-nginx-controller-admission   ClusterIP      10.106.223.114   <none>          443/TCP                      56s
[root@master mylab]#
```

Task 3: Deploying application

1. We will deploy 3 different nginx-based application.

2. We will use kubectl run command to deploy 3 deployments.

```
kubectl create deployment hotel --image=quay.io/gauravkumar9130/hotel --replicas=2
kubectl create deployment tea --image=quay.io/gauravkumar9130/tea --replicas=2
kubectl create deployment coffee --image=quay.io/gauravkumar9130/coffee --replicas=2
```

3. Now we will deploy three services exposing our deployment.

```
kubectl expose deployment hotel --target-port=80 --port=80
kubectl expose deployment tea --target-port=80 --port=80
kubectl expose deployment coffee --target-port=80 --port=80
```

4. Confirm the deployment and service status

```
kubectl get deploy,svc
```

```
[root@master mylab]# kubectl get deployment,svc
NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coffee   2/2     2            2           63s
deployment.apps/hotel    2/2     2            2           87s
deployment.apps/tea      2/2     2            2           75s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/coffee       ClusterIP   10.98.2.58     <none>        80/TCP    10s
service/hotel        ClusterIP   10.96.20.156   <none>        80/TCP    22s
service/kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   10d
service/tea          ClusterIP   10.101.34.25   <none>        80/TCP    16s
[root@master mylab]#
```

KOENIG
step forward

Task 4: Creating Ingress Object

1. Create the following ingress.yaml file for ingress Controller.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hotel-app-ing
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - http:
      paths:
      - path: /hotel
        pathType: Prefix
        backend:
          service:
            name: hotel
            port:
              number: 80
      - path: /tea
        pathType: Prefix
        backend:
          service:
            name: tea
            port:
              number: 80
      - path: /coffee
        pathType: Prefix
        backend:
          service:
            name: coffee
            port:
              number: 80
```

2. We will create the ingress object.

```
kubectl apply -f  ingress.yaml
```

```
[root@master mylab]# vim ingress.yaml
[root@master mylab]# kubectl create -f ingress.yaml
ingress.networking.k8s.io/hotel-app-ing created
[root@master mylab]#
```

3. Now we will verify the ingress object.

```
kubectl get ing
```

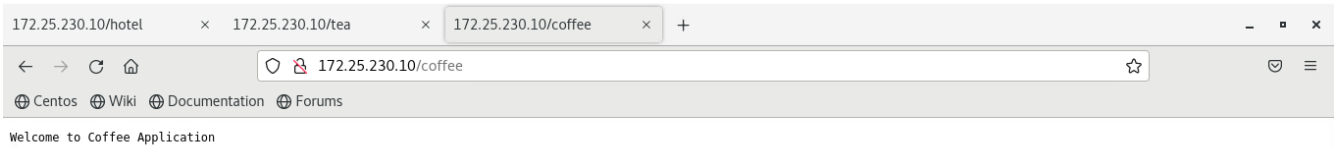
```
[root@master mylab]# k get ing -w
NAME            CLASS   HOSTS   ADDRESS         PORTS   AGE
hotel-app-ing   nginx   *                       80      15s
hotel-app-ing   nginx   *       172.25.230.10   80      41s
^C[root@master mylab]#
```

4. Confirm the path based routing using the ingress address via browser.

   `<ingress-IP>/hotel`

   `<ingress-IP>/tea`

   `<ingress-IP>/coffee`



End Lab

1. Exit from the terminal SSH session.
2. You have completed the lab.
3. Once you have completed the steps, **delete** the resources.

Lab Details

1. This hands-on lab will guide you through the implementation and usage of namespaces in Kubernetes. You will create two namespaces using different methods and deploy simple Nginx applications within each namespace.
2. You will practice using the terminal .

Introduction

Understanding Kubernetes namespace

- Namespaces in Kubernetes provide a way to virtually partition and isolate resources within a cluster. They allow multiple virtual clusters to share the same physical cluster, providing logical separation between different teams, projects, or environments.
- Namespaces provide a scope for Kubernetes resources such as pods, services, and replication controllers.
- Resources created within a namespace are accessible only to other resources within the same namespace, preventing naming conflicts and providing a clear organizational structure.
- Namespaces play a crucial role in implementing Role-Based Access Control (RBAC) in Kubernetes. By segregating resources into namespaces, administrators can apply different access policies and permissions to different teams or users. This helps enforce the principle of least privilege and enhances overall cluster security.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. List existing namespace
4. Create Namespace
5. Create deployments

6. List pods

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening **Master** machine.
2. Open the terminal.

Task 2: List existing namespace

In this step, we will Verify existing namespaces in your Kubernetes cluster by running the following command:

```
kubectl get namespaces
```

```
[root@master mylab]# kubectl get ns
NAME              STATUS   AGE
default           Active   11d
ingress-nginx     Active   47h
kube-node-lease   Active   11d
kube-public       Active   11d
kube-system       Active   11d
metallb-system    Active   10d
[root@master mylab]#
```

Task 3: Create namespace

There are two ways to create a namespace in kubernetes. In this task we are going to create two namespaces each with a different method.

1. Let's start by creating the first namespace i.e **ns-1** with the first method. To do so paste the following command in the terminal and press **Enter.**

```
kubectl create namespace ns-1
```

```
[root@master mylab]# kubectl create namespace ns-1
namespace/ns-1 created
[root@master mylab]#
```

2. Now, let's create the second namespace **ns-2**. Paste the following command in the terminal.

```
vim ns-2.yaml
```

3. In the terminal that opens up, paste the following manifest and then press **CTRL+X** and the **Y** and then **Enter** to save and then Exit the terminal.

```
apiVersion: v1
kind: Namespace
metadata:
  name: ns-2
```

4. Apply the changes in the namespace with the help of the following command:

```
kubectl apply -f ns-2.yaml
```

```
[root@master mylab]# kubectl apply -f ns-2.yaml
namespace/ns-2 created
[root@master mylab]#
```

Task 4: Create Deployments

In this task, we are going to deploy an **Nginx** application in both the create namespaces.

1. Let's create the deployment firstly in **ns-1** namespace with the help of the following command.

```
kubectl create deployment my-dep --image=quay.io/gauravkumar9130/nginx --
replicas=1 --namespace ns-1
```

```
[root@master mylab]# kubectl create deployment my-dep --image=quay.io/gauravkumar9130/nginx --replicas=1 --namespace=ns-1
deployment.apps/my-dep created
[root@master mylab]#
```

2. Now use the following command to deploy **Nginx** application in **ns-2**.

```
kubectl create deployment my-dep2 --image= quay.io/gauravkumar9130/nginx --
replicas=1 --namespace ns-2
```

```
[root@master mylab]# kubectl create deployment my-dep2 --image=quay.io/gauravkumar9130/nginx --replicas=1 --namespace=ns-2

deployment.apps/my-dep2 created
[root@master mylab]#
```

Task 6: List pods

Now, list all the pods across namespaces using either of the following command:

```
kubectl get pods --all-namespaces
```

```
[root@master mylab]# kubectl get pods --all-namespaces
NAMESPACE        NAME                                       READY   STATUS      RESTARTS        AGE
default          app-deployment-5995975c44-4dfs8            1/1     Running     1               23h
default          app-deployment-5995975c44-dbz89            1/1     Running     1               23h
default          defaultsa-pod                              1/1     Running     0               113m
default          lab-sa-pod                                 1/1     Running     0               103m
ingress-nginx    ingress-nginx-admission-create-m4lgj       0/1     Completed   0               47h
ingress-nginx    ingress-nginx-admission-patch-2x522        0/1     Completed   0               47h
ingress-nginx    ingress-nginx-controller-5d974c544-t2ll5   1/1     Running     1               47h
kube-system      calico-kube-controllers-5f58fdf897-kjnvg   1/1     Running     1               11d
kube-system      calico-node-4b9tl                          1/1     Running     1               11d
kube-system      calico-node-mft59                          1/1     Running     1               11d
kube-system      calico-node-r2b7s                          1/1     Running     1               11d
kube-system      coredns-5dd5756b68-bwhr4                   1/1     Running     1               11d
kube-system      coredns-5dd5756b68-smb6d                   1/1     Running     1               11d
kube-system      etcd-master                                1/1     Running     1               11d
kube-system      kube-apiserver-master                      1/1     Running     1               11d
kube-system      kube-controller-manager-master             1/1     Running     1               11d
kube-system      kube-proxy-4ghbl                           1/1     Running     1               11d
kube-system      kube-proxy-bjxp6                           1/1     Running     1               11d
kube-system      kube-proxy-wtzhn                           1/1     Running     1               11d
kube-system      kube-scheduler-master                      1/1     Running     1               11d
kube-system      kube-state-metrics-777b7bfdb5-k8l84        1/1     Running     0               33m
metallb-system   controller-67d9f4b5bc-65jf7                1/1     Running     1               10d
metallb-system   speaker-8v8nk                              1/1     Running     2 (5h58m ago)   10d
metallb-system   speaker-dk6zv                              1/1     Running     2 (5h58m ago)   10d
metallb-system   speaker-xgm4k                              1/1     Running     2 (5h58m ago)   10d
ns-1             my-dep-846d4746db-5nspn                    1/1     Running     0               2m38s
ns-2             my-dep2-5f7c79cc7b-fgtcn                   1/1     Running     0               62s
[root@master mylab]#
```

```
kubectl get pods -n ns-2
```

```
[root@master mylab]# kubectl get pods -n ns-2
NAME                          READY   STATUS    RESTARTS   AGE
my-dep2-5f7c79cc7b-fgtcn      1/1     Running   0          2m19s
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created the namespaces
2. You have successfully created the deployments in both namespaces
3. You have successfully listed the pods across the namespaces.

End Lab

1. You have completed the lab.
2. Once you have completed the steps, **delete** the resources.

Lab Details

1. This lab walks you through the steps to launch and configure a Metal LB in the Kubernetes cluster.
2. You will use terminal.

Introduction

What is Metal LB?

MetalLB is an open-source load balancer implementation designed for bare-metal Kubernetes clusters. In Kubernetes, LoadBalancer services typically rely on cloud providers to provision external load balancers. However, on bare-metal environments where such cloud providers are absent, MetalLB steps in to offer load balancing functionality. Key aspects of MetalLB include:

Load Balancing for Bare Metal:

MetalLB extends Kubernetes with the capability to allocate and manage external IP addresses for services in bare-metal environments.

Layer 2 and BGP Support:

MetalLB supports both Layer 2 (ARP) and Border Gateway Protocol (BGP) modes, providing flexibility in how it advertises and manages IP addresses.

Integration with Network Topology:

It integrates with the underlying network topology, allowing it to distribute load across nodes in the cluster.

Configuration with ConfigMaps:

MetalLB configurations are managed through Kubernetes ConfigMaps, providing a declarative and Kubernetes-native approach to configuration.

Service Type LoadBalancer Support:

Users can define services of type LoadBalancer in Kubernetes, and MetalLB will allocate external IP addresses and handle load balancing for those services.

Scalability and High Availability:

MetalLB is designed to scale with the number of nodes in the cluster and can be configured for high availability setups.

Open Source and Community-Driven:

MetalLB is an open-source project with an active community contributing to its development and maintenance.

By offering load balancing capabilities for bare-metal Kubernetes clusters, MetalLB facilitates the deployment of applications in on-premises or private data center environments where cloud-based load balancers may not be available.

Pre-requisites:

Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Using terminal
3. Install an Metal LB
4. Validation Test

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment opening **Master** machine.
2. Open the terminal.

Task 2: Verify the cluster configuration

1. Run the below command to check the version of `kubectl` package. Kubectl allows us to control the cluster via a command-line interface. One can run the kubectl commands followed by the resource arguments to send API requests to the master node and run the intended task over the worker node.

   kubectl version

   ```
   [root@master mylab]# kubectl version
   Client Version: v1.28.2
   Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
   Server Version: v1.28.5
   [root@master mylab]#
   ```

2. Now, verify the kubelet service is running by the following command. Kubelet service pods are running on both the master and worker nodes and this service is responsible for sending and receiving API requests within the cluster.

systemctl status kubelet

```
[root@master mylab]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2024-01-03 21:24:17 IST; 6 days ago
     Docs: https://kubernetes.io/docs/
 Main PID: 4993 (kubelet)
    Tasks: 11
   Memory: 46.1M
   CGroup: /system.slice/kubelet.service
           └─4993 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --confi...
```

3. Last step is to verify all the prerequisite pods are up and running. These pods are the helping hand in performing cluster operations like scheduling, network policy and route management, API request handling, proxy service and so on. Run the following command to fetch the pods running in all the namespaces.

kubectl get pods --all-namespaces

```
[root@master mylab]# kubectl get pods --all-namespaces
NAMESPACE        NAME                                       READY   STATUS    RESTARTS   AGE
kube-system      calico-kube-controllers-5f58fdf897-kjnvg   1/1     Running   0          5d22h
kube-system      calico-node-4b9tl                          1/1     Running   0          5d22h
kube-system      calico-node-mft59                          1/1     Running   0          5d22h
kube-system      calico-node-r2b7s                          1/1     Running   0          5d22h
kube-system      coredns-5dd5756b68-bwhr4                   1/1     Running   0          6d
kube-system      coredns-5dd5756b68-smb6d                   1/1     Running   0          6d
kube-system      etcd-master                                1/1     Running   0          6d
kube-system      kube-apiserver-master                      1/1     Running   0          6d
kube-system      kube-controller-manager-master             1/1     Running   0          6d
kube-system      kube-proxy-4ghbl                           1/1     Running   0          5d23h
kube-system      kube-proxy-bjxp6                           1/1     Running   0          6d
kube-system      kube-proxy-wtzhn                           1/1     Running   0          5d23h
kube-system      kube-scheduler-master                      1/1     Running   0          6d
metallb-system   controller-67d9f4b5bc-65jf7                1/1     Running   0          4d23h
metallb-system   speaker-8v8nk                              1/1     Running   0          4d23h
metallb-system   speaker-dk6zv                              1/1     Running   0          4d23h
metallb-system   speaker-xgm4k                              1/1     Running   0          4d23h
[root@master mylab]#
```

Make sure all pods display status **RUNNING** which implies all the required pods are up and running and we can perform the further steps.

Task 2: Add and verify Metal LB to cluster

1. In this step, we will launch add metal lb. We will use the official link to deploy metal lb to cluster. Run the following command to deploy

wget https://raw.githubusercontent.com/metallb/metallb/v0.13.7/config/manifests/metallb-native.yaml

kubectl apply -f metallb-native.yaml

2. Verify the deployment of metal lb.

kubectl get pods -n metallb-system

```
[root@master mylab]# kubectl get pods -n metallb-system
NAME                          READY   STATUS    RESTARTS       AGE
controller-67d9f4b5bc-65jf7   1/1     Running   1              11d
speaker-8v8nk                 1/1     Running   2 (16h ago)    11d
speaker-dk6zv                 1/1     Running   2 (16h ago)    11d
speaker-xgm4k                 1/1     Running   2 (16h ago)    11d
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully verified Kubernetes cluster components.
2. You have successfully deployed a metal LB inside K8s cluster.

End Lab

1. You have successfully completed the lab.
2. Once you have completed the steps, **delete** the resources.

KOENIG
step forward

Lab Details

1. This lab walks you through the steps to understand and configure service networking with Network type.
2. You will practice using the terminal.
3. Duration: **60 minutes**

Introduction

What is Network Policy in Kubernetes?

- A Network Policy in Kubernetes is a specification that outlines the rules for controlling the communication between groups of pods within a cluster.
- It provides granular control over ingress (incoming) and egress (outgoing) traffic to and from pods, allowing administrators to define communication rules.
- Network Policies use a pod selector to identify groups of pods based on their labels. Rules are then applied to these selected pods.
- Ingress rules define how incoming traffic is handled. These rules specify the sources from which traffic is allowed or denied and the ports that are accessible.
- Egress rules govern outgoing traffic from pods, specifying the destinations that pods are allowed to communicate with and the ports that are accessible.
- Network Policies support two main types: Ingress policies and Egress policies, each tailored for controlling traffic in their respective directions.
- Rules can be based on the labels of peer pods or specified using IP blocks, offering flexibility in defining the allowed traffic sources and destinations.
- Network Policies can include default policies that apply to all pods within a namespace in the absence of specific policies for individual pods.

  They enhance the security and isolation of pods, enabling administrators to enforce network segmentation within a Kubernetes cluster.

  Network Policies are created and managed using the Kubernetes Network Policy API. Administrators define policies in YAML or JSON format.

  Network Policies are particularly valuable in scenarios where multi-tier applications with different components (frontend, backend, database) require controlled and secure communication.

Pre-requisites:

- Basic Linux Commands

Task Details

1. Launching Lab Environment.
2. Open the terminal
3. Create the YML declarative file for Network Policy
4. Verify the Network Policy using Kubectl command

Lab Steps:

Lab Steps

Task 1: Launching Lab Environment

1. Launch the lab environment by opening the **Matser** machine.
2. Open the **terminal**.

Task 2: Create the YML declarative file for Network Policy

In this step, we will create the manifest file for Network Policy

Here, we have to mention 2 types of ports to route the traffic to the application running inside the pod. One is the port to be exposed from the node(can be picked from the default range). The other port is the actual application port. For example: for a web server running inside the pod, the target port would be 80.

Using YAML approach

1. Create a YAML file for the sample nginx pod using the command below:

```
vim webpod.yml
```

2. Copy the below code into the YML file which will create a pod of nginx server and adding "Net_RAW" capability to the container.

```
apiVersion: v1
kind: Pod
metadata:
 name: webpod
 labels:
  app: web
spec:
 containers:
 - name: c1
   image: quay.io/gauravkumar9130/nginxdemo
   securityContext:
    capabilities:
     add: ["Net_RAW"]
```

3. The next step is to apply the YML file to launch the deployment using the command below:

```
kubectl apply -f webpod.yml
```

4. Now comes the part to create the service and define the networking rule using which we will expose the nginx pod to the public world. Create a YML file using the command below:

```
vim networkpolicy.yml
```

5. Now pass the below sample yml code. Here, apiVersion is v1 which means this resource is included and maintained under the mentioned version of the Kubernetes API. Also, the keyword kind suggests the type of resource which is service in this scenario. We will use label selectors for the service to detect the apps running on the same hierarchy and eventually it will distribute traffic equally based on the nodes.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-nginx-policy
spec:
 podSelector:
  matchLabels:
   app: web
 policyTypes:
 - Ingress
 ingress:
 - from:
 - podSelector:
  matchLabels:
   app: db
```

6. The next step is to apply this yml file to create the service as declared.

```
kubectl apply -f networkpolicy.yml
```

7. Till now, we have launched nginx server and created network policy.

Task 3: Verify the network policy using Kubectl command

1. Let's verify the deployed resources with kubectl commands. First, we will check the pod and verify the running status of the pod.

```
kubectl get pods
```

2. After verifying of nginx pod is running, run the below command to check whether the network policy is applied or not:

```
kubectl get networkpolicy
```

3.

```
[root@master mylab]# k get pods,netpol
NAME          READY   STATUS    RESTARTS   AGE
pod/webpod    1/1     Running   0          11s

NAME                                                    POD-SELECTOR   AGE
networkpolicy.networking.k8s.io/allow-nginx-policy      <none>         11s
[root@master mylab]#
```

Completion and Conclusion

1. You have successfully created a NetworkPolicy in Kubernetes.

End Lab

1. You have successfully **completed** the lab.
2. Once you have completed the steps, **delete** the resources.