Lab 1 : Kubernetes cluster

Step 1: Prerequisties Setup

On node1

```
# yum install vim -y
# hostname master
# echo 127.0.0.1  master >> /etc/hosts
```

logout and login to check the hostname setup on the master machine

On node2

```
# yum install vim -y
# hostname worker1
# echo 127.0.0.1  worker1 >> /etc/hosts
```

logout and login to check the hostname setup on the worker1 machine

On node3

```
# yum install vim -y
# hostname worker2
# echo 127.0.0.1  worker2 >> /etc/hosts
```

logout and login to check the hostname setup on the worker2 machine

---

Step 2: Install Docker CE

Set up the repository and Install required packages.

```
# yum install yum-utils device-mapper-persistent-data lvm2
# yum-config-manager   --add-repo   https://download.docker.com/linux/centos/docker-ce.repo
```

Install Docker CE.

```
# yum update && yum install docker-ce-18.06.2.ce
```

```
# Restart Docker
systemctl daemon-reload
systemctl restart docker
systemctl enable docker
```

Step 3: Install kubeadm

You will install these packages on all of your machines:

- kubeadm: the command to bootstrap the cluster.
- kubelet: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- kubectl: the command line util to talk to your cluster.

```
# cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

```
# yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes
```

```
# systemctl enable --now kubelet
```

Step 4: Enable Net packet fowarding
Some users on RHEL/CentOS 7 have reported issues with traffic being routed incorrectly due to iptables being bypassed. You should ensure net.bridge.bridge-nf-call-iptables is set to 1 in your sysctl config, e.g.

```
# cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

```
# sysctl –system
```

Step 5: Setup master node
The control-plane node is the machine where the control plane components run, including etcd (the cluster database) and the API server (which the kubectl CLI communicates with).

```
# kubeadm init  --pod-network-cidr=192.16.0.0/16
```

To make kubectl work for your non-root user, run these commands, which are also part of the kubeadm init output:

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

Step 6: Enable Network CNI
We are using calico to setup the pod cluster network

```
# kubectl apply -f https://docs.projectcalico.org/v3.8/manifests/calico.yaml
```

Step 7: Adding worker node
Run the following command on the master node to generate command to add worker on the master node.

```
# kubeadm token create --print-join-command
```

copy the ouput

Now login into each worker node and paste the command.
Once done, login back to the master node and run the following command to check the cluster staus

```
# kubectl get nodes
```

You should be able to see all three nodes in ready state.
Congrats you have successfully deployed the kubernetes cluster using kubeadm.

Lab 2: Pods lab

Step 1: Namespace

Create a name space with the name new-ns with the manifest file

```
# vim ns.yml
```

```
apiVersion: v1
kind: Namespace

metadata:
  name: new-ns
```

save and quit the file

```
# kubectl create -f ns.yml
# kubectl get ns
```

Step 2: Pod

Create a pod name pod-data with the manifest file in the new-ns namespace with nginx image

```
# vim pod.yml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  namespace: new-ns
spec:
  containers:
    - name: cont1
      image: nginx
```

save and quit the file

```
# kubectl create -f pod.yml
# kubectl get pods  - n new-ns
```

Step 3: Pod with label

Create a pod named pod2 with label env=test with redis image

```
# vim pod.yml

apiVersion: v1

kind: Pod

metadata:

  name: pod2

  labels:

    env: test

  spec:

  containers:

    - name: cont1

      image: redis
```

save and quit the file

```
# kubectl create -f pod2.yml
# kubectl get pods -o wide
# kubectl get pods –show-labels
```

Lab 3: Replica's

Step 1: RC

Setup a replica controller rc1 with 3 replicas and the pod selector to be env=prod

```
# vim rc.yml

apiVersion: v1

kind: ReplicationController

metadata:

  name: rc1

spec:

  replicas: 3

  selector:

    env: prod

  template:

    metadata:

      labels:

        env: prod

    spec:

      containers:

      - name: cont1

        image: nginx
```

save and quit the file

```
# kubectl create -f rc.yml
```

```
# kubectl get rc
```

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

Step 2: ReplicaSet

Create a Replica set rs1 with 3 replicas and two pod selection as env = test or env = prod

```
# vim rs.yml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

  name: rs1

spec:

  replicas: 3

  selector:

    matchExpressions:

      - key: env

        operator: In

        values:

          - prod

          - test

  template:

    metadata:

      labels:

        env: prod

    spec:

      containers:

      - name: cont1

        image: nginx
```

save and quit the file

```
# kubectl create -f rs.yml
```

```
# kubectl get rs
```

```
# kubectl get pods
```

Step 3: DaemonSet

Create the daemonset and see it’s operation

```
# vim ds.yaml

apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: ds1

spec:

  selector:

    matchLabels:

      abc: xyz

  template:

    metadata:

      labels:

        abc: xyz

    spec:

      containers:

      - name: newcont

        image: nginx
```

save and quit the file

```
# kubectl create -f ds.yml

# kubectl get daemonset

# kubectl get pods -o wide
```

Lab 3: Services

Step 1: ClusterIP Service

Create a pod with label new=old and nginx image and expose it with clusterIP service on port 8080

```
# vim l3pod.yml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: l3pod
  labels:
    new: old
spec:
  containers:
    - name: cont1
      image: nginx
```

Save and quit the file

```
# kubectl create -f l3pod.yml
# kubectl get pods -o wide
```

```
# vim cip.yml
```

```
apiVersion: v1

kind: Service

metadata:

  name: cipservice

spec:

  type: ClusterIP

  selector:

    new: old

  ports:

    - name: port1

      port: 8080

      TargetPort: 80
```

save and quit the file

```
# kubectl create -f cip.yml

# kubectl get svc

# curl <serviceipaddress>
```

Step 1: NodePort Service

Create a pod with label old=new and nginx image and expose it with clusterIP service on port 8080 and node port 32080

```
# vim l3pody.yml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: l3pody
  labels:
    old: new
spec:
  containers:
    - name: cont1
      image: nginx
```

Save and quit the file

```
# kubectl create -f l3pod.yml
# kubectl get pods -o wide
```

```
# vim nip.yml
```

```
apiVersion: v1

kind: Service

metadata:

  name: cipservice

spec:

  type: NodePort

  selector:

    new: old

  ports:

    - name: port1

      port: 8080

      TargetPort: 80

       nodePort: 32080
```

save and quit the file

```
# kubectl create -f nip.yml
```

```
# kubectl get svc
```

Open your desktop brower and access http://mastereip:32080

Lab 4: Storages

Step 1: Host path Volume
Create a host path volume to mount /data directory into /vishal of the container

```
# vim l4pod.yml

apiVersion: v1
kind: Pod

metadata:
  name: pod3

spec:
  containers:
    - name: cont1
      image: nginx
      volumeMounts:
        - name: vol1
          mountPath: /vishal

  volumes:
    - name: vol1
      hostPath:
        path: /data

# kubectl create -f l4pod.yml
# kubectl get pods
```

Lab 5: Injecting Data

Step 1: Configmap

Create a config map with name qwe with key value pair to be v1=vishal
Export this variable in the pod l5pod with environment variable name as username

```
# kubectl create configmap qwe --from-literal=v1=vishal
```

```
# kubectl get configmap
```

```
# vim l5pod.yml
apiVersion: v1
kind: Pod

metadata:
  name: l5pod
spec:
  containers:
    - name: cont1
      image: nginx
      env:
        - name: username
          valueFrom:
            configMapKeyRef:
              name: qwe
               key: v1
```

save and quit the file

```
# kubectl create -f l5pod.yml
```

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

```
# kubectl exec -it pod l5pod
```

inside the pod # echo $username
the output should be vishal

Step 2: Secret

Create a config map with name sec1 with key value pair to be pass=redhat
Export this variable in the pod l5pod with environment variable name as password

```
# kubectl create secret generic sec1 --from-literal=pass=redhat
```

```
# kubectl get secret
```

```
# vim l5pod1.yml
apiVersion: v1
kind: Pod

metadata:
  name: l5pod1
spec:
  containers:
    - name: cont1
      image: nginx
      env:
        - name: username
          valueFrom:
            secretKeyRef:
              name: sec1
               key: pass
```

save and quit the file

```
# kubectl create -f l5pod.yml
```

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

```
# kubectl exec -it pod l5pod
```

inside the pod # echo $password
the output should be redhat

Lab 6: Deployment

Step 1: Deployment
Deploy a deployment dp1 with redis image and 3 replicas.

```
# vim l6dp.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dp1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: cont1
        image: redis
```

save and quit the file

```
# kubectl create -f l6dp.yml

# kubectl get deployment

# kubectl get pods -o wide
```

Lab 7: Authentication

Step 1: User creation
Create a user on the master machine then go into its home directory to perform the remaining steps.

```
# useradd jean &&  cd /home/jean
```

Step 2: Certification generations
Create the private key and certificate signing request

```
# openssl genrsa -out jean.key 2048
# openssl req -new -key jean.key -out jean.csr -subj "/CN=jean"
```

Step 3: Sign the CSR
Sign the CSR with the Kubernetes CA. We have to use the CA cert and key which are normally in /etc/kubernetes/pki/. Our certificate will be valid for 500 days

```
# openssl x509 -req -in jean.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -
CAcreateserial -out jean.crt -days 500
```

Step 4: Securing identification
Create a “.certs” directory where we are going to store the user public and private key.

```
# mkdir .certs && mv jean.crt jean.key .certs
```

Step 5: K8s User creation
Create the user inside kubernetes

```
# kubectl config set-credentials jean --client-certificate=/home/jean/.certs/jean.crt
--client-key=/home/jean/.certs/jean.key
```

Step 6: Context
Create a context for the user.

```
# kubectl config set-context jean-context --cluster=kubernetes –user=jean
```

Step 7: Set RBAC

```
# mkdir /home/jean/.kube/config
```

Ask the instruction from trainer to create the config file.
# chown -R jean: /home/jean

```
# kubectl create ns mynstest
```

From root, create a file name rb.yml and put the following entry

```
# vim rb.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
```

```
metadata:
  name: jean
  namespace: mynstest
subjects:
- kind: User
  name: jean
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

```
# kubectl create -f rb.yml
# kubectl get rolebinding  -n mynstest
```

login from jean user account and try creating the pod in mynstest namespace, you should be able to do the same.

Lab 8: Security

Scenario : Setting up host namespace in pod
Create a pod with network namespace to be shared with that of the host namespace.

Step1: Create pod manifest
Create the following manifest defination with details

```
# vim podns.yml
apiVersion: v1
kind: Pod

metadata:
  name: podns1

spec:
  hostNetwork: true
  containers:
    - name: cont1
      image: nginx
```

save and quit the file

```
# kubectl create -f podns.yml
# kubectl get pods -o wide
```

check the ip associate with the pod, it should be the ip address of the worker node on which it is deployed.

Scenario : Using Security Context
In the scenario we will try to run a pod with root access denied as let’s see if it run or throws the error.

Step 1: Create the manifest file with following details

```
# vim podsc.yml
apiVersion: v1
kind: Pod

metadata:
  name: podsc1

spec:
  containers:
    - name: cont1
      image: nginx
      securityContext:
        runAsNonRoot: true
```

Save and quit the file.

```
# kubectl create -f podsc.yml
# kubectl get pods -o wide
```

You should not be able see the pod in the running state as there is some issue with the container configuration

```
# kubectl describe pod podsc1
```

Scroll down to the bottom to see the error mentioning the container need root privilege to run.

Scenario : Managing Network Policies

In this scenario, we will manage network policies

First scenario will be restricted pod to communicate with each other inside a namespace.

Step 1: Create a namespace and deploy two nginx pods inside it.

```
# vim pod.yml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1

spec:
  containers:
    - name: cont1
      image: nginx
```

save and quit.

```
# kubectl create -f pod.yml
```

Repeat the same step by changing pod name to pod2 now.

Check the ip for both pods

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

Login into pod1 and try doing the curl for another pod

# kubectl exec -it pod1 */bin/bash*

*Inside the pod # curl <**ip of second pod**>*

You should be able to connect with the nginx application running on the pod.

Step 2: Apply the default deny network policy

```
# vim default-ns.yml
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
```

save and quit the file

```
# kubectl create -f default-ns.yml
# kubectl get networkpolicy
```

Now login into pod1 again and try to access nginx on pod2

```
# kubectl exec -it pod1 /bin/bash
inside the pod # curl ip of pod 2
```

This time you should not be able to connect.

Step 3: Allow controlled traffic between these two pods

```
# kubectl label pod pod1 env=test
# kubectl label pod pod2 env=prod
```

write the network policy to enable the communication from pod1 to pod2 for port 80.

```
# vim ns.yml
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-nspolicy
spec:
  podSelector:
    matchLabels:
      env: prod
  ingress:
  - from:
    - podSelector:
        matchLabels:
          env: test
    ports:
- port: 80
```

save and quit the file.

```
# kubectl create -f ns.yml
```

Follow the steps to login into pod1 and try access pod2
You should be able to access it this time.

Lab 8: Computing Resources

Scenario : Set request
In this exercise we will setup the request for container in a pod defination

Step 1: Create the pod resource with following resource request

```
# vim podcr.yml
apiVersion: v1
kind: Pod
metadata:
  name: podcr1

spec:
  containers:
    - image: busybox
      command: ["dd", "if=/dev/zero", "of=/dev/null"]
      name: main
      resources:
        requests:
          cpu: 200m
          memory: 10Mi
```

save and quit the file.

```
# kubectl create -f podcr.yml
```

To verify the memory allocated run the following command to check the status

```
# kubectl exec -it podcr1 top
```

In the output you should be able to see the memory allocated at the top.

Scenario : Create limit range
In this scenario, we will be creating the limit range to define default limit and request for a namespace.

```
# kubect create ns limit-ns
```

```
# vim limitns.yml
apiVersion: v1
kind: LimitRange
metadata:
  name: example
spec:
  limits:
    - type: Pod
      min:
        cpu: 50m
        memory: 5Mi
      max:
        cpu: 1
        memory: 1Gi
    - type: Container
      defaultRequest:
        cpu: 100m
        memory: 10Mi
      default:
        cpu: 200m
        memory: 100Mi
      min:
        cpu: 50m
        memory: 5Mi
      max:
        cpu: 1
        memory: 1G
```

save and quit the file

```
# kubectl create -f limitns.yml -n limit-ns
# kubectl get limitrange -n limit-ns
```

Scenario : Using Node affinity

In this scenario, we will deploy the pod to a node with nodeselector option

```
# kubectl label node worker1 disk=ssd
# kubectl get nodes –show-labels
```

```
# vim podsel.yml
apiVersion: v1
kind: Pod

metadata:
  name: podsel1

spec:
  nodeSelector:
    disk: ssd
  containers:
- name: cont1
   image: nginx
```

save and quit the file.

```
# kubectl creat -f podsel.yml
# kubectl get pods -o wide
```

The output should show you that the pod is deployed on the worker node 1