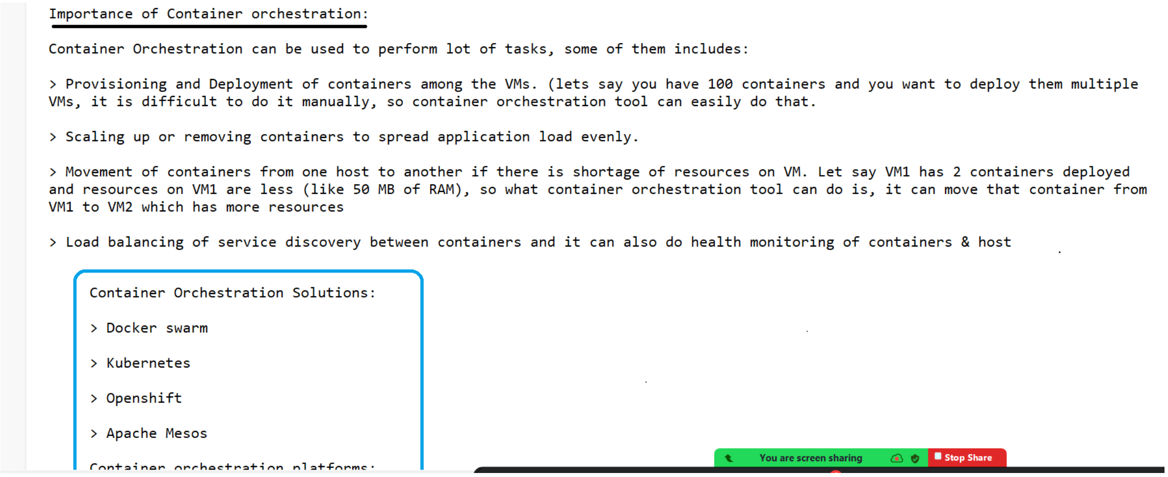
Notes - By Sonal Mittal



Importance of Container orchestration:

Container Orchestration can be used to perform lot of tasks, some of them includes:

> Provisioning and Deployment of containers among the VMs. (lets say you have 100 containers and you want to deploy them multiple VMs, it is difficult to do it manually, so container orchestration tool can easily do that.

> Scaling up or removing containers to spread application load evenly.

> Movement of containers from one host to another if there is shortage of resources on VM. Let say VM1 has 2 containers deployed and resources on VM1 are less (like 50 MB of RAM), so what container orchestration tool can do is, it can move that container from VM1 to VM2 which has more resources

> Load balancing of service discovery between containers and it can also do health monitoring of containers & host

Container Orchestration Solutions:

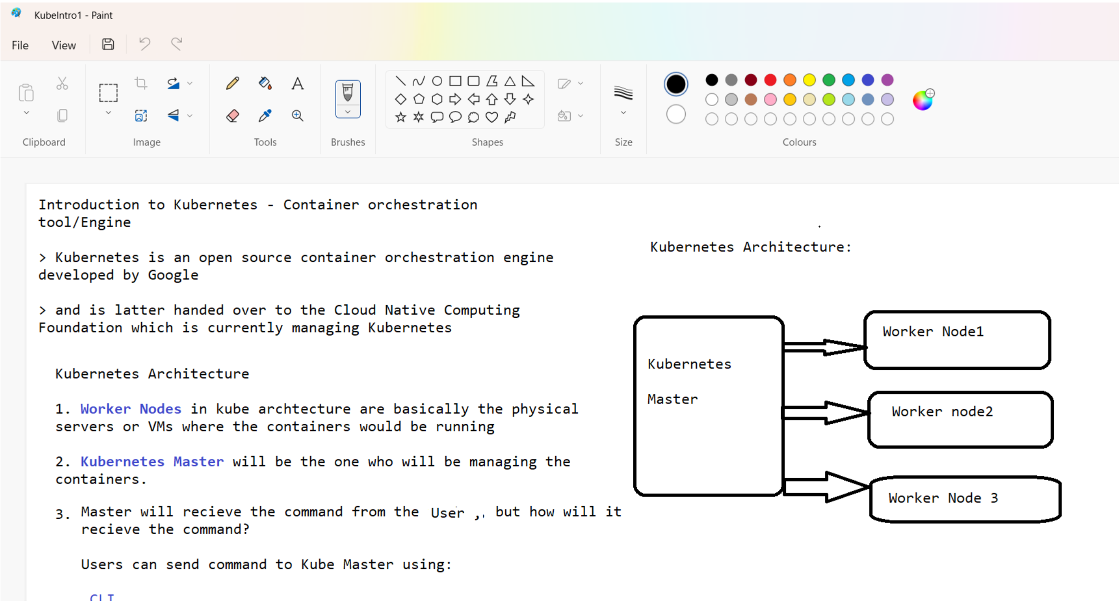
> Docker swarm

> Kubernetes

> Openshift

> Apache Mesos

Container orchestration platforms:



Introduction to Kubernetes - Container orchestration tool/Engine

> Kubernetes is an open source container orchestration engine developed by Google

> and is latter handed over to the Cloud Native Computing Foundation which is currently managing Kubernetes

Kubernetes Architecture

1. Worker Nodes in kube archtecture are basically the physical servers or VMs where the containers would be running

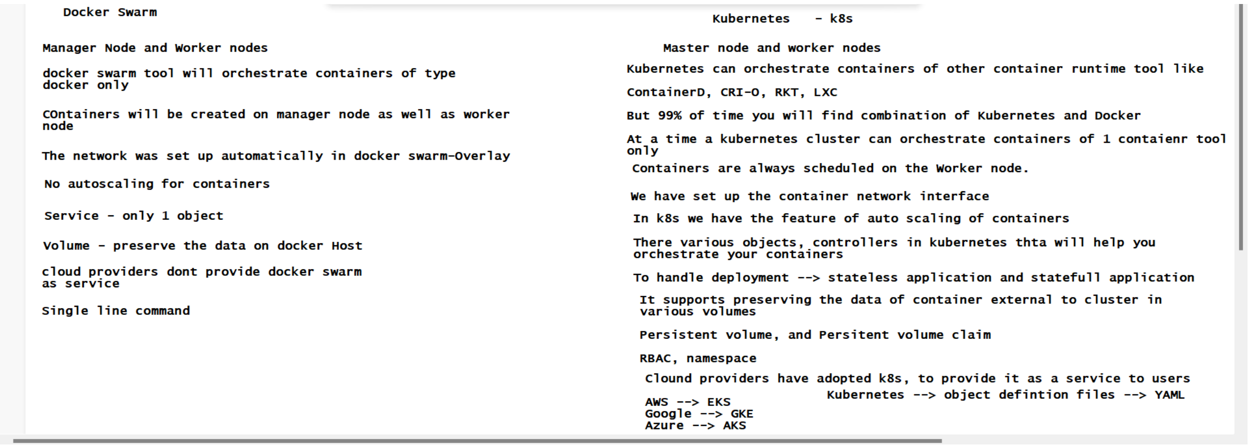
2. Kubernetes Master will be the one who will be managing the containers.

3. Master will recieve the command from the User ,, but how will it recieve the command?

Users can send command to Kube Master using:

CLI

Kubernetes Architecture:



Docker Swarm

Manager Node and Worker nodes

docker swarm tool will orchestrate containers of type docker only

COntainers will be created on manager node as well as worker node

The network was set up automatically in docker swarm-Overlay

No autoscaling for containers

Service - only 1 object

Volume - preserve the data on docker Host

cloud providers dont provide docker swarm as service

Single line command

Kubernetes - k8s

Master node and worker nodes

Kubernetes can orchestrate containers of other container runtime tool like

ContainerD, CRI-O, RKT, LXC

But 99% of time you will find combination of Kubernetes and Docker

At a time a kubernetes cluster can orchestrate containers of 1 contaienr tool only

Containers are always scheduled on the Worker node.

We have set up the container network interface

In k8s we have the feature of auto scaling of containers

There various objects, controllers in kubernetes thta will help you orchestrate your containers

To handle deployment --> stateless application and statefull application

It supports preserving the data of container external to cluster in various volumes

Persistent volume, and Persitent volume claim

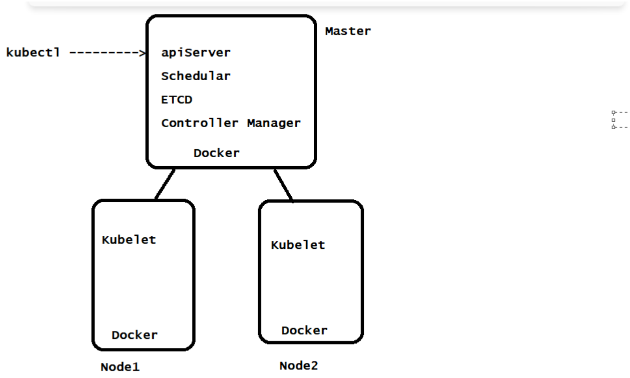
RBAC, namespace

Clound providers have adopted k8s, to provide it as a service to users

Kubernetes --> object defintion files --> YAML

AWS --> EKS
Google --> GKE
Azure --> AKS

Kubernetes Architecture:



****

Use Kubeadm to install and set up kubernetes on VMs

==

Setup of Kubernetes on VM

Step1: On Master Node Only

```
## Install Docker

sudo wget https://raw.githubusercontent.com/lerndevops/labs/master/scripts/installDocker.sh -P /tmp
sudo chmod 755 /tmp/installDocker.sh
sudo bash /tmp/installDocker.sh
sudo systemctl restart docker

## Install kubeadm,kubelet,kubectl

sudo wget https://raw.githubusercontent.com/lerndevops/labs/master/scripts/installK8S-v1-23.sh -P /tmp
sudo chmod 755 /tmp/installK8S-v1-23.sh
sudo bash /tmp/installK8S-v1-23.sh
```

```
   71  docker -v
   72  kubeadm version -o short
   73  kubelet --version
   74  kubectl version --short --client
```

```
## Initialize kubernetes Master Node

   sudo kubeadm init --ignore-preflight-errors=all

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

   ## install networking driver -- Weave/flannel/canal/calico etc...

   ## below installs calico networking driver

   kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.24.1/manifests/calico.yaml

   # Validate:  kubectl get nodes
```

Step2: On All Worker Nodes

```
## Install Docker

sudo wget https://raw.githubusercontent.com/lerndevops/labs/master/scripts/installDocker.sh -P /tmp
sudo chmod 755 /tmp/installDocker.sh
sudo bash /tmp/installDocker.sh
sudo systemctl restart docker

## Install kubeadm,kubelet,kubectl

sudo wget https://raw.githubusercontent.com/lerndevops/labs/master/scripts/installK8S-v1-23.sh -P
/tmp
sudo chmod 755 /tmp/installK8S-v1-23.sh
sudo bash /tmp/installK8S-v1-23.sh

   71  docker -v
   72  kubeadm version -o short
   73  kubelet --version
   74  kubectl version --short --client

## Run Below on Master Node to get join token

sudo kubeadm token create --print-join-command

    copy the kubeadm join token from master & run it on all nodes

    Ex: kubeadm join 10.128.15.231:6443 --token mks3y2.v03tyyru0gy12mbt \
           --discovery-token-ca-cert-hash
sha256:3de23d42c7002be0893339fbe558ee75e14399e11f22e3f0b34351077b7c4b56
```

****

To check kube system pods:

…
kubectl get pods -n kube-system

Set up kubernetes → GCP → Google kubernetes Engine

Kubernetes commands:

Will list the worker nodes attached to the Master machine

```
…
kubectl get nodes
…
kubectl get nodes -o wide
…
```

Create a Pod of nginx image:

```
mkdir mykubefiles

cd mykubefiles

vim pod-defintion.yml
```

```
apiVersion: v1
kind: Pod
metadata:
 name: pod1
 labels:
  author: sonal
  app: webserver
spec:
 containers:
  - name: c1
    image: nginx
```

```
…
kubectl create -f pod-defintion.yml
…
kubectl get pods
```

```
…
kubectl get pods -o wide
…
kubectl describe pod pod1 | less
…
kubectl logs pod1
…
kubectl logs -f pod1 -c c1
…

kubectl get pods --show-labels
…
kubectl get pods -l author=sonal
```

MultiContainer POD:
****

```
apiVersion: v1
kind: Pod
metadata:
 name: multicont
 labels:
  author: sonal
  role: dev
spec:
 containers:
  - name: cont1
    image: httpd
  - name: cont2
    image: tomcat
```

```
kubectl create -f pod-defintion.yml
kubectl get pods -o wide
kubectl describe pod multicont | less

kubectl delete pods --all
```

MultiContainer POD - Example 2:

```
apiVersion: v1
kind: Pod
metadata:
 name: multicont3
 labels:
  author: sonal
  role: dev
spec:
 containers:
  - name: cont1
    image: httpd
  - name: cont2
    image: tomcat
  - name: cont3
    image: ubuntu
    command: ["bash", "-c", "sleep 6000"]
```

****

Example 3:

```
apiVersion: v1
kind: Pod
metadata:
 name: multicont3
 labels:
  author: sonal
  role: dev
spec:
 containers:
  - name: cont1
    image: nginx
  - name: cont2
    image: busybox
    command:
     - sleep
     - "6000"
```

```
kubectl create -f pod-defintion.yml
kubectl get pods -o wide
kubectl exec -it multicont3 -c cont1 -- bash
kubectl exec -it multicont3 -c cont2 -- sh
```

****

```
# kubectl get pods multi-pod2 -o jsonpath='{.spec.containers[*].name}'
```

this will give you names of containers on the pod

==

Static Pods

****

Which process in kubernetes is responsible to create PODs

Kubelet is the process which is used to create and run PODS

By default we send request to apiserver to create pods, but it is kubelet that created pods on the scheduled node

So can we directly instruct kubelet to create a pod on the node of a cluster
Yes that is possible: using the concept of Static pods

- Static pods are created by Kubelet directly
- We should give Pod yaml to kubelet so that kubelet can create pod
- if you have to give YAML to kubelet Then we have to logon to the node where the pods has to be created
- kubelet process running on the node reads the YAMl file from a location/path on that node or VM where we can place the yaml
- By default kubelet has a configuration path called StaticPODpath
- When Pod created by the kubelet it appends the node name to the podname in the YAML

Example:
****

Go to a worker node:

execute following command:

```
# service kubelet status
```

Now go to this location

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

Create a file with name as

```
vim static-pod.yml
```

Copy the text from master machine(static-pod.yml file) into this file

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: static-pod
  name: static-pod
spec:
  containers:
  - image: nginx
    name: static-pod
```

Save the file

****

Go to Master node

****

kubectl get pods --> static pod wil be there , here node name will be appeneded to pod name

kubectl get pods -o wide --> pod will be scenduled on the same node by kubelet

Now delete the pods and observe the behaviour

```
$ kubectl delete pod <podname>
```

You will observe kubelet will recreaate the pod on itself

The only way to delete static pod is to delete the yaml file on slave node in the directory cd /etc/kubernetes/manifests

go to slave

****

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

```
rm static-pod.yml
```

go to master and check the if pod is available or not

****

If in case the pod is failing and you want to check the kubelet logs on the SLAVE:

```
# journalctl -u kubelet.service | less
```

****

CKA practice test questions:

1. Create a static pod named static-busybox that uses the busybox image and the command sleep 1000

2. Create a static pod on node01 called nginx-critical with image nginx. Create this pod on node01 and make sure that it is recreated/restarted automatically in case of a failure.

   Use /etc/kubernetes/manifests as the Static Pod path for example.

   Kubelet Configured for Static Pods
   Pod nginx-critical-node01 is Up and running

DAEMON SETS:

****

Demon sets ensure that all the nodes in the cluster runs single copy of a Pod.

As a new node is added to the cluster, Single pod will automatically be created on that node

If the node is deleted those pods will also get deleted

Commonly used:

Run a pod on every node, that run a log collection deamoon

Run a pod on every node for cluster backup

Run a pod on every node for monitoring

Example:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: myds
spec:
 selector:
  matchLabels:
    type: webserver
 template:
  metadata:
   name: mypod
   labels:
    type: webserver
  spec:
   containers:
    - image: nginx
      name: c1
```

===

Demo 2:

Configure a DaemonSet to run the image k8s.gcr.io/pause:2.0 in the cluster.

Solution:

```
kubectl create deployment testds --image=k8s.gcr.io/pause:2.0
--dry-run=client -o yaml > testds.yaml
```

then edited it as Daemonset to get it running, you don't do replicas in a daemonset, it runs on all nodes

remove the replicas field from Yaml, ensure the Yaml looks like below.

```
apiVersion: apps/v1
kind: DaemonSet   ## update Deployment to DaemonSet
metadata:
  labels:
     app: testds
 name: testds
spec:
 selector:
     matchLabels:
      app: testds
 template:
     metadata:
      creationTimestamp: null
      labels:
           app: testds
     spec:
      containers:
      - image: k8s.gcr.io/pause:2.0
           name: pause
```

```
kubectl create -f testds.yaml
kubectl get ds  ## ensure the pods are running
```

Here are some of the main reasons why you might use a DaemonSet in Kubernetes:

Running system-level services: Since DaemonSets ensure that a specific Pod is running on each node in a cluster, they are ideal for running system-level services that need to be present on every node in the cluster. This includes services like log collectors, monitoring agents, and other system daemons.

Resource utilization: DaemonSets can help optimize resource utilization by spreading the workload across all nodes in a cluster. By running a single instance of a Pod on each node, DaemonSets can ensure that resources are being used efficiently.

Scaling: DaemonSets can be used to automatically scale services based on the number of nodes in a cluster. As new nodes are added to the cluster, new instances of the Pod are automatically created to run on those nodes.

Rolling updates: DaemonSets can also be used to manage rolling updates of system-level services. By updating the DaemonSet configuration, Kubernetes will automatically cr

****

> Deployment

====================================

It is a high level object in kubernetes which provides you 3 main features:

Here Replicas = PODs

> create multiple replicas of an Image, high availability of replicas
> A deployment object will create an object called as ReplicSet
> A replicaset object will create the desired replicas(Pods)
> Scale up and scale down the replicas

We mention labels in the pod template
Labels are used for selecting pods of same group in the cluster
Labels are mandatory in deployment

vim deployment.yml

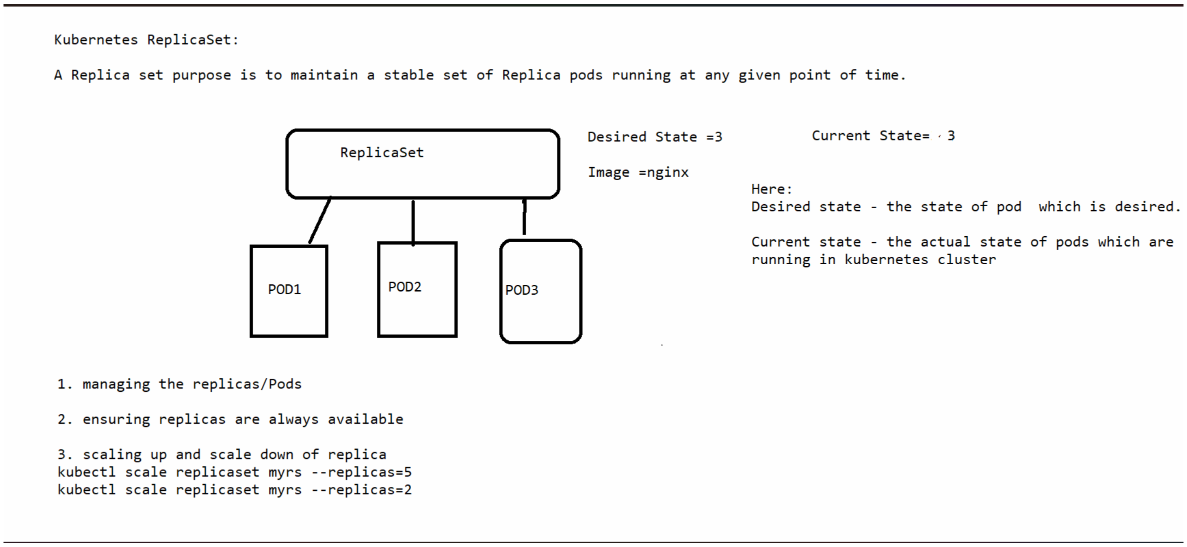
```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mydep
spec:
 replicas: 4  # replicas= desired count of pods of same  image to b created
 selector: # query the cluster to see the current count of replicas
  matchLabels:
   type: webserver
 template: # pod specification for the replicas
  metadata:
   labels:
    type: webserver
  spec:
```

```
  containers:
   - name: c1
     image: nginx
```

```
kubectl apply deployment.yml
kubectl get all
kubectl delete pod mydep-86ddfbc958-2gm5c
kubectl get all
kubectl scale deployment mydep --replicas=6
kubectl scale deployment mydep --replicas=2
```

> Rolling update feature - update the image on the deployment

ReplicaSet:



Kubernetes ReplicaSet:

A Replica set purpose is to maintain a stable set of Replica pods running at any given point of time.

Desired State =3

Image =nginx

Current State= 3

Here:
Desired state - the state of pod which is desired.

Current state - the actual state of pods which are running in kubernetes cluster

1. managing the replicas/Pods

2. ensuring replicas are always available

3. scaling up and scale down of replica

```
kubectl scale replicaset myrs --replicas=5
kubectl scale replicaset myrs --replicas=2
```

ReplicaSet.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: myrs
  # labels optional
spec:
 replicas: 3
 selector:
  matchLabels:
   type: webserver
# provide the pod template
 template:
  metadata:
   name: mypod
   labels:
    type: webserver
  spec:
   containers:
    - name: c1
      image: nginx
```

```
kubectl create -f replicaset.yml
kubectl get all
kubectl describe replicaset myrs | less
kubectl scale --replicas=2 replicaset myrs
```

For you information:
****

```
# kubectl explain ReplicaSet | less
```

to know the values of the YAML file

```
# kubectl run pod2 --image nginx --dry-run=client -o yaml
```

Generate the YAMl file template

```
# kubectl scale --replicas=6 replicaset myrs --dry-run=client -o yaml | less
```

****

Service Object:
===

Delete the previous deployment:

```
# kubectl delete deployment mydep
```

Check the resources:

```
# kubectl get all
```

Step1: Create a nginx pod which will be our end point

```
apiVersion: v1
kind: Pod
metadata:
 name: pod
 labels:
  author: sonal
  type: webserver
spec:
 containers:
  - name: c1
    image: nginx
```

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

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

Make note of the ipaddress= that will the end point ipaddress

Pod ip address = endpoint = 192.168.22.10

Step2: Create Cluster IP service for above pod

ClusterIP:

```
vim service1.yml
```

```
apiVersion: v1
kind: Service
metadata:
 name: mysvc1
spec:
 type: ClusterIP  # type of service
 selector: # endpoints selector
  type: webserver
 ports:
  - port: 80 # service port
    targetPort: 80 # container Image port
```

```
#  kubectl create -f service1.yml
# kubectl get service mysvc1 -o wide
# kubectl describe service mysvc1 | less
```

```
service/mysvc1      ClusterIP   10.96.248.116   <none>       80/TCP    18s
```

Step 3: Create a Test pod of image ubuntu that will send request to service ip:80, which inturn will forward request to nginx pod(end point)

```
vim testpod.yml
```

```
apiVersion: v1
kind: Pod
metadata:
 name: testpod
 labels:
  author: sonal
spec:
 containers:
  - name: c1
    image: ubuntu
    args: [/bin/bash, -c, 'sleep 6000']
```

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

Lets validate if test pod is able to communicate with nginx pod using cluster IP

```
kubectl exec -it testpod -- bash
```

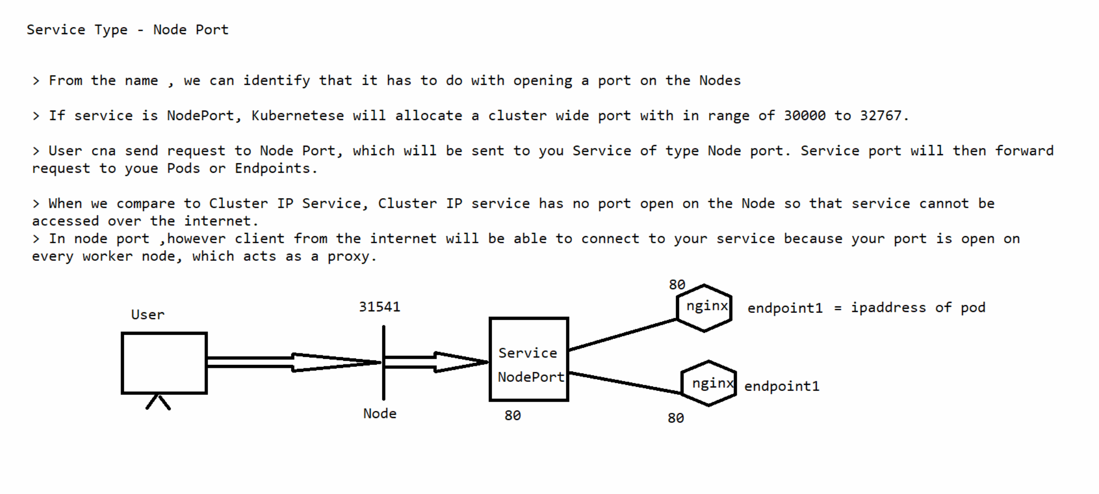
Inside the pod on the ubuntu container install curl

```
apt-get update && apt-get install curl -y
```

```
curl serviceip:80
```

You will be able to see nginx message.

Node Port:
****



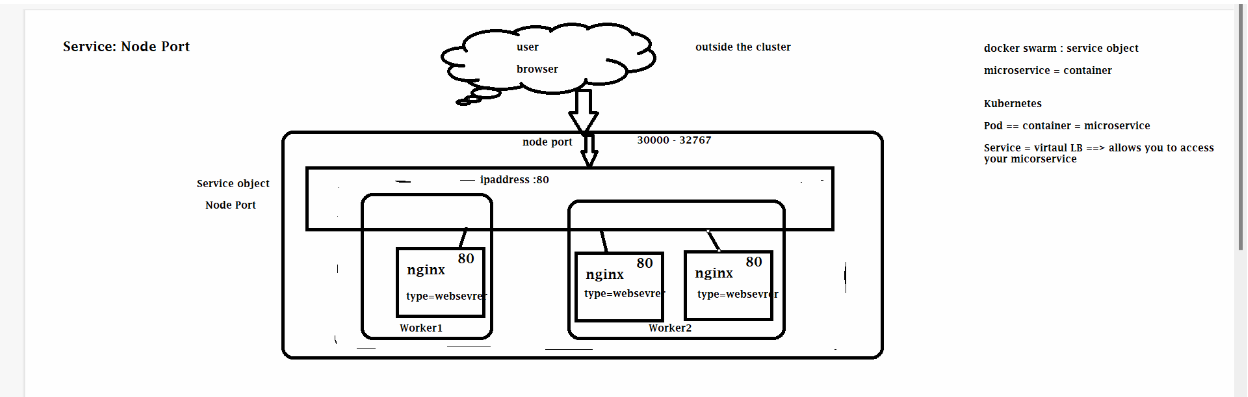
Service Type - Node Port

> From the name , we can identify that it has to do with opening a port on the Nodes

> If service is NodePort, Kubernetese will allocate a cluster wide port with in range of 30000 to 32767.

> User cna send request to Node Port, which will be sent to you Service of type Node port. Service port will then forward request to youe Pods or Endpoints.

> When we compare to Cluster IP Service, Cluster IP service has no port open on the Node so that service cannot be accessed over the internet.
> In node port ,however client from the internet will be able to connect to your service because your port is open on every worker node, which acts as a proxy.



Demo:

==================================

Delete all exisitng pod

kubectl delete pods --all

Create new Deployment and Nodeport Service

vim deployment.yml

apiVersion: apps/v1

```
kind: Deployment
metadata:
 name: mydep
spec:
 replicas: 3  # replicas= desired count of pods of same  image to b created
 selector: # query the cluster to see the current count of replicas
  matchLabels:
   type: webserver
 template: # pod specification for the replicas
  metadata:
   labels:
    type: webserver
  spec:
   containers:
    - name: c1
      image: leaddevops/kubeserve:v1
```

```
# kubectl create -f deployment.yml

# kubectl get all

# kubectl get pods -o wide
mydep-6b575bc547-nwzbv   1/1     Running   0          3m39s   192.168.22.11    worker1-kube   <none>           <none>
mydep-6b575bc547-qb9ch   1/1     Running   0          3m39s   192.168.127.10   worker2-kube   <none>           <none>
mydep-6b575bc547-xp5bt   1/1     Running   0          3m39s   192.168.127.11   worker2-kube   <none>           <none>
```

Step 2: Create service object of type node Port that will forward request to above created endpoints

```
---
apiVersion: v1
kind: Service
metadata:
 name: mysvc2
spec:
 type: NodePort  # type of service
 selector: # endpints selector
  type: webserver
 ports:
  - port: 80 # service port
    targetPort: 80 # container Image port
```

Create the service:

```
# kubectl create -f service1.yml
# kubectl get all
# kubectl get service
```

output:

```
service/mysvc2      NodePort    10.106.155.32   <none>       80:31830/TCP   5s
```

Here 31830 is the nodePort mapped service port

Endpoints: kubectl describe service mysvc2

```
192.168.127.10:80,
192.168.127.11:80,
192.168.22.11:80
```

Validate:

```
kubectl get all
```

```
kubectl get svc
```

```
service/mysvc     NodePort     10.105.31.132    80:32111/TCP
```

Copy the node port.

Go to browser, take worker node ipaddress and node port number

You will access the application.

****

Service 3: Load balancer (access the application on the Pod via the browser of your machine)

===

> it can be implemented only and only if kubernetes cluster is on the cloud

> with this service kubernetes will create a load balancer on GCP --> which will give an external IP --> this ip will be mapped to service ip:portnumber--> forward request to pods(endpoints)

Step 1: Create a GKE cluster and connect to it.

Step 2: Create a deployment for 3 replicas

```
vim deployment.yml

apiVersion: apps/v1
kind: Deployment
metadata:
 name: mydep
spec:
 replicas: 3  # replicas= desired count of pods of same  image to b created
 selector: # query the cluster to see the current count of replicas
  matchLabels:
   type: webserver
 template: # pod specification for the replicas
  metadata:
   labels:
    type: webserver
  spec:
   containers:
    - name: c1
      image: leaddevops/kubeserve:v1
```

Load Balancer:

---

apiVersion: v1

kind: Service

```
metadata:
 name: mysvc
spec:
 type: LoadBalancer
 ports:
  - targetPort: 80
    port: 80
 selector:
  type: webserver
```

```
kubectl create-f service3.yml
kubectl get svc
```

An external Ip will be generated, go to browser and paste the external Ip

You will see the application

****

INGRESS RESOURCE:

==

Create 2 Pods

```
kubectl run pod1 --image nginx
kubectl run pod2 --image nginx
kubectl get pods
```

Update each pod index.html file, so that different application is deployed on them

```
# kubectl exec -it pod1-6ffc6c4d6-r4sxw -- bash

# cd /usr/share/nginx/html

echo “This is Learners webpage for courses” > index.html
```

Exit out of the pod

Execute same steps on pod 2

```
# kubectl exec -it pod2-6fd7b6c4bb-f8rwx -- bash

# cd /usr/share/nginx/html

echo “This is Trainers webpage for Batches” > index.html
```

Exit out of the pod

```
NAME                   READY   STATUS    RESTARTS   AGE
pod1-6ffc6c4d6-r4sxw    1/1     Running   0          7m44s
pod2-6fd7b6c4bb-f8rwx   1/1     Running   0          7m3s
```

Create 2 services:

======================================

```
# kubectl get pods
```

Copy the name of each pod as shown below and create service using below commands

```
NAME                   READY   STATUS    RESTARTS   AGE
pod1-6ffc6c4d6-r4sxw    1/1     Running   0          7m44s
pod2-6fd7b6c4bb-f8rwx   1/1     Running   0          7m3s
```

Create 1st service

=========================

```
kubectl expose pod <podname> --name service1 --port=80 --target-port=80
```

For example:

```
kubectl expose pod pod1-6ffc6c4d6-9z7tw --name service1 --port=80
--target-port=80
```

Create 2nd service:

==

```
kubectl expose pod <podname> --name service2 --port=80 --target-port=80
```

```
kubectl expose pod pod2-6fd7b6c4bb-nrlw6 --name service2 --port=80
--target-port=80
```

```
kubectl get svc  ==> copy the ipaddress
```

```
service1    ClusterIP   10.8.9.241    <none>      80/TCP    3m34s
service2    ClusterIP   10.8.14.162   <none>      80/TCP    74s
```

Create a front end pod

===

```
kubectl run frontend-pod --image ubuntu --command -- sleep 36000
```

go inside frontend pod

```
kubectl exec -it <podname> -- bash
```

```
apt-get update && apt-get install curl nano -y
```

```
curl service1 ipaddress
```

```
curl service2 ipaddress
```

Exit from the frontend pod

Command to install Nginx Controller

```
=============================
```

https://github.com/kubernetes/ingress-nginx/blob/main/docs/deploy/index.md

Execute this below command in the cluster

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.2.1/deploy/static/provider/cloud/deploy.yaml
```

```
# kubectl get ingressclass
```

===

Create Ingress object:

==

vim ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:

  rules:
  - host: website01.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: website02.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service2
            port:
              number: 80
```

```
kubectl create -f ingress.yml
```

```
kubectl get ingress

kubectl describe ingress name-virtual-host-ingress

kubectl get ingressclass
```

Update ingress class in the yaml file

```
apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: name-virtual-host-ingress

spec:

  ingressClassName: nginx

  rules:

  - host: website01.example.com

    http:

      paths:

      - pathType: Prefix

        path: "/"

        backend:

          service:

            name: service1

            port:
              number: 80
  - host: website02.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service2
            port:
              number: 80
```

```
kubectl get ingressclass
```

Update YAML file and apply

```
kubectl apply -f ingress.yml
```

OR use this link to create ingress:

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/ingress.yml
kubectl get svc
```

```
kubectl get svc -n ingress-nginx
```

Copy the service details

Copy the load balancer Ip address..

```
ingress-nginx-controller           LoadBalancer   10.8.8.154    34.67.95.94
80:31524/TCP,443:30
```

OR

go to GCP--> burger menu -->networkservice --> load balancer click on it --frontend check for ip

Now go the frontend pod and update the hostdetails to ping the ingress service

```
kubectl exec -it frontend-pod -- bash
```

```
nano /etc/hosts
```

make an entry

```
## loadbalancer ip hoatname1 hostname2
```

```
34.135.243.240 website01.example.com website02.example.com
```

ctl x y enter

```
curl website01.example.com
```

```
curl website02.example.com
```

Now to access for laptop browser.. you need to update the hosts file on windows

Press the Windows key.

Type Notepad in the search field.

In the search results, right-click Notepad and select Run as administrator.

From Notepad, open the following file: c:\Windows\System32\Drivers\etc\hosts.

Make the necessary changes to the file.

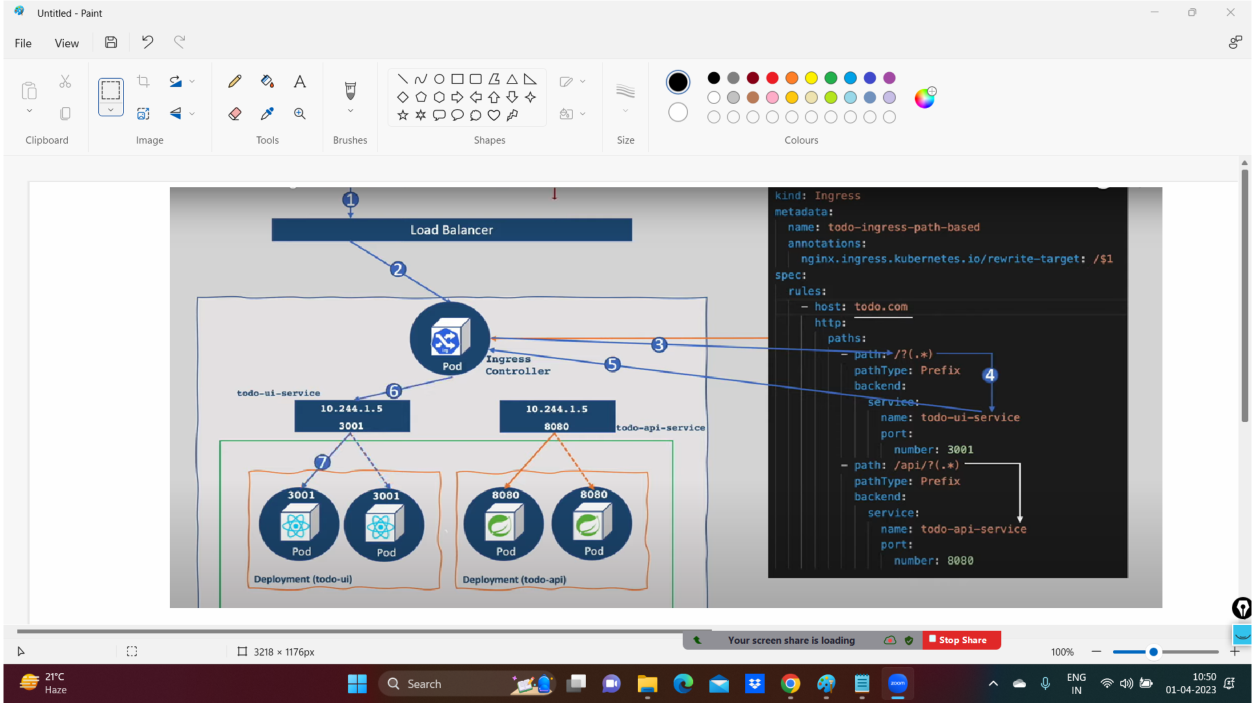
Select File > Save to save your changes.

Make an entry like this:

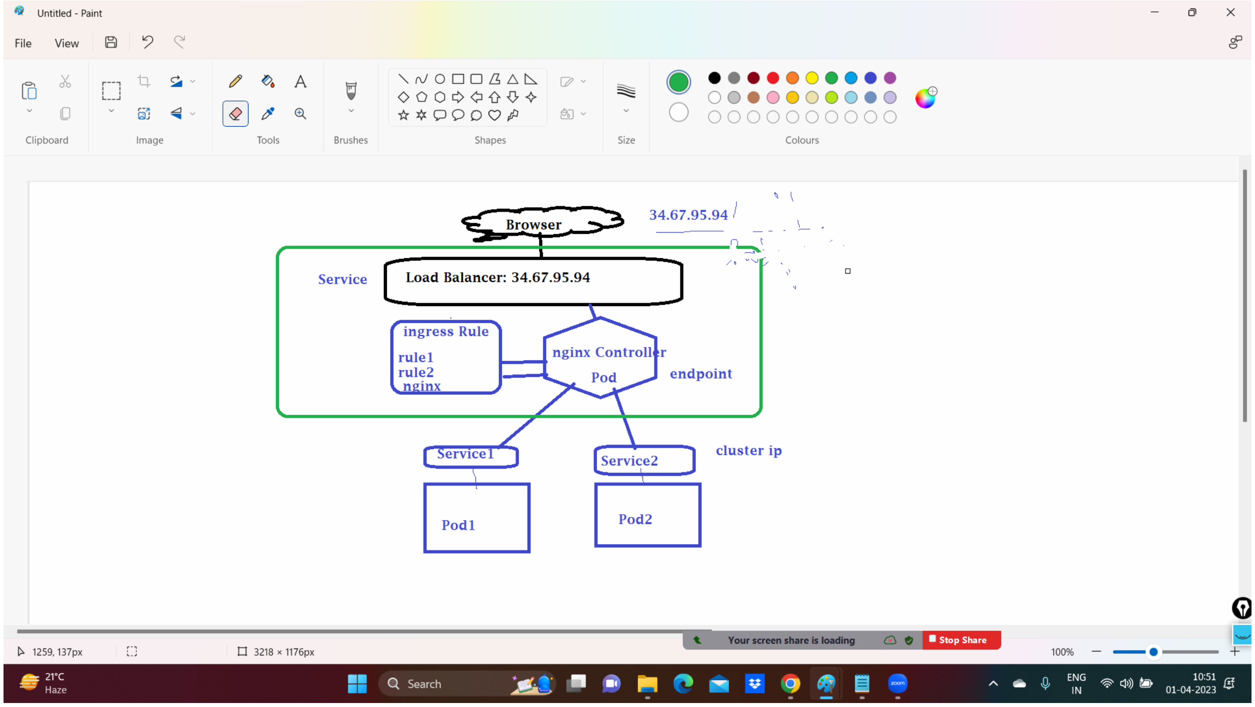
```
34.135.243.240 website01.example.com website02.example.com
```

save the file

go to browser and type website01.example.com



```
kind: Ingress
metadata:
  name: todo-ingress-path-based
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
    - host: todo.com
      http:
        paths:
          - path: /?(.*)
            pathType: Prefix
            backend:
              service:
                name: todo-ui-service
                port:
                  number: 3001
          - path: /api/?(.*)
            pathType: Prefix
            backend:
              service:
                name: todo-api-service
                port:
                  number: 8080
```



****

NameSpaces in kubernetes:
****

> provides a mechanism to partition your cluster so as to isolate a group of objects and controllers

> In k8s from the very begining we have a namespace called as DEFAULT

> Any object that we have created so far is created in DEFAULT name space

> but you can further create a new namespace that will allow you to isolate some specfic pods, replicas of an application

> resources of one namespace by default cannot talk to another namespace

> Once a resource has been created in a namespace,
in order get its details, we have to pass the namespace details

example: kubectl get pods ==> fetch details from namespace default

kubectl get pods -n teama ==> fetch details from namespace teama

-n = namespace

Demo:

Show the namespace already available in kubernetes

When we created the cluster a namespace called kube-system got created

in this namespace all the pods and services of kubernetes components are present

> apiserver, service
> coreDNS
> etcd
> scheduler

kubectl delete all -->delete pods form default and the kube system pods are safe as they are in different namespace.

demo2:

create our own namespace

kubectl create namespace teama

In nsmae space teama we have created

pod1 => httpd
traffic to this pod1 will be sent via service ==> service1teama

pod2 ==> ubuntu

we went inside ubuntu pod and we want to send traffic to service serviceteama

In k8s we can send traffic to another pod using service ipaddress or its name

execute curl servicename
<html><body><h1>It works!</h1></body></html>

==

kubectl create namespace teamb

In namespace teamb we have created

pod1teamb => ubuntu

we will go inside this pod and install curl and then ping service of namespace teama
==

1. k8s allows us to create pods/services in different namespaces

2. In Kubernetes, namespaces provides a mechanism for isolating groups of resources within a single cluster.

3. Names of resources need to be unique within a namespace

4. Namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc)

5. Namespaces are intended for use in environments with many users spread across multiple teams, or projects.

6. Whenever we create a cluster, it will create a namespace called as kube-system, this namespace includes all the k8s components pods & services

example: kubectl get pods -n kube-system

7. Whenever we create a cluster, it will also create a namespace called as default. So if you are the only user of the cluster then you cangoahead and create objects in the default namespace. However if we have multiple user we can isolate creation of resources in multiple namespaces

8. How does kubernetes know that it has to create objects in the defualt namespace only:

User can go and check the namespace details of cluster by going to

```
cd .kube

vim config
```

in this file under context detaiuls if there is no namespace then it means we are using default namespace

OR

use this command to see config details:

```
kubectl config view
```

9. Let's see now how we can create a namespace

```
# kubectl create namespace sonal
```

```
# kubectl get ns
```

Create a pod in your namespace

here:

-n = namespace

```
kubectl run pod1 --image nginx -n sonal
```

Create a service in the your namespace

```
kubectl expose pod pod1  --name service1 --port=80 --target-port=80 -n sonal
```

```
kubectl get all -n sonal
```

=====================================

10. If a user wants that the cluster should always create objects in the custom namespace, then you can update context details in the config file of kubernetes

Command is :

```
kubectl config set-context -- current --namespace=sonal
```

Change the context of cluster to default

Command is :

```
kubectl config set-context -- current --namespace=default
```

==

Config is created when user is setting up the cluster

Config file consists of default authroization and authentication details of the cluster

==

Network Policy in kubernetes:

==

1. By default pods are non-isolated, they can access traffic form any sources

2. Using services in k8s we are able to access pods from internet or from another pods

3. Network policies are set of rules which specifies if a group of pods are allowed to communicate with eachother or not.

4. We can isolate the pods by attaching to a network policy, if netpol is applioed, traffic from other ip/namespace or pod will not be allowed

5. there are 3 options using which we will apply network policy

> list of ipaddress
>
> namespace
>
> labels of pods

6. Network policy are of 2 type :

> allow traffic
>
> deny traffic

7. Two types of traffic:

> ingress => incoming traffic to pod

> egress   => outgoing traffic

8. default policy in k8s is Allow ALL

DEMO:

===

Deploy Spring Java Application & MongoDB Pods

kubectl apply -f
https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/policies/springboot-mongo-app.yml

Access the Sping Java Application & write some data to mongo db

kubectl get services springboot-app-svc

use the NodPort to access the springboot java in the browser

this proves that the You are able to access application from app pods

app is able to communicate to mongodb pods & write the data to it

Now lets block the request / traffic to spring app & mongo db using Network Policies

kubectl apply -f
https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/policies/deny-ingress-to-mongodb-and-springapp.yaml

kubectl get netpol

Now try to access application from browser it shoudn't respond

kubectl get services springboot-app-svc

use the NodPort to access the springboot java in the browser

This proves we successfully block all ingress (incoming) traffic to spring app

Now lets allows ingress(incoming) traffic to spring java app fromm all using Network Policies

kubectl apply -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/policies/allow-ingress-to-springapp-from-all.yaml

kubectl get services springboot-app-svc

use the NodPort to access the springboot java in the browser

This should allow the traffic to Spring java App & you should see the app in browser

But if you try to submit the data to DB it will not respond, we still need to allow traffic to mongodb

Now lets allow ingress(incoming) traffic to mongodb only from spring app pods using Network Policies

kubectl apply -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/policies/allow-ingress-to-mongodb-from-springapp.yaml

Now we should be able to write the data to mongodb from spring java app

==

CORE DNS information:

DNS for Services and Pods :

ØKubernetes creates DNS records for Services and Pods.

ØWe can contact Services with consistent DNS names instead of IP addresses.

ØKubernetes publishes information about Pods and Services in the Core DNS.

ØKubelet configures Pods' DNS i.e the /etc/resolv.conf file so that running containers can lookup Services by name rather than IP

ØServices defined in the cluster are assigned DNS names.

Ø By default, a Pod's /etc/resolv.conf file will contain DNS search list that includes the Pod's own namespace and the cluster's default domain(cluster.local)

ØWhenever in a POD a DNS query is made it may return different results based on the namespace of the Pod making it

ØDNS queries that don't specify a namespace are limited to the Pod's namespace.

ØAccess Services in other namespaces by specifying fully qualified domain name

ØFor example,

consider a Pod in a test namespace. A data Service is in the prod namespace.

A query for data returns no results, because it uses the Pod's test namespace.

A query for data.prod returns the intended result, because it specifies the namespace.

DNS queries may be expanded using the Pod's /etc/resolv.conf. Kubelet configures this file for each Pod.

DNS Records

What objects get DNS records?

1.Services

2.Pods

●

The following example gives detail of the supported DNS record types:

Services

my-svc.my-namespace.svc.cluster-domain.example - This resolves to the cluster IP of the Service.

Pods

In general a Pod has the following DNS resolution:

pod-ip-address.my-namespace.pod.cluster-domain.example.

For example,

if a Pod in the default namespace has the IP address 172.17.0.3, and the domain name for your cluster is cluster.local,
 then the Pod has a DNS name: 172-17-0-3.default.pod.cluster.local.
Pod's hostname and subdomain fields:
==================================
Currently when a Pod is created, its hostname (as observed from within the Pod) is the Pod's metadata.name value.
The Pod spec has an optional hostname field, which can be used to specify a different hostname.
When specified, it takes precedence over the Pod's name to be the hostname of the Pod
For example, given a Pod with spec.hostname set to "my-host", the Pod will have its hostname set to "my-host“
The Pod spec also has an optional subdomain field which can be used to indicate that the pod is part of sub-group of the namespace.
For example, a Pod with spec.hostname set to "foo", and spec.subdomain set to "bar", in namespace "my-namespace", will have its hostname set to "foo" and its fully qualified domain name (FQDN) set to "foo.bar.my-namespace.svc.cluster.local" (once more, as observed from within the Pod).

==

Practice questions:

https://github.com/Sonal0409/Container-Orchestration-using-Kubernetes/blob/main/CKAquestions_answers.txt

===

Jobs in kubernetes:

===

```
vim  job.yml
```

```
apiVersion: batch/v1

kind: Job

metadata:

 name: job1
```

```
spec:
 backoffLimit: 4
 template:
  spec:
   restartPolicy: Never
   containers:
    - name: c1
      image: perl:5.34.0
      command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
```

```
kubectl delete  jobs --all
kubectl create -f job.yml
kubectl get pods
kubectl logs job1-n959g
kubectl delete job job
```

==

Cron Job OR Scheduled Job

Cron syntax: min Hours date month day

Syntax Example: * * * * * or H/15 * * * * or * 8 15 1 *

```
vim cronjob.yml
```

=======================

```
apiVersion: batch/v1
kind: CronJob
```

```
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      backoffLimit: 4
      template:
        spec:
          restartPolicy: Never
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
```

```
# kubectl create -f cronjob.yml

# kubectl get cronjob

# kubectl get cronjob hello

# kubectl get jobs --watch

# kubectl get cronjob hello
```

You should see that the cron job hello successfully scheduled a job at the time specified in LAST SCHEDULE.

There are currently 0 active jobs, meaning that the job has completed or failed.

```
# kubectl delete cronjob hello
```

****

By default, cron job will keep record of last 3 pods log only

if you will give kubectlg et pods, only last 3 pods will be displayed created by each job every minute

if you want to keep more pods in the log, then you can set the successfull threashhold limit or a failure threashhold limit

for example:

```
vim cronjob2.yml
```

```
apiVersion: batch/v1

kind: CronJob

metadata:

  name: hello

spec:

  schedule: "* * * * *"

  successfulJobsHistoryLimit: 10
  failedJobsHistoryLimit: 10
  jobTemplate:
    spec:
      backoffLimit: 4
      template:
        spec:
          restartPolicy: Never
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
```

Here:

```
  successfulJobsHistoryLimit: 10   #number of successfull pod history to be kept
  failedJobsHistoryLimit: 10       # number of failed pod history to be kept
```

****

You can suspend the cron if needed for certain scenarios like for maintencne, add the suspend parameter in spec

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  successfulJobsHistoryLimit: 10
  failedJobsHistoryLimit: 10
  suspend: true
  jobTemplate:
    spec:
      backoffLimit: 4
      template:
        spec:
          restartPolicy: Never
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
```

In class Notes:
What is a task?

> get me logs of an application and send it to me on email --> STOP

activity which starts and completes

2 types of job in k8s

> RUN to completion --> simply called as Jobs

> Scheduled Job --> CronJob

RUN to completion Jobs:

In k8s there is a controller which is called as JOB

> when we create a JOB object--> create a pod --> create a container

> container will perform the task

> Once the task is completed successfully , the pod will shutdown or it status will be completed

> Once the task is completed successfully ==> POD will not get deleted automatically

> Pod will be available in the k8s as well as its logs will be availble, to check what task it performed or if there were any errors

> the pod will shutdown or it status will be completed, it will not use any resources of the k8s cluster

> suppose we created a JOb --> create a pod --> tasks starts executing-->but task was fails

> The status of the POD will be Failure and it will show the error --> pod is not deleted--> logs will be there

> Job will recreate a new POD

> Job will never try to restart the failed POD

> On the job specification we can mention a backoffLimit = 4, to control the number of pods that a job can recreate if a tasks fails

> Even after restarting for 4 times, if task is still failed, job status will become failed.

> If we delete the job, it will automatcially delete the pods that it has created.

> A job can generete one pod or multiple pods

> if your task needs multiple pods for completion, job can set a parameter in its specification called as completions

> if completions = 4 ==> job will create 4 pods , one after the other

> by default a job will create 1 pod only as its completion parmener is set to 1

> if a job is creating many pods and you want the pods to be created in parallel, then use the parameter parallelism = 2
> its default value is 1

****

Scheduled or Cron JOB

In linux we have the crontab file --> task of this file is to execute the mentioned task to be executed on the given scehdule

in kubernetes, cron job is a controller which will run your job on a specfic schedule

Create a cron job to run at 8AM -> cronjob will create a Job --> a job will create a pod--> a pod willc reate container and perform the task.

This can be called as scheduled or Cron job

==

Scheduling in Kubernetes:

==

1. **nodeName:**

```
# kubectl get nodes
```

Copy the name of the node on which we have to schedule the pods

```
vim deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mydep
spec:
 replicas: 3  # replicas= desired count of pods of same  image to b created
 selector: # query the cluster to see the current count of replicas
 matchLabels:
  type: webserver
template: # pod specification for the replicas
 metadata:
  labels:
   type: webserver
 spec:
  nodeName: worker1-kube
  containers:
   - name: c1
     image: leaddevops/kubeserve:v1
```

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

All pods will be scheduled on desired worker nodes.

2. NodeSelector

   ==

   In this first we will set the node label for worker1 and worker2

   ```
   # kubectl label node worker2-kube region=us-east-1
   ```

   Now add the parameter NodeSelector in the YAML file

```
apiVersion: apps/v1

kind: Deployment

metadata:

 name: mydep

spec:

 replicas: 3  # replicas= desired count of pods of same  image to b created

 selector: # query the cluster to see the current count of replicas

  matchLabels:

   type: webserver

 template: # pod specification for the replicas

  metadata:

   labels:

    type: webserver

  spec:

   nodeSelector:

    region: us-east-1

   containers:

    - name: c1

      image: leaddevops/kubeserve:v1
```

```
kubectl create -f deployment.yml

kubectl get pods -o wide
```

3. Taints & tolerations

   Effect : noSchedule

   Existing pods on the tainted node continue to run but no new pod will be scheduled on the tainted node.

   ```
   # kubectl taint node worker-1-kube color=red:NoExecute
   ```

Horizontal Pod Autoscaler:

===================================

Create a deployment:

```
apiVersion: apps/v1

kind: Deployment

metadata:

 name: nginx

spec:

 replicas: 1

 selector:

  matchLabels:
```

```
   app: nginx
 template:
  metadata:
   name: nginxpod
   labels:
    app: nginx
  spec:
   containers:
    - image: nginx
      name: nginx
      resources:
       limits:
        cpu: 10m
```

Create a service:

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

Create HPA:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 5
```

Create load generator pod:

```
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never --
/bin/sh -c "while sleep 0.01; do wget -q -O- http://10.8.4.194:80; done"
```

==

Volumes: Storage in kubernetes

===

1. Volumes in kubernetes are implemented using Persistent Volume object and persistent volume claim object.

2. Storage in kubernetes can be :

> on the hosts of the cluster

> outside the cluster i.e. on the cloud like AWS(elastic block store) or GCP(Persistent disk or filestore) or Azure (azuredisk), NFS server

3.Pods in k8s can preserve the data in the cluster(on the host where the pod is scheduled) or outside the cluster

4. Persistent Volume object in k8s can be created manually and dynamically

5. In order to create storage dynamically, we have to use another object called as storage class

Demo:

volume type is hostpath

this is volume which gets created on the host machine itself (within the cluster)

its access mode is read write once

Step1: you will create object persistent volume

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/storage/pv.yml
```

```
kubectl get pv
```

Step 2: you will create persistent volume claim

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/storage/pvc.yml
```

```
kubectl get pvc
```

```
kubectl get pv
```

Step 3: we will define volume in pod spec

Step 4: mount the volume on the container

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/storage/pod-pvc.yml
```

```
kubectl get pods
```

```
kubectl describe pod pod-pvc | less
```

```
kubectl exec -it pod-pvc -- bash
```

Demo2 : dynamic volume provisioning

Create storage outside the cluster and it will be created automatically. The PV also will be created automatically

In this case, we will give permissions to kubernetes engine to goahead and create storage on the cloud

to do this, we use an object called storgae class

Step1: we will create a storage class wherein defining kubernetes to use provisioner kubernetes.io/gce-pd to create persistent disk on GCP

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/storage/DynamicStorage/sc.yml
```

```
  kubectl get sc
```

Step 2: Create a Persistent volume claim

in the claim we will write:

> storageClass : customName

> capacity: 10GB

> accessmode: rwo

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/storage/DynamicStorage/PD-pvc.yml
```

```
kubectl get pvc
```

```
kubectl get pv
```

Immediatly : storge of type PD of size 10GB will be created in GCP

A PV will also get created and bounded to the claim

Step 3: add the volume section to Pod spec with claim detail

Step 4: mount volume on containers

```
kubectl create -f https://raw.githubusercontent.com/Sonal0409/Container-Orchestration-using-Kubernetes/main/Day%203%20-%20Notes/Networking/storage/DynamicStorage/PD-pod.yml
```

CONFIGMAP:

```
==========================

# kubectl get configmap


 # kubectl create configmap dev-config --from-literal=app.mem=2048m


 # kubectl get configmap


 # kubectl get configmap dev-config -o yaml

 # vim dev.properties

app.env:dev

app.mem=2048m

app.properties=dev.env.url

:wq!


# kubectl create configmap dev-config1 --from-file=dev.properties

# kubectl get configmap
# kubectl get configmap dev-config1 -o yaml
```

Use configmap for a pod

```
vim pod-configmap.yml
```

```
kind: Pod
apiVersion: v1
metadata:
 name: pod-configmap
spec:
 containers:
  - image: nginx
    name: c1
    volumeMounts:
     - name: config-volume
       mountPath: /etc/config
 volumes:
  - name: config-volume
    configMap:
     name: dev-config1
 restartPolicy: Never
```

```
:wq!
```

```
# kubectl apply -f pod-configmap.yml
# kubectl exec -it pod-configmap bash
# cd /etc/config
```

you will find the dev.properties file and configurations

Edit the configMAP

```
kubectl edit configmap -n <namespace> <configMapName> -o yaml
```

This opens up a vim editor with the configmap in yaml format. Now simply

Secrets:
==============================

Create a secret:

```
apiVersion: v1
kind: Secret
metadata:
 name: secret2
type: opaque
```

```
data:
 username: YWRtaW4=
 password: YWRtaW5AMTIz
```

Create a pod and mount the secret

```
apiVersion: v1
kind: Pod
metadata:
 name: pod-secret
spec:
 containers:
  - image: nginx
    name: c1
    volumeMounts:
     - name: foo
       mountPath: "/etc/foo"
       readOnly: true
 volumes:
  - name: foo
    secret:
     secretName: secret2
```

Example 2:

======================

```
kind: Secret
apiVersion: v1
metadata:
   name: mysql
data:
   password: "YWRtaW4="
```

Create a deployment to use the secret as env variable:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:8
          name: mysql
          args:
            - "--default-authentication-plugin=mysql_native_password"
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                   name: mysql
                   key: password
```

==

Authentication and Authorization:

===

To manage a Kubernetes cluster and the applications running on it,

the kubectl binary or the Web UI are usually used. The kubectl tool call the API Server.

When a request is sent to the API Server, it first needs to be authenticated (to make sure the requestor is known by the system) before it’s authorized (to make sure the requestor is allowed to perform the action requested).

The authentication step is done through the use of authentication plugins. There are several plugins as different authentication mechanisms can be used:

Client certificates (the one we will talk about in this demo)

Bearer tokens

Authenticating proxy

HTTP basic auth

DEMO:

============

Lets say we have just set up a brand new Kubernetes cluster.

It will be used across multiple teams and we already have a team member, Dave from the development team,

who wants to start deploying and testing his new microservices application on it.

What are the High level steps we can do to get him access?

1. We will start by creating a namespace named development dedicated to the development team

2. Dave needs to deploy standard Kubernetes resources like pods, deployments and services.

He will then be provided the right to create, list, update, get and delete Pod,Deployments and Services resources only.

Additional rights could be provided later on if needed.

We will ensure those rights are limited to the development namespace.

3. As a pre-requiste Dave needs to have kubectl installed on his server, and he also needs openssl as

he will generate a private key and a certificate sign-in request.

create a new Ubuntu Server for Dave and install kubectl, docker on it. Execute below steps:

```
## Install Docker

sudo wget https://raw.githubusercontent.com/lerndevops/labs/master/scripts/installDocker.sh -P /tmp

sudo chmod 755 /tmp/installDocker.sh

sudo bash /tmp/installDocker.sh

sudo systemctl restart docker
```

```
## Install kubeadm,kubelet,kubectl

sudo wget https://raw.githubusercontent.com/lerndevops/labs/master/scripts/installK8S-v1-23.sh -P /tmp
sudo chmod 755 /tmp/installK8S-v1-23.sh

sudo bash /tmp/installK8S-v1-23.sh
```

```
   71  docker -v

   72  kubeadm version -o short

   73  kubelet --version

   74  kubectl version --short --client
```

==

Lets us start the demo and create a user Dave, gives access to kubernetes cluster :

This is all part of Authentication .

We will use the method of authentication as : X509 Client Certs -> Client certificate authentication

On the main Kubernetes Master:

1. Create a namespace by using the following command:

```
# kubectl create namespace dev
```

2. Create a directory dev

```
mkdir dev
```

```
cd dev
```

3. Generating an RSA private key and certificate requests for Dave

First we need to generate a private rsa key and a CSR. The private key can easily be created with this command:

```
# sudo openssl genrsa -out dave.key 2048
```

4. Use the following command to generate certificate requests:

```
# sudo openssl req -new -key dave.key -out dave.csr
```

Common Name (CN) field: this will be used to identify him against the API Server ==> dave

Uses the group name in the Organisation (O) field: this will be used to identify the group against the API Server ==> dev

5. Once the .csr file is created, Dave needs to send it to us (admins) so we can sign it using the cluster Certificate Authority.

Run the following command to link an identity to a private key using a digital signature.

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

6. The following openssl command shows the certificate has been signed by the Kubernetes cluster CA (Issuer part),

the subject contains dave in the CN (CommonName) field and dev in the O (Organisation) field as Dave specified when creating the .csr file.

```
# openssl x509 -in ./dave.crt -noout -text | less
```

==

7. Building a Kube Config for Dave

Setting credentials to the user

Set credentials to dave:

```
# kubectl config set-credentials dave --client-certificate=/root/dev/dave.crt --client-key=/root/dev/dave.key
```

Set context to dave:

```
# kubectl config set-context dave-context --cluster=kubernetes --namespace=dev
--user=dave
```

Run the following command to display current contexts:

```
# kubectl config get-contexts
```

Everything is set up.

We now have to send Dave the information he needs to configure his local kubectl client to communicate with our cluster.

==

Copy the config file from the master node in the home directory to the new node(dave).

On master Machine

```
cd ..
```

```
cat .kube/config
```

Paste the copied config file into the client machine in root directory iteself.

```
vi myconf
```

copy the master config file contents to this file

In the Dave's node

create a directory with name as dev

```
mkdir dev
```

```
cd dev
```

Copy the crt and key files from the master node to the dave's node in the /role directory.

keep the filename same as on master node

```
vim dave.crt
```

```
vim dave.key
```

Locate the home directory.

```
cd ..
```

```
kubectl get pods --kubeconfig=myconf
```

```
kubectl config get-contexts
```

```
kubectl config use-context dev-user-context
```

===

By creating a certificate, we allow Dave to authenticate against the API Server, but we did not specify any rights so he will not be able to do many things

We will change that and give him the rights to create, get, update, list and delete Deployment and Service resources in the dev namespace.

In a nutshell: A Role (the same applies to a ClusterRole) contains a list of rules. Each rule defines some actions that can be performed (eg: list, get, watch, …) against a list of resources (eg: Pod, Service, Secret) within apiGroups (eg: core, apps/v1, …).

While a Role defines rights for a specific namespace, the scope of a ClusterRole is the entire cluster.

8. Creation of a Role

Let’s first create a Role resource with the following specification:

```
kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

   namespace: dev

   name: dave-role

rules:

- apiGroups: ["", "extensions", "apps"]

  resources: ["deployments", "pods", "services"]

  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

```
kubectl create -f role.yml
```

```
kubectl get roles -n dev
```

For your information:

==================

Pods and Services resources belongs to the core API group (value of the apiGroups key is the empty string), whereas Deployments resources belongs to the apps API group.

For those 2 apiGroups, we defined the list of resources and the actions that should be authorized on those ones.

9. Creation of a RoleBinding

The purpose of a RoleBinding is to link a Role (list of authorized actions) and a user or a group.

In order for Dave to have the rights specified in the above Role, we need to bind him to this Role.

We will use the following RoleBinding resource for this purpose:

```
kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

 name: role-dave

 namespace: dev

subjects:

- kind: User

  name: dave

  apiGroup: ""

roleRef:

  kind: Role

  name: dave-role

  apiGroup: ""
```

```
kubectl create -f rolebinding.yml
```

```
kubectl get rolebinding -n dev
```

This RoleBinding links:

A subject: our user Dave.

A role: the one named dev that allows to create/get/update/list/delete the Deployment and Service resources that we defined above.

For your information:

=============================

as Dave belongs to the dev group, we could use the following RoleBinding in order to bind the previous Role with the group instead of with an individual user. Remember: the group information is provided in the Organisation (O) field within the certificate that is sent with each request.

```
kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

 name: dev

 namespace: development

subjects:

- kind: Group

  name: dev

  apiGroup: rbac.authorization.k8s.io

roleRef:

 kind: Role

 name: dev

 apiGroup: rbac.authorization.k8s.io
```

Run the following commands to verify roles we have generated:

```
kubectl get pods --kubeconfig=myconf
```

```
kubectl create deployment test --image=docker.io/httpd -n dev --kubeconfig=myconf
```

```
kubectl get pods --kubeconfig=myconf
```

```
kubectl get deployment --kubeconfig=myconf
```

The worker node can create, update, remove, and list pods, services, and deployments after using the master config settings.

===

Helm, The Kubernetes package manager

`Helm` **helps you manage Kubernetes applications with** `Helm Charts` **which helps you define, install, and upgrade even the most complex Kubernetes application.**

The main building block of Helm based deployments are Helm Charts: these charts describe a configurable set of dynamically generated Kubernetes resources.

The charts can either be stored locally or fetched from remote chart repositories.

The Basic Architecture / Helm Version 3

Helm 3 is a single-service architecture. One executable is responsible for implementing Helm. There is no client/server split, nor is the core processing logic distributed among components.

Implementation of Helm 3 is a single command-line client with no in-cluster server or controller. This tool exposes command-line operations, and unilaterally handles the package management process.

some key words to understand in helm

Chart

A Chart is a Helm package. It contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. Think of it like the Kubernetes equivalent of a Homebrew formula, an Apt dpkg, or a Yum RPM file.

Repository

A Repository is the place where charts can be collected and shared. It's like Perl's CPAN archive or the Fedora Package Database, but for Kubernetes packages.

Release

A Release is an instance of a chart running in a Kubernetes cluster. One chart can often be installed many times into the same cluster. And each time it is installed, a new release is created. Consider a MySQL chart. If you want two databases running in your cluster, you can install that chart twice. Each one will have its own release, which will in turn have its own release name.

Install Helm version3

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
```

```
chmod 700 get_helm.sh
```

```
./get_helm.sh
```

```
helm version --short
```

```
v3.0.2+g19e47ee
```

DEMO:

==

helm repo add bitnami https://charts.bitnami.com/bitnami

helm install my-release bitnami/jenkins

helm list

#Wait for 2 mins to see Jenkins UP

kubectl get deploy

kubectl get pods

kubectl get svc

Get username as : user

Get password by executing the command:

echo Password: $(kubectl get secret --namespace default my-release1-jenkins -o jsonpath="{.data.jenkins-password}" | base64 -d)

===

Kubernetes Dashboard & Service Account

===

execute below commands:

```
# kubectl apply -f https://raw.githubusercontent.com/Sonal0409/educka/master/dashboard/dashboard-insecure-v2.4.0.yml
```

```
# kubectl get pods -n kubernetes-dashboard
```

Access from browser using node port

Ipaddress:30009

==