

## Lec - 01 :- Complete theory of Kubernetes.

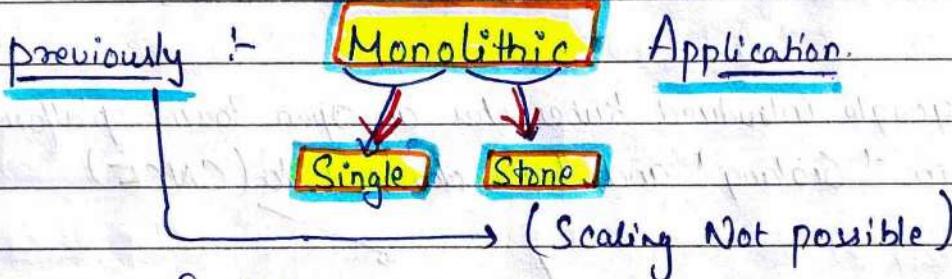


↓  
(K8s)

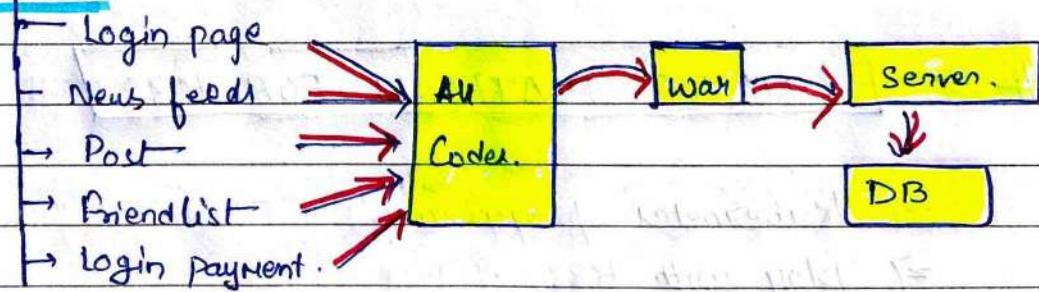
7 lines in the logo.  
( project 7 ).

- ↳ Container Management tool
- ↳ High Availability.
- ↳ Scalability

Kubernetes is an open-source Container-Orchestration system for automating Computer application deployment, Scaling, and Management.



Facebook,

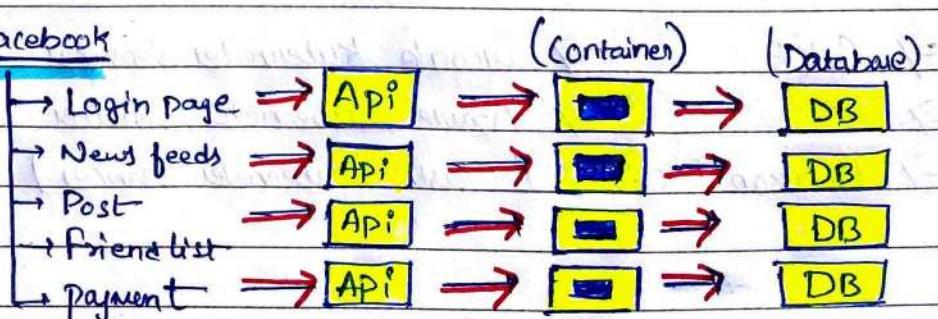


Now :-

Micro Services.

1 → (works independently.)

Facebook



⇒ Kubernetes is an Open-Source Container Management tool which automates Container deployment, Container scaling & Load Balancing.

⇒ It schedules, runs and manages isolated containers which are running on virtual / physical / cloud machines.

⇒ All top cloud providers support.

### History

⇒ Google developed an internal system called 'borg' (later named as omega) to deploy and manage thousand google applications and services on their clusters.

⇒ In 2014, Google introduced Kubernetes an open source platform written in 'GoLang' and later donated to (CNCF)

(Cloud Native Computing Foundation)

### # ⇒ ONLINE PLATFORM FOR K8S

⇒ Kubernetes playground.

⇒ play with K8s

⇒ play with kubennets classroom.

### # ⇒ CLOUD BASED K8S SERVICES

⇒ GKE → Google Kubernetes Service.

⇒ AKS → Azure Kubernetes Service

⇒ Amazon EKS → (Blastic Kubernetes Service)

## # 7 KUBERNETS INSTALLATION TOOL

- ⇒ Minicube.
- ⇒ Kubeadm.

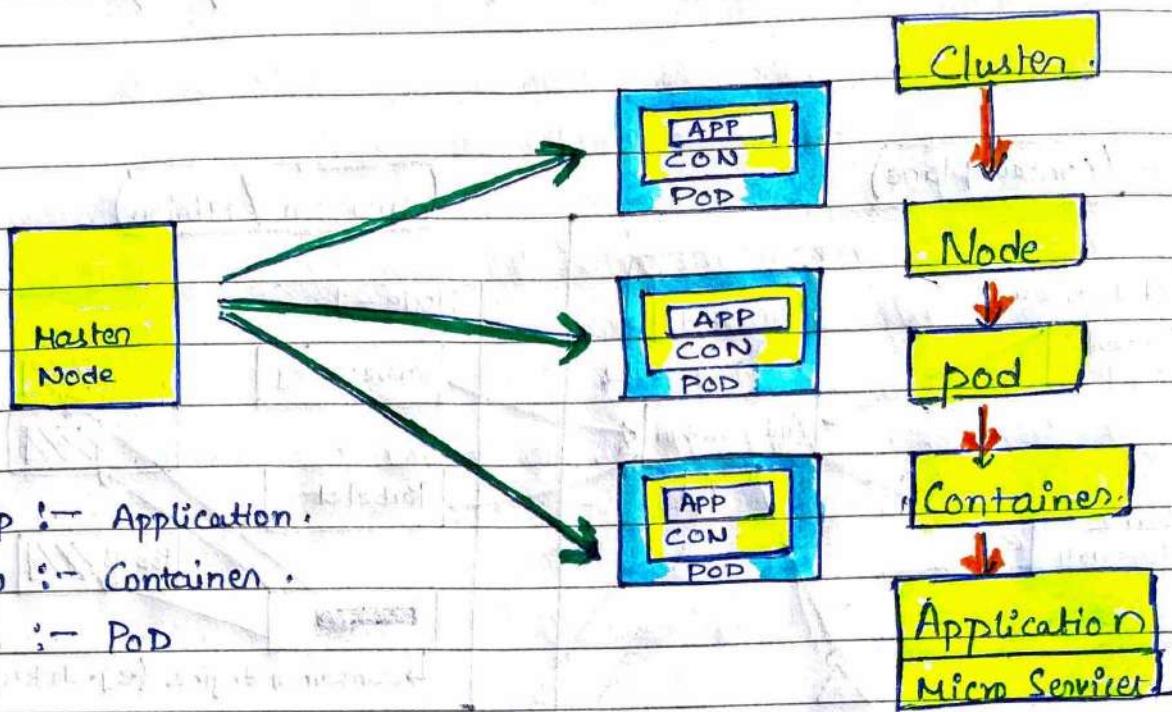
## Problems with Scaling up the Containers.

- ⇒ Containers Can Not Communicate with Each other.
- ⇒ Auto-scaling and Load Balancing was Not possible.
- ⇒ Containers had to be managed Carefully.

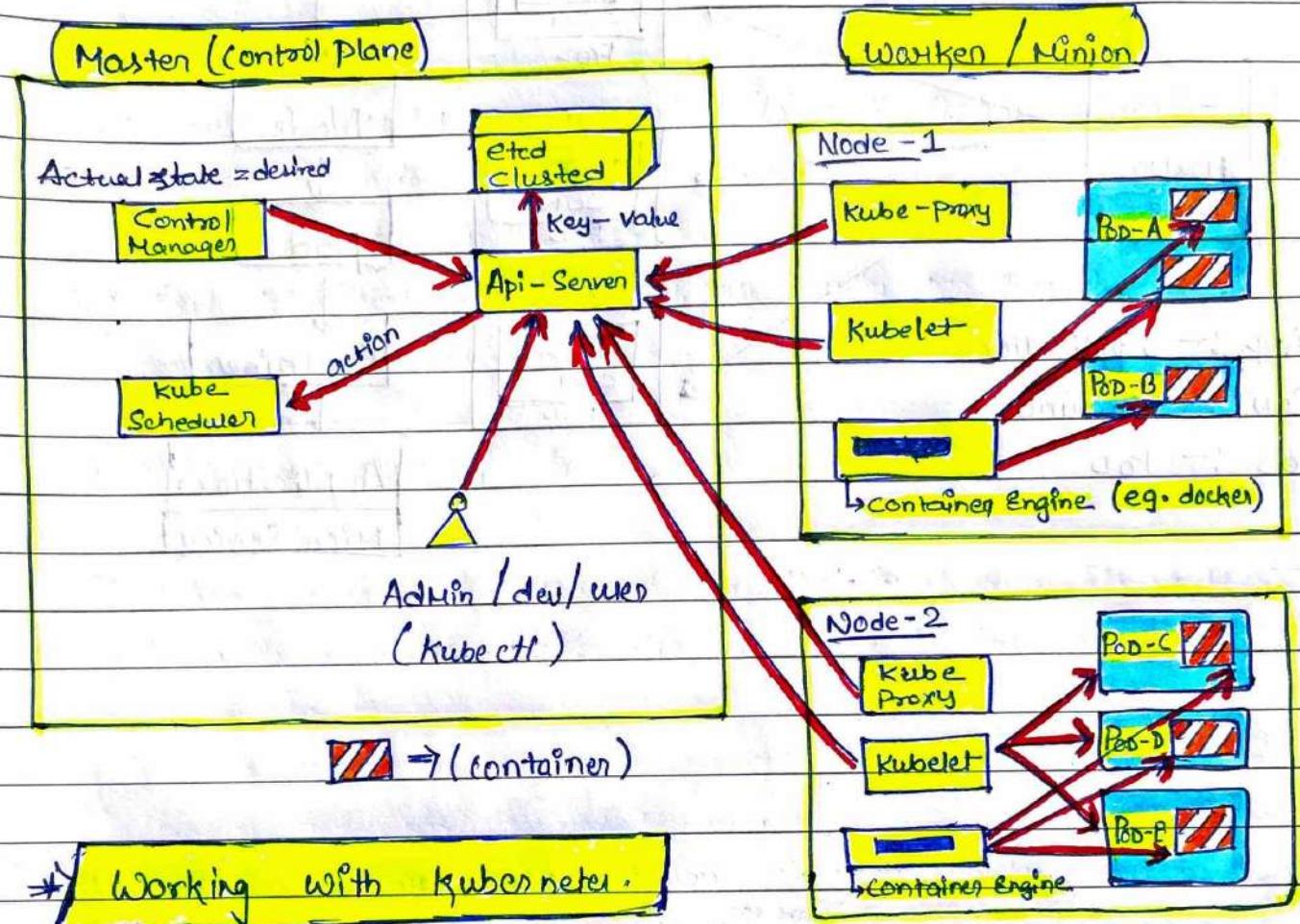
## Features of Kubernetes (JSON) (YAML)

- ⇒ Orchestration (clustering of any no of Containers running on different v/w)  
(vertical & horizontal) Prefend.
- ⇒ Auto Scaling      ⇒ Auto - healing.
- ⇒ Load Balancing      ⇒ platform independent (cloud / virtual / physical)
- ⇒ Fault Tolerance (Node / pod failure)
- ⇒ Rollback (going back to previous version)
- ⇒ Health Monitoring of Containers.
- ⇒ Batch Execution (one time Sequential, Parallel)

| FEATURES                               | KUBERNETES   | DOCKER SWARM  |
|--|--|---|
| Installation and Cluster Configuration | Complicated and time consuming   | Fast and Easy.  |
| Supports                               | k8s Can work with almost all container types like Rocket, Docker Container | Work with docker only.  |
| GUI                                    | GUI Available  | GUI Not Available.  |
| Data Volumes                           | Only Shared with Containers in Same pod.                                   | Can be shared with any other container.                               |
| Updates & Rollback                     | Process Scheduling to Maintain Services while updating                     | progressive update & Service health Monitoring throughout the update. |
| Auto Scaling                           | Support Vertical and Horizontal Auto scaling                               | Not Support Auto Scaling.   |
| Logging and Monitoring                 | Inbuilt tool present for Monitoring  | Used 3rd party tools like Splunk.                                     |



## Lec-02 Architecture of Kubernetes & its demo.

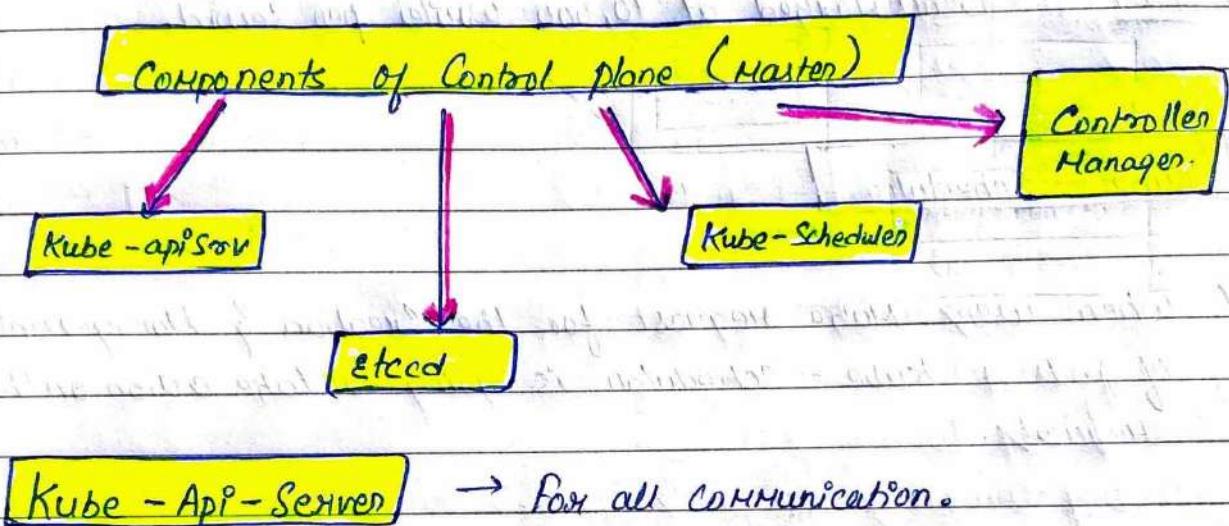


- ⇒ We Create Manifest (.yaml)
- ⇒ Apply this to Cluster (to Master) to bring into desired-state
- ⇒ pod runs on node, which is controlled by Master.

### Role of Master Node.

- ⇒ Kubernetes cluster contains containers running on Bare Metal / VM instances / cloud instances. / all mix.
- ⇒ Kubernetes designates one or more of these as Master and all others as Worker.

- ⇒ The Master is now going to run set of K8s processes - these process will ensure smooth functioning of cluster. These process are called "Control plane".
- ⇒ Can be Multi-Master for high availability.
- ⇒ Master runs Control plane to run cluster smoothly.



- ⇒ This api-server interacts directly with your user (ie we apply YAML or JSON manifest to Kube api-server)
- ⇒ This Kube-api-server is meant to scale automatically as per load.
- ⇒ Kube api-server is front-end of Control-plane.

### etcd

- ⇒ Stores metadata and status of cluster.
- ⇒ etcd is consistent and high availability store (key-value store)
- ⇒ Storehouse of truth for cluster state.  
(information about the state of cluster).

etcd has following features :-

1. Fully Replicated :- The entire state is available on every node in the cluster.
2. Secure :- Implements automatic TLS with optional client - certificate authentication.
3. Fast :- Benchmarked at 10,000 writes per second.

### Kube-Scheduler

- ⇒ When user make request for the creation & management of pods, Kube-Scheduler is going to take action on the requests.
- ⇒ Handles pod creation and management.
- ⇒ Kube-Scheduler match / assign any node Create and run pods.
- ⇒ A scheduler watcher for newly created pods that have no node assigned for every pod that the scheduler discover, the scheduler becomes responsible for finding best node for that pod to run on.
- ⇒ Scheduler gets the information for hardware configuration .yaml file and schedules the pods on nodes accordingly.

## Controller - Manager.

→ Make Sure actual state of Cluster Matcher to desired State.

Two possible choices for Controller Manager.

- ① if K8s on cloud, then it will be cloud - controller - Manager.
- ② if K8s on Non-cloud then it will be Kube - controller - Manager.

Components on Master that runs Controller.

Node - Controller → For checking the Cloud provider to determine if a Node has been detected in the cloud after it's stop responding.

Route - Controller → Responsible for setting up network, Router on your Cloud.

Service - Controller :— Responsible for load Balancers on your cloud against Services or type load Balancers.

Volume - Controller :— For Creating, attaching and Mounting volumes and interacting with the cloud provider to orchestrate volume.

## Nodes (Kubelet and Container Engine)

⇒ Node is going to run 3 important piece of software / process.

### Kubelet

- ⇒ Agent running on the node.
- ⇒ Listens to Kubernetes Master. (e.g. pod creation request)
- ⇒ use port **10255**
- ⇒ Sends Success and Fail report to Master.

### Container Engine (e.g. Docker)

- ⇒ Works with Kubelet
- ⇒ pulling images.
- ⇒ Start / stop containers.
- ⇒ Exposing Containers on ports specified in Manifest.

### Kube - Proxy

- ⇒ Assign IP to each pod.
- ⇒ It is required to assign (IP) address to pods (Dynamic IP)
- ⇒ Kube-proxy runs on each node & this make sure that each pod will get its own unique IP address.

These 3 Components Collectively Constitute "Node"

## POD

- ⇒ Smallest units in Kubernetes.
- ⇒ pod is a group of One or More Containers that are deployed together on the same host.
- ⇒ A cluster is a group of Nodes.
- ⇒ A cluster has at least one worker node and Master node.
- ⇒ In Kubernetes, the Control Unit is the pod, not Containers.
- ⇒ Consist of one or more tightly coupled Containers.
- ⇒ pod runs on node, which is controlled by Master.
- ⇒ Kubernetes only knows about pods  
(does not know about individual containers)
- ⇒ Cannot start Container without a pod.
- ⇒ One pod usually contains one controller.

### Multi-Container Pods.

- ⇒ Share access to memory space.
- ⇒ Connect to each other using (local host).
- ⇒ Share access to the same volume.
- ⇒ Containers within pods are - deployed in an all nothing manner.
- ⇒ Entire Pod is hosted on the same Node (Scheduler will decide about which Node).

### POD LIMITATION

- ⇒ No auto-healing or scaling.
- ⇒ Pod crashes.

### Higher level k8s Object

Replication Set :- Scaling and auto healing.

Deployment :- Versioning and Rollback  
Service :- Static IP and Networking  
Volume :- Non-Ephemeral Storage

### Important

Kubectl :- Single cloud.  
Kubeadm :- On-premise  
Kubefed :- Federated (Hybrid Cloud)

## Lec-03 :- K8s Setup Master and Node on Aws.

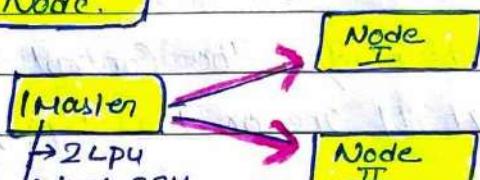
Login into Aws account → Launch 3 instance  
⇒ ubuntu 16.04 (t2. medium)

- ⇒ Now, using puttygen Create private key and save.
- ⇒ Access all the instance (1 master, 2 nodes)

### COMMON COMMANDS FOR Master Node.

```
$ #> sudo su  
$ #> apt-get update.  
$ #> apt-get install apt-transport-https.
```

```
$ #> apt install docker.io -y  
$ #> docker --version.  
$ #> systemctl start docker  
$ #> systemctl enable docker.
```



( This https is needed for intra cluster communication particularly from control plane to individual node )

Setup Open GPG key this required for intra cluster communication it will be added to source key on this node ie . when k8s sends signed info to our host , it is going to accept those information because this open gpg key is present in the source key .

```
$ #> sudo curl -s https://packages.cloud.google.com/apt/ | sudo apt-key add.
```

```
$ #> vim /etc/apt/sources.list.d/kubernetes.list.
```

↳ deb http://apt.kubernetes.io/ kubernetes-xenial main

⇒ Save and exit .

## BOOTSTRAPPING THE MASTER NODE (IN MASTER)

`$#) kubeadm init`

You will get one long command started from "kubeadm join  
172.165.158.6443 .."

Copy this command and save to notepad.

Create both .kube and its parent directories (-p)

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

Copy Configuration to kube directory (in config file)

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

Provide permission for config file.

`$#) sudo chown $(id -u):$(id -g) $HOME/.kube/config`

Deploy flannel node network for its repository path Flannel is going to place a binary in each node.

`$#) sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/documentation/kube-flannel.yaml`

`$#) sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/documentation/k8s-manifests/kube-flannel-rbac.yaml`

\*> Configure Worker Node :-> paste long command in both the nodes

\*> Go to Master Node :> `$#) kubectl get nodes`

(it will provide the node status)

## Lec-04 - Kubernetes in detail.

### Kubernetes Objects.

- ⇒ Kubernetes uses objects to represent the state of the your cluster.
- ⇒ What Containerized applications are running (and on which node)
- ⇒ The policies around how those applications behave, such as restart policies, upgrades and fault tolerance.
- ⇒ Once you Create the object, the Kubernetes system will constantly work to ensure that object exist and maintain's cluster's desired state.
- ⇒ Every Kubernetes object includes two nested fields that govern the object Config the Object Spec and the object status.
- ⇒ The Spec, which we provide, describes your desired state for the object - the characteristics that you want the object to have.
- ⇒ The status describes the actual state of the object and is updated and updated by the Kubernetes system.
- ⇒ All objects are identified by a unique name and a UID.

### The Basic Kubernetes Objects include.

- 1.7 Pod
- 2.7 Service
- 3.7 Volume
- 4.7 Namespace
- 5.7 Replicates
- 6.7 Secrets
- 7.7 Configmaps
- 8.7 Deployment
- 9.7 Jobs
- 10.7 Daemonsets

## Relationship between these objects.

- ⇒ pod Manager Containers.
- ⇒ Replicaset Managed pods.
- ⇒ Service expose pod processes to the Outside world.
- ⇒ Configmaps and Secrets helps you configure pods.

## Kubernetes Object.

- ⇒ it represents as JSON or YAML file.
- ⇒ you Create them and then push them to the Kubernetes API with kubectl.

## State of the Object

- |                 |                          |
|-----------------|--------------------------|
| ⇒ Replica (2/2) | ⇒ image (Tomcat /Ubuntu) |
| ⇒ None          | ⇒ part                   |
| ⇒ Volume        | ⇒ startup                |
|                 | ⇒ detached (default)     |

## Kubernetes Objects Management.

The kubectl command line tool supports several different ways to create and manage Kubernetes objects.

| Management Technique      | Operate ON                  | Recommend Env      |
|---------------------------|-----------------------------|--------------------|
| Imperative Commands       | Live Objects                | Deployment Project |
| Declarative object Config | Individual files (YML/JSON) | Production         |

Declarative is about describing what you are trying to achieve, without instructing how to do it.

Imperative, explicitly tells "how to accomplish it"

## Fundamental of pods

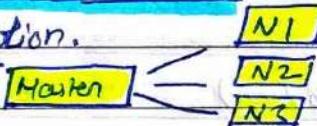
- ⇒ When a pod gets created, it is scheduled to run on a node in your cluster.
- ⇒ The pod remains on that node until the process is terminated or the pod object is deleted, the pod is evicted for lack of resource or the node fails.
- ⇒ If a pod is scheduled to a node that fails, or if the scheduling operation itself fails, the pod is deleted.
- ⇒ If a node dies, the pods scheduled to that node are scheduled for deletion after a timeout period.
- ⇒ A given Pod (uid) is not "rescheduled" to a new node, instead it will be replaced by an identical pod, with even the same name if desired, but with a new UID.
- ⇒ Volume in a pod will exist as long as that pod (with that UID) exists. If that pod is deleted for any reason, volume is also destroyed and created again on new pod.
- ⇒ A Controller can Create and Manage multiple pods, handling replication, rollout and providing self-healing capabilities.

## Kubernetes Configuration.

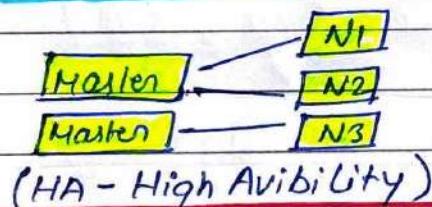
All in One Single node installation.

↳ 2 vcpu (For practice)

Single - Node etcd, Single Master, and Multi - worker installation.



Single node etcd, ~~Multiple~~ Multi Master and Multiworker install



with all in - one, all the master and worker Components are installed on a Single node. This is very useful for learning, development and testing. This type should not be used in production. Minikube is one such example, and we are going to explore it.

In this setup, we have a single Master node, which also runs a single - node etcd instance. Multiple worker nodes are Connected to the Master node.

In this step, we have multiple Master nodes which works in an HA mode, but we have a single - node etcd instance. Multiple worker nodes are Connected to the Master node.

Go to Aws account  $\Rightarrow$  Launch instance

$\Rightarrow$  Ubuntu 18.04  $\Rightarrow$  t2 medium (2 vcpu)

Now access Fc2 via putty  $\Rightarrow$  Login as "ubuntu"

$\Rightarrow$  sudo su

$\Rightarrow$  apt update & apt -y install docker.io

$\Rightarrow$  install kubectl

⇒ install Minikube.

⇒ apt install Contrack

⇒ Minikube Status.

⇒ Kubecli Version

⇒ minikube start

Then check the nodes details :-

⇒ kubectl get nodes. ↴ (output will be like)

| o/p | None            | Status | Roles  | Age  | Version |
|-----|-----------------|--------|--------|------|---------|
|     | ip.172.31.34.55 | Ready  | Master | 2min | v1.90.7 |

⇒ kubectl describe node 172.31.34.35 ↴ (Node ip)

⇒ Sample file to write Manifest file for kubernetes.

Kind: Pod

apiVersion: v1

Metadata:

name: testpod

Spec:

Containers:

- name: COO

image: ubuntu

Command: ["/bin/bash", "-c", "while true; do echo Hello-Saurav; sleep 5; done"]

restartPolicy: Never # Defaults to Always.

⇒ kubectl apply -f pod1.yaml (to apply the yaml file)

## Multi Container pod Environment

Kind: Pod

apiVersion: v1

Metadata:

name: testpod3

Spec:

Containers:

- name: CO0

image: ubuntu

command: ["#!/bin/bash", "-c", "while true; do echo Sawan; sleep 5; done"]

- name: CO1

image: ubuntu

command: ["#!/bin/bash", "-c", "while true; do echo Sawan; sleep 5; done"]

## Pod Environment Variables.

Kind: Pod

apiVersion: v1

Metadata:

name: environments

Spec:

Containers:

- name: CO0

image: ubuntu

Command: ["#!/bin/bash", "-c", "while true; do echo Sawan; sleep 5; done"]

Env: # List of environments.

- name: MYNAME

value: SAWAN

## Pod with PORTS

Kind : Pod

apiVersion : V1

Metadata :

name : testpod4

Spec :

Containers :

- name : COO

image :

ports :

- ContainerPort : 80

⇒ Kubectl apply -f pod2.yaml

⇒ Kubectl get pods → (To check the pod and Container details)

⇒ Kubectl describe pod testpod3 → (describe command to check details.)

⇒ Kubectl logs -f testpod3

↳ you will get error, need to specify Container if we need to check the logs of multiple containers.

⇒ Kubectl logs -f testpod3 -c COO → (Container name)

⇒ Kubectl logs -f testpod3 -c COO1 → (Container name)

⇒ Kubectl exec testpod3 it -c CO1 - /bin/bash → (Container name) → (Container name)

To login .

## Lec - 05 - Kubernetes in detail II

### Labels and Selections

- ⇒ Labels are the mechanism you use to organise Kubernetes objects.
- (None: Sawan)
- ⇒ A label is a key-value pair without any predefined meaning that can be attached to the objects.
- ⇒ Multiple labels can be added to a single object.
- ⇒ Labels are similar to tags in AWS or git where you use a name to quickly reference.
- ⇒ So you are free to choose labels as you need it to refer an Env which is used for dev or testing or prod, refer a prod group like Department A - Department B

### EXAMPLE OF LABELS.

Kind: Pod

apiVersion: v1

Metadata:

name: Sawanpod

Labels:

env: development

class: pods

Spec:

Containers:

- name: COO

Image: Ubuntu

Command: ["/bin/bash", "-c", "while true; do echo Hello-Sawan; sleep 5; done"]

⇒ Multiple labels we can add into a single object.

⇒ kubectl apply -f pods.yaml

⇒ kubectl get pods --show-labels  
└ (nodes)

↳ if we want to check the labels  
we can add nodes option.

→ Now, if you want to add a label to an existing pod.

⇒ kubectl label pods devhipod myname = bhupinder  
└ (object name)      ↴ (key)      ↴ (Selector)  
'object type'

⇒ kubectl get pods --show-labels

↳ now, list pods matching a label.

⇒ kubectl get pods -l env=development.

↳ now, give a list, where 'development' label is not present

⇒ kubectl get pods -l env!=development

↳ now if you want to delete pod using labels.

⇒ kubectl delete pod -l env!=development

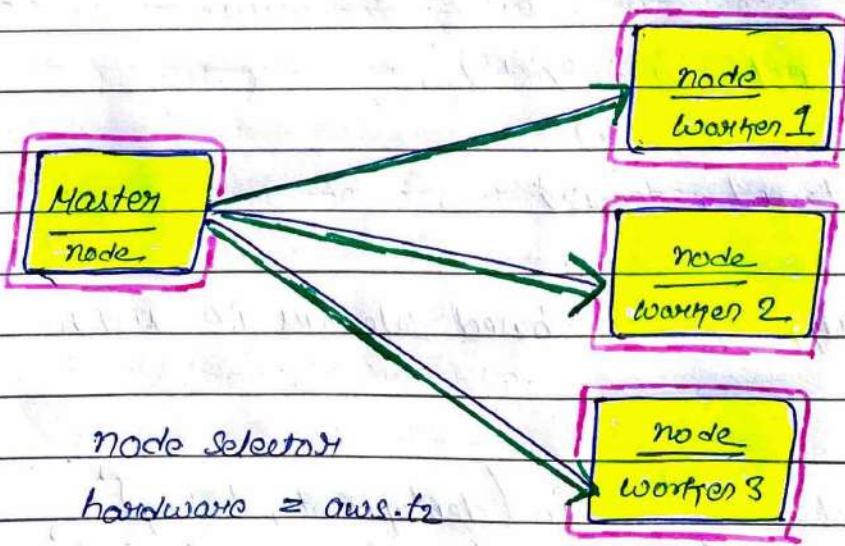
⇒ kubectl get pods.

## Label - Selectors

- ⇒ Unlike name / UIDs, labels do not provide uniqueness, as in general, we can expect many objects to carry the same label.
- ⇒ Once labels are attached to an object, we would need filters to narrow down and these are called as Label Selectors.
- ⇒ The API currently supports two types of Selectors — Equality based and set based.
- ⇒ A label selector can be made of multiple requirement which are comma-separated.
  - ⇒ Equality Based ( $=, !=$ )
    - name: sawan
    - class: Node1
    - project: development.
  - ⇒ Set Based : (in/notin and exists)
    - env in (production, dev)
    - env notin (team1, team2)
- ⇒ Kubernetes also supports set-based selectors i.e. match multiple values.
  - ⇒ kubectl get pods -l 'env in (development, testing)'
  - ⇒ kubectl get pods -l 'env notin (development, testing)'
  - ⇒ kubectl get pods -l class=pod1, myname=bhupinder

## Node - Selector

- ?) One way for selecting labels is to constrain the set of nodes onto which a pod can schedule i.e. you can tell a pod to only be able to run on particular nodes.
- ?) Generally such containers are unnecessary as the scheduler will automatically do a reasonable placement, but in certain circumstances we might need it
- ?) We can use labels to tag nodes
- ?) If the nodes are tagged, you can use the label selector to specify the pods run only of specific nodes.
- ?) First we give label to the node.
- ?) Then we need to use node selector to the pod configuration.



## Node Selector Example

Kind: Pod

apiVersion: v1

Metadata:

name: nodeTables

tables:

env: development

Spec:

Containers:

- name: COO

image: ubuntu

command: ["/bin/bash", "-c", "while true; do echo Hello-Sawan; sleep 5; done"]

nodeSelector:

hardware: t2-medium

## Scaling and Replication.

⇒ Kubernetes was designed to orchestrate multiple containers and replication.

⇒ Need for multiple containers / replication helps us with these

Reliability :- By having multiple versions of an application, you prevent problems if one or more fail.

Load Balancing :- Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node.

Scaling :- When load does become too much for the number of existing instances.

Kubernetes enables you to easily scale up your application by adding additional instances as needed.

Rolling update :- Updates to a service by replacing pods one by one.

### Replication Controller

- ⇒ A replication controller is an object that enables you to easily create multiple pods, then make sure that number of pods always exists.
- ⇒ If a pod created using RC will be automatically replaced if they die, crash, failed, or terminated.
- ⇒ RC is recommended if you just want to make sure 1 pod is always running, even after system reboots.
- ⇒ You can run the RC with 1 replica & the RC will make sure the pod is always running.

Kind : Replication controller.

## Replication Controller

Kind: ReplicationController → this defines to create the object of  
apiVersion: v1

Replication type

Metadata:

name: myreplica

Spec:

replicas: 2 → this element defines the desired number of pods

Selector: → tells the controller which pods to watch/belong to RC

Myname: sawan → this must match the label.

template: → template element defines a template to launch a new pod

Metadata:

name: testpod0

labels: → Selector value needs to match the label values specified

Myname: sawan. in the pod template.

Spec:

Containers:

- name: con

image: ubuntu

command: [ "bin/bash", "-c", "while true; do echo hello-sawan; sleep 5; done" ]

\$#) kubectl apply -f myrc.yaml

\$#) kubectl get rc

↳ (replication controller)

| Name      | Desired | Current | Ready | Age |
|-----------|---------|---------|-------|-----|
| myreplica | 5       | 5       | 5     | 48s |

\$#) kubectl describe rc myreplica

\$#) kubectl get pods

↳ (name)

\$#> kubectl get pods --show-labels.  $\Rightarrow$  To check the labels on Pod

| Name          | Ready | Status  | Replicas | Age   | Labels          |
|---------------|-------|---------|----------|-------|-----------------|
| Myreplica-h12 | 1/1   | Running | 0        | 2M40S | myname = Sawan  |
| Myreplica-t79 | 1/1   | Running | 0        | 2M50S | myname = Sawn   |
| Myreplica-189 | 1/1   | Running | 0        | 4M40S | my name = Sawan |
| Myreplica-xyz | 1/1   | Running | 2        | 3H10S | my name = Sawan |

\$#> kubectl scale --replicas=8 rc -l myname = bhupinder.

$\hookrightarrow$  This will scale the myname

= bhupinder template to 8 pods.

\$#> kubectl get pods  $\Rightarrow$  to get the pod details.

\$#> kubectl scale --replicas=1 rc -l myname = bhupinder.

$\hookrightarrow$  This command will scale down the pods from 8 to 1.

\$#> kubectl get rc.

$\hookrightarrow$  To check the replication controller details.

## Replica Set

$\Rightarrow$  Replica Set is a next generation Replication Controller.

$\Rightarrow$  The Replication Controller only supports Equality based Selection whereas the replica set supports Set-based Selection i.e filtering according to set of values.

$\Rightarrow$  ReplicaSet rather than the Replication Controller is used by other objects like deployment.

## Example of Replica Set

Kind : Replicaset

apiVersion : apps/v1

Metadata :

name : myrs

Spec :

replicas : 2

Selector :

MatchExpressions:

- [key: myname, operator: In, values: [Pawan, SAWAN, SAWAN]]

- [key: env, operator: NotIn, values: [production]]

template :

Metadata :

name : testpod7

Labels :

Myname : Pawan

Spec :

Containers :

- name : COO

image : ubuntu

command : ["/bin/bash", "-c", "while true; do echo Hello-Sawan; sleep 5; done"]

\$#> kubectl apply -f myrs.yaml

\$#> kubectl get rs .

\$#> kubectl get pods .

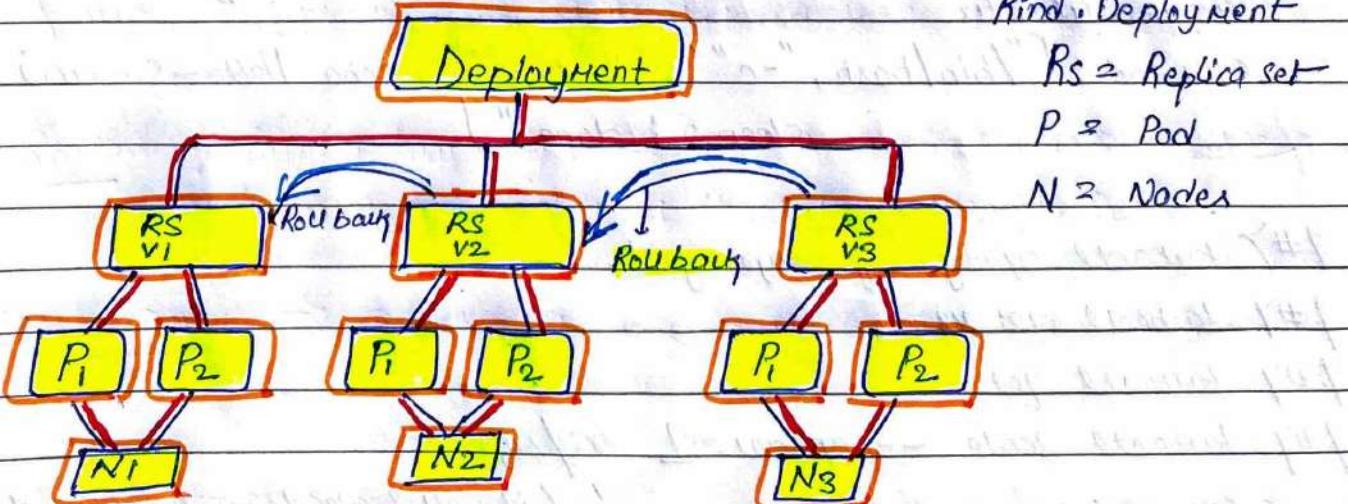
\$#> kubectl scale --replicas=1 rs/myrs

↳ (it will scale down from 2-1)

## Lec :- 06 - Deployment & Rollback

Replication Controller & Replica Set is not able to do update & rollback apps in the cluster.

- ⇒ A deployment object act as a supervisor for pods ; giving you fine - grained control over how and when a new pod is rolled out, updated or rolled back to a previous state.
- ⇒ When using deployment object , we first define the state of the app. then K8s Cluster Schedule Mentioned app instances onto specific individual Nodes.
- ⇒ K8s then Monitors , if the node hosting an instance goes down or pod is deleted the deployment controller replaces it.
- ⇒ This provides a self-healing mechanism to address machine failure or maintenance.



- ⇒ A deployment provides declarative update for pod & Replicated

The following are typical use cases of Deployments :-

- 1.) Create a deployment to rollout a Replicaset  $\rightarrow$  The replicaset creates pods in the background checks the status of the rollout to see if it succeeded or not.
- 2.) Declare the new state of the pods  $\rightarrow$  By updating the pod template spec of the deployment. A new replicaset is created and the Deployment Manager moves the pods from the old replicaset to the new one at a controlled rate. Each new replicaset updates the revision of the deployment.
- 3.) Rollback to an earlier deployment Revision  $\rightarrow$  if the current state of the deployment is not stable each rollback updates the revision of the deployment.
- 4.) Roll back to an earlier deployment Revision:— if the current state of the deployment is not stable each rollback updates the revision of the deployment.
- 5.) Scale up the deployment to facilitate more load.
- 6.) Pause the deployment to apply multiple fixes to its pod template spec and then resume it to start a new rollout.
- 7.) Cleanup older replicates that you don't need anymore.

⇒ if there are problems in the deployment, k8s will automatically rollback to the previous version, however you can also explicitly rollback to a specific revision, or in our case to Revision 1 (the original pod version)

⇒ you can rollback to a specific version by specifying it with --to-revision.

#⇒ For eg ⇒ kubectl rollout undo  Previous Version.

Type → deploy/mydeployment --to-revision=2

Note ⇒ That the name of the Replicaset is always formatted as  
[Deployment-name]  
[Random]

CMD ⇒ kubectl get deploy.

\* When you inspect the deployment in your cluster, the following field are displayed:

NAME ⇒ List the names of the deployments in the namespace.

READY ⇒ Display how many replicas of the application are available to your user if follows the pattern ready / desired.

UP-TO-DATE ⇒ Display the number of replicas that have been updated to achieve the desired state.

AVAILABLE ⇒ Displays how many replicas of the application are available to your user.

AGE ⇒ Display the amount of time that the Application has been Running.

Lab Login to AWS Management Console, Create one Ubuntu(t2-medium) instance. Take access using putty.

\$#> Sudo su

\$#> Sudo apt update && apt -y install docker.io

\$#> install kubectl

\$#> install minicube also

\$#> apt install Contrack

\$#> minicube start --VM-driver=none

\$#> minicube status

\$#> Kubectl version

\$#> Kubectl get nodes

### Deployment YAML Example

Kind: Deployment

apiVersion: apps/v1

Metadata:

name: mydeployment

Spec:

Replicas: 2

Selector:

MatchLabels:

name: deployment

Template:

Metadata:

name: testpod

Labels:

name: deployment

Spec:

Containers:

- name: eco

image: ubuntu

command: ["/bin/bash", "-c", "while true; do echo Hello-Saurav; sleep 5; done"]

⇒ To check deployment was created or not.

\$#> kubectl get deploy.

⇒ To check how deployment Created RS & pods

\$#> kubectl describe deploy mydeployments

\$#> kubectl get rs

⇒ To scale up or Scale down

\$#> kubectl scale --replica=1 deploy mydeployments.

Object type      ↳ object name

To check, what is running inside container.

\$#> kubectl logs -f < podname >

\$#> kubectl rollout status deployment mydeployments

\$#> kubectl rollout history deployment mydeployments

\$#> kubectl rollout undo deployment mydeployments.

**Failed Deployment** your deployment may get stuck trying to deploy its newest Replicaset with out ever completing.

This can occurs due to some following reasons.

i) Insufficient Quotas

ii) Readyness probe failures.

iii) Image pull errors

iv) insufficient permission

v) Limit Range

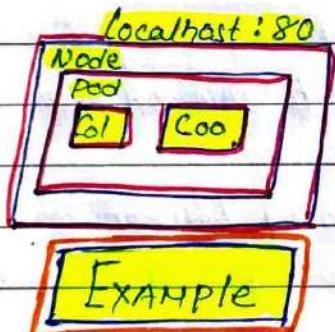
vi) Application runtime Misconfig.

## Lec-07 - Kubernetes Networking, Services, Nodeport, Volumes

Kubernetes Networking address four Concerns :

- i) Containers within a pod use networking to communicate via Loopback.
- ii) Cluster Networking provides communication between different pods.
- iii) The Service resource lets you expose an application running in pods to be reachable from outside your cluster.
- iv) You can also use services to publish services only for consumption inside your cluster.

Cluster : Container to Container communication on same pod happens through localhost within the container.



Example

Kind : Pod  
apiVersion : v1

Metadata :  
name : testpod

Spec :

Containers :

- name : coo

image : ubuntu

command : [ "/bin/bash", "-c", "while true; do echo Hello - sawan ; sleep 5; done" ]

- name : co1

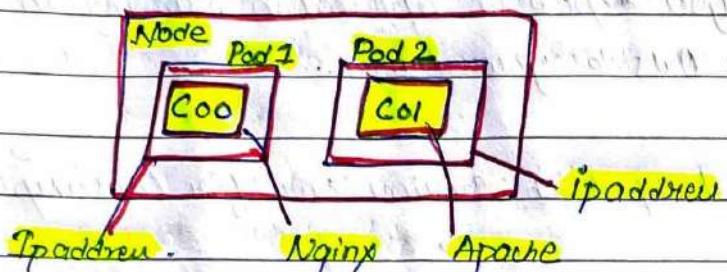
image : httpd

Ports :

- containerPort : 80

Now try to establish communication between two different pods within same node.

- ⇒ Pod to pod communication on worker nodes happens through pods
- ⇒ By default pod IP will not be accessible outside the node.



### EXAMPLE FOR Nginx

Kind: Pod

apiVersion: v1

Metadata:

name: testpod1

Spec:

Containers:

- name: CO1

image: nginx

ports:

- containerPort: 80

\$ #> kubectl apply -f pod2.yml

## Example for Apache

Kind : Pod

apiVersion : v1

Metadata :

name : testpod4

Spec :

Containers :

- name : pod3

image : httpd

Ports :

- ContainerPort : 80

\$#> kubectl apply -f pod3.yaml.

\$#> kubectl get pods -o wide      #> for all details like ip address

\$#> curl 172.\*.\*.1      → pod 1 ip address

\$#> curl 172.\*.\*.4      → pod 2 ip address.

## Object - Service :-

Each pod gets its own ip address, however in a deployment, the set of pods running in one moment in the time could be different from the set of pods running that application a moment later.

This leads to a problem : If some sets of pods (call them 'backend') provide functionality to other pods (call them frontend) inside your cluster, how do the frontends find out and keep track of which ip address. To connect, so that the frontend can use the backend part of the workload.

- ⇒ When using RC, pods are terminated and created during scaling or replication operations.
- ⇒ When using deployments, while updating the image version the pods are terminated and new pods take the place of other pods.
- ⇒ pods are very dynamic i.e. they come & go on the k8s cluster and on any of the available nodes & it would be difficult to access the pods as the pod ip changes once it's recreated.
- ⇒ Service object is an logical bridge between pods and end users, which provides virtual ip.
- ⇒ Services allows clients to reliably connect to the containers running in the pod using the VIP
- ⇒ The VIP is not an actual IP connected to a network interface, but its purpose is purely to forward traffic to one or more pods.
- ⇒ Kube proxy is the one which keeps the mapping between the VIP and the pods up-to-date. which queries the API server to learn about new services in the cluster.
- ⇒ Although each pod has a unique IP address, those IPs are not exposed outside the cluster.
- ⇒ Services helps to expose the VIP mapped to the pods & allows application to receive traffic.

- ⇒ Labels are used to select which are the pods to be put under a service.
- ⇒ Creating a Service will create an endpoint to access the pods / applications in it.
- ⇒ Services can be exposed in different ways by specifying a type in the Service Spec:

#⇒ Cluster IP

#⇒ NodePort

#⇒ Load Balancer

Loadbalancer :- Created by cloud providers that will route external traffic to every node on the Nodeport. (Eq F1B on AWS)

Headless :- Creates several endpoints that are used to produce DNS Records. Each DNS record is bound to a pod.

- ⇒ By default Service can run only between ports 30,000 - 32767
- ⇒ The set of pods targeted by a service is usually determined by a selector.

## Example of Deployment

Kind: Deployment

apiVersion: apps/v1

Metadata:

Name: mydeployments

Spec:

Replicas: 1

Selector:

MatchLabels:

name: deployment

template:

Metadata:

Name: testpod1

Labels:

Name: deployment

Spec:

Containers:

- name: c00

image: httpd

ports:

- containerPort: 80

\$#> kubectl apply -f deploymenthttpd.yaml

\$#> kubectl get pods

\$#> kubectl get pods -o wide

\$#> curl 172.16.12.\* → pod IP to check httpd is working or not.

## Example of Service

Kind : Service # Define to create Service type Object

Service

apiVersion : v1

Metadata :

name : demo service

Spec :

Ports :

- port : 80 # Container port exposed.

targetPort : 80 # pod Port

Selector :

name : deployment # Apply this service to any pod which has the

type : ClusterIP      specific label

→ specifies the service type i.e. Cluster IP or Nodeport

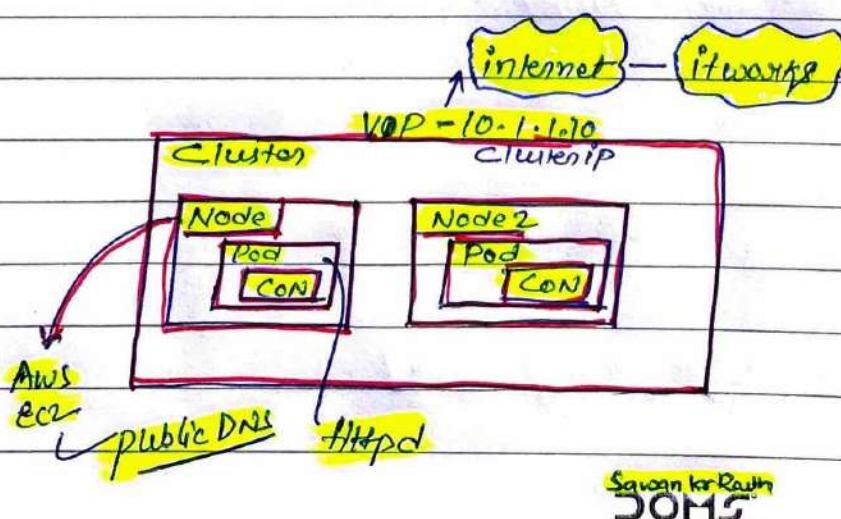
\$#> kubectl apply -f service.yaml

\$#> kubectl get svc

\$#> curl 10.231.10.\* → cluster ip

NODEPORT :- → makes a service accessible from outside the cluster.

→ Exposes the service on the same port of each selected node  
int the 'using' NAT



## Example of NODEPORT

```
Kind: Service      # Defines to create service type object
apiVersion: v1
Metadata:
  name: demoservice
Spec:
  ports:
    - port: 80      # Container port exposed
      targetPort: 80 # Pod port
Selector:
  name: deployment # Apply this service to any pod which has the
  type: Nodeport   # Specific label.
                    # Specifies the service type ie clusterIP or
                    # Nodeport
```

```
$#> kubectl apply -f deployment.yaml
$#> kubectl apply -f svc.yaml
$#> kubectl get svc
                    it will show the Nodeport for example
                    80:31341/TCP
$#> kubectl describe svc demoservice
```

## K8s Volumes

- ⇒ Containers are short lived in Nature.
- ⇒ All data stored inside a container is deleted if the container crashes. However the kubelet will restart it with a clean state, which means that will not have any of the old data.
- ⇒ To overcome this problem, K8s uses volume. A volume is essentially directly backed by a storage medium. The storage medium and its content are determined by the volume type.
- ⇒ In K8s, a volume is attached to a pod and shared among the containers of that pod.
- ⇒ The volume has the same life span as the Pod, and its lifetime is limited to the containers of the pod. This allows data to be preserved across container restarts.

## K8s Volume Types

- ⇒ A volume type decides the properties of the directory, size, size, content etc. Some examples of volume types are:-
- ⇒ Node-local types such as emptydir and hostpath.
- ⇒ File sharing types such as nfs.
- ⇒ Cloud provider-specific types like aws EBS, Azure disk.
- ⇒ Distributed file system types, for example glusterfs or cephfs.
- ⇒ Special purpose types like secret, git repo.

## Empty Dir. :-

↳ Volume Type

- ⇒ Use this when we want to share contents between multiple containers on the same pod & not to the host machine.
- ⇒ An EmptyDir volume is first created when a pod is assigned to a node and exits as long as the pod is running on that node.
- ⇒ As the name says, it is initially empty.
- ⇒ Containers in the pod can all read and write the same files in the EmptyDir volume, through that volume can be mounted at the same or different paths in each container.
- ⇒ When a pod is removed from a node for any reason, the data in the empty dir is deleted forever.
- ⇒ Container crashing does not remove a pod from a node, so the data in an empty dir volume is safe across container crashes.

### Example of EmptyDir

apiVersion: v1

kind: Pod

Metadata:

name: myvolemptydir

Spec:

Containers:

- name: c1

image: Centos

command: ["bin/bash", "-c", "sleep 5000"]

Volume Mounts:

- name : xchange

MountPath : "/tmp/xchange"

- name : c2

image : Centos

command : ["/bin/bash", "-c", "sleep 1000"]

volumeMounts :

- name : xchange

Mountpath : "/tmp/data"

Volume :

- name : xchange

emptyDir: { }

\$#> kubectl apply -f emptydir.yaml

\$#> kubectl get pods

\$#> kubectl exec myvolemptydir -c c1 -it -- /bin/bash  
↳ pod name      ↳ Container

\$#> cd /tmp

\$#> ls ⇒ you will find the volume name.

\$#> then go to the volume volume and create some txt file.

\$#> Then go to C2 ⇒ Container 2

\$#> kubectl exec myvolemptydir -c c2 -it -- /bin/bash

\$#> cd /tmp

\$#> ls ⇒ All files will be also here.

## Host Path :-

⇒ we use this when we want to access the Content of a pod/container from hostnames.

⇒ A hostpath Volume Mounts a file or directory from the host node's filesystem into your pod.

## Example of Hostpath

apiVersion : v1

Kind : Pod

Metadata :

name : volumehostpath

Spec :

Containers :

- image : Centos

name : testc

command : ["/bin/bash", "-c", "sleep 15000"]

Volume Mounts :

- Mount Path : /tmp/hostpath

name : testvolume

Volumes :

- name : testvolume

hostPath :

path : /tmp/data

\$#) kubectl apply -f hostpath.yaml

\$#) kubectl get pods.

\$#) kubectl exec volumehostpath -- ls /tmp

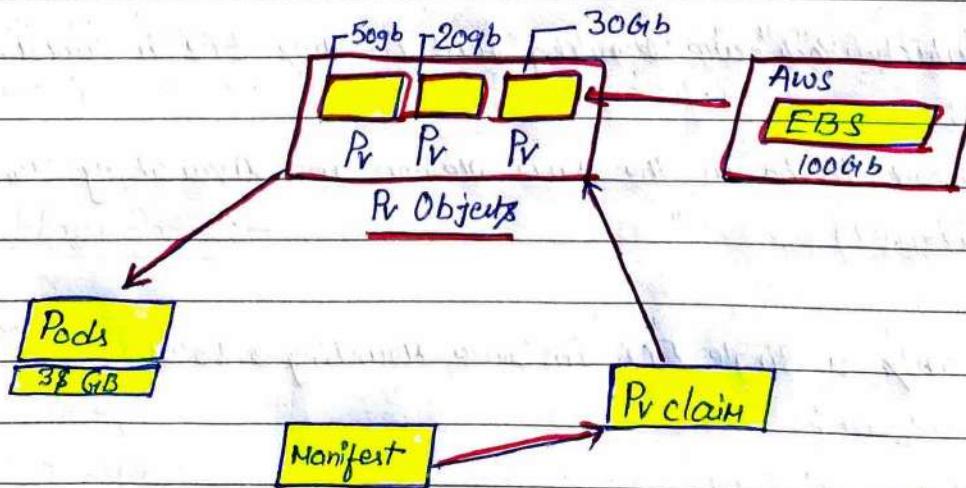
\$#) echo -e "I am sawan" > myfile

\$#) kubectl exec volumehostpath -- ls /tmp/hostpath

\$#) kubectl exec myvolhostpath -- cat /tmp/hostpath/myfile

## Lec-08 - Persistent Volume & Liveness Probe

- ⇒ In a typical IT env, storage is managed by the storage / system admin. The end user will just get instructions to use the storage, but does not have to worry about the underlying storage management.
- ⇒ In the containerized world, we would like to follow similar rules, but it becomes challenging, given the many volume types we have seen earlier. Kubernetes handles this problem with the persistent Volume subsystem.
- ⇒ A persistent Volume is a cluster-wide resource that you can use to store data in a way that is persist beyond the life-time of a pod.
- ⇒ The pv is not backed by locally attached storage on a worker node but by networked storage system such as EBS or NFS or a distributed filesystem like ceph.
- ⇒ K8s provides API for users and admin to manage and consume storage to manage the volume, it uses the persistent volume API resource type and to consume it, uses the persistent volume-claim API resource type.



## Persistent Volume Claim

- ⇒ In order to use a PV you need to claim it first, using a persistent volume claim.
- ⇒ The PVC request a PV with your desired specification (size, access mode, speed etc) from k8s and once a suitable Persistent Volume is found, it is bound to a persistent volume claim.
- ⇒ After a successful bind to a pod, you can mount it as a volume.
- ⇒ Once it is finished its work, the attached PV can be released then underlying PV can then be reclaimed and recycled for future usage.

## Aws EBS

- ⇒ An aws ebs volume mounts on AWS EBS volume into your pod unlike EmptyDir, which is erased when a pod is removed; the contents of an EBS volume are preserved and the volume is merely unmounted.

## There are some restrictions:-

- i) The nodes on which pods are running must be AWS EC2 instances.
- ii) Those instances need to be in the same region and availability zone as the EBS volume.
- iii) EBS only supports a single EC2 instance mounting a volume.

## Example of Persistent Volume

apiVersion : v1

kind : PersistentVolume

Metadata :

name : mypvcvol

Spec :

capacity :

storage : 1Gi

accessMode :

- ReadWriteOnce

PersistentVolumeReclaimPolicy : Recycle

awsElasticBlockStorage :

volumeID : vol-0dc02983b1xyz # Put EBS volume ID Here

fsType : ext4

\$#> kubectl apply -f mypv.yaml

\$#> kubectl get pv

## Example of Persistent Volume Claim

apiVersion : v1

kind : PersistentVolumeClaim

Metadata :

name : mypvcvolumeclaim

spec :

accessModes :

- ReadWriteOnce

resources :

requests :

storage : 1Gi

\$ #> kubectl apply -f mypv.yaml

\$ #> kubectl get pvc

### Example of Deployment.

apiVersion: apps/v1

kind: Deployment

metadata:

name: mydeploy

spec:

replicas: 1

selector: # tells the controller which pods to watch / belong to

matchLabels:

app: mypv

template:

Metadata:

labels:

app: mypv

spec:

containers:

- name: shell

image: CentOS

command: ["/bin/bash", "-c", "sleep 1000"]

volumeMounts:

- name: mypd

mountPath: "/tmp/persistent"

volumes:

- name: mypd

persistentVolumeClaim:

claimName: mypvc.vol.claim

```
$#> kubectl apply -f deployrc.yaml  
$#> kubectl get deploy  
$#> kubectl get rs  
$#> kubectl get pods  
$#> kubectl exec pvdeploy-0589xyz -it -- /bin/bash  
$#> cd /tmp/persistent
```

Now delete the existing pod and create an RS will create one new pod to Under that pod in the tmp dir and check the same file will be there.

### Health Check / Liveness Probe

- ⇒ A pod is considered ready when all of its containers are ready.
- ⇒ In order to verify if a container in a pod is healthy and ready to serve traffic, k8s provides for a range of healthy checking connection.
- ⇒ Health checks or probes are carried out by the kubelet to determine when to ~~restart~~ <sup>recreate</sup> a container (For liveness probe) and used by services and deployments to determine if a pod should receive traffic.

For eg. Liveness probes could catch a deadlock, where an application is running but unable to make ~~any~~ progress. Restoring - Restarting a container in such a state can help to make the application more available despite bugs.

- ⇒ One use of readiness probe is to control which pods are used as backends for services when a pod is not ready, it is removed from Service Load Balancer.
- ⇒ For running healthcheck, we would use cmd specific to the application
- ⇒ if the cmd succeeds, it returns 0, and kubelet considers the container to be alive and healthy. if the command returns a non-zero value the kubelet kills the pod and removes it.

### Example of Liveness Probe

Kind : Pod

Metadata :

Labels :

test : liveness

name : mylivenessprobe

Spec :

Containers :

- name : liveness

image : ubuntu

args :

- /bin/sh

--c

- touch /tmp/healthy ; sleep 1000

livenessprobe : # define health check

exec :

command :

# command to run periodically

- cat :

- /tmp/healthy

initialDelaySeconds: 5 # Wait for the specified time before it runs first probe.  
periodSeconds: 5  
timeoutSeconds: 30 # Run the above command every 5 sec

\$#> kubectl apply -f liveness.yaml

\$#> kubectl get pods

\$#> kubectl describe pod mylivenessprobe

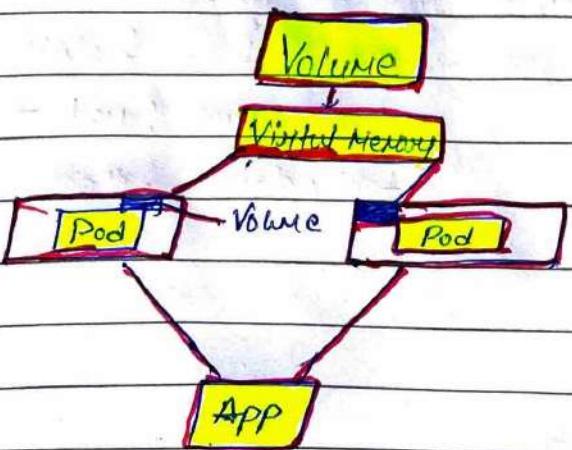
\$#> kubectl exec mylivenessprobe -it -/bin/bash

\$#> cat /tmp/healthy

\$#> echo \$? # to check the status.

## Lec -09 - ConfigMap and Secrets

- ⇒ while performing application deployments on k8s cluster, sometimes we need to change the application Configuration file depending on environment like dev, QA, stage or prod.
- ⇒ Changing this application Configuration file means we need to change source code, Commit the change, creating a new image and then go through the completed deployment process.
- ⇒ Hence these Configuration should be decoupled from image Content in Order to keep Containerised application portable.
- ⇒ This is where k8s Configmap Come handy it allows us to handle Configuration files much more efficiently.
- ⇒ Configmaps are useful for storing and sharing non-sensitive; unencrypted Configuration information we Secretly otherwise.
- ⇒ Configmap can be used to store fine-grained information like individual properties or entire config files.
- ⇒ Configmap are not intended to act as a replacement for properties file.



⇒ ConfigMap can be created in following ways :-

i) As Env Variable.

ii) As volume in the pod.

Kubectl create configmap <mapname> --from-file = <file to read>

### SECRETS

⇒ You don't want sensitive information such as a database password or an API key kept around in clear text.

⇒ Secrets provide you with a mechanism to use such information in a safe and reliable way with the following properties :-

⇒ Secrets are namespace objects, that exist in the context of namespace

⇒ you can access them via a volume or an environment variable from a container running in a pod.

⇒ The secret data on nodes is stored in tmpfs volume (tmpfs is a file system which keeps all files in virtual memory everything in tmpfs is temporary in the sense that no files will be created on your hdd).

⇒ A per-secret size limit of 1mb exist.

⇒ The API Server stores secrets as plaintext in etcd.

Secrets can be created.

1.) From a text file

2.) From a YAML file.

```
$#> vi sample.conf # create sample config file.  
$#> kubectl create configmap mymap --from-file=sample.conf  
$#> kubectl get configmap # to check the details under the file.  
$#> kubectl describe ConfigMap mymap.
```

### Example of ConfigMap

apiVersion: v1

kind: Pod

Metadata:

name: myvolcunfig

spec:

containers:

- name: c1

image: Centos

command: ["bin/bash", "-c", "while true; do echo 'Hello Sawan'; sleep 5; done"]

volumeMounts:

- name: testconfigmap

mountPath: "/tmp/config" # the config files will be mounted at readonly here

volumes:

- name: testconfigmap

ConfigMap:

name: mymap # This should match the Config Map name created

items:

- key: sample.conf

path: sample.conf

```
$#> kubectl apply -f deployconfigmap.yaml  
$#> kubectl get pods.  
$#> kubectl exec myenvconfig -it -- /bin/bash
```

apiVersion: v1

### Example of MYENV

kind: Pod

Metadata:

name: MyEnvConfig

Spec:

Containers:

- name: C1

image: CentOS

command: ["/bin/bash", "-c", "while true; do echo Hello Sawan; sleep 5; done"]

Env:

- name: MYENV # env name in which value of the key is stored.

valueFrom:

configMapKeyRef:

name: mymap # name of the config created.

key: sample.conf

```
$#> kubectl apply -f deployenv.yaml
```

```
$#> kubectl get pods.
```

```
$#> kubectl exec myenvconfig -it -- /bin/bash
```

```
$#> env
```

```
$#> echo $MYENV
```

```
$#> exit.
```

## Example of Secret

```
$#> echo "Moust" > username.txt ; echo "mурзикword" > password.txt  
$#> cat password.txt  
$#> kubectl create secret generic mysecret --from-file=username.txt --from-file=password.txt
```

apiVersion: v1

kind: Pod

Metadata:

name: Myvolsecret

Spec:

containers:

-name: c1

image: centos

command: ["/bin/bash", "-c", "while true; do echo Hello-Saavn; sleep 5; done"]

volumeMounts:

-name: testsecret

mountPath: "/tmp/mysecret" # the secret files will be mounted

volumes:

as readonly by default here

-name: testsecret

Secret:

SecretName: mysecret.

```
$#> kubectl apply -f deployment.yaml
```

```
$#> kubectl get pods
```

```
$#> kubectl exec Myvolsecret -it -- /bin/bash
```

```
$#> cd /tmp
```

```
$#> cat password.txt
```

```
$#> cat username.txt
```

## Lec-10 - Namespaces, Limits & Request

Namespaces:- you can name your object, but if many are using the cluster then it would be difficult for managing.

⇒ A namespace is a group of related elements that each have a unique name or identifier. Namespace is used to uniquely identify one or more names from other similar names of different objects or groups or the namespace in general.

⇒ Kubernetes names help different projects teams or customers to share a Kubernetes cluster & provider :-

⇒ A scope for every name.

⇒ A mechanism to attach authorization and policy to a subsection of the cluster.

⇒ By default, a k8s cluster will instantiate its default namespace when provisioning the cluster to hold the default set of pods, services and deployments used by the client cluster.

⇒ We can use resource quota on specifying how many resources each namespace can use.

⇒ Most k8s resources (e.g. pods, service replication controllers and others) are in some namespaces and low-level resources such as nodes and persistent volumes, are not in any namespace.

⇒ Namespaces are intended for use in environment with many users spread across multiple teams, or projects for clusters with a few to tens of users, you should not need to create or think about namespaces and all.

### Create New Namespace

⇒ Let us assume we have shared K8s cluster for Dev and production use cases.

⇒ The dev team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services and Deployments they use to build and run their application. To this, no restrictions are put on who can or cannot modify resources to enable agile development.

⇒ For prod team we can enforce strict procedure on who can or cannot manipulate the set of pods, services and deployments.

Cmd :- **kubectl get namespaces**.

### **Namespace YAML**

APIVersion : v1

Kind : Namespace

Metadata:

name: dev

Labels:

name: dev

\$#> kubectl config set-context \$(kubectl config current-context) --namespace=dev