# Lec 45 → Complete Theory of Kubernetes

→ Kubernetes is an open-source container management tool which automates container deployment, container & load balancing.

→ It schedules, runs & manages isolated containers which are running on virtual/Physical/cloud Machines.

→ All top cloud providers support Kubernetes.

## * History

↳ Google developed an internal system called 'borg' (later named as Omega) to deploy & manage thousands google application & services on their clusters.

→ In 2014, google introduced Kubernetes an open source platform written in 'Golang' & later donated to CNCF.

↳ Cloud Native computing foundation

## * ONLINE PLATFORM FOR K8S

↳ Kubernetes playground.
→ Play with K8s
→ Play with Kubernetes classroom.

## * Cloud Based K8s Services.

↳ GKE → Google Kubernetes Services.
→ AKS → Azure Kubernetes Services.
→ Amazon EKS → (Elastic Kubernetes Services).

## * KUBERNETES INSTALLATION TOOL

- Minicube.
- Kubeadm.

## * Problems with Scaling up the containers

- Containers cannot communicate with each other.
- Autoscaling & load Balancing was not possible.
- Containers had to be managed carefully.

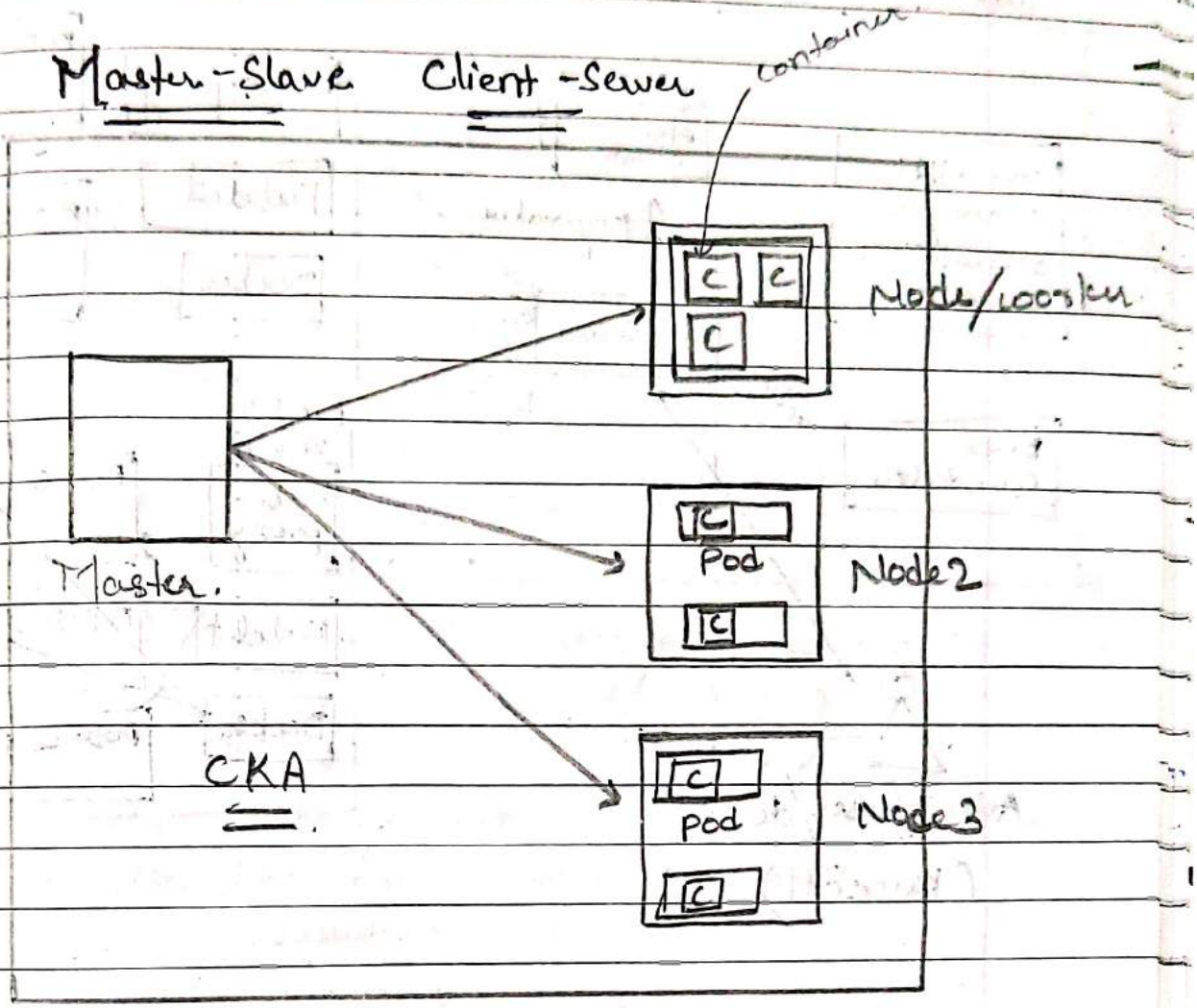
## * Features of Kubernetes. → supports Hybrid Cloud.

- Orchestration (Clustering of any no. of containers running on different m/c).
- Auto Scaling. (Vertical & Horizontal 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).

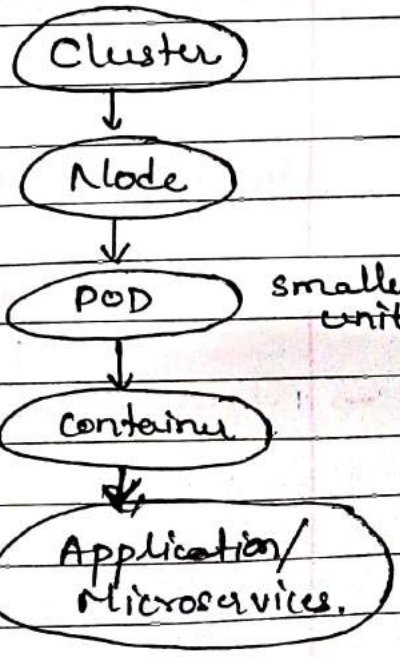
| Features | Kubernetes | Docker SWARM |
|---|---|---|
| Installation & cluster Configuration | complicated & time consuming | Fast & Easy. |
| Supports | K8s can work with almost all containers type like Rocket, container, containerD | work with docker only. |
| GUI | GUI Available | GUI not available. |
| Data Volumes | only shared with container in same POD | can be shared with any other container. |
| Updates & Rollback | Process scheduling to maintain services while updating. | Progressive updates & services health monitoring throughout the updates. |
| Auto Scaling | Support Vertical & Horizontal Autoscaling | Not support Autoscaling. |
| Logging & Monitering. | Inbuilt tool present for monitoring | used 3rd Party tools like splunk. |

# * Architecture of Kubernetes.

↳ Master-Slave Client-Sever

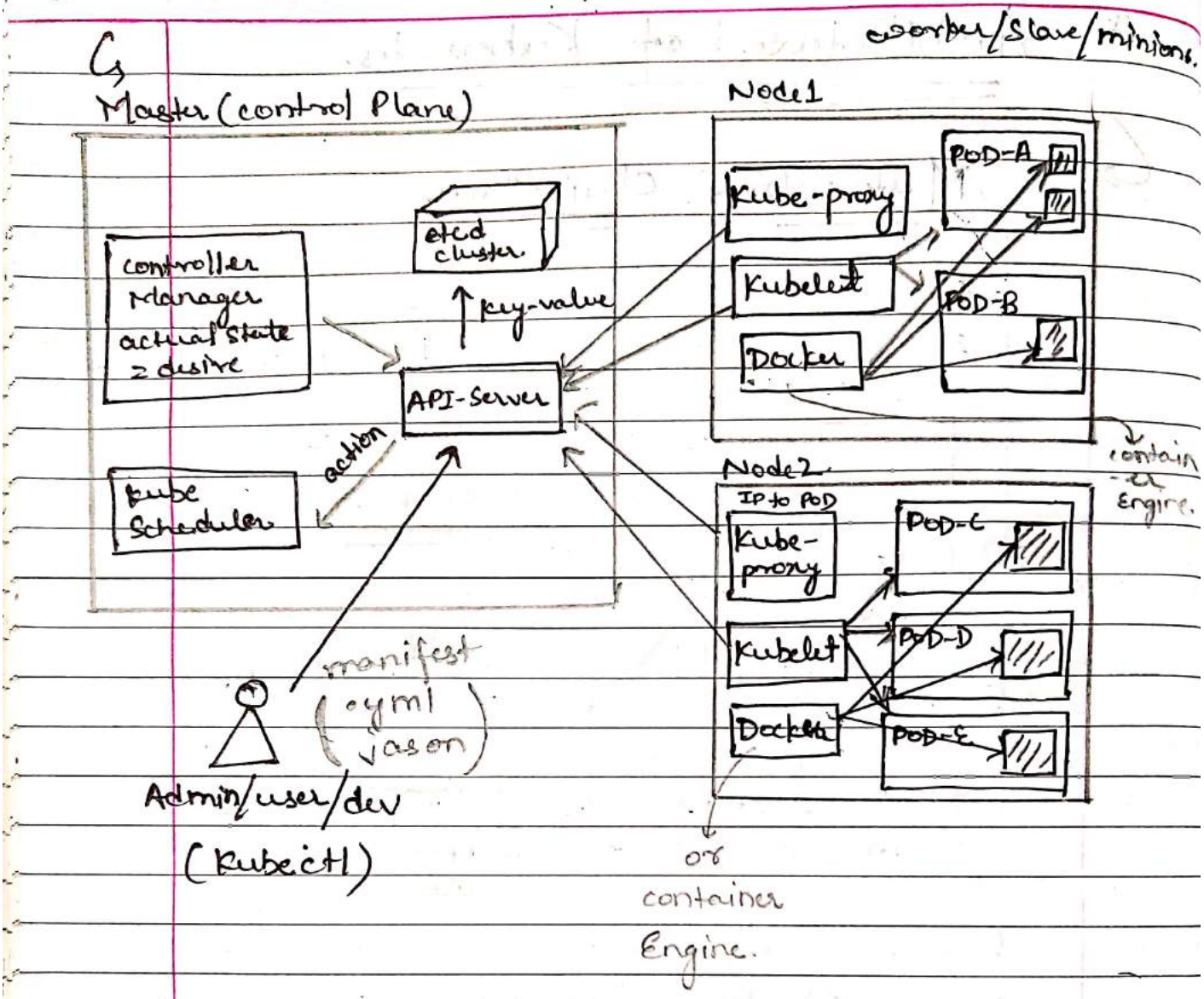



Cluster



# Lec-46 : Architecture of Kubernetes & its demo

Date
Page No



## * Working with Kubernetes.

↳ we create manifest (.yml)

→ 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 or Bare Metal / VM instances / cloud instances / all mix.

→ Kubernetes designates one or more of these as master & all others as workers.

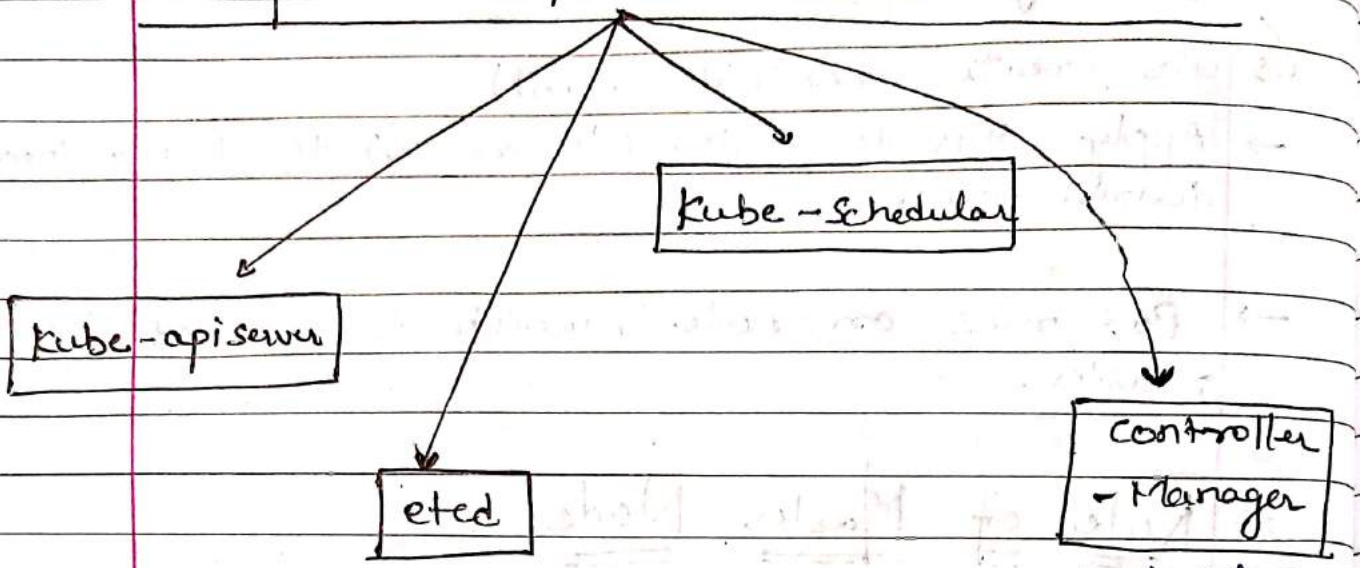
→ The master is now going to run set of K8s processes. These processes will ensure smooth functioning of cluster. The process are called "control Plane".

→ Can be Multi-master for high availability.

→ Master runs Control Plane to run cluster smoothly.

Date / /
Page No.

# * Components of control Plane (Master)




actual state = desired state.

① Kube-API-Server (For all communications)

↳ The api-server interacts directly with user (i.e., we apply .yml or json manifest to kube-apiserver)

→ This kube-apiserver is meant to scale automatically as per load.

→ Kube api-server is front-end of control panel.

② etcd

↳ Stores metadata & status of cluster

→ etcd is consistent & high-available store (key-value store).

→ Source of touch for cluster state (info about 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.

## (3) → Kube-Scheduler (action)

↳ When users make request for the creation & Management of PODS, Kube-Scheduler is going to take action on these requests.

→ Handles POD Creation & Management

→ Kube-Scheduler match/assign any node to create and run pods.

→ A Scheduler watches for newly created Pods that have no node assigned for every Pod that the Scheduler discovers, the Scheduler becomes Responsible for finding best node for that pod to run on.

→ Scheduler gets the information for hardware configuration from configuration files & schedules the Pods on nodes accordingly.

## (4) → Controller Manager

↳ Make sure actual state of cluster matches 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 stops responding.

→ Route-controller → responsible for setting up network routes on your cloud.

→ Service Controller → Responsible for load Balancers on your cloud against services of type Load Balancer.

→ 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.

## ① Kubelet →

↳ Agent running on the node.

↳ Listen to Kubernetes master (eg - pod creation request)

→ use Port 10255

→ send success / fail reports to master.

## ② → Container Engine (Docker, Rocket)

↳ works with kubelet

→ Pulling images.

→ start / stop containers.

→ Exposing containers on ports specified in manifest.

## (3) Kube - proxy

↳ Assign IP to each Pod.

→ It is required to assign IP addresses to pods (dynamic)

→ kube-proxy runs on each node & this make sure that each pod will get its own unique IP address.

→ These 3 components collectively consist "node".

## * POD

↳ Smallest unit 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 atleast one coorker 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 control by master.

→ Kubernetes only knows about PODs (does not know about individual container)

→ Cannot start containers without a POD.

→ One POD usually contains one container.

## ⇒ Multi-Container PODs

↳ Share access to memory space.

→ connect to each other using local host. <container port>

→ share access to the same volume.

→ containers within pod are deployed in an all-or-nothing manner.

→ Entire pod is hosted on the same node (scheduler will decide about which node).

## ~~⇒~~ POD Limitations

↳ No auto healing or auto scaling

→ POD crashes.

# * Higher level Kubernetes Objects.

1. Replication set → auto Scaling & auto healing
2. Deployment → Versioning & Rollback.
3. Service → Static (Non-ephemeral) IP and Networking.
4. Volume → Non-ephemeral Storage.

## Important

1. Kubectl → Single cloud.
2. Kubeadm → on premise.
3. Kubefed → Federated.

# 47 → Setup Kubernetes Master & Node on AWS

↳ Login into AWS account → lanch 3 Instances → ubuntu (t2.medium)

Master must have 2CPU & 4GB RAM

→ Now, using puttygen, create private key & save.

→ Access all the 3 instances (1 Master, 2 nodes) using putty

**Commands common for Master & Node.**

↳ sudo su
→ apt-get update

Now install https package
→ apt-get install apt-transport-https

This https is needed for intra cluster common (particularly from control plane to individual pods)

Now install docker on all 3 instances
→ sudo apt install docker.io -y.

To check, whether docker is installed or not

→ docker --version
→ systemctl start docker
→ systemctl enable docker

Setup open GPG key This is required for intra cluster communication. It will be added to source key on this node i.e when K8s send signed info to our host, it is going to accept those information because this open GPG key is present in the source key.

→ `curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add`

Edit sources.list file (apt-get install nano)

→ `nano /etc/apt/sources.list.d/kubernetes.list`

→ `deb http://apt.kubernetes.io/ kubernetes-xenial ma`

exit

→ `apt-get update` → Install all Packages

→ `apt-get install -y kubelet kubeadm kubectl kubernetes-cni`

*. Bootstrapping the Master Node (in Master)

↳ To initialize K8s cluster.

→ kubeadm init

You will get one long command started from "kubeadm join 172.31.6.165:6443 ....
copy this command & save on notepad.

↓

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

→ `mkdir -p $HOME/.kube` → default command

↳ Copy configuration to kube directory (in config file)

→ `sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config.` → default command

Provide user permissions to config file

→ `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.yml`

→ sudo KubeCH apply -f https://raw.github
-usercontent.com/coreos/flannel/master/
Documentation/K8S-manifests/kube-flannel
-rbac.yml.

Configure worker node.

→ paste long command in both the nodes

go to master.

→ kubectl get nodes

# Lec 48 → Kubernetes in Detail

→ Kubernetes uses objects to represent the state of your cluster.

→ what containerized applications are running (and on which node)

→ The policies around how those applications behave, such as restart policies, upgrades & fault tolerance.

→ Once you create the object, the kubernetes system will constantly work to ensure that object exist & maintain's cluster's desired state.

→ Every kubernetes object includes two nested fields that govern the object config. the object spec & 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 & is supplied and updated by the kubernetes system.

→ All objects are identified by a unique name and a UID.

→ The Basic Kubernetes objects include:-

1. POD
2. Service
3. Volume
4. Namespace
5. Replicasets
6. Secrets
7. Config Maps
8. Deployments
9. Jobs.
10. Daemonsets.

→ It represents as JSON or YAML files. (manifest)

→ You create these and then push them to the Kubernetes API with Kubectl.

## * Relationship betw these Objects.

↳ POD manages containers.
→ Replicaset manages pods.
→ Services expose pod processes to the outside world.
→ Configmaps and secrets helps you configure pods.

## * State of the objects

↳ Replicas (2/2)
→ Image (Tomcat/ubuntu)
→ Name
→ Port
→ Volume
→ startup cmd
→ Detached (default).

# * Kubernetes Objects Management.

↳ The Kubectl command line tool supports several different ways to create and manage kubernetes object.

| Management Technique | Operates on | Recommended Environment. |
|---|---|---|
| Imperative commands | Live objects | Development Projects |
| Declarative commands | Live Individual | Production. |
| Object configuration | files (Yml/Json) | |

→ 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 untill the process is terminated, the pod object is deleted, the pod is evicted for lack of resources, 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.

→ The given (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 exists as long as that POD (with that UID) exist if that POD is deleted for any reason, volume is also destroyed & created as new on new pod.

→ A controller can create & manage multiple pods, handling replication, rollout & Providing self-healing capabilities.

# * Kubernetes Configuration

## (1) All-in-One single node installation.

↳ 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 soon.

## (2) Single-Node eted, Single-Master & Multi-worker Installation.

↳ In this step, we have a single master node which also runs a single-node etcd instance. Multiple worker nodes are connected to the master node.

## (3) Single-node etcd, Multi-master and Multiworker installation.

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

Lab →

→ Go to aws account → Launch instance
→ ubuntu 18.04 → t2.medium (2vCPU)

Now access EC2 via putty → login as "ubuntu"

→ sudo su
→ apt update && apt -y install docker.io

Now install kubectl (link, I will provide you in description).

Then install minikube

→ apt install conntrack → need to install for proper functioning of minikube
→ minikube start --vm-driver=none.
→ minikube status
→ kubectl version.

Now onwards, we will use kubectl commands.

* → kubectl get nodes. → to check all nodes in kubernetes.

↳ O/P =

| Name | Status | Roles | Age | Version |
|---|---|---|---|---|
| ip-172-31-34-55 | ~~Not~~ Ready | Master | 2m | v1.20.7 |

*

→ kubectl describe node ip-172-31-34-55

→ vi pod1.yml.

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

Spec:
    containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while
                  true;
                  do echo Hello-Bhupinder; sleep5;
                  done"]
    restart policy: Never
```

represents group

→ :wq.

→ Kubectl apply -f pod1.yml → pod will create/ready

↳ o/p → pod/testpod created

→ kubectl get pods

↳ o/p → testpod 1/1 Running.

(1/1 → container, 1 container)

→ if you want to see, where exactly pod is running

↳ kubectl gets pods -o wide

→ `kubectl describe pod testpod`

or

`kubectl describe pod/testpod.`

→ To check info about the pod.

↳ `kubectl logs -f testpod.` `-c coo`

↳ to check specific container.

→ if you want to delete pod

↳ `kubectl delete pod testpod`

or

`kubectl delete -f pod1.yml`

# Example of Annotation

for adding extra info

→ vi pod1.yml

```
metadata:
    name: testpod
    annotations:
        description: Hamara pehla pod create
                     hone ja rha hai
```

:wq →

→ `kubectl apply -f pod1.yml` → to apply those changes.

→ `kubectl describe pod testpod.`

↳ Now you can see annotation somewhere.

→ `kubectl delete pod testpod.`

↳ to delete that pod.

Eg: Multicontainer Pod.

↳ vi pod2.yml

```
kind: Pod
apiVersion: v1
metadata:
    name: testpod3

Spec:
  containers:
    -name: c00
     image: ubuntu
     command: ["/bin/bash", "-c", "while true,
              do echo Techincal-Gufta: sleep5;
              done"]
    -name: c01
     image: ubuntu
     command: ["/bin/bash", "-c", "--"]
```

→ :wq →

→ kubectl apply -f pod2.yml

→ kubectl get pods

↳ O/P → testpod3 2/2

→ kubectl describe pod testpod3

→ `kubectl logs -f testpod3`

↳ you will get error, need to specify container.

→ `kubectl logs -f testpod3 -c c00` → andar kya chal rha dekhne liye

→ `kubectl logs -f testpod3 -c c01`

→ `kubectl exec testpod3 -it -c c02`

↳ -/bin/bash.

→ ps

→ ps -ef

} to work inside container.

→ exit

→ `kubectl delete pod testpod3`

## * Eg → Environment Variables in Pod

↳ vi pod3.yml

```
kind : Pod
apiVersion: V1
metadata:
    name: environments
spec:
  containers:
    -name: c00
     image: ubuntu
     command: ["/bin/bash","-c","while true;
               do echo Hello-Bhupinder;
               sleep 5; done"]

  env:
    -name: MYNAME
     value: BHUPINDER
```

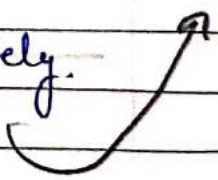
→ :wq

→ `kubectl apply -f pod3.yml`

O/P → pod/environment created.

→ `kubectl get pods`

if you want to go inside interactively.



→ kubectl exec environments -it --/bin/bash

→ env
→ echo $ MyNAME.
o/p → Bhupinder

Inside container

→ exit

→ kubectl delete -f pod3.yml.

* Eg → Pod with Ports

↳ vi pod4.yml

```
kind: Pod
apiVersion: V1
metadata:
    name: testpod4

spec:
  containers:
    -name: c00
     image: httpd
     ports:
      - containerPort: 80
```

↳ :wq

→ kubectl apply -f pod testpod4.yml.

→ kubectl get pods.

→ kubectl get pods -o wide

↳ O/P → you will get pod ip.

→ curl 172.17.0.3:80

↳ O/P → It works

→ kubectl delete -f pod4.yaml

→ kubectl gets pods.

↳ O/P → No resources found.

# Lec 49:→ Kubernetes in Details

## Labels and Selector

↳ Labels are the mechanism you use to organise Kubernetes Objects.

→ A label is a key-value pair without any pre-defined meaning that can be attached to the objects.

→ Labels are similar to tags in AWS or git where you use a name to Quick reference.

→ So you are free to choose labels as you need it to refer an environment which is used for dev or Testing or Production, refer a product group like Department A, Department B.

→ Multiple labels can be added to a single object.

→ vi pod5.yml

```
Kind: Pod
apiVersion: v1
metadata:
    name: delhipod
    labels:
        env: development
        class: pods.
        Name: Bhupinder
        comp: Technical Trainer
```

```
spec:
  containers:
    -name: c00
     image: ubuntu
     command: ["bin/bash", "-c", "while true;
          do echo Hello-bhupinder; sleep5; done"]
```

: wq. →

→ kubectl apply -f pods.yml

→ kubectl get pods --show-labels.

objects.

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

imperative method

→ kubectl label pods dilhipod myname=bhupindu

→ 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`

## * Labels - 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 & 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 requirements which are comma-separated.

### Equality Based ( =, != )

↳
```
name: bhupinder
class: nodes
project: development.
```

* Set based → (in, notin & exists)

→ env in (production, dev)
→ env not in (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 not in (development, testing)'`
(no space)

→ `kubectl get pods -l class=pods, myname=bhupinder`

# * Node Selector

One use case 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 constraints are unnecessary, as the scheduler will automatically do a reasonable placement, but on certain circumstances we might need it.

→ we can use labels to tag nodes.

→ You the nodes are tagged, you can use the label selectors to specify the pod run only of specific nodes.

→ first we give label to the node.

→ Then use node selector to the pod configuration.

lab→ vi pod6.yml

```
kind: Pod
apiVersion: v1
metadata:
    name: nodelabels
    labels:
        env: development
```

```
spec:
  containers:
    -name: c00
     image: ubuntu
     command: ["/bin/bash", "-c", while true;
              do echo Hello-Bhupinder; sleep5;
              done"]
     nodeselector:
          hardware: t2-mediem.
```

→ kubectl apply -f pod6.yml.

→ kubectl get pods. → will throw error because we have not applied label on node.

→ kubectl get nodes. → to get node.

→ kubectl label nodes ip-172-31-34-55 hardware=t2.medium

↳ O/P : labeled

↓ Imperative method.

→ kubectl delete -f pod6.yml.

↳ to delete pod.

# * Scaling & Replication

↳ Kubernetes was designed to orchestrate multiple containers & 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 fails.

⇒ Load Balancing → Having multiple versions of a containers enables you to easily send traffic to different instances to prevent overloading of a single instances or node.

⇒ Scaling → when load does become too much for the number of existing instances, kubernetes enables you to easily scale up your application, adding additional instances as needed.

⇒ Rolling Updates → Updates to a service by replacing pods one by one.

kind: replication controller.

* Replication Controller

↳ A replication controller is a object that enables you to easily create multiple pods, then make sure that number of pods always exist.

→ If a pod created using RC will be automatically replaced if they does 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.

(abs) = ↳ vi ~~my~~rc.yml → object.

```
kind: ReplicationController → this defines to create the object of replication type
apiVersion: v1
metadata:
  name: myreplica
spec:
  replicas: 2 → this element defines the desired no. of pod
  selector: → tells the controller which pods to watch/belong to this rc
    myname: bhupinder → this must match the label
  template: → template element defines a template to launch a new pod.
    metadata:
      name: testpod6
```

labels: → selectors values need to match, the labels values specified in the pod template.
*myname: bhupinder

spec:
containers:
- name: c00
image: ubuntu
command: ["bin/bash", "-c", "while-true; do echo hello-bhupinder; sleep 5; done"]

! log →

+ Note → selector's name & label's name must be same.

→ kubectl apply -f myrc.yml

→ kubectl get rc → to get replicas.

→ kubectl describle rc myreplica

→ kubectl get pods

→ kubectl delete pod (pod name) myreplica-2worst

↳ after deleting again 1 pod will be created

To scale up

↳ kubectl scale --replicas=6 rc -l myname=bhupinder

↳ o/p: replicationcontroller / myreplica scaled.

→ To scale down

↳ kubectl scale --replicas=1 rc -l myname=bhupinder

↳ it will terminate all other pods. except 1.

→ kubectl delete -f myrc.yml

↳ o/p: replicationcontroller "myreplica" deleted.

→ kubectl get rc

↳ o/p: no resources found in default namespace

→ kubectl get pods

↳

→ advance version of replication controller.

## * Replica Set

↳ Replica set is a next generation replication controller.

→ The replication controller only supports equality-based selector whereas the replica set supports set-based selector i.e, filtering according to set of values.

→ Replicaset rather than the Replication controller is used by other objects like deployment.

↳ kind: ReplicaSet
apiVersion: apps/v1
} changes.

tab →

→ vi myrs.yml.

```
kind: ReplicaSet
apiVersion: apps/v1
metadata:
    name: myrs
spec:
  replicas: 2
  selector:
    matchExpressions:
      - {key: myname, operater: In,
         values: [Bhupindr, Bupiner,
                  Bhopendra]}
      - {key: env, operator: NotIn,
         values: [production]}

  template:
    metadater:
      name: testpod7
      labels:
        myname: Bhupinder
    spec:
      containers:
        -name: c00
         image: ubuntu
         command: ["/bin/bash", "-c",
                   "while true; do echo
         Technical-Guftgu ; sleep 5 ; done"]
```

esc :wq →

→ kubectl apply -f myrs.yml.

→ kubectl get pods.

→ kubectl scale --replicas=1 rs/myrs
↳ o/p → replicaset.apps/myrs scaled.

→ kubectl get rs → to get replicaSet details.

→ kubectl delete pod myrs-77nmt
↳ pod name.
↳ after delete 1 pod, 1 will be created.

→ kubectl get pods.

→ kubectl delete rs/myrs

→ kubectl get rs
↳ No resource found in default namespace.

→ To delete files
↳ kubectl -rf pod1.yml
↳ file name.

# Lec 50 :- Deployment & Rollback

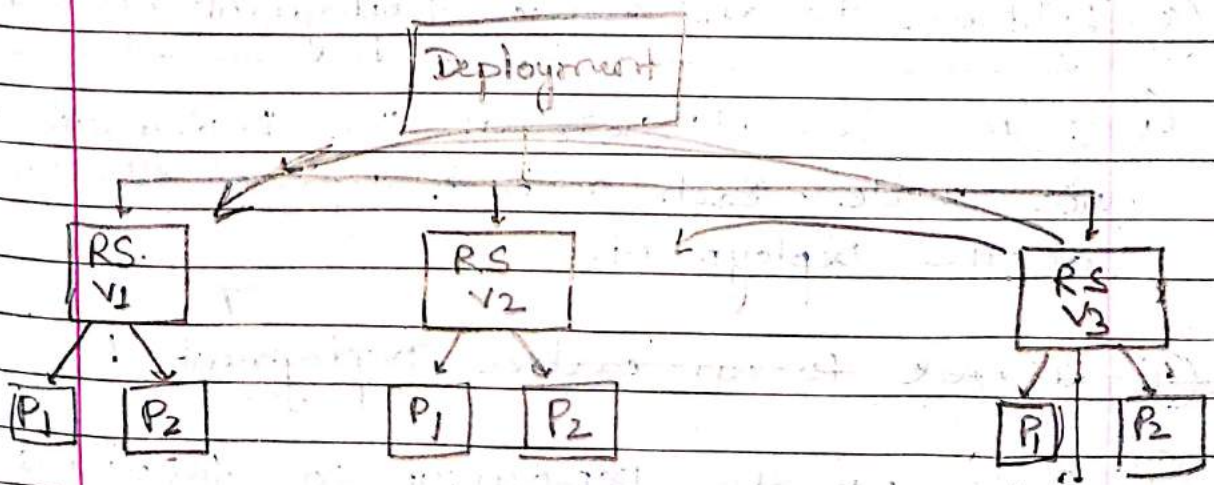
↳ Replication controller & Replica set is not able to do updates & Rollback apps in the cluster.

→ 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 schedules mentioned app instances onto specific individuals nodes.

→ K8s then monitors, if the node hosting an instance goes down or pod is selected the deployment controller replaces it.

→ This provides a self-healing mechanism to address machine failure or maintenance.




→ A deployment provides declarative updates for Pods & Replicaset.

Date / /
Page No.

* → The following are typical use cases of Deployments: →

(1) Create a deployment to rollout a Replicaset → The replicaset creates pods in the Background. Check the status of the Rollout to see if it succeeds or not:

(2) Declare the new state of the Pods →

↳ By updating the PodTemplateSpec of the deployment. A new Replicaset is created and the Deployment manages moving 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 →

↳ If the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.

(4) ~~Rollback to an earlier Deployment~~ !

(4) Scale up the Deployment to facilitates more load.

(6) Pause the Deployment to apply multiple fixes to its PodTemplateSpec & then resume it to start a new Rollout.

(7) Cleanup older Replicaset that you do not need anymore.

*⇒ If there are problems in the deployment, = kubernetes will automatically roll back to the previous version, however you can also explici-tly rollback to a specific revision, as in our case to Revision 1 (the Original Pod version)

→ You can rollback to a specific version by specifying it with `--to-revision`

ForEg ⇒ `kubectl rollout undo` → previous version.

Type ↳ `deploy/my deployment --to-revision=2`

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

cmd → kubectl get deploy

when you inspect the deployments in your cluster, the following field are display.

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

READY → Display how many replicas of the application are available to your users. If follows the pattern ready/Desired.

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

AVAILABLE → Displays how many replicas of the application are available to your users.

AGE → Display the amount of time that the application has been Running.

Lab

Login to AWS Management console, Create one ubuntu (t2.medium) instances. Take access using putty.

→ sudo su
→ sudo apt update && apt -y install docker.io
→ install kubectl from given link.
→ install minikube from link.
→ apt install conntrack
→
Create one yaml file now.

→ Vi mydeploy.yml

```
kind: Deployment
apiVersion: apps/v1
metadata:
    name: mydeployment
spec:
    replicas: 2
    selector: # tell controller which pods to
                watch / belongs to

  matchLabels:
      name: deployment
  template:
      metadata
        name: testpod
        labels:
          name: deployment
```

Should be same

```
spec:
  containers:
    - name : c00
      image : ubuntu
      command : ["bin/bash", "-c"
        "while true ; do echo Technical-
          Gurukul ; sleep 5; done "]
```

→ kubectl apply -f mydeploy.yml.

⇒ To check deployment was created or not

↳ `kubectl get deploy`

→ To check, how deployment creates RS & pods

↳ `kubectl describe deploy mydeployments.` (+yaml)

⇒ kubectl get rs

→ To scale up or scale down.

↳ `kubectl scale --replicas=1 deploy mydeployments`

↳ newest created pod will be deleted

→ To check, what is running inside containers.

↳ `kubectl logs -f <podname>`

→ `kubectl rollout status deployment mydeployments`

(to check versions)

→ `kubectl rollout history deployment mydeployments`

→ `kubectl rollout undo deploy/ mydeployments.`

* Failed Deployment

↳ Your deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

1. Insufficient Quota.
2. Readiness probe failures.
3. Image pull errors.
4. Insufficient permission
5. Limit Ranges.
6. Application runtime misconfiguration.

# Lec 51 : Kubernetes Networking, Services, Nodeport & Volumes

→ Kubernetes Networking addresses four concerns:-

1. Containers within a pod use networking to communicate via loopback.

2. Cluster Networking provides communication between Pods.

3. The services resources lets you expose an application running in Pods to be reachable from outside your cluster.

4. You can also use services to publish services only for consumption inside your cluster.

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

Labs:

vi pod1.yml

```
kind: Pod
apiVersion: v1
metadata:
    name: testpod
spec:
  containers:
    -name: c00
     image: ubuntu
     command: ["/bin/bash", "-c", "while true;
              do echo Hello-Bhupinder; sleep5;
              done"]
    --name: c01
      image: httpd
      ports:
        -containerPort: 80
```

→ kubectl apply -f pod1.yml.

→ kubectl get pods

→ kubectl exec testpod -it -c c00 --/bin/bash  (space here)

↳ To enter in a container to communicate with other container.

· testpod:/# → apt update && apt install curl -y

↳ to download curl package.

→ → curl localhost:80

↳ o/p → <html><body><h1> It works! </h1></body></html>

→ /:# exit

→ kubectl delete -f pod1.yml

* Now try to established communication betn two different Pods within same machine.

→ Pod to Pod communication on same worker node happens through Pod IP

→ By default Pod's IP will not be accessible outside the node.

Labs vi pod2.yml

```
kind: Pod
apiVersion: v1
metadata:
    name: testpod1
spec:
    containers:
      - name: c00
        image: ~~ubuntu~~ nginx
        ~~command: ["bin/bash", "-c", "while true; do echo Hello-Bhupinder; sleep 5; done"]~~
      ~~- name: c01~~
        ~~image: httpd~~
        ports:
          - containerPort: 80
```

→ vi pod3.yml

```
kind: pod
apiVersion: v1
metadata:
    name: testpod4
spec:
  containers:
    - name: c03
      image: httpd
      ports:
        - containerPort: 80
```

→ kubectl apply -f pod2.yml

→ kubectl apply -f pod3.yml

→ kubectl get pods.

→ kubectl gets pods -o wide. → to get pods details.

→ curl testpod1_IP_Address:80

O/P ⇒ welcome to Nginx.

→ curl testpod4_IP_Address:80

O/P ⇒ It works.

→ kubectl delete -f pod2.yml.

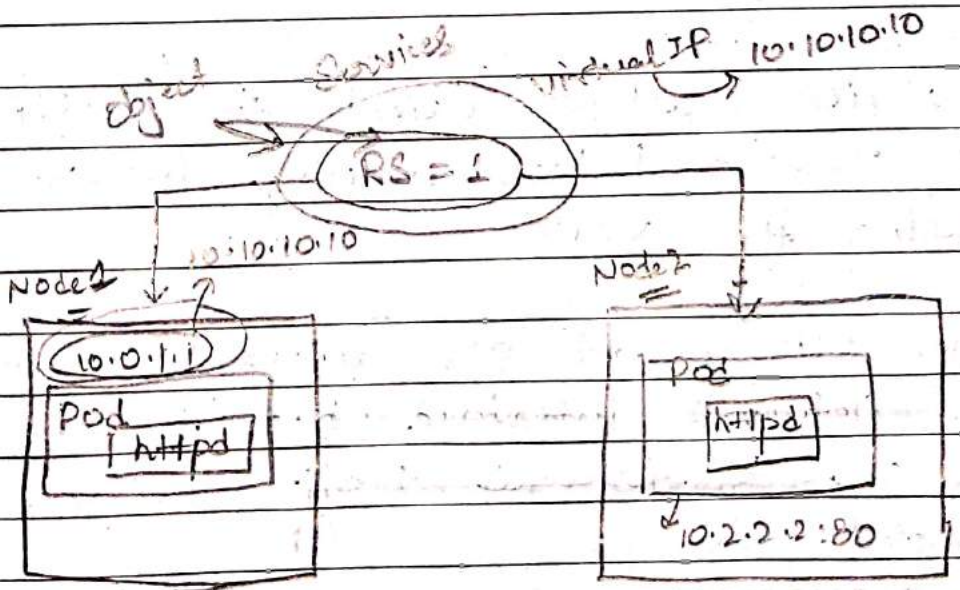
→ kubectl delete -f pod3.yml

Date / /
Page No.

* object - Services → more powerful than Replicaset.
↳ It provides virtual IP to Replicaset

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

→ This leads to a problem: if some set of pods (call them 'backends') provides functionality to other pods ('call the frontend') inside your cluster, how do the frontends find out & keep track of which IP address to connect to, so that the frontend can use the backend part of the workload.




X If it fails.

→ When using RC, pods are terminated & created during scaling or replication operations

→ When using deployments, while updating the image version the pods are terminated & new pods take the place of other pods.

→ Pods are very dynamic i.e. they come & go on the K8s cluster & on any of the available nodes & it would be difficult to access the pods as the pods IP changes once it's recreated.

→ Services object is an logical bridge between pods & end users, which provides virtual IP (VIP)

→ Services allows client to reliably connect to the container 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 & 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 IP's 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 services.

→ Creating a service will create an endpoint to access the pods/application in it.

→ Services can be exposed in different ways by specifying a type in the service spec.

- Cluster IP (By default)
- NodePort
- Load Balancer
- Headless.

} type of Services

→ Load Balancer → Created by cloud Provider that will route external traffic to every node on the nodeport. (Eg. ELB on AWS)

→ Headless → Creates several endpoints that are used to produce DNS Records. Each DNS record is bound to a Pod.

→ By default services can run only between ports 30,000 - 32767.

→ The set of pods targeted by a service is usually determiner by a selector.

① Cluster IP

↳ Exposes VIP only reachable from within the cluster.

→ Mainly used to communicate between components of micro services.

Lab: vi deployhttpd.yaml

```
kind: Deployment
apiVersion: apps/v1
metadata:
    name: mydeployments
spec:
    replicas: 1
    selector:          # tells the controller which pods to
    matchLabels:                    watch/belong to
        name: deployment
    template:
      metadata:
        name: testpod1
        labels:
          name: deployment
      spec:
        containers:
          - name: c00
```

```
    image: httpd
    ports:
    - containerPort: 80
```

→ `kubectl apply -f deployhttpd.yml`

→ `kubectl get pods`

→ `kubectl get pods -o wide`

→ `curl 172.17.0.5:80` → pod IP

↳ o/p → It works!

→ `vi service.yml`

```
kind: Service          # Defines to create Service type object
apiVersion: v1
metadata:
    name: demoservice
spec:
    ports:
      - port: 80          # containers port exposed
        targetPort: 80    # Pods port
    selector:
      name: deployment  # Apply this services to any
    type: ClusterIP       pods which has the specific
                          label.
```

Specifies the service type i.e. Cluster IP or Nodeport.

→ kubectl apply -f service.yml

→ kubectl get svc

→ curl 10.103.231.98:80
  → cluster virtual/static IP.

→ kubectl get pods

→ kubectl delete pod podname

→ kubectl get pods

→ curl 10.103.231.98:80
  → Port
  → virtual IP
  → O/P → It works!

→ kubectl delete -f service.yml
  kubectl delete -f deployhttpd.yml

## ② Node Port → cluster IP is mapped with port.

↳ Makes a service accessible from outside the cluster.

→ Exposes the service on the same port of each selected node in the cluster using NAT.

Labs

→ first create a file `deployhttpd.yml`

→ `kubectl apply -f deployhttpd.yml`

→ `vi svc.yml`

↳ in previous service.yml file change following things :—

`type: NodePort`

↳ `kubectl apply -f svc.yml`

→ `kubectl get svc`

↳ O/P → PORT(S) → 80:31341/TCP

→ `kubectl describe svc demoservice`

↳ name of service.

→ Public DNS: PORT(S)

↳ O/P → It works!

→ kubectl delete -f svc.yml

## * 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 it will not have any of the old data.

→ To overcome this problem, kubernetes uses volumes. A volume is essentially a directory backed by a storage medium. The storage medium & its content are determined by the volume type.

→ In kubernetes, a volume is attached to a Pod & shared among the containers of that Pod.

→ The volumes has the ~~volume~~ same life span as the Pod, & it outlives the containers of the Pod this allows data to be preserved across container Restart.

# Volume Types

→ A volume type decides the properties of the directory, like size, content etc. some eg. of volume types are :-

1. Node-local type such as emptydir & hostpath
2. File sharing type such as nfs. → n/w file sharing system.
3. Cloud provider-specific types like awselastic-blockstore, azuredisk.
4. distributed file system types, for example glusterfs or cephfs.
5. special purpose types like secret, git repo.

## * Empty Dir

↳ 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, & exists as long as that pod is running on that node.

→ As the name says, it is initially empty.

→ Containers in the pod can all read & write the same file in the emptydir Volume, though that volume can be mounted at the same or different paths in each containers.

→ When a pod is removed from a node for any reason, the data in the emptydir is deleted forever.

→ A container crashing does not remove a pod from a node, so that data in an emptydir volume is safe across container crashes.

Lab → vi emptydir.yml

```
apiVersion: v1
kind: Pod
metadata:
    name: myvolemptydir
spec:
  containers:
    -name: c1
     image: centos
     command: ["/bin/bash", "-c", "sleep 15000"]
     volumeMounts: #mount difination inside the container
        -name: xchange
         mountPath: "/tmp/xchange"

    -name: c2
     image: centos
```

```
command: ["/bin/bash", "-c", "sleep 10000"]
volumeMounts:
    - name: xchange
      mountPath: "/tmp/data"
volumes:
  - name: xchange
    emptyDir: {}
```

→ kubectl apply -f emptydir.yml

→ kubectl get pods.

→ To go inside container

→ kubectl exec myvolemptydir -c c1 -it -- /bin/bash

myvolemptydir → pod name
c1 → container name

↳ c1 → ~~cd/tmp~~ cd /tmp
→ ls
→ cd xchange/
→ vi abc.yml

technical Guftgu zindabad.
esc. :wq

→ ls
o/p → abc.yml.

→ exit → to exit container.

→ kubectl exec myvolemptydir -c c2 -it -- /bin/bash

→ cd /tmp

→ ls

→ O/P → data

→ cd /data

→ ls

↳ abc.yml

→ cat abc.yml

→ vi abc.yml

→ i am container 2.

esc :wq

→ Go to container 1, inside xchange folder to see the changes done in container 2.

→ kubectl delete -f emptydir.yml

## * HostPath

↳ Use this when we want to access the content of a Pod / container from hostmachine

→ A hostpath volume mounts a file or directory from the host node's filesystem into your pod.

Lab → vi hostpath.yml

```
apiVersion: V1
kind: Pod
metadata:
    name: myVolhostpath
spec:
  containers:
  - image: centos
    name: testc
    command: ["/bin/bash", "-c", "sleep 15000"]
    VolumeMounts:
     - mountPath: /tmp/hostpath
       name: testvolume.

  Volumes:
   - name: testvolume
     hostPath:
       path: /tmp/data.
```

→ kubectl apply -f hostpath.yml.

→ kubectl get pods.

→ ls /tmp

↳ O/P → data → check for data directory.

→ cd /tmp/data

↳ ls

↳ /tmp/data # kubectl get pods.

→ /tmp/data # kubectl exec myvolhostpath -- ls /tmp/hostpath

→ exec myvolhostpath -- ls /tmp

→ /tmp/data # echo "i love techincal gutgu" > myfile.

→ pwd

↳ O/P → tmp/data

→ kubectl exec myvolhostpath -- ls /tmp/hostpath

OR

↳ kubectl exec myvolhostpath -- cat /tmp/hostpath/myfile.

↳ i love techinical Gutgu.

→ ~~cd /tmp/hostpath/~~

→ kubectl exec myvolhostpath -it -- /bin/bash

↳ [root@myvolhostpath /]# cd /tmp /hostpath

[ hostpath ]# ls  O/P-s myfile.

[ / ]#

[: hostpth]# ~~exit~~ echo "this is my 2nd file" > file2

→ ls [ hostpath ]# ls

→ file2 myfile

→ [ hostpath ]# exit.

→ ls

→ cat file2

↳ O/P-s this is my 2nd file.

# Lec 52 → Persistent Volume & Liveness Probe.

↳ It is an object

Date / /
Page No.

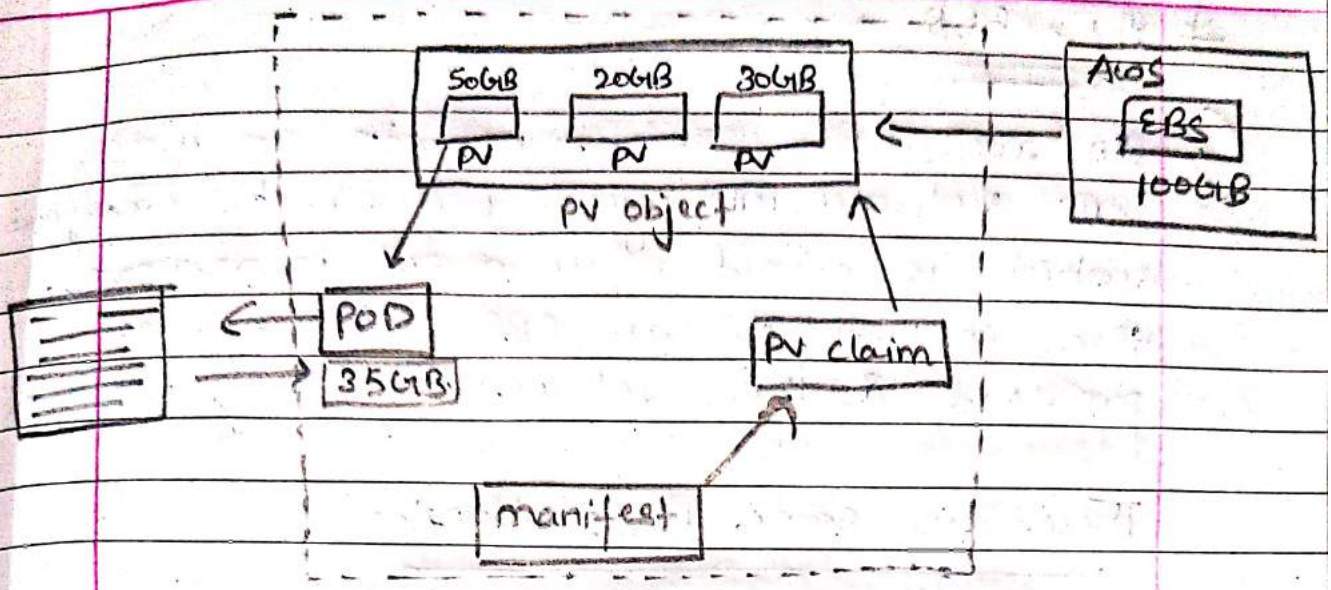
↳ In a typical IT environment, storage is managed by the storage/system administrator. The end user will just get instructions to use the storage, but does not have to worry about the underlying storage management.

→ 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 resolves this problem with the Persistent Volume (PV) Subsystem.

→ A persistent volume (PV) is a cluster-wide resource that you can use to store data in a way that it persists 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 APIs for users & administrator to manage & consume storage. To manage the volume, it uses the Persistent Volume API resource type and to consume it, uses the PersistentVolumeClaim API resource type.




## * Persistent Volume Claim.

↳ In order to use a PV you need to claim it first, using a Persistent Volume Claim (PVC)

→ The PVC request a PV with your desired specification (size, access modes, speed etc) from kubernetes & once a suitable Persistent Volume is found, it is bound to a Persistent Volume Claim.

→ After a successful bound to a pod, you can mount it as a volume.

→ Once a user finishes its works, the attached Persistent Volume can be released. The underlying PV can then be reclaimed & Recycled for future usage.

# * AWS EBS

An aws EBS volume mounts an Aws EBS Volume into your Pod unlike emptyDir, which is erased when a pod is removed, the contents of an EBS Volume are preserved & the volume is merely unmounted.

There are some Restrictions!-

1. The nodes on which Pods are running must be aws EC2 instances.
2. Those instances need to be in the same region & Availability zone as the EBS volume.
3. EBS only supports a single EC2 instance mounting a volume.

Labs ⇒ vi mypv.yml. (creating persistent vol.)

```
apiVersion: V1
kind: PersistentVolume.
metadata:
    name: myebsvol
spec:
    capacity:
        storage: 1Gi
    accessModes:
        - ReadWriteOnce
```

```
persistentVolumeReclaimPolicy: Recycle
awsElasticBlockStore:
    volumeID: vol-0dca9336795f520
    fsType: ext4
```

Need to paste EBS vol. ID.

→ `kubectl apply -f mypv.yml`

→ `kubectl get pv`

→ `vi mypvc.yml`

claiming persistent vol.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: myebsvolclaim
spec:
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
            storage: 1Gi
```

→ `kubectl apply -f mypvc.yml`

→ `kubectl get pvc`

→ `vi deploypvc.yml` → To use that PV

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: pvdeploy
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 10000"]
           volumeMounts:
           - name: mypd
             mountPath: "/tmp/persistence
         volumes:
           -name: mypd
            persistentVolumeclaim:
               ClaimName: myebsvolclaim
```

→ kubectl apply -f deploypvc.yaml.

→ exec pvdeploy-8585679-8d4hf -it --/bin/bash
↳ pod name

↳ cd /tmp/persistent/
→ ls
→ vi testfile
↳ hello world!
↳ ls
O/P ⇒ testfile.
→ exit.

→ kubectl delete pod podname

→ kubectl get pods ↳ After deletion a new pod will be created

→ kubectl podname -it --/bin/bash

→ cd /tmp/persistent/.
→ ls
↳ testfile. → file will be there as it is.

→ cat testfile.
exit.
→ kubectl delete -f podnames.

# * 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 & ready to serve traffic, kubernetes provides for a range of healthy checking mechanism.

→ Health checks or probes are carried out by the kubelet to determine when to ~~restart~~ recreate a container (for liveness Probe) & used by services & 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 progress. Restoring a container in such a state can help to make the application more available despite bugs.

→ One use of readiness probes is to control which pods are used as backends for services. When a pod is not ready, it is removed from service load Balancers.

→ For running healthchecks, we would use cmd specific to the application.

→ If the cmd succeeds, it returns 0, & the kubelet considers the container to be alive & healthy. If the command returns a non-zero value, the kubelet kills the pod & recreate it.

Labs = vi liveness.yml

```
apiVersion: v1
kind: Pod
metedata:
    labels:
      test: liveness
      name: mylivenessProbe
  spec:
    containers:
      -name: liveness
       image: ubuntu
       args:
         - /bin/sh
         - -c
         - touch /tmp/healthy; sleep 1000

  liveness probe:
        exec:
        command:
        - cat
        - /tmp/healthy
          initialdelayseconds: 5
            periodseconds: 5
            timeoutseconds: 30
```

→ kubectl apply -f liveness.yml

→ kubectl get pods.

→ kubectl describe pod podname

↳ kubectl exec podname -it -- /bin/bash

↳ → cat /tmp/healthy
→ echo $?
O/P → 0

→ cat /tmp/healthyfying
O/P: not found
→ echo $?
O/P → 1

→ kubectl delete -f liveness.yml

# Lec 53:→ Configmap and Secrets

Date / /
Page No.

→ 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 & then go through the complete deployment process.

→ Hence these configurations should be decoupled from image content in order to keep containerised application portable.

→ This is where kubernetes configmap comes handy. It allows us to handle configuration files much more efficiently.

→ Configmap are useful for storing & sharing non-sensitive unencrypted configuration information use secrets otherwise.

→ Configmap can be used to store fine-grained information like individual properties or entire configfiles.

→ Configmap are not intended to act as a replacement for a properties file.

→ configmap can be accessed in following ways :-

① As environment variables.
② As volumes in the Pod.

→ kubectl create configmap <mapname> --from-file = <file to read>

## * Secrets

↳ You do not 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 & reliable way with the following properties :-

→ Secretes are namespaced objects, that is exist in the context of a namespace.

→ You can access them via a volume or an environment variable from a container running in a Pod.

→ The secrets data on nodes is stored in tmpfs volumes (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 hard drive.

→ A per-secret size limit of 1MB exists.

→ The API server stores secrets as plaintext in etcd.

Secrets can be created

→ from a text file.
→ from a yaml file.

Lab: vi sample.conf

this is my configuration file for any application.

:wq ⇒

→ kubectl create configmap mymap --from-file=sample.conf

kubectl get configmap

→ kubectl describe configmap mymap.

→ vi deployconfigmap.yml

```
apiVersion: v1
kind: Pod
metadata:
    name: myvolconfig.
spec:
    containers:
      -name: c1
       image: centos
       command: ["/bin/bash", "-c",
       "while true; do echo Technical-Guftgu
        ; sleep 5 ; done"]

       VolumeMounts:
            -name: testconfigmap
             mountPath: "/tmp/config"
```

the config files will be mounted as ReadOnly by default here.

```
       volumes:
         -name: testconfigmap
          configMap:
            name: mymap
            items:
              -key: sample.conf
               path: sample.conf.
```

this should match the config map name created in the first step.

→ kubectl apply -f deployconfigmap.yml.

→ kubectl get pods.

→ kubectl exec podname -it -- /bin/bash
(podname → myvolconfig)

↳ @myvolconfig /]→ cd /tmp
/]→ ls
/]→ ls -l
/]→ cd config
/]→ ls
sample.conf.
/]→ cat sample.conf.

→ exit

→ kubectl delete -f deployconfigmap.yml.

# Lec54 Namespaces, Limits & Request

Date / /
Page No.

## 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, groups or the namespace in general.

→ Kubernetes namespace help different projects, teams or customer to share a Kubernetes cluster & provides:—

↳ A scope for every names.

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

→ By default, a Kubernetes cluster will instanti-ate a default namespace when provisioning the cluster to hold the default set of pods, services and deployments used by the cluster.

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

→ Most Kubernetes resources (eg. pods, services, replication controllers & others) are in some namespaces & low-level resources such as nodes & persistent volumes, are not in any namespace. (not comes in namespaces)

→ Namespaces are intended for use in environments 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 at all.

* Create new Namespace

↳ Let as assume we have shared K8s cluster for Dev & 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. In 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 & deployment.

cmd → `kubectl get namespace.`

vi devns.yml

```
apiVersion: V1
kind: Namespace
metadata:
  name: dev
  labels:
    name: dev
```

→ kubectl config set-context $(kubectl config current-context) --namespace=dev

→ kubectl apply -f devns.yml

→ kubectl get namespace.

→ vi pod.yml.

```
kind: Pod
apiVersion: V1
metadata:
  name: testpod
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Technical Guftgu; sleep 5; done"]
```

→ kubectl apply -f pod.yml -n dev

→ kubectl get pods → O/P → No resource found in default namespace.

→ `kubectl get pods -n dev`

↳ O/P → testpod running.

→ To delete namespace.

↳ `kubectl delete -f pod-yml -n dev`

→ To set default namespace

↳ `kubectl config set-context $(kubectl config current-context) --namespace=dev`

→ `kubectl config view | grep namespace:`

↳ To check details of namespace.

→ kubectl get pods -n default

→ `kubectl delete -f pod.yml.`

Date / /
Page No.

# * Managing Compute Resources for Containers.

↳ A pod in kubernetes will run with no limits on CPU & memory.

→ You can optionally specify how much CPU and memory (RAM) each container needs.

→ Scheduler decides about which nodes to place pods, only if the Node has enough CPU resources available to statisfy, the pod CPU request.

→ CPU is specified in units of cores & memory is specified in units of Bytes.

→ Two types of containers can be set for each resource type - Request & limits.

↳ A request is the amount of that resources that the system will guarantee for the container & kubernetes will use this value to decide on which node to place the pod.

→ A limit is the max. amount of resources that kubernetes will allow the container to use. In the case that request is not set for a container, it default to limits. If limit is not set then it default to 0.

→ CPU values are specified in 'millicpu' & memory in MiB.

⇒ if request = not mention
limit = mention

then,

request = limit.

↳ vise-versa not true.

## * Resource Quota

↳ A kubernetes cluster can be divided into namespaces. If a container is created in a namespace that has a default CPU limit, & the container does not specify its own CPU limit, then the container is assigned the default CPU limit.

→ Namespaces can be assigned Resource Quota objects, this will limit the amount of usage allowed to the objects in that namespace.

You can limit :—

↳ ① compute (CPU)
② Memory. (RAM)
③ Storage.

* Here are two restrictions that a resource quota imposes on a namespace.

→ Every container that runs in the namespace must have its own CPU limit.

→ The total amount of CPU used by all containers in the namespace must not exceed a specified limit.

Lab → vi podresources.yml

```
apiVersion: V1
kind: Pod
metadata:
    name: resources.        → pod name.
spec:
    containers:
      -name: resource       → container name.
       image: centos
       command: ["/bin/bash", "-c", "while true;
                 do echo Technical-Gruftgu; sleep 5;
                 done"]
       resources:
         requests:
          memory: "64Mi"
          cpu: "100m"
         limits:
          memory: "128Mi"
          cpu: "200m"
```

→ kubectl apply -f podresources.yml.

→ kubectl describe pod resources.
                  ↳ podname.

→ kubectl delete -f podresources.yml.

# Lec-55 → Resource Quota & Horizontal Scaling.

→ prod-Namespace.

limit CPU: "600m"
limit memory: "800Mi"

request CPU: "200m"
request memory: "500Mi"

Default Range.

CPU:
  request - 0.5
  limit - 1
memory:
  request - 500M
  limit - 1G.

# * Horizontal Pod Autoscaler

↳ Kubernetes has the possibility to automatically scale pods based on observed CPU utilization, which is horizontal Pod Autoscaling object.

→ Scaling can be done only for scalable objects like controller, deployment or Replica set.

→ HPA is implemented as a kubernetes API resource & a controller.

→ The controller periodically adjust the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.

→ The HPA is implemented as a controlled loop with a period controlled by the controller manager's - horizontal-pod-autoscaler -sync-period flag (Default value of 30 sec)

→ During each period, the controller manager queries the resource utilization against the metrics specified in each horizontalPod-Autoscaler definition.

→ for per-pod resource metrics (like CPU), the controller feteches the metric from the resource metrics API for each pod targeted by the HorizontalPodAutoscaler.

→ Then if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each pod.

→ If a target raw-value is set, the raw metric values are used directly. The controller then takes the mean of the utilization on the raw value across all targeted pods, & produces a ratio used to scale the number of desired replicas.

→ Cooldown period to wait before another downscale operation can be performed is controlled by --horizontal-pod-autoscaler-downtime-stabilization flag (Default value of 5 min).

→ Metric server needs to be deployed in the cluster to provide metric via the resource metric API.

→ --kubelet-insecure-tls.

autoscale → kubectl autoscale deployment mydeploy --cpu-percent=20 --min=1 --max=10

(deployment → object)

# Lec 56 → Jobs, Init Containers & Pod Lifecycle.

Date / / Page No.

## * Jobs

↳ we have replicasets, Daemonsets, statefulsets and deployments they all share one common property: they ensure that their pods are always running if a pod fails, the controller restarts it or reschedules it to another node to make sure the application the pods is hosting keeps running.

## * Use Cases

1. Take Backup of a DB
2. Helm charts uses jobs.
3. Running Batch Processes.
4. Run a task at an schedule interval.
5. Log rotation.

Labs

```
apiVersion: batch/v1
kind: Job
metadata:
    name: myjob
spec:
  template:
    metadata:
      name: myjob

  spec:
    containers:
      - name: c1
```

```
image : Centos7
command : ["bin/bash", "-c", "echo TG ; sleep 5"]
restart policy : Never.
```

→ Job does not get deleted by itself, we have to delete it.

→ `kubectl delete -f job.yml`

## * Parallelism

```
apiVersion: batch/v1
kind: Job
metadata:
    name: testjob
spec:
  parallelism: 5        # Runs for Pod in Parallel.
  activeDeadlineSeconds: 10   # Timeout after 30sec.
  template:
    metadata:
      name: testjob
      spec:
        containers:
          -name: counter
           image: centos7
           command: ["bin/bash", "-c", "echo
            Technical-Guftgu ; sleep 30"]
        restartpolicy: Never.
```

* The Cron Job Pattern

↳ If we have multiple nodes hosting the application for high availability, which nodes handles cron?

→ what happens if multiple identical cron job run simultaneously.

Lab→

```
apiVersion: batch/v1beta1
kind: CronTab
metadata:
   name: Bhupi
spec:
 schedule: "* * * * *"
 jobTemplate:
      spec:
        template:
            spec:
              containers:
              -image: ubuntu
                name: Bhupi
                command: ["/bin/bash", "-c",
"echo Technical-Guftgu", sleep 5"]

 restartPolicy: Never.
```

## * Init Containers

→ Init containers are specialised containers that run before app containers in a pod.

→ init containers always run to completion.

→ If a pod's init container fails, Kubernetes repeatedly restart the pod until the init container succeeds.

→ init containers do not support readiness Probe.

## * Use Cases

↳ Seeding a Database

→ Deploying the application launch until the dependencies are ready.

→ clone a git repository into a volume.

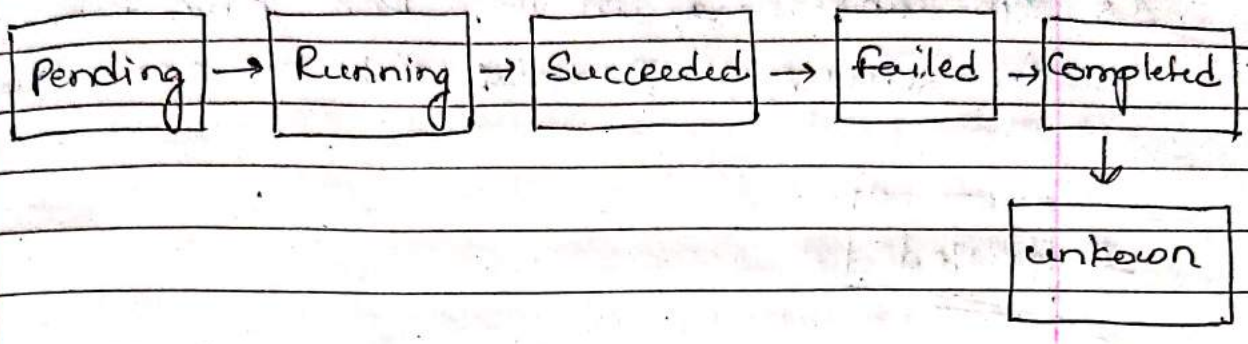
→ Generate configuration file dynamically.

Date / /
Page No.

Lab:

```
apiVersion: v1
kind: Pod
metadata:
  name: initcontainer
spec:
  initContainers:
    - name: c1
      image: centos
      command: ["/bin/sh", "-c", "echo LIKE & SUBSCRIBE Techincal Cruffgu> /tmp/xchange/testfile; sleep 30"]
      volumeMounts:
        - name: xchange
          mountPath: "/tmp/xchange"
  containers:
    - name: c2
      image: centos
      command: ["/bin/bash", "-c", "while true; do echo 'cat /tmp/data/testfile'; sleep 5; done"]
      volumeMounts:
        - name: xchange
          mountPath: "/tmp/data"
  volumes:
    - name: xchange
      emptyDir: {}
```

# * Pod Life Cycle




→ The phase of a Pod is a simple, high-level summary of where the pod is in its life-cycle.

## * Pending

↳ The pod has been accepted by the K8s system, but it's not running.

→ One or more of the container images is still downloading.

→ If the pod cannot be scheduled because of resource constraints.

## * Running

↳ The pod has been bound to a node.

→ All of the containers have been created

→ Atleast one containers is still running or is in the process of starting or restarting.

* **Succeded**

↳ All containers in the Pod have terminated in success, & will not be restarted.

* **failed**

↳ All containers in the Pod have terminated & atleast one container has terminated in failure.

→ The container either exited with non-zero status or was terminated by the system.

* **Unknown**

↳ State of the pod not be obtained

→ Typically due to an error in network or communicating with the host of the Pod.

* **Completed**

↳ The pod has run to completion as there's nothing to keep it running.

Eg → completed jobs.

## * Pod Conditions

A pod has a Pod Status, which has an array of Pod conditions through which the pod has or has not passed.

→ Using 'kubectl describe pod PODNAME' you can get condition of a Pod.

These are the possible types.

(1) PodScheduled → The pod has been scheduled to a node.

(2) Ready → The pod is able to serve request & will be added to the load Blancing pools of all matching services.

(3) Initialized → All init containers have started Succesfully.

(4) Unscheduled → The scheduler cannot schedule the pod right now, for eg. due to lacking of Resources or other constraints.

(5) ContainerReady → All containers in the pod are ready.

# [c57] → What is Helm & Helm Chart?

Date / /
Page No.

## * HELM (Packaging Manager)

↳ Introduced first time in 2015.

→ Helm helps you manage K8s applications with Helm chart which helps you define, install & upgrade even the most complex kubernetes application.

→ Helm is the K8s equivalent of yum or apt.

→ Helm is now an official K8s Project and is part of CNCF.

→ The main Building Block of Helm Based deployments are Helm charts these charts describe a configurable set of dynamically generated K8s resources.

→ The chart can either be stored locally or fetched from remote chart Repositories

* Why Use Helm?

↳ Writing and maintaing kubernetes YAML manifest for all the required kubernetes objects can be a time consuming & tedious task. For the simplest of deployments, you would need atleast 3 YAML manifest with duplicated & Hardcoded values.

Helm simplifies this process & create a single package that can be advertised to your cluster.

→ HELM 3 was released in NOV 2019. HELM K8s automatically maintains a database of all versions of your releases. So whenever something goes wrong during deployment, rolling back to the previous version is just one command away.

Date / /
Page No.

## * Some keywords to Understand 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 K8s cluster. Think of it like the kubernetes equivalent of a Homebrew formula, apt, yum or RPM file.

OR

Helm charts are simply K8s YAML manifests combined into a single package that can be advertised to your K8s cluster. (install/refer/upgrade)

### RELEASE

↳ A release is an instance of a chart running in a K8s 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 can install that chart twice. Each one will have its own release, which will in turn have its own release name.

| Revision | Request |
|---|---|
| 1 | Install chart |
| 2 | Upgrade to 1.1 |

→ Helm keep tracks of all chart execution (Install / Upgrade / Rollback)

→ REPOSITORY → Location where packaged charts can be stored & shared.

## * Normal Kubernetes deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Release-name-springboot
spec:
  replicas: 2 → hardcoding.
  selector:
    matchLabels:
      app.kubernetes.io/name: springboot.
```

VS

## HELM deployment YAML

↳
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Release-name-springboot
  Spec:
  replicas: {{values.replicaCount}}
                ↳ no hardcoding
  Selector:
  matchLabels:
    app.kubernetes.io/name: springboot
```

## * HELM-3 ARCHITECTURE

↳ 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 & unilaterally handles the package management process.

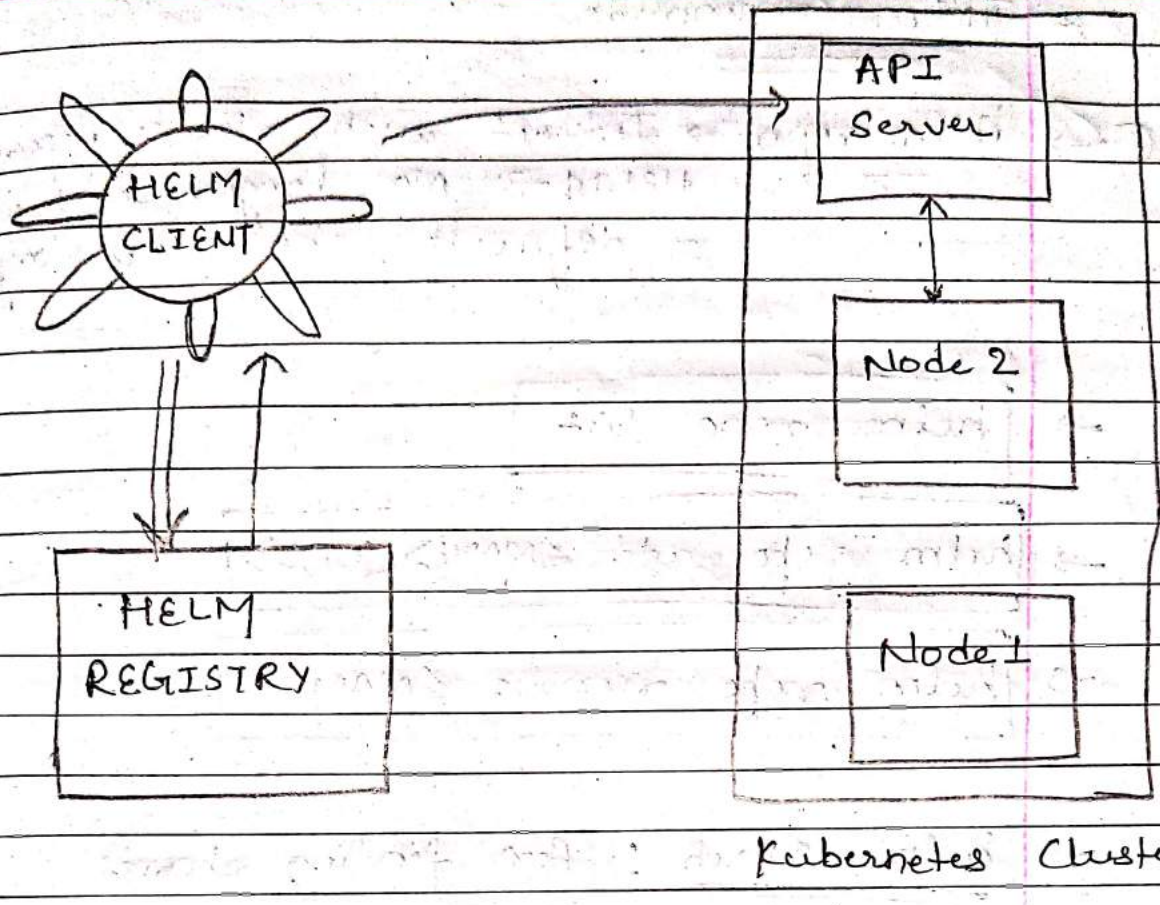
HELM client & Library
↳ Go language

HELM - 2 → Tiller

↳ Client - Server

drawback → RBAC ✗

Keys Port Expose



# Lec 58 :- HELM Commands & Concepts

Date / /
Page No.

* Helm Commands

① Helm repo → Interact with charts Repository
HELM-3 No longer ships with a default chart Repository.

→ helm repo list

→ helm repo add <NAME> <URL>

→ helm repo remove <NAME>

→ helm search : for finding charts

for Eg :- helm search repo <chart>

→ helm show → Information about a chart

for Eg helm show <values | chart | readme | all> <chartName>

→ helm install :- Install a package.

for Eg helm install <Release Name> <chart Name>

```
helm install --dry-run testchart stable/tomcat
```

Date / /
Page No.

→ wait untill all subjects are ready

for eg → `helm install mychart stable/tomcat --wait --timeout 10s`

→ helm install --wait --timout 20s testtomcat stable/tomcat,

→ `helm create` → Create a new chart with given name.

for eg → `helm create helloworld`.

```
Helloworld
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── services.yaml
│   ├── Serviceaccount.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml.
```

when we create chart then it also created by default.

→ `helm delete testjenkins`

→ To install specific chart/app version

```
helm install testchart stable/tomcat
        --version 0.4.0
```

## * Helm Commands

↳ There are two ways to pass configuration data during install.

↳ **--set**: Specify overrides on the commands line

→ **--values**: Specify a YAML file with overrides (or -f) (forcefully)

For Eg ⇒

```
helm install mychart stable/tomcat
--set servicetype = NodePort
```

```
helm install testchart stable/Jenkins
--set master.servicetype = NodePort
```

→ 'helm get' Information about from a Named release.

```
helm get <all | manifest | values> <Rel. name>
```

For Eg ⇒ helm get all mychart

→ helm show → before installation to check what parametes are given.

→ helm list → List all the Named Release

→ helm status → Display the status of the Named Release.

→ helm status <Releasename>

Eg: helm status mychart.

→ helm history : → Feteh Release history

helm history <Release Name>

→ helm delete : → Uninstall the deployed Release

helm delete <Release Name>

→ helm upgrade : → "Upgrade a Release"

helm upgrade <ReleaseName> <chartName>

for Eg: helm upgrade mychart stable/tomcat

→ helm rollback → Rollback a Release to any previous version.

helm rollback <ReleaseName> <Revision>

for Eg: helm rollback mychart.

→ helm pull → download a chart from a repository.

helm pull <chartName> # download the tar

helm pull --untar <chartname> # download & untar

Eg → helm pull --untar stable/tomcat

→ Install from a local chart archieve

Eg, helm install mychart tomcat-0.4.3.tgz

→ Install from an unpacked chart directory

for Eg → helm install mychart

→ Install from a full URL

Eg → helm install mychart <URL>