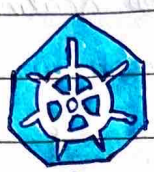
Lec - 01 :- Complete theory of Kubernetes.



7 lines in the logo.
(project 7).

(K8s)

(Container Management tool)
↳ (High Availability.)
↳ (Scalibity)

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

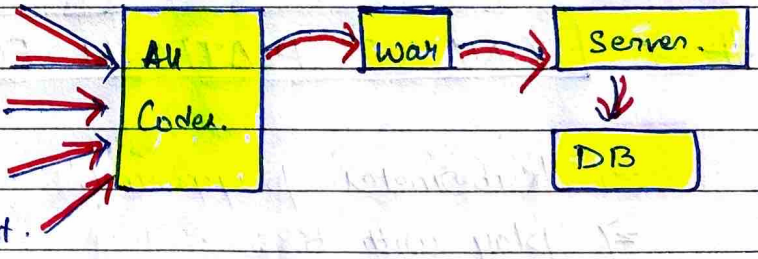
previously :- Monolithic Application.

Single Stone.

→ (Scaling Not possible)

Facebook.

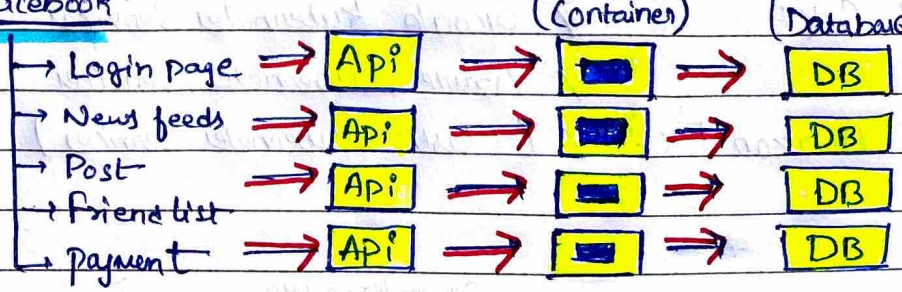
- Login page
- News feeds
- Post
- Friendlist
- Login payment.



Now :- Micro Services.

→ (works independly.)

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 application and Services on there cluster.

⇒ 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 Kubernets Classroom.

⇒ CLOUD BASED K8s SERVICES

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

⇒ KUBERNETS INSTALLATION TOOL

⇒ Minicube.
⇒ Kubeadm.

Problems with Scaling up the Containers.

⇒ Containes Can Not Communicate with Each other.

⇒ Autoscaling 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 N/w)

⇒ Auto Scaling (vertical & horizontal) ⇒ Auto - healing. — Prefered.

⇒ Load Balancing ⇒ platform Indepent (cloud / virtual / physical)

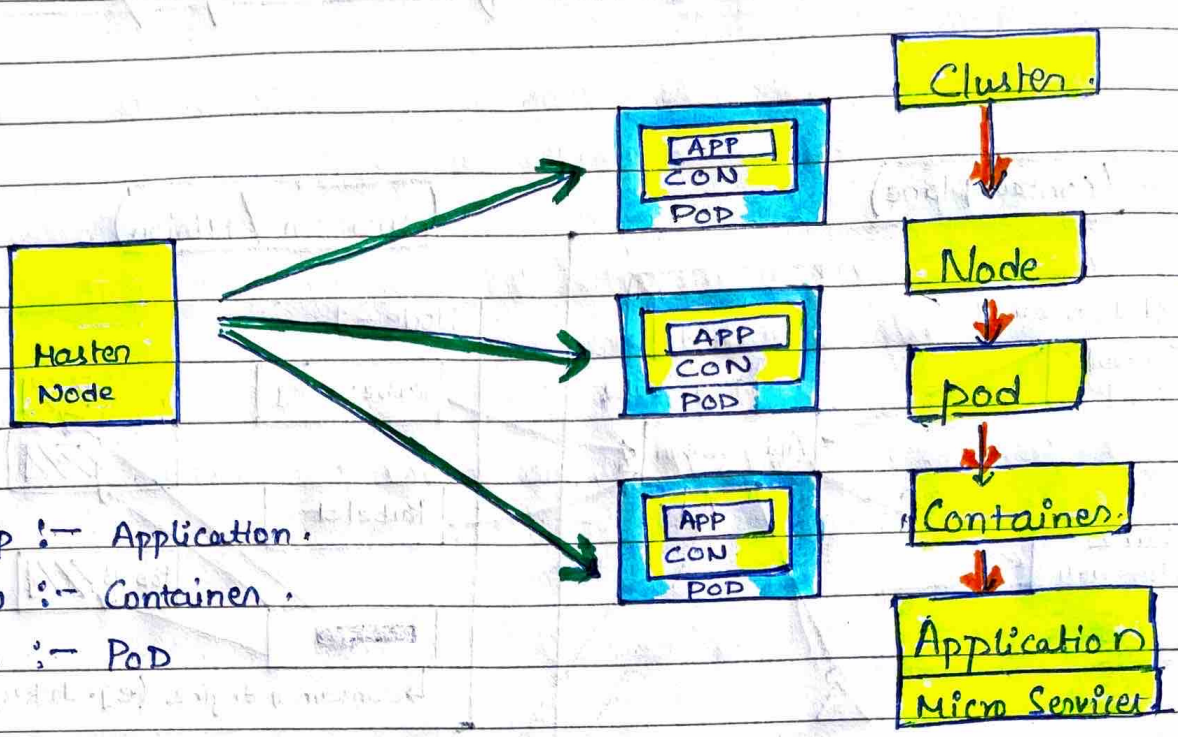
⇒ Fault Tolerence (Node / pod failure)

⇒ Rollback (going back to previous Version)

⇒ Health Monitoring of Containers.

⇒ Batch Execution (one time Sequential, Parellel)

FEATURES	KUBERNETES	DOCKER SWARM
Installation and Cluster Configuration	Complicated and time Consuming	Fast and Easy.
Supports	K8s Can work with almost all Containers types like Rocket, Docker Container	Work with docker only.
GUI	GUI Avilable	Gui Not Avilable.
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 updates & Service health Monitoring throughout the update.
Auto Scaling	Support Vertical and Horizontal Autoscaling	Not Support Auto Scaling.
Logging and Monitoring.	inbuilt tool present for Monitoring	used 3rd party tools like Splunk.

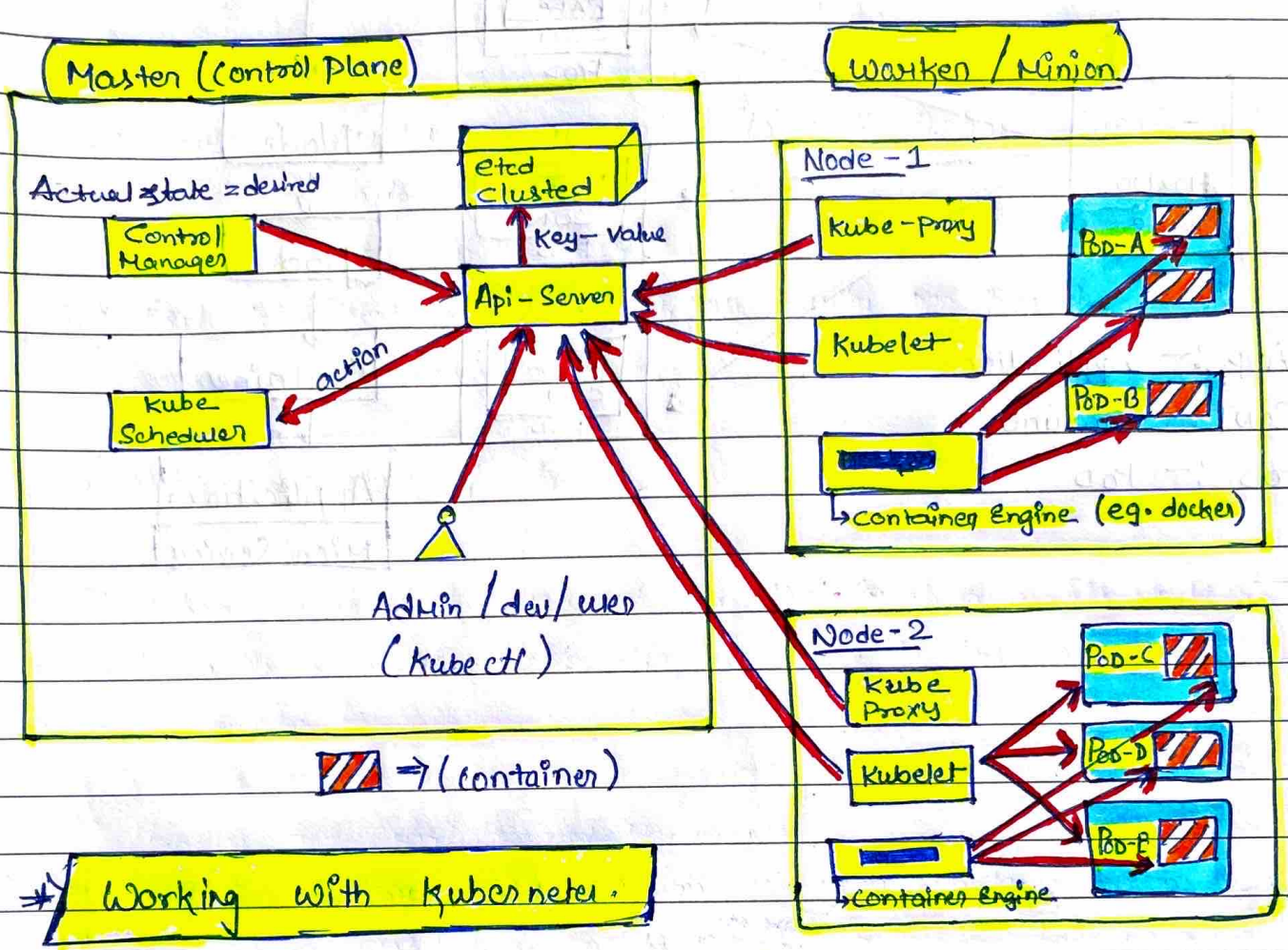


App :- Application.

Con :- Containen.

Pod :- Pod

Lec-02 Architecture of Kubernetes & its demo.



* Working with Kubernetes.

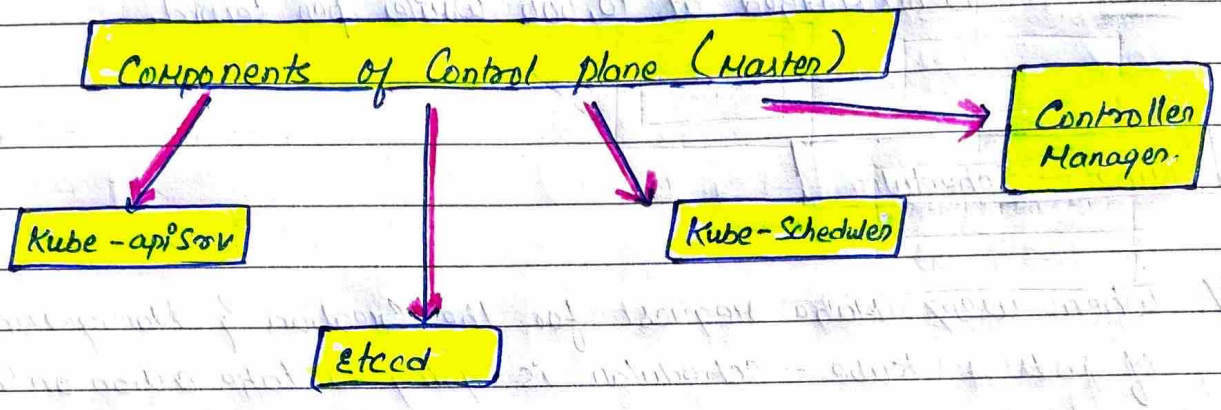
⇒ We Create Manifest (.yml)
⇒ Apply this to Cluster (to Master) to bring into desired state.
⇒ pod runs on node, which is controlled by Master.

Role of Master Node.

⇒ Kubernetes cluster contains Container running on Bare Metal / VM instances / cloud instances. / all Mix.
⇒ Kubernetes designates one or more of these as Master and all other as Worker.

=> The Master is now going to run set of K8s process. these process will ensure smooth functioning of cluster. These process are called "Control plane"

=> Can be Multi-Master for high availibity.
=> Master runs Control plane to run cluster smoothly.



Kube-Api-Server → For all communication.

=> This api-server interacts directly with your user (ie we apply .yml or JSON Manifest to kube api-server)

=> This kube-apiserver 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)
=> Source 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 users 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 watches for newly created pods that have no node assinged for every pod that the scheduler, discover, the Schedular becomes Responsible for finding best node for that pod to run on.

⇒ Scheduler gets the information for hardware configuration from file and Schedules the pods on nodes accordingly.

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's stop responding.

Route - Controller → Responsible for setting up network, routes on your cloud.

Service - Controller :— Responsible for load Balancers on your cloud against Services of types 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 pice of Software / process.

Kubelet

- Agent running on the node.
- Listens to Kubernetes Master. (Eg. pod creation Request)
- use PORT 10255
- Sends Sucess and fail report to Master.

Container Engine (Eg. 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 asign (IP) address to pods (Dynamic ip)
- Kube-proxy runs on each Nodes & this Make Sure that each pod will get its own unige ip address.

These 3 Components Collectivily Consest "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 atleast One worker node and Master node.

- In Kubernents, 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

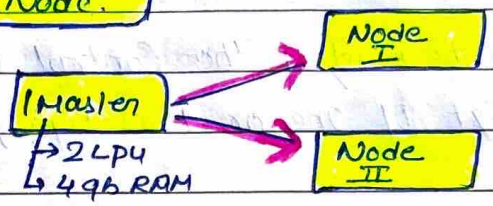
Kubectl :- Single cloud.
Kubeadm :- On premise
Kubefed :- Federated (Hybrid Cloud)

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 (1Master, 2 Nodes)

COMMON COMMANDS For Master Node.



$#) Sudo su
$#) apt-get update.
$#) apt-get install apt-transport-https.

(This https is needed for intra cluster Common particually from control plane to individual pods)

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

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 it's 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.yml.

$#] sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/k8s-manifests/kube-flannel-rbac.yml

*] 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.

=> Kubernets 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 polices, upgrades and fault tolerance.

=> Once you Create the object, the Kubernets System will Constantly work to Ensure that object exist and maintain's cluster's desired state.

=> Every Kubernetes objet includes two nested fields that gover the object Config the Object Spec and the object status.

=> The spec, which we provide, describes your desired state for the objeet – the characteristics that you want the object to have.

=> The status describes the actual state of the object and is supplied and updated by the Kubernets 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.) Replicats
6.) Secrets 7.) ConfigMaps 8.) Deployment 9.) Jobs 10.) DaemonSets.

Relationship between these objects.

- pod Manages Containers.
- Replicaset Manages pods.
- Services expose pod processes to the Outside World.
- ConfigMaps and Secrets helps you configure pods.

Kubernetes Object.

- it represents as JSON or YAML files.
- you Create these and then push them to the kubernetes Api with kubectl.

State of the Object

- Replicas (2/2)
- None
- Volume
- image (Tomcat/ubuntu)
- port
- Startup
- deattached (default)

Kubernetes Objects Managements.

The kubectl Command line tool Supports Several different way to Create and Manage Kubernetes Object.

Management Technique	Operates ON	Recommend Env
Imperative Commands	Live objects	Deployment Project
Declarative object Config	Individual files (YML/JSON)	Production

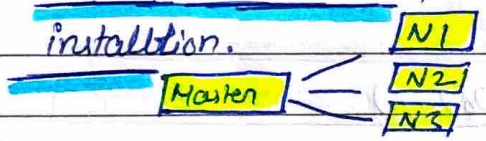
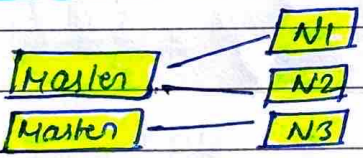
Declarative is about describing what you are trying to achive, without instructing how to do it

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 ~~will~~ until the process is terminated 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 (wid) 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 and Created as new 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)	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.
Single-Node etcd, Single Master, and Multi-worker installation. 	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.
Single node etcd, ~~Single~~ Multi Master and Multiworker install  (HA - High Avibility)	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 ⇒ Launch instance
→ ubuntu 18.04 ⇒ t2 Medium (2 vcpu)

Now access Ec2 via putty ⇒ Login as "ubuntu"
⇒ sudo su
⇒ apt update && apt -y install docker.io

⇒ install Kubectl

- install Minikube.
- apt install Conntrack
- Minikube Status.
- Kubectl Version
-

Then check the nodes details :—

- Kubectl get nodes. ↳ (output will be like)

O/p

NAME	STATUS	ROLES	AGE	VERSION
ip-172-31-34-55	Ready	Master	2Min	v1.20.7

- 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: C00
   image: ubuntu
   Command: ["/bin/bash", "-c", "while true; do echo Hello-Saumm;
              sleep 5; done" ]
restartPolicy: Never  # Default to Always.
```

- Kubectl apply -f pod1.yml (to apply the yml file)

Multi Container POD Environment

```
Kind: Pod
apiVersion : V1
Metadata :
  name : testpod3
Spec :
 Containers :
  - name: C00
    image: ubuntu
    command : ["/bin/bash", "-c", "while true ; do echo Sawan ;
              sleep 5; done"]
  - name : C01
    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 : C00
  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 : testpod 4
Spec :
  Containers :
  - name : C00
    image :
    ports :
    - ContainerPort : 80
```

1) Kubectl apply -f pod2.yml
2) Kubectl get pods. → (To check the pod and Container details)

3) Kubectl describe pod testpod 3 → (describe command to check detail.)
4) Kubectl logs -f testpod 3

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

5) Kubectl logs -f testpod3 -C C00 → (Container name)
6) Kubectl logs -f testpod3 -C C001 → (Container name)

7) Kubectl exec testpod3 it -C C01 --/bin/bash
(Pod name) (Container name)

↳ To login.

Lec-05 - Kubernetes in detail II

Labels and Selectors

=> Lables are the Mechanism you use to Organise Kubernetes Objects.

(Name: Sawan)

=> A label is a key - value pair without any predefined Meaning that can attached to the objects.

=> Multiple lables can be added to a single object.

=> Lables 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 on Env which is used for dev or testing or prod, refer a prod group like Department A - Department B

EXAMPLE of LABLES.

```
Kind: Pod
apiVersion: v1
Metadata:
  name: Sawanpod
  lables:
    Env: development
    class: pods
Spec:
  Containers:
    - name: COO
      Image: ubuntu
      Command: ["/bin/bash","-c" "while true; do Echo Hello-Sawan; sleep 5;
               done"]
```

=> Multiple lables we can add into a single object.

=> kubectl apply -f pod5.yml

=> kubectl get pods --show-lables.
(nodes)
↳ if we want to check the lables we can add nodes option.

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

=> kubectl label pods delhipod MyName = bhupinedn.
(object type) ↳ (object name) ↳ (key) (Selector)

=> kubectl get pods --show-labeles
↳ now, list pods matching a lables.

=> 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 / UID,s 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 labels 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
  projet development.

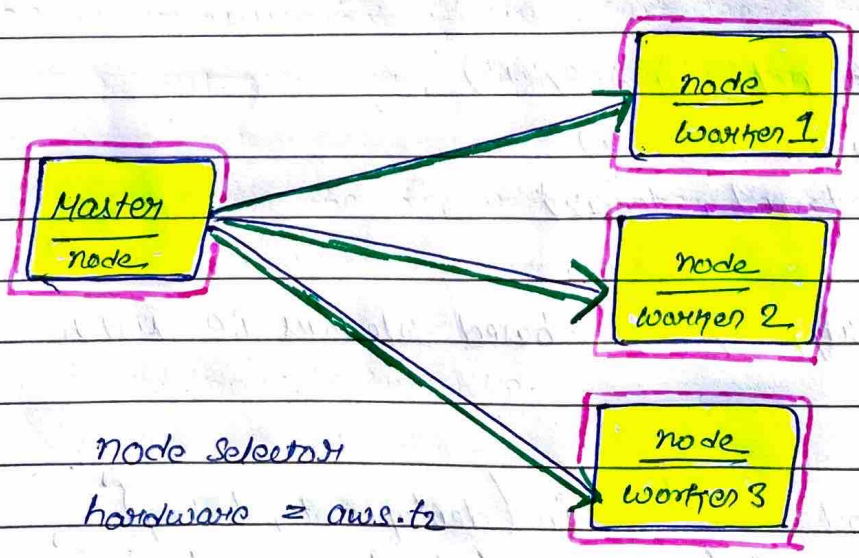
- Set Based : (in notin and exists)
  env in (production, dev)
  env notin (team1, team2)

- 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 = pods, myname = bhupinder

Node - Selector

- One way we can for selecting lables is to Constrain the set of nodes onto which a pod can schedule ie 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 on certain circumstances we might need it
- We can use lables to tag nodes.
- you the nodes are tagged, you can use the label selectors to specify the pods run only of specific nodes.
- First we give label to the node.
- Then ~~we need~~ use node selector to the pod configuration.



Node Selector Example

```
Kind: Pod
apiVersion: V1
Metadata:
  name: nodelables.
  lables:
    env: development
Spec:
  Containers:
    - name: COO
      image: ubuntu
      Command: ["/bin/bash", "-c", "while true; do echo Hello-Sawan;
                sleep 5; done"]
node Selector:
  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 fails.

Load Balancing :— Having Multiple version of a Container enables you to Easily Send traffic to different instonstans to prevent overloading of a single instance or node.

Scaling :- When load does become too much for the numbers of existing instance.

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.

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.

Kind : ReplicationController.

Replication Controller

Kind: ReplicationController → this defines to create the object of Replication type
apiVersion: v1
Metadata:
name: My replica
Spec:
replica: 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 labels.
template: ⟶ template element defines a template to launch a new pod
Metadata:
name: testpod0
labels: ⟶ Selector values needs to match the labels values specified in the pod template.
Myname: Sawan.
Spec:
Containers:
- name: C00
image: ubuntu
command: ["bin/bash", "-c", "while true"; do echo hello - Sawan; Sleep 5; done "]

$#] Kubectl apply -f myrc.yml

$#] Kubectl get rc ⟶ (replication Controller)

NAME	Desired	Current	Ready	Age
myreplica	5	5	5	48s

$#] Kubectl describe rc myreplica ⟶ (name)
$#] Kubectl get pods

$#] Kubectl get pods --show-lables. => To check the lebels on Pods

Name	Ready	STATUS	Restarts	Age	labels.
My replica-h12	1/1	Running	0	2M40s	myname = Sawan
Myreplica-179	1/1	Running	0	2M50s	My name = Sawan
My replica-189	1/1	Running	0	4M40s	My name = Sawan
My replica-xyz	1/1	Running	2	3M10s	My name = Sawan

$#] Kubectl scale --replicas=8 rc -l My name = bhupinder.

This will scale the myname = bhupinder template to 8 pods.

$#] Kubectl get pods => to get the pod details.

$#] Kubectl scale --replicas=1 rc -l myname = bhupinder.

↳ This command will scale down the pods from 8 to 1.

$#] kubectl get rc.

↳ To check the replication controller details.

Replica Set

=> Replica Set is a next generation Replication controller

=> The Replication Controller only supports Equality based Selectors whereas the replica set supports Set-based selectors ie filtering according to set of values.

=> Repliaset 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 :                                  # there must match the labels.
    MatchExpressions :
      -{key : myname, Operator : In, values : [Sawan, SAWAN, sAwAn]
      -{key : Env, Operator : NotIn, values : [production]
  template :
    Metadata :
      name : testpod7
      labels :
        Myname : Sawan
  Spec :
    Containers :
    - name : C00
      image : ubuntu
      command : ["/bin/bash, "-c" "while true; do Echo Hello-Sawan;
                                    sleep 5; done "]
```

$#7 Kubectl apply -f myrs.yml

$#7 Kubectl get rs

$#7 Kubectl get pods

$#7 Kubectl scale --replicas=1 rs/myrs

↳ (it will scale down from 2 → 1)

Lec :- 06 - Deployment & Rollback

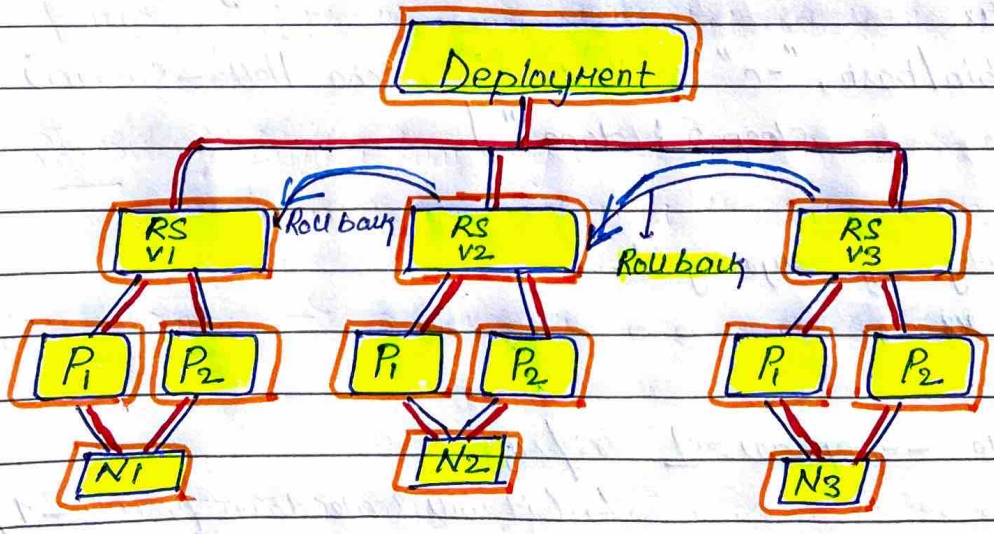
Replication Controller & Replica Set is not able to do update & Rollback apps in the cluster.

=> 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 Speific 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 Maintaince.



(Objet name
Kind : Deployment
Rs = Replica set
P = Pod
N = Nodes

=> A deployments provides declarative update for pod & Replicaset

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 pod template Spec of the deployment. A new Replicaset is created and the Deployment Manager 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.) 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 resumes it to start A new Rollout.

7.) Cleanup Older Replicasets that you don't need anyone.

⇒ if there are problems in the deployment, K8s will automatically rollback to the previous version, however you can also explicity 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.

#⇒ 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 ~~deploy~~ display:-

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

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

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

AVAILABLE ⇒ Displays how many replicas of the application are avilable 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) instance. Take access using putty.

$#) Sudo su
$#) Sudo apt update && apt -y install docker.io
$#) install Kubectl
$#) install Minicube also
$#) apt install Conntrack
$#) Minicube start --vm-driver = none
$#) Minicube Status
$#) Kubectl version
$#) Kubeut get nodes.

Deployment YAML Example

```
Kind : Deployment
apiVersion : apps/v1
Metadata :
  name : my deployments
Spec :
  Replicas : 2
  Selectors :
  MatchLabels :
  name : deployment
template :
  Metadata :
  name : testpod
  labels :
    name : deployment
Spec :
Containers :
  - name : c00
    image : ubuntu
    Command : ["/bin/bash", "-c", "while true ; do Echo Hello-sawan ; 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.
(deploy: object type; mydeployments: object name)

To check, what is running inside containers.

$#> kubectl logs -f <podname>

$#> kubectl rollout status deployment mydeployments

$#> kubectl rollout history deployment Mydeployments

$#> kubectl rollout undo deployment My deployments.

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

This can occour due to some following reasons.

i) Insufficant Quotas
ii) Readyness probe failures.
iii) Image pull errors
iv) insufficent permission
v) Limit Ranges
vi) Application runtime Misconfig.

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

Kubernetes Networking addresses four concerns :-

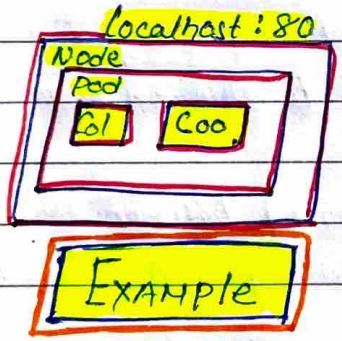
i) Containers within a pod use networking to communicate via loopback.

ii) Cluster Networking provides communication between different pods.

iii) The Service resources 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.



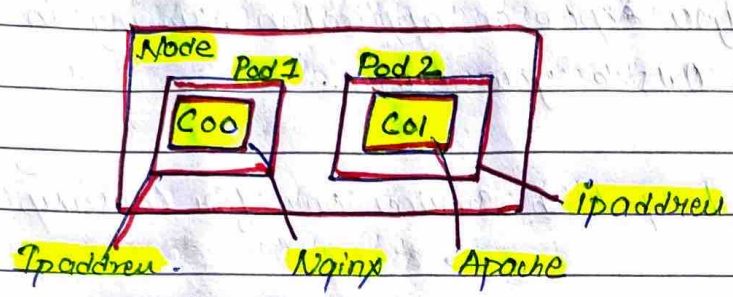
EXAMPLE

```
Kind : Pod
apiversion : v1
Metadata :
  name : testpod
Spec :
 Containers :
  - name : c00
    Image : ubuntu
    command : [ "/bin/bash", "-c", "while true; do echo Hello - Sawan ; sleep5;
                                    done" ]
  - name : c01
    image : httpd
    Ports :
    - containerPort : 80
```

Now try to established communication between two different pods within same Node.

=> pod to pod communication on worker node happens through pods

=> By default pods IP will not be accessible outside the node.



EXAMPLE FOR Nginx

```
Kind : Pod
apiversion : v1
Metadata :
   name : testpod1
Spec :
 Containers :
 - name : c01
   image : nginx
   ports :
    - ContainerPort : 80
```

$#> kubectl apply -f pod2.yml

Example for Apache

```
Kind : Pod
apiVersion : v1
Metadata:
  name : testpod4
Spec:
  Containers:
   - name : c03
     image: httpd
     Ports:
       - ContainerPort: 80
```

$#] kubectl apply -f pod3.yml.
$#] 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 - Services :-

Each pods 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 'backends') provides functionality to other pods (call them 'frontends') 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 ie they come & go on the K8s cluster and on any of the avilable nodes & it would be difficult to access the pods as the pods ip changes once its 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 it's 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 upto 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 recive traffic.

=> ~~Kubectl~~ 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 / applicati 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. (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 Service can run only between ports 30,000 - 32767

=> The set of pods targeted by a service is usually determined by a selectors.

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 deployenthttpd.yml
$#] Kubectl get pods
$#] Kubectl get pods -O wide.
$#] Curl 172.16.12.* → pod Ip to check httpd is working or not.

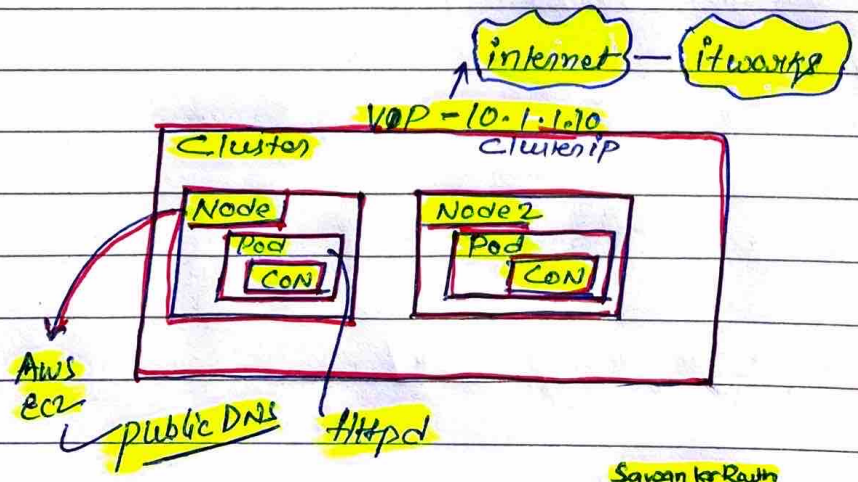
Example of Service

```
Kind : Service      # Define to Create Service type Object
Service
apiVersion : v1
Metadata :
  name : demo service
spec :
  ports :
  - port : 80       # Containers port exposed.
    target port : 80    # pods Port
  Selector :
    name : deployment   # Apply this service to any pods which has the
    type : ClusterIP          specific label
```

↳ specifies the service type i.e ClusterIp or Nordport

```
$#) kubectl apply -f service.yml
$#) kubectl get svc
$#) Curl 10.231.10.*   => cluster ip
```

NODEPORT :- => Makes a service accesible from outside the cluster.
=> Expose the service on the same port of each selected node int the using NAT



Example of NodePort

```
Kind : Service          # Defines to creates services type Object
apiVersion : V1
Metadata :
  name : demoservice
Spec :
  ports :
   - port : 80          # Containers port Exposed.
     targetPort : 80    # pods port
  Selector :
    name : deployment   # Apply this service to any pods which has the
type : NodePort           Specific label.
                    # Specifies the services type ie clusterIp or
                      Nodeport.
```

```
$#) Kubectl apply -f deployment.yml
$#) Kubectl apply -f svc.yml
$#) 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 use Volume. A Volume is essentially directory 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 it outlives 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, like size, Content etc. Some example 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

⇒ We 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 exit as long as the pod is running on that node.

⇒ As the name says, it is initally 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 containers.

⇒ 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 is an empty dir volume is safe across container crashes.

Example of Eptydir

```
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.yml
$#) kubectl get pods
$#) kubectl exec myvolemptydir -c c1 -it -- /bin/bash
(myvolemptydir ↳ pod name; c1 ↳ 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 myvolumeemptydir -c c2 -it -- /bin/bash
$#) cd /tmp
$#) ls ⇒ All files will be also here.

HOST PATH :-

⇒ We use when we want to access the Content of a pod / container from hostname.
⇒ 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 : volume hostpaist
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 pod.
$#] Kubectl exec volume hostpath --ls /tmp
$#] Echo -i " i am sawan " >myfile
$#] Kubectl exec volume hostpath --ls /tmp/hostpath
$#] Kubectl exec myvolhostpath --cat /tmp/hostpath/myfile.
```

Lec-08 - Persistent Volume & Liveness Probe

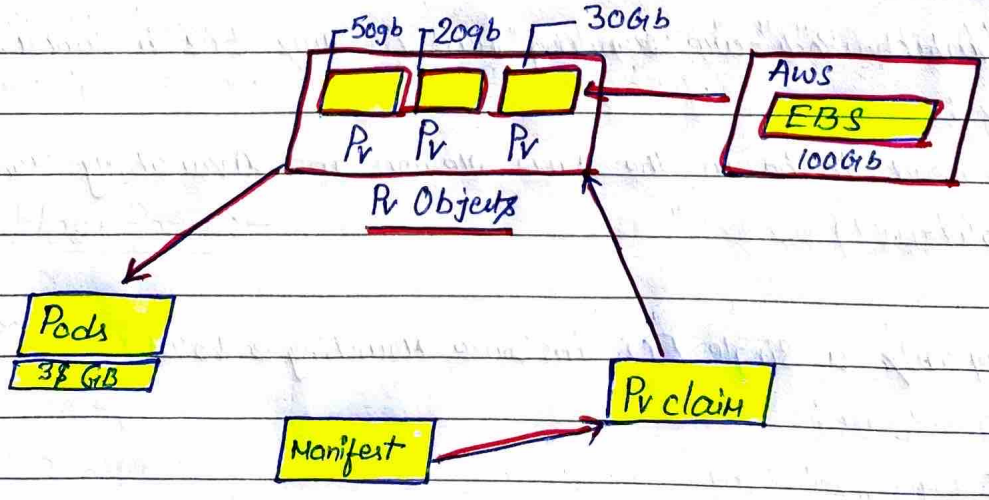
⇒ In a typical IT env, storage is managed by the storage/system admin. The end user will just get instructions to use the storage, but does not have to worry about the underlying storage management.

⇒ 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 resolves 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 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 API for user 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 persist 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 modes, Speed etc) from K8s and 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 PV can be released the underlying PV can then be reclaimed and 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 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 : myebsvol
Spec :
  Capacity :
    Storage : 1Gi
  accessMode :
    - ReadWriteOnce
  PersistentVolumeReclaimPolicy : Recycle
  awsElasticBlockStorage :
    VolumeID : vol-0dca2983B1XYZ   # Put EBS volume ID Here
    FsType : ext4
```

```
$#7 kubectl apply -f mypv.yml
$#7 kubectl get pv
```

Example of Persistent Volume Claim

```
apiVersion : v1
Kind : PersistentVolumeClaim
Metadata :
  name : myebsvolumeclaim.
Spec :
  accessModes :
    - ReadWriteOnce
  Resources :
    Requests :
      Storage : 1Gi
```

$#] Kubectl apply -f mypvc.yml

$#] Kubectl get pvc

Example of Deployment.

```
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 1000"]
  volume Mounts :
  - name : mypd
    MountPath : "/tmp/persistent"
Volume :
- name : mypd
  persistentVolumeClain :
  clainName : mypvsvolclain.
```

```
$#> kubectl apply -f deployPvc.yml
$#> kubectl get deploy
$#> kubectl get rs
$#> kubectl get pods
$#> kubectl exec pvdeploy-0589x42 -it -- /bin/bash
$#> cd /tmp/persistent
```

Now delete the existing pod and create or RS will creat 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 considend ready when all of its containers are ready.
=> In order to very 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~~ recreate 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 ~~payment~~ progress. Restoring - Restarting a container in such a state can help to make the application more avilable 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 cmds specific to the application

=> if the cmd succeeds, it returns 0, and kubelet considers the containers to be alive and healthy. if the command returns a non-zero value the kubelet kills the pod and restarts it.

Example of Liveness Probe

```
Kind : Pod
metadata :
  labels :
    test : liveness
  name : myliveprobe
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
```

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

$#) kubectl apply -f livenes.yml
$#) 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.

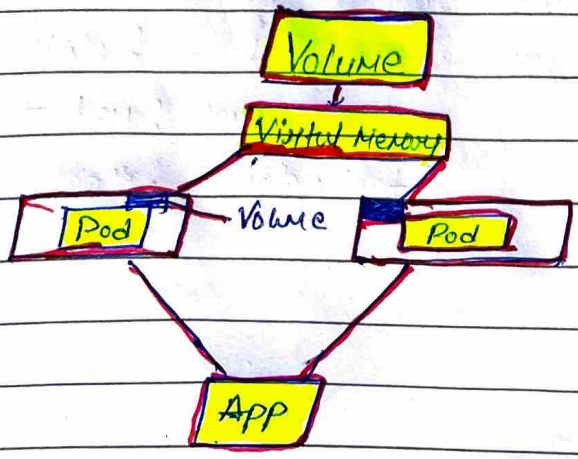
⇒ Hence these configuration should be decoupled from image content in order to keep containerised application portable.

⇒ 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, unencryped configuration information we secretly otherwise.

⇒ ConfigMap can be used to store fine-grained information like individual properties or entire config files.

⇒ ConfigMap are not intened to act as a replacement for properties file.



⇒ ConfigMap can be accessed 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 namedspace Objects, that is exist in the context of namespace

⇒ you can access them via a Volume or an Environment variables from a Containers 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: myvolconfig
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
                              as readonly here
Volumes:
-name: testconfigmap
 configmap:
   name: mymap # This should match the Config Map name Created
   items:          in the first step.
   - key: sample.conf
     path: sample.conf
```

$#] Kubectl apply -f deployconfigMap.yml
$#] Kubectl get pods.
$#] Kubectl exec Myvolconfig -it -- /bin/bash

Example of MYENV

```
apiVersion : v1
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.yml
$#] Kubectl get pods.
$#] Kubectl exec Myenvconfig -it -- /bin/bash
$#] Env
$#] echo $MYENV

$#] exit.

Example of Secrect

```
$#) echo "root" > username.txt ; echo "mypassword" > 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-Sawan ; sleep 5
            ; done"]
  VolumeMounts :
   - name : testsecret
    MountPath : "/tmp/mysecrets" # the secret files will be mounted
Volumes :                          as readonly by default here
 -name : testsecret
  Secret :
   Secretname : mysecret .
```

```
$#) kubectl apply -f deploysecret.yml
$#) kubctl get pods
$#) kubectl exec myvolsecret -it --/bin/bash
$#) cd /tmp
$#) cat passwd.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, groups or the namespace in general.

=> Kubernetes names help different projects teams or customer to share a kubernetes cluster & provide :-

=> A scope for every names.

=> A Mechanism to attach authorization and policy to a subsection of the cluster.

=> By default, a K8s cluster will instantiate to default namespace when provising the cluster to hold the default set of pods, Services and deployments used by the ~~clients~~ cluster.

=> We can use resource Quota on specifying how many resources each name space can use.

=> Most K8s resources (eg pods, services replication controllers and others) are in some namespaces and low-level resours such as nodes and peristentVolumes, are not in any namespace.

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

Create New Namespaces.

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

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

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

CMD :- Kubectl get namespaces.

Namespace YML

```
apiVersion : v1
Kind : Namespace
Metadata:
  name: dev
  labels:
    name: dev
```

$#⇒ Kubectl config set-context $(kubectl config current-context) --namespace=dev

Creating Sample Pod YAML

```
Kind : Pod
apiVersion : v1
Metadata :
  name : testpod
Spec :
  Containers :
  -name : cOO
   image : ubuntu
   command : ["/bin/bash","-c","while true; do echo Technical guftgu;
             sleep 5; done"]
Restart Policy : Never.
```

$#) kubectl apply -f pod.yml -n dev → it will create the pod under dev namespace.

$#) kubectl get pod -n dev

(*) To change the pod view from default namespace to dev namespace command is as follows.

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

(*) To check which is the default namespace.

$#) kubectl config view | grep namespace:

$#) kubectl get pods.

Managing Compute Resources for Containers

⇒ A pod in K8s will run with no limits on CPU and 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 satisfy the pod CPU request.

⇒ Cpu is specified in units of cores and Memory is specified in units of Bytes.

Two types of ~~Containers~~ Constraints Can be set for each resource type :—

Request and Limits.

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

⇒ A limit is the max amount of resources that K8s 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 if default to 0

⇒ Cpu values are specified in 'Millicpu' and Memory in MIB.

Resource Quota

⇒ A K8s cluter can be divided into namespaces if a Container is created in a namespace that has a default cpu limit and the container does not specify its own cpu limit, then the Container is assigned the default Cpu limit.

⇒
↳ Namespaces can be assigend Resource Quota objects, this will limit the amount of usage allowed to the objects in that namespaces.

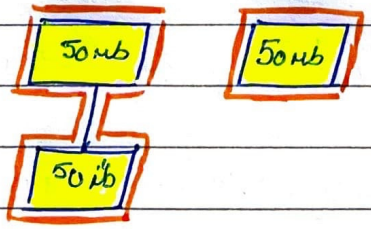
You can limit:
1. Compute
2. Memory
3. Storage

Here are the 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.

Namespace → 200 Mb → RAM



Example of Resource

```
apiVersion : V1
Kind :
Metadata :
  name : Resource1
Spec :
  Containers :
  - name : resource1
    image : Centos
    command : ["/bin/bash", "-c", "while true; do echo Hello -Sagar
              ; sleep 5 ; done"]
    Resource1 :
      requests :
      Memory : "64Mi"
      Cpu : "100M"
    Limits :
      Memory : "128 Mi"
      Cpu : "200 M"
```

$#] kubectl apply -f resource1.yml.

$#] kubectl describe pod resource1.

Example of Resource Quota

```
apiVersion : v1
Kind: Resource Quota
Metadata:
  name: myquota
Spec:
  hard:
   Limits.cpu: "400m"
   Limits.memory: "400Mi"
   requests.cpu: "200m"
   Requests.memory : "200Mi"
```

#$> Kubectl apply -f resourcequota.yml

Example of Testpod and ReplicaSet

```
Kind : Deployment
apiversion: apps/v1
Metadata:
  name: deployments
Spec:
  replicas: 3
  selector:
   MatchLables:
    objtype: deployment
template:
  Metadata:
  name: testpod8
  lables:
   Objtype: deployment
Spec:
```

```
Containers:
  - name: c00
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Hello - Savan
             ; sleep 5; done"]
    resources:
      requests:
        cpu: "200m"
```

→ reduce it to 150mi"

$#] kubectl apply -f testpod.yml

$#] kubectl get deploy

Lec -11 - Resource Quota And Horizontal Scaling

Cpu Limit → 500 mi
Request → ⇑⇓
Request = Limit

Cpu Request = 600mi
Limit = 1
Limit ≠ Request

Example for Memory

apiVersion: v1
Kind: Limit Range
Metadata:
name: Mem-Min-Max-demo-lr
Spec:
Limits:
- Max:
Memory: 1Gi
Min:
Memory: 500 Mi
type: Container

prod-namespace

Limits: Cpu: "600M"
limits. Memory "800 Mi"
Requests. Cpu: "200M"
Requests: Memory: "500M"

Default Range

Cpu
(Min) Request - 0.5
(Max) Limit - 1
Memory:
(Min) request :- 500 M
(Max) Limit :- 1Gi

$#) Kubectl describe pod podname.
$#) Kubectl apply -f yml file name.

Horizontal pod Autoscaler :-

=) Kubernetes has the possibility to automatically scale pod based on observed cpu utilization,
Which is horizontal pod Autoscaling.

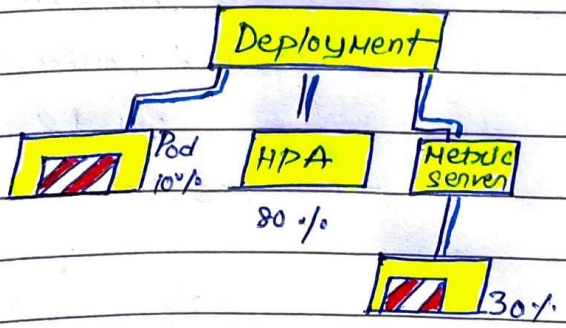
=) Scaling can be done only for scalable objects like controller / deployment or Replica set.

=) HPA is implemented as a K8s Api Resources and 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 - horizontal - pod - auto - scaler - sync - period flag (Default value of 30 sec)

=) During each period, the controller Manager Queries the resource utilization against the metrics specified in each horizontal pod Autoscaler defination —



Horizontal Pod Autoscaler (HPA)

⇒ For per-pod resource metrics (like cpu), the controller fetches the metrics from the resource metrics API for each pod targeted by the Horizontal pod Autoscalers

⇒ 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.

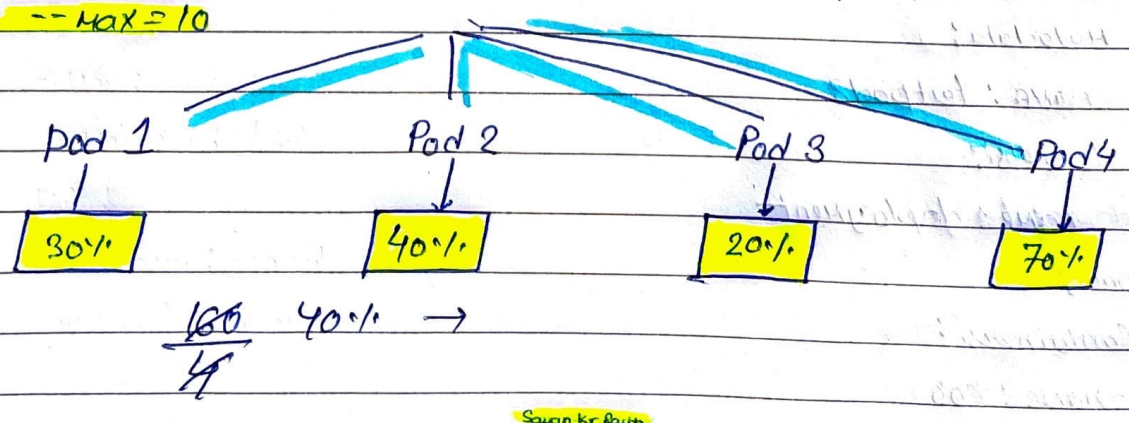
⇒ if a Target raw-value is set, the raw metric values are used directly The controller then takes the mean of the utilization on the raw value across all targeted pods, and produce a ratio we to scale the number of desired replicas.

⇒ Cooldown period to wait before another down scale 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 metrics via the resource metrics API

--kubectl-insecure-tls

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



Command to install Metric server on K8s :-

$#) wget -O metricserver.yml https://github.com/kubernetes-sigs/metric-server/releases/latest/download/components.yml

it will download the metricserver.yml file, then open that file under vi tool and go under Metric-Server line. and add the line after

```
- --secure-port = 4443
- --kubelet-insecure-tls   # add this line.
```

$#) kubectl apply -f metricserver.yml

Then Create a Deployment and a container to watch Real time scaling.

```
Kind : Deployment
apiVersion : apps/v1
Metadata :
   name : mydeployment
Spec :
  replicas : 1
  selector :
   matchLabels :
    name : deployment
  template :
   Metadata :
    name : testpod8
    labels :
     name : deployment
  Spec :
   Containers :
   - name : c00
```

image: httpd
ports:
- containerport: 80
resources:
limits:
cpu: 500m
requests:
cpu: 200m

$#) kubectl apply -f deploy.yml
$#) kubectl get all

Command to start autoscale on deployment :-

$#) kubectl autoscale deployment mydeploy --cpu-percent=20 --min=1 --max=10

Now enter into the pod and give some load.

$#) kubectl exec mydeploy-7899fccxyz -it -- /bin/bash
$#) apt-get update

Now open that terminal of kubernetes server.

$#) watch kubectl get all

it will check and scale automatically in every 2 sec and create the pod according to the wage.

Lec-12- Jobs, Init Containers & Pod Lifecycle

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 pods is hosting keeps running.

Use Cases :-

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

Example of Batch

```
apiVersion : batch/v1
kind : Job
Metadata:
  name: testjob
Spec:
  template:
    Metadata:
      name: testjob
    Spec:
      Containers:
      - name: Counter
```

```
image: centos:7
command: ["/bin/bash", "-c" "echo Hello-sawan; sleep 5"]
restartpolicy: Never
```

$#) kubectl apply -f job.yml
$#) kubectl get pods.
$#) watch !!

```
apiVersion: batch/v1
Kind: Job
Metadata:
  name: testjob
Spec:
  parallelelism: 5      # Runs for Pods is parallel
  activeDeadlineSeconds: 10  # Timeout after 30 sec
  template:
    Metadata:
      name: testjob
    Spec:
      containers:
      -name: Counter
       image: centos:7
       command: ["bin/bash", "-c", "echo Technical-Guftgu; sleep 20"]
    RestartPolicy: Never.
```

$#) kubectl apply -f job2.yml
$#) kubectl get pods.
$#) watch !!
$#) watch kubectl get pods

The "Cron job pattern" :-

⇒ if we have multiple nodes hosting the application for high availability which node handles Cron ?

⇒ What happens if multiple identical Cron jobs run simultaneously.

Example of Cron

```
apiVersion : batch / v1beta1
Kind : Cronjob
Metadata :
  name : bhupi
Spec :
  Schedule : " * * * * * "
  Job Template :
    spec :
      Containers :
      - image : ubuntu
        name : Sawan
        command : [ "/bin/bash ", "-c", " Hello sawan ; sleep 10 ; ]
      restart policy : Never
```

$#7 Kubectl apply -f cron.yml
$#7 kubectl get pods
$#7 watch !!
$#7 watch kubectl get pods.

Init Containers :-

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

⇒ Init containers always runs to completion.

⇒ If a pod's init container fails kubernetes repeatedly restarts the pod until the init container succeeds.

⇒ init containers do not support readiness probe.

Use Cases :-

⇒ Seeding a Database.

⇒ Delaying the application launch Until the dependencies are ready.

⇒ Clone a git repo into a Volume.

⇒ Generate configuration file dynamically.

Example of Init

```
apiVersion : v1
Kind: Pod
Metadata:
  name: initContainer
Spec:
  initContainers:
  -name : c1
  image: Centos
  command: ["/bin/sh", "-c", " echo Hello all > /tmp/xchange/testfile;
                sleep 5s "]
  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 "]
    Volume Mounts;
      -name : xchange
      Mountpath : " /tmp/data"
    Volumes:
    -name: xchange
    Empty Dir : {}
```

```
$#] kubectl apply -f init.yml
$#] kubectl get pods.
$#] watch !!
$#] watch kubectl get pods.
$#] kubectl logs -f pod/initcontainer
```

Pod Lifecycle.

Pending → Running → Succeeded → Failed → Completed → Unknown

⇒ The phase of a pod is a simple high-level summary of where it is in its lifecycle.

Pending

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

⇒ One or more of the container 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 container is still running or is in the process of starting or restarting.

Succeeded

⇒ All Containers in the pod have terminated in success, and will not be restarted.

Failed.

⇒ All container in the pod have terminated and atleast one Container has terminated in failure.

⇒ The Container either exited with non-zero status or was terminated by the system.

Completed

⇒ The pod has run to completion as there's nothing to keep it running eg - completed Jobs.

Unknown

⇒ State of the pod could not be obtained.

⇒ Typically due to an Error in network or communicating with the pod host of the pod.

Pod Conditions.

A pod has a podstatus, which has an array of podconditions trough which the pod has or has not passed.

⇒ using "Kubectl describe pod Podname" you can get condition of a pod.

These are the posible types :—

podscheduled :— The pod has been scheduled to a node.

Ready :— The pod is able to serve request and will be added to the load Balancing pools of all matching service.

Initialized :— All init Containers have started succesfully

Unscheduled :— The scheduler cannot schedule the pod right now. for e.g due to lacking of Resource or other constraints.

Container Ready :— All containers in the pod are ready.