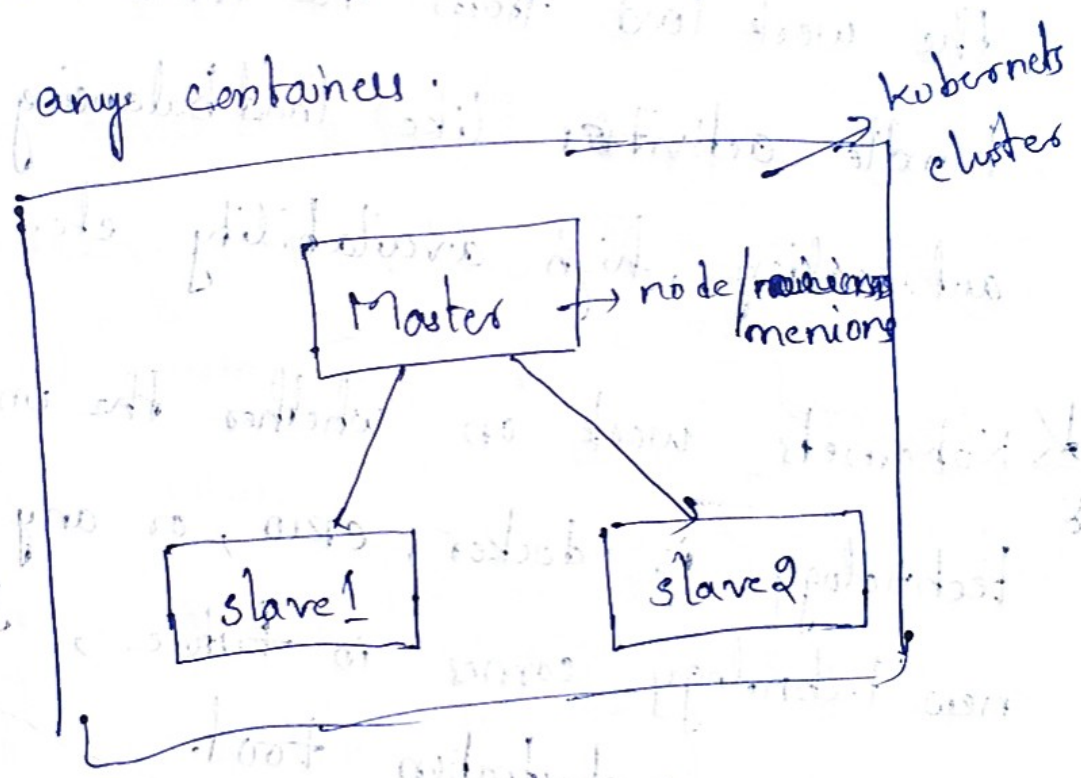
KUBERNETES

→ It is a container orchestration tool. where as docker swarm is docker container orchestration tool.

→ It means docker swarm work on only docker containers. where kubernets work on any containers.



→ We have one Master machine and slaves in the cluster

→ The individual node is known as menions

→ The combination of these nevions is known as kubernetes cluster.

→ Master is the main machine which triggers the containers orchestration. It distributes the work load to the slaves.

→ Slaves are the nodes that accept the work load from the master and handle activities like load balancing, autoscaling, high availability etc.

★ → Kubernets work on whether the underlining technology is docker, CRIO, or any other new technology comes in future, It is a generic orchestration tool.

Kubernets objects :-

① Pod :-

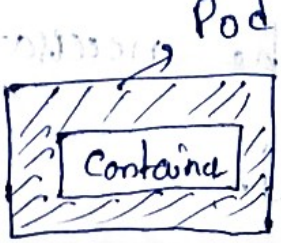
→ It is the layer of abstraction on top of a container.

→ It is the smallest object that kubernetes can work on.

→ In the Pod we have container.

→ The advantage of Pod is that kubectl commands will work on the Pod and Pod communicates these instructions to the container irrespective of which technology containers are in the Pod.

→ It means it acts like a translator between containers of any technology with kubernetes commands.



→ Kubectl commands can work upto the level of Pod, then Pod translates in such a way that underlining container can understands

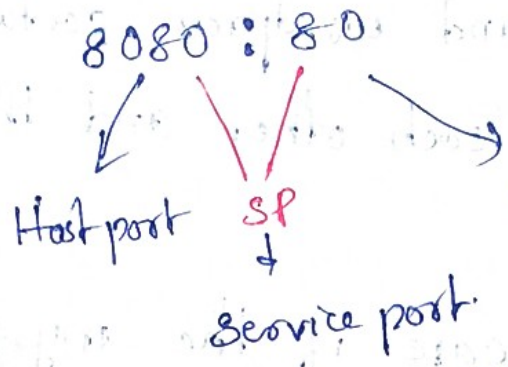
Interview:-

⊛ what is pod? why it is required?

→ Because kubernetes is container orchestration tool its not restricted to docker, It can work on any kind of containers it could be docker, crio etc. So they have designed the kubernetes in such a way that kubernetes can communicate upto the level of Pod. It is the responsibility of Pod to understand which kind of container is inside and according do the necessary translations is called as Pod.

② Service :—

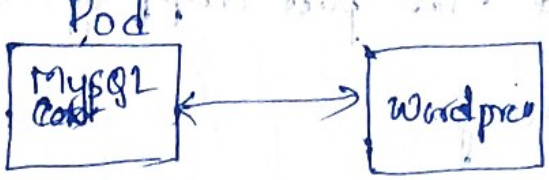
→ It is used for port mapping and n/w load balancing...



→ In between host port and container port we have service port.

→ Service port ensures that same ip address is assigned to the re-created container (if container crashes, there is a chance of having another ip address for newly recreated container, then communication between that container and other will be lost), so to avoid that miscommunication, service port taking care of assigning same ip address to the container

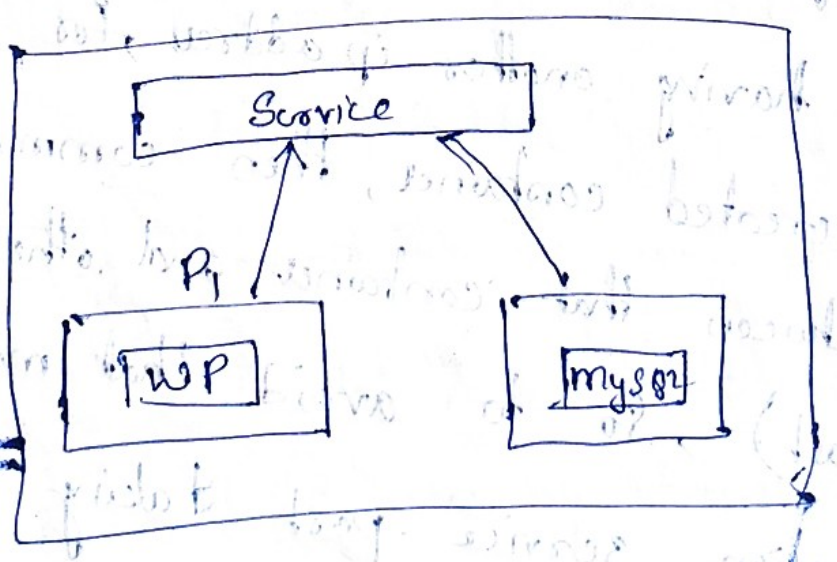
eg:-



→ The mysql and wordpress containers are linked with each other and they communicate.

→ But in any case if the mysql gets crashes, then kubernetes creates another replica automatically but it comes up with different ip address, then communication between them is lost.

→ To avoid that we have service



→ Service is separately storing the ip

address of each Pod, and assigns the same ip address for newly created Pod.

→ whenever same ip address is assigned then communication between ~~conta~~ Pods remain same.

③. Replication Controller :-

It is used for performing two activities of container orchestrations, they are

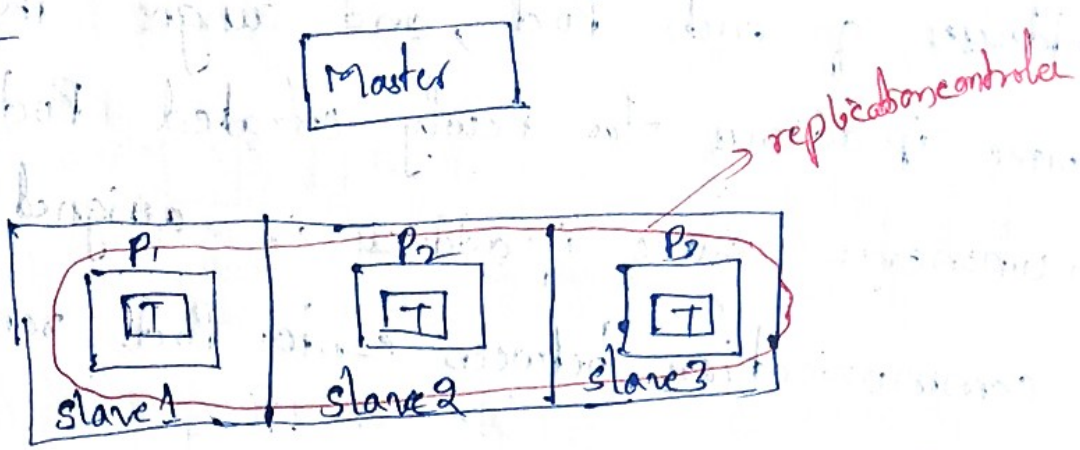
(i) Load balancing

(ii) Scaling.

→ This is used for managing multiple replicas of Pods and also performing scaling.

→ To increase the no. of replicas or decrease the no. of replicas we have replication controller.

eg:-



→ We have 3 slaves, on each of them has Pod, in the Pod tomcat container is there.

→ To maintain all these pods on slaves we have another layer called replication controller.

→ For load balancing, and scaling of replicas then use replication controller.

(4) Replicset :-

→ It is also similar to replication Contd. but some it has some advanced future like selector is implemented.

→ It is also used for maintaing replicas which are running on different slaves.

→ It also performs load balancing and scaling. but it has attribute called selector

→ The selector has ability to pick up particular element based on label and use them.

Eg:-

→ We want to create 3 replicas of tomcat.

→ But already already 2 replicas of tomcat are running.

→ Then selector sees that already two tomcat Pods are running and then it creates one Pod of tomcat and reaches the desired state of 3.

→ It checks the components which you are creating in the cluster, if they

are available it will use them otherwise it will create.

→ In this we have a reusability factor

⑤ Deployment :-

→ It can do loadbalancing, scaling and also rolling update.

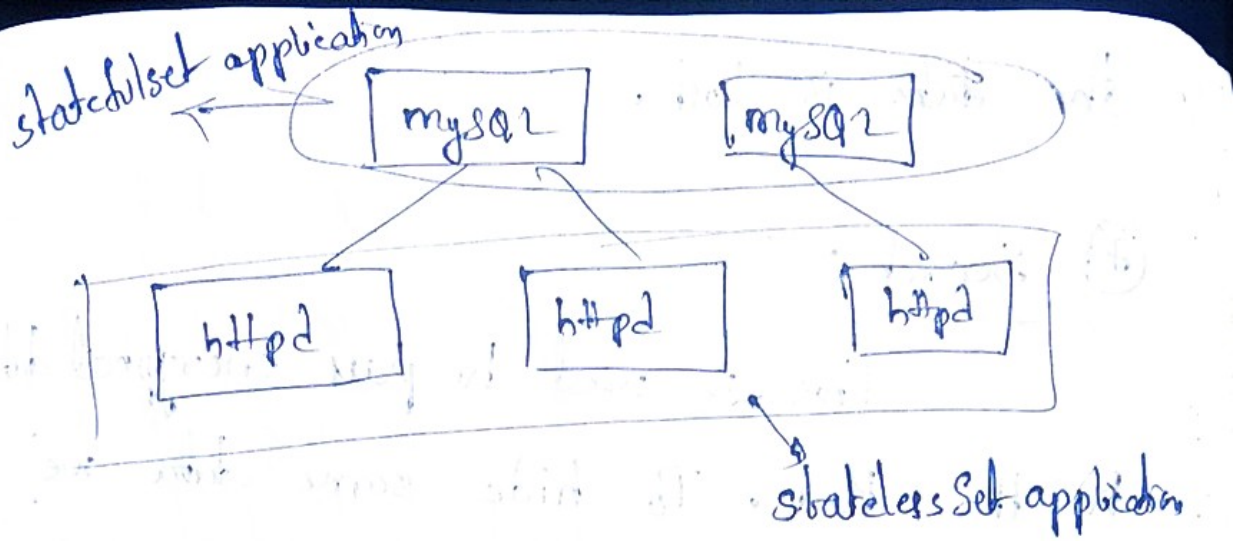
⑥ StatefulSet :-

These are used to handle stateful applications like databases where that data has to be preserved even if the container got crashed

→ The newly created container must come up with the previous data.

→ For this purpose we use statefulset

eg :- All databases



→ The httpd act as user interface, it takes the data and pases it to mysql database, So these are called statelessSet applications where mysql data bases are called StatefulSet application

Interview:

(*) How you mainted databases in docker swarm.

→ We don't have databases in the dockerswarm cluster, but we are maintaining databases, We go with regular database installation and linking with docker swarm cluster. because if the data base is maintained in the swarm cluster, then when the container crashes

the data is lost.

⑦ Secret :-

This is used to pass encrypted data to the Pods. To hide some data we use secrets.

⑧ Ingress :-

The mapping of the domain name with sub domains is done with ingress.

eg:- amazon.in / something → sub domain
↳ main domain

⑨ Horizontal Pod Autoscaller :-

→ The no. of Pods will automatically increase or decrease depending on the load.

→ In docker swarm we don't have autoscaling

(10) Persistant Volume :-

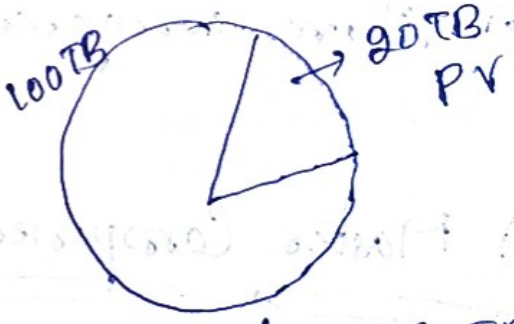
It is used to specify the section of storage that should be used for volumes.

(11) Persistant Volume Claim :-

It is used to reserve a certain amount of storage for pod from the persistant volume.

→ These two are used for backup mechanism.

eg :-



Let's say from 100TB of cluster, 20TB is reserved for volumes, it is called PV

Now I want to use 1TB for creating a container from 20TB of PV, then it is called PVC.

(12). NameSpace! –

It is used for creating partitions in the cluster. Pods are running in a namespace cannot communicate with other Pods running in other namespace.

Master Slave Components / Master Slave architecture

(1) Master components

(2) Slave components.

(1) Master Components :– It is responsible for managing the execution and lifecycle of containers within the K8s.

a) Container runtime :– It is generally a docker running on the master machine. We call that is kubernetes as container runtime.

→ don't use docker word here.

(b) Kube apiserver :-

→ It acts as a validator.
→ It is a gateway to the cluster.
→ It works as a gatekeeper for authentication and it validates if a specific user is having permissions to execute a specific command.
→ If he has permissions, then it passes to scheduler

eg:- I am trying delete a cluster, then I executed delete command for that cluster. then Kube apiserver, checks whether he has permission to delete or not. If I have permissions it passes to the scheduler who performs the operations which I want to do.

(c) Kube scheduler :-

→ This process accepts the instruction from apiserver after validation and starts an application on a specific node or set of nodes.

→ The hardware resources are estimated by kube scheduler.

eg:- I want to create 3 tomcat replicas then kube scheduler first checks how many slaves we have, how much amount of storage is present on each slave, and on which machine to create the replicas, all these are estimated by kube scheduler.

(d) Control manager or Controller :-

→ Maintaining the ~~desired~~ ~~desired~~ desired

count of replicas is done by this controller.

eg:- Let say I have desired count of 6 replicas, suddenly one replica crashes. then control manager automatically creates replica and maintains desired count.

(e) etcd :-

→ It is a repository or database.

→ It stores all the information about the cluster i.e how many slaves are there, no. of replicas, ip addresses, every information on about the cluster.

→ It is the core repository, the entire information of cluster is stored in etcd.

Slave Components :-

a) Container runtime :-

Some containerization technology should be there, generally docker.

(b) Kubelet :-

This is the actual process that takes the orders from the scheduler and deploy an application or any operation on a slave.

→ It is present on both master and slave.

(c) Kubeproxy :-

→ It will take the request from services to pod.

→ It has the intelligence to forward a request to a near by pod.

eg :- If an application pod wants to

communicate with db pod then kube-proxy will take that request to the near by pod. So that latency will be reduced.

Kubernetes installations:

→ These are divided into two types, they are

Unmanaged / self managed Kubernetes setup

1) KOPS
2) Kubeadm
3) KIND

Managed Kubernetes setup

1) EKS (AWS)
2) GKE (GCP)
3) AKS (azure)

→ In managed kubernetes setup, the cloud automatically setup the cluster where as in self/unmanaged we manually

have to setup the cluster.

① KOPS :-

→ It uses S3, in AWS, it is like a drive for storage, they call it as bucket.

→ Route53 :- It acts like DNS (it converts the domain name into ip-address)

→ Autoscalling :- It also activates autoscalling i.e if a server gets crashed, it automatically creates a server in the cluster.

→ If the KOPS server wants to activate all these services, it needs some permissions, the permissions are given by IAM

S3
EC2
VPC
Route53
Autoscalling etc

step-I :-

→ Launch Linux EC2 instance in AWS

→ Create and attach IAM role to EC2 instance.

step-II :-

→ Install kops on EC2

```
curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_name | cut -d '"' -f 4)/kops-linux-amd64
```

→ Now change execute permissions of kops-linux-amd64 folder

```
chmod +x kops-linux-amd64
```

→ Now move that file into /usr/local/bin/kops

```
sudo mv kops-linux-amd64 /usr/local/bin/kops
```

step-III :- Install kubectl

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
```

→ change permissions of kubectl

```
chmod +x ./kubectl
```

→ Now move that file into /usr/local/bin/ kubectl

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

step-IV :- Create S3 bucket in AWS

```
aws s3 mb s3://name of bucket.in.k8s --region us-east-1
```

→ Give any name for the bucket, it must be unique.

→ give the region, in which region your kops server present.

step 5:- Create private hosted zone in Aws Route53

Head over to aws Route53 and create

Aws/ Route53 hosted zone

choose name for example (sudarshan.in)

→ Now choose type as privated hosted zone for VPC

→ Select default vpc in the region you are setting up your cluster.

→ Hit 'create'.

step 6:- Now configure environment variable

open .bashrc file

```
vim ~/.bashrc
```

Add following contents into .bashrc,

you can choose any arbitary name for cluster and make sure bucket name matches the one you created in previous step.

```
export KOPS_CLUSTER_NAME = project.in
export KOPS_STATE_STORE = s3://your
bucket name.in.k8s
```

→ Then run command to reflect variably added to .bashrc

```
source .bashrc
```

step-VI: Create SSH key pair

→ This key pair is used for ssh into kubernetes cluster

```
ssh-keygen
```

step-VII:

Create a kubernetes cluster

```
kops create cluster \
--state = ${KOPS_STATE_STORE} \
--node-count = 2 \
--master-size = t3.medium \
--node-size = t3.medium \
--zones = us-east-1a \
--name = ${KOPS_CLUSTER_NAME} \
--dns private \
--master-count 1
```

step-IX: Create kubernetes cluster

```
Kops update cluster --yes --admin
```

step-X: Execute the command now and then unbill the cluster gets created.

```
Kops validate cluster
```

step-XI: To delete the cluster permanently

```
Kops delete cluster --yes
```

Step - VII :- To connect to the master

```
ssh admin @ api.javahome.in
```

We can change number of nodes and no. of masters using following commands.

```
kops edit ig nodes change minsize and
maxsize to 0
```

```
Kops get ig - to get master node name
```

```
Kops edit ig - change min and max size
to 0
```

```
Kops update cluster --yes
```

③ KIND :-

step - I :-

→ Install docker

step - II :-

Install kubectl

→ go to kubectl install site and select for which flavour you are using. then copy and paste those commands

step - III :-

Install KIND software

→ go to kind install site

copy those commands and paste

step - IV :-

→ It requires some configuration file

copy those commands in

```
vim config.yml
```

step - V :- Now create / cluster

```
kind create cluster --name mycluster --config config.yml
```

Managed Kubernetes setup

(1) EKS (Elastic kubernetes Service)

step-I :-

First launch linux server and give IAM permission and attach it to EC2

step - II) -

→ first install eksctl

→ go into google and select eksctl install then install it

step - II) -

→ Now install kubectl

step - IV) :- To activate the cluster or to create the cluster

eksctl create cluster \
--region us-east-1 \
--node-type t3.medium \
--nodes 3 \

--name new-cluster

step-v:-

```
eksctl create cluster --name my-cluster --region
us-east-1
```

```
eksctl delete cluster --name my-cluster --region us-east-1
```

step-vi:-

To see the cluster search in AWS as EKS.

step-vii:-

Now install certain plugins from the site

https://docs.aws.amazon.com/eks/latest/userguide/install-aws-iam-authenticator.html

(or)

when type kubernetes commands, it will show the website url.

(2) GKE (Google Cloud Kubernetes Engine)

step-I :- Create google cloud account and then go to console.

step-II :-

Then go into navigation symbol ☰ click on kubernetes engine → cluster → switch to standard cluster → create.

step-III :-

If you get error as insufficient regional quota then go to default-pools → nodes → in that give 3, where the no. is showing 100. then enter create cluster

1. To see the list of all the nodes

```
kubectl get nodes
```

2. To see info about the nodes in wider o/p

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

3. To see the detailed info about a node

```
kubectl describe nodes node_name
```

4. To create a pod

```
kubectl run --image image_name pod_name
```

5. To see the list of all the pods that are created

```
kubectl get pods
```

6. To see the info of pods in wider o/p

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

7. To get detailed info about a pod

```
kubectl describe pods pod_name
```

(8) To go into the bash shell of a pod

```
kubectl exec -it pod_name -- bash
```

(9) To delete a running pod

```
kubectl delete pod pod_name
```

(*) Create a mysql pod

```
kubectl run --image mysql mydb --env MYSQL_ROOT_PASSWORD=sudarshan
```

→ We have yml files in the kubernetes these files are called as definition files or manifest files, We have top level elements in this every files

```
---
apiVersion:
kind:
metadata
spec:
```

kind : → it represents which type of object you are using eg:- pod, service, deployment, etc.

apiVersion : — Here we have some api versions for different objects.

These are the resource of K8s cluster

	kind	apiVersion
1	Pod	v1
2	Service	v1
3	Namespace	v1
4	Secrets	v1
5	Persistentvolume	v1
6	Persistentvolumeclaim	v1
7	Horizontal Pod AutoScaller	v1
8	Replicaset	apps/v1
9	Deployment	apps/v1
10	Statefulset	apps/v1

Interview

RDS = apps/v1

metadata :- It is data about the data

Spec :- The containers information is given in this module.

eg :- If I am creating a Pod, then in the spec: section I will give container information

Labels :-

You can give any data about the object you are creating. In this key and value are not predefined

eg :- author : intelliqit

key value

You can use any keys and any values in the label section.

To run the definition file

```
kubectl apply -f file_name
```

(*) Create a nginx pod

```
---
apiVersion : v1
kind : Pod
metadata :
    name : nginx-pod
    labels :
        author: intelliqit
        type : proxy
Spec :
    containers :
        - name : mynginx
          image : nginx
...
```

```
kubectl apply -f pod-definitionfile1.yml
```

→ To delete the pod which is created using definitionfile

```
kubectl delete -f definitionfile_name
```

(*) Create the postgres database

```
---
apiVersion: v1
kind: Pod
metadata:
    name: postgres-pod
    Labels:
      type: db
      author: sudarshan
Spec:
  containers:
    - name: mypostgres
      image: postgres
      env:
        - name: POSTGRES_PASSWORD
          value: intelliqit
        - name: POSTGRES_USER
          value: myuser
```

```
    - name : POSTGRES_DB
      value : mydb
...
```

(*) Create a jenkins pod

```
---
apiVersion : v1
kind : Pod
metadata :
    name : jenkins-pod
    labels :
        author : sudarshan
        type : CI-CD implementation
spec :
    containers :
        - name : myjenkins
          image : jenkins/jenkins
          ports :
              - containerPort : 8080
                hostPort : 8686
...
```

→ To create yaml files of kubernetes in pycharm.

step ①:-

Download a "kubernetes and openshift resource support" plugin

step ②:-

Now open pycharm, go to file → settings → plugins → click on gear icon → click on → install plugin from disk then go to download section and select the downloaded plugin

Namespace:-

We have different types of namespaces in the kubernetes setup. To see the namespace

```
kubectl get namespace
```

→ In this we a namespace called default

→ The pods which are created will be running on default namespace. The namespace is nothing but making the partition in the cluster.

→ We can create our own namespaces.

vim definitionfile.yml

```
---
apiVersion: v1
kind : Namespace
metadata :
    name : test-ns
...
```

To run our pod in the test-ns.

vim definitionfile.yml

```
---
apiVersion: v1
kind : Pod
metadata : ghost-pod
...
```

vim definitionfile.yaml

---

apiVersion : v1

kind : Pod

metadata :

  name : ghost-pod

  namespace : test-ns

  Labels :

    type : cms

    author : intelliqit

Spec :

  containers :

    - name : ghost

      image : ghost

      env :

        - name : NODE_ENV

          value : development

...

To check the customised namespace

```
kubectl get pods -n namespace_name
```

eg:- kubectl get pods -n testing

(or) The command which shows all the pods in all the namespaces

Criteria

```
kubectl get pods --all-namespaces
```

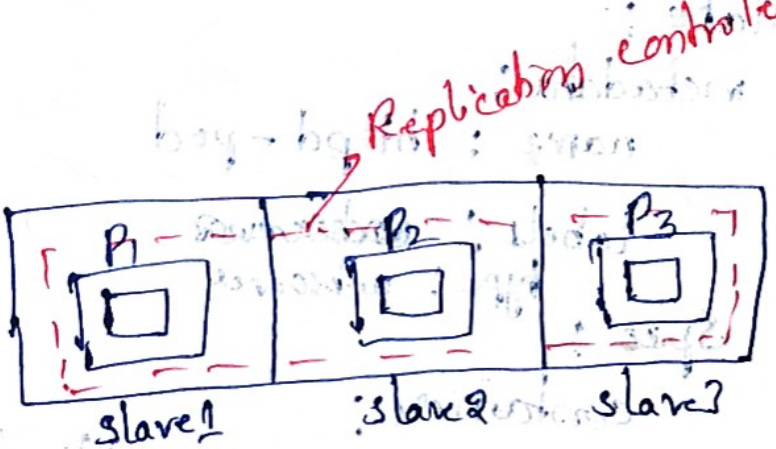
Replication Controller:-

→ It can perform two activities, they are (i) loadbalancing (ii) scaling.

→ The layer which controls/maintains all the replicas in a cluster is called Replication Controller

→ In the Replication Controller we have Pods

→ in the pods we have containers.



template:- It is just like blue print, i.e all the replicas created, structure must be same i.e, it maintains the structure of all the replicas same.

It has two child elements, they are

metadata and spec

vim definitionfile.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
    name: httpd-rc
    labels:
        type: webserver
        author: intelliqit
Spec:
    replicas: 3
    template:
        metadata:
            name: httpd-pod
            labels:
                type: webserver
        Spec:
            containers:
                - name: myhttpd
```

```
image: httpd
ports:
  - containerPort: 80
    hostPort: 8080
...
```

kubectl get all

It shows all the pods information.

Replicaset :- It can perform the same action of replication Controller, but it has a another element i.e selector, the selector first it checks whether the replicas are present or not,

→ If the replicas are present, it uses them accordingly otherwise it creates and achieves the desired count.

→ It is mainly used for Reuseability

Syntax

```
selector:
  matchLabels:
    type:
```

vim definitionfile.yml

```
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
    name: tomcat-rs
    labels:
        type: appserver
        author: intelliqit
Spec:
    replicas: 2
    selector:
        matchLabels:
            type: appserver
    template:
        metadata:
            name: tomcat-pod
            labels:
                type: appserver
        Spec:
            containers:
                - name: mytomcat
                  image: tomcat
                  ports:
                    - containerPort: 8080
                      hostPort: 8181
```

Scaling can be done in two ways

① open your definitionfile, in that change the replicas number, then run the command as

```
kubectl replace -f definitionfile_name
```

② Another way is from the command line itself we can scale as

```
kubectl scale --replicas=2 -f definitionfile.yml
```

eg:- kubectl scale --replicas=2 -f definitionfile 2.yml

To delete all the pods in the cluster

```
kubectl delete all --all
```

Deployment :-

It is absolutely same as replicaSet but it can perform rolling updat
→ It can perform ReplicaSet activities too

Vim definitionfile.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    Labels:
        type: proxy
        author: intelliqit
Spec:
    replicas: 3
    selector:
        matchLabels:
            type: proxy
    template:
        metadata:
            name: nginx-pod
            Labels:
                type: proxy
```

```
        auther : intelliqit
    Spec :
        containers :
            - name : myngingx
              image : nginx
              ports :
                - containerPort : 80
                  hostPort : 9090
...
```

Deamon set :-

It is used to set no. of replicas = no. of slaves, i.e how many slaves are there, that many replicas must be created automatically.

→ Deployment definitionfile and deamon definition file are same but in deamon file we won't give replicas count.

→ It automatically checks how many slaves are there, that becomes the desired count.

vim definitionfile.yml

```
---
apiVersion: apps/v1
kind : Deploy, DeamonSet
metadata :
    name : httpd-deamon
    labels :
        type : webserver
Spec :
    selector :
        matchLabels :
            type : webserver
    template :
        metadata :
            name : httpd-pod
            Labels :
                type : webserver
        Spec :
            containers :
                - name : myhttpd
                  image : httpd
                  ports :
                    - containerPort : 80
                      hostPort : 8888
```

Scenario:-

The deamonSet is used when you want to have log collectors on each server. i.e the copy of the log collecting software/applicatn must present on each server, in this scenario we use deamonset

Secrets:-

This is used to encrypt the password or any data.

Syntax:-

```
type: Opaque
stringData:
  a: your password/any data
```

→ Now create a scret definitionfile and use that data or variable in other definitionfiles in which you want to encrypt.

(*) Create a mysql database and the password must be encrypted or not directly mentioned in the definitionfile.

→ First create Secret definitionfile

vim definitionfile.yml

```
---
apiVersion: v1
kind: Secret
metadata:
    name: mysql-secret
    labels:
        author: intelliqit
type: Opaque
stringData:
    a: intelliqit
...
```

→ Now pass this variable 'a' in the definitionfile of database

vim definitionfile.yml

```
---
apiVersion : v1
kind : Pod
metadata :
    name : mysql-pod
    labels :
        type : db
        auther : intelliqit
Spec :
    containers :
        - name : mydb
          image : mysql
          env :
            - name : MYSQL_ROOT_PASSWORD
              valueFrom :
                secretKeyRef :
                    name : mysql-secret
                    key : a
...
```

→ We can also encrypt the password intelliqit as

```
echo "intelliqit" | base64
```

→ And change the permission of secret definition file, then group and others won't even touch the file.

(*) Create a secret file for postgres database

vim definitionfile.yml

```
---
apiVersion: v1
kind: Secret
metadata:
    name: postgres-secret
    Labels:
        author: intelliqit
type: Opaque
stringData:
    password: intelliqit
    username: myuser
    dbname: mydb
```

→ Instead of using a, b, c, use some meaningfull names as above

→ Now create postgres deployment file.

```
---
apiVersion : apps/v1
kind : Deployment
metadata :
    name : postgres-deployment

    labels :
        type : db
        author : intelligit

Spec :
  replicas : 2
  selector :
    matchLabels :
        type : db

  template :
    metadata :
        name : postgres-pod

        labels :
            type : db

    Spec :
        containers :
            - name : ~~postgres~~ mydb
              image : postgres
              env :
                - name : POSTGRES_PASSWORD
```

```
valueFrom:
  secretKeyRef:
    name: postgres-secret
    key: password
- name: POSTGRES_USER
  valueFrom:
    secretKeyRef:
      name: postgres-secret
      key: username
- name: POSTGRES_DB
  valueFrom:
    secretKeyRef:
      name: postgres-secret
      key: dbname
...
```

→ To ~~decode~~ decode the password which is encrypted, search in google as "base64 to string", there paste the code, you will get actual password.

Service :-

It is used for portmapping and making your application externally available (i.e we can access from browser)

→ It is classified into 3 types

1. NodePort
2. Load Balancer
3. Clusterip
   i) Headless

① NodePort :-

It is used for network load balan

-cing :

example :-

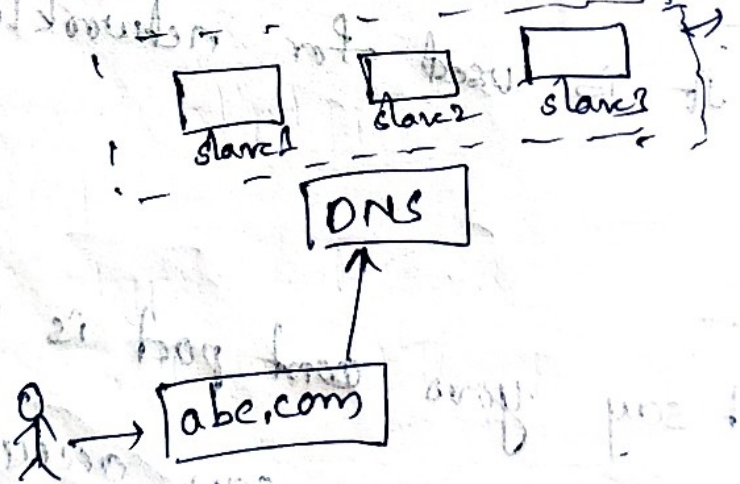
let say your one pod is running on the 1st slave, you can access that application using 1st slave ip address, you can't access with the ip address of other slaves which are in the cluster.

→ So when you use nodeport, you can access from other slaves also, this is called as Network Load balancing.

② LoadBalancer :- (also do n/w load balancing)

It will generate one ip for the entire cluster, nodo register that one ip in the DNS, it will work when you delete some slaves or add some slaves in the cluster.

Scenario:-



for the entire cluster generate one ip address and register in the DNS server

→ This kind of feature is only available in

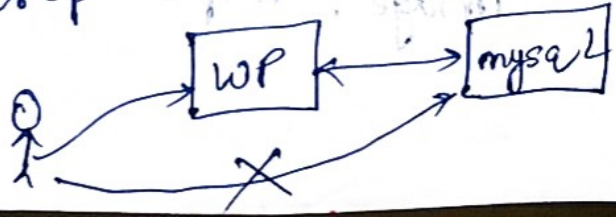
kubernetes only, and that too on managed kubernetes

③ Clusterip :-

It is used when some pods in the cluster which don't want to communicate with external world but they have to communicate with other pods in the cluster internally.

Scenario :-

lets say we WordPress and mysql Now wordpress can communicate with mysql and vice versa. but customers can access only wordpress but not the mysql database → The database access can't be given to external world, in this scenario we use clusterip.



① NodePort :-

(*) Create a service definition file for pod mapping of httpd Pod.

→ first create httpd Pod definitionfile and then create service definitionfile based on the labels given in the definitionfile.

o vim definitionfile.yml

```
---
apiVersion: v1
kind: Pod
metadata:
    name: httpd-pod
    labels:
        type: webserver
        author: intelliqit
Spec:
    containers:
        - name: myhttpd
          image: httpd
...
```

vim service1.yml

```
---
apiVersion: v1
kind: Service
metadata:
    name: httpd-service
    labels:
        author: intelliqit
Spec:
  type: NodePort
  ports:
    - targetPort: 80
      port: 80
      nodePort: 30001
  selector:
      author: intelliqit
      type: webserver
...
```

→ targetPort :- It is nothing but containerPort

nodePort :- It is nothing but hostPort

port :- It is service port

→ The service files make changes based on the labels given in the selector section

→ To get the list of services.

```
kubectl get svc
```

Service port:-

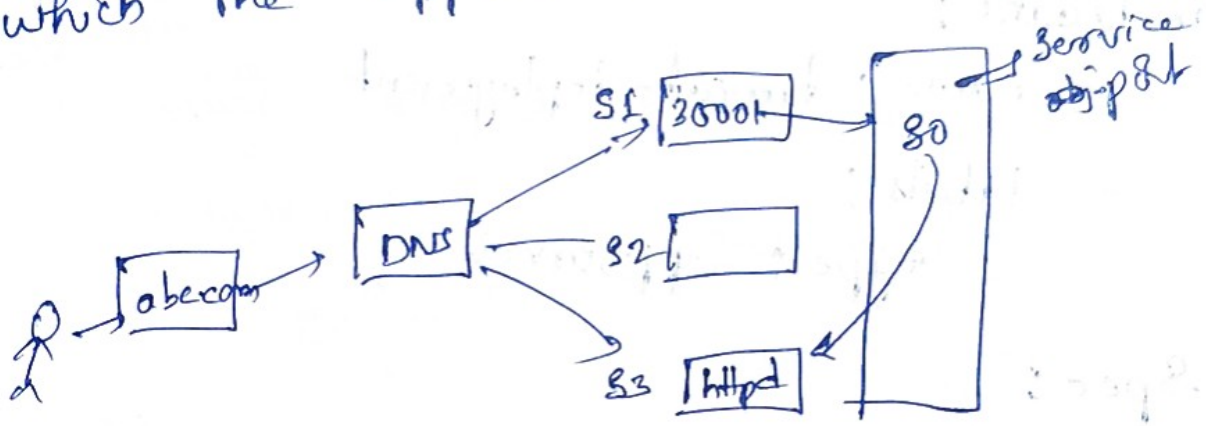
It is the common port mapping between the DNS and the slaves.

Let's say, we have 3 slaves linked with DNS, and my httpd application is running on 3rd slave, if the DNS is sending request to 3rd slave then httpd will respond, or if it is sending request to 1st and 2nd slave, then it creates a problem.

→ To overcome that problem we are assigning

it with service port.

→ Then service port take the request from DNS, and sends it to the slave on which the application is running.



→ nodePort range starts from 30,000 to 32767

→ If you give nodePort it will assign that number

→ If you don't give nodePort number, it will automatically do port mapping for nodePort (hostPort)

(*) Create a tomcat Deployment and service file for it.

vim definitionfile.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: tomcat-deployment
    labels:
        type: appserver

Spec:
    replicas: 2
    selector:
        matchLabels:
            type: appserver
    template:
        metadata:
            name: tomcat-pod
            labels:
                type: appserver
        Spec:
            containers:
                - name: mytomcee
                  image: tomee
...
```

vim service2.yml

```
---
apiVersion: v1
kind: Service
metadata:
    name: tomcat-service
    labels:
        type: appserver

Spec:
    type: NodePort
    ports:
      - targetPort: 8080
        port: 80
        nodePort: 30002
    selector:
        type: appserver

...
```

→ Deploy both the files in the cluster and access from any ip address of any slave, application will respond. This is called network load balancing

② LoadBalancer :- It assigns one unique ip for entire cluster; with that ip we can access the application from browser.

(*) Create a nginx Pod and service file

vim definitionsfile.yml

```
---
apiVersion : v1
kind : Pod
metadata :
    name : nginx-pod
    labels :
        author : intelligit
        type : proxy
Spec :
    containers :
        - name : mynginx
          image : nginx
```

vim service3.yml

```
---
apiVersion: v1
kind: Service
metadata:
    name: nginx-service
    labels:
        type: proxy
        author: intelliqit

Spec:
    type: LoadBalancer
    ports:
    - targetPort: 80
      port: 80
      nodePort: 30008
    selector:
        type: proxy
        author: intelliqit

...
```

③ Clusterip :-

(*) Create postgres Pod, see that it should not be accessed by the external world.

→ clusterip is the default service of kubernetes cluster.

vim definitionfile.yml

```
---
apiVersion: v1
kind: Pod
metadata:
    name: postgres-pod
    labels:
        type: db
        author: intelliqit.

Spec:
    containers:
        - name: mydb
          image: postgres

...
```

vim service4.yaml

```
---
apiVersion: v1
kind: Service
metadata:
    name: postgres-service
    labels:
        type: db
        author: intelliqit

Spec:
    ports:
      - targetPort: 5432
        port: 5432

    selector:
        type: db
        author: intelliqit
...
```

→ We are not giving access to external world, so we are not mentioning nodePort,

Kompose :- (docker compose + kubernetes = Kompose)

→ Create one docker compose file and apply the command as

`kompose convert`

It will convert that file into all necessary definition files.

Install kompose :-

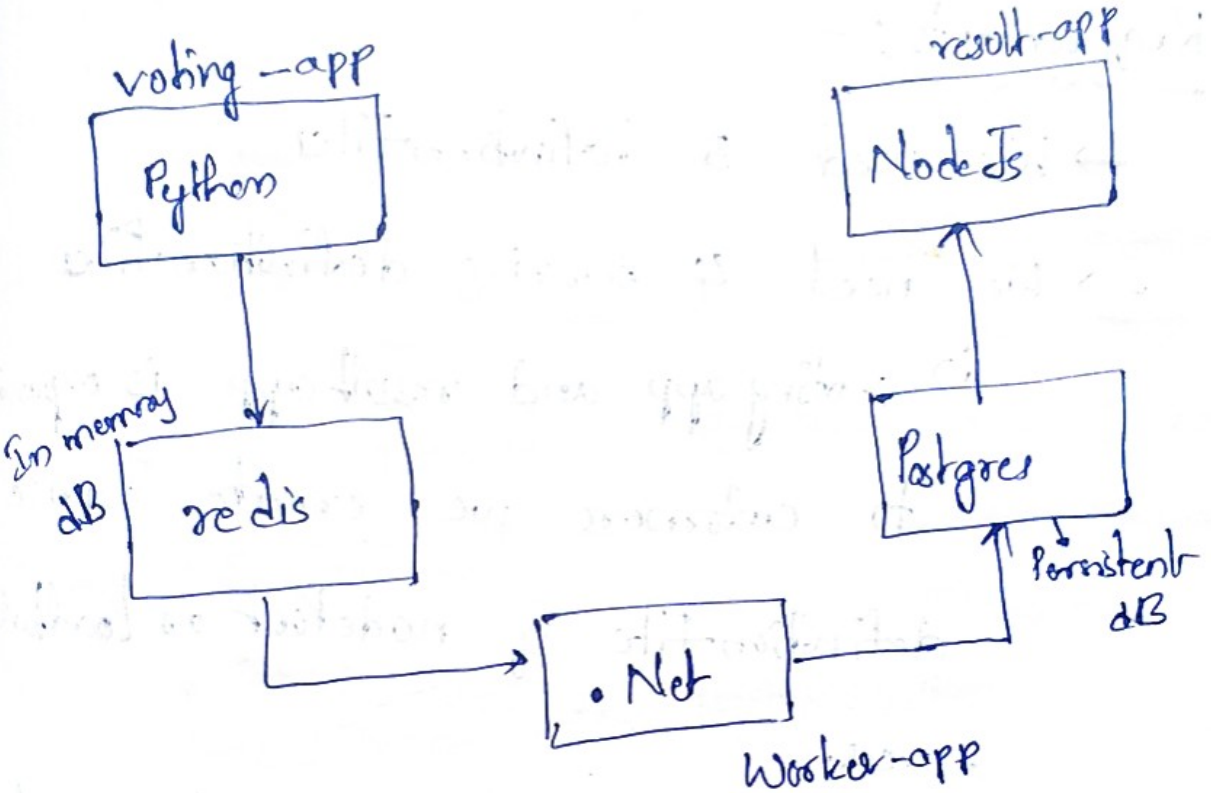
Search in google and install kompose

eg:- Design docker compose file for mysql and wordpress and convert into kubernetes definition files

`kompose convert`

Project :- Voting system

Architecture :-



→ voting app is created by python jango frame work. , it is used to cast their votes

→ redis , is used to collect the data and pass through the worker app

→ Worker app will filter the data and send it into postgres

→ Then postgres stores the data permanently

→ Result-app is created by NodeJs, and is used to view the results.

Requirements :-

→ We need 5 definition files

→ We need 4 service definition files

(i) voting app and result app is exposed to customers we, create service definition file of nodePort or loadbalancer.

(ii) redis and postgres are not exposed to customers and they are used to communicate internally, so create 2 clusterip service files.

→ For worker, it is just filtering the data and passing through to postgres it is not exposed to customers.

→ create a folder

`mkdir project`

→ Inside the folder create 5 deployment definitionfiles and 4 service definitionfile

→ ① vim voting-app-deployment.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: voting-app-deployment
    labels:
        name: voting-app
        author: intelliqit
Spec:
    replicas: 2
    selector:
        matchLabels:
            name: voting-app
    template:
        metadata:
            name: voting-app-pod
            labels:
                name: voting-app
```

```
    Spec:
      containers:
        - name: mj-voting-app
          image: dockersamples/example
                 votingapp_vote
...
```

② vim result-app-deployment.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: result-app-deployment
    labels:
      name: result-app
      author: intelligit

Spec:
  replicas: 2
  selector:
    matchLabels:
      name: result-app

  template:
    metadata:
      name: result-app-pod
      Labels:
```

```
        name : result-app
    Spec :
      containers :
        - name : my-result-app
          image : dockersamples/example
                  votingapp-result
...
```

③ → vim redis-app-deployment.yml

```
---
apiVersion : apps/v1
kind : Deployment
metadata :
    name : redis-app-deployment
    Labels :
        name : redis-app
        auther : intelliqit
Spec :
  replicas : 1
  selector :
    matchLabels :
        name : redis-app
  template :
    metadata :
        name : redis-app-pod
        Labels :
```

```
        name: redis-app
    spec:
      containers:
        - name: my-redis-app
          image: redis
...
```

④ → vim postgres-app-deployment.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: postgres-app-deployment
    labels:
        name: postgres-app
        author: intelliqit
spec:
  replicas: 1
  selector:
    matchLabels:
        name: postgres-app
  template:
    metadata:
        name: postgres-app-pod
        labels:
            name: postgres-app
    spec:
      containers:
```

```
    - name : my-postgres-app
      image : postgres
      env :
        - name : POSTGRES_PASSWORD
          value : intelliqit
        - name : POSTGRES_USER
          value : myuser
        - name : POSTGRES_DB
          value : mydb
```

(5) vim worker-app-deployment.yml

```
---
apiVersion : apps/v1
kind : Deployment
metadata :
    name : worker-app-deployment
    Labels :
        name : worker-app
        author : intelliqit
Spec :
    replicas : 1
```

```
    selector:
      matchLabels:
        name: worker-app
  template:
    metadata:
      name: worker-app-pod
      labels:
        name: worker-app
    spec:
      containers:
        -name: my-worker-app
         image: dockersamples/
          examplevotingapp_worker
...
```

→ Now create service files for voting app and resultapp.

① vim voting-app-service.yml

```
---
apiVersion: v1
kind: Service
metadata:
    name: voting-app-service
```

```
    labels :
        author :
        author : intelliqit

Spec :
    type : LoadBalancer
    ports :
      - targetPort : 80
        port : 80
        nodePort : 30008
    selector :
        name : voting-app
```

...

② vim result-app-service.yml

```
---
apiVersion : v1
kind : Service
metadata :
    name : result-app-service
    labels :
        author : intelliqit

Spec :
    type : LoadBalancer
    ports :
      - targetPort : 80
        port : 80
```

```
    nodePort : 30009
  selector :
      name : result-app
...
```

③ vim redis-app-service.yml

```
---
apiVersion : v1
kind : Service
metadata :
      name : redis-app-service
      labels :
          author : intelliqit
Spec :
    ports :
        - targetPort : 6379
          port : 6379
    selector :
          name : redis-app
...
```

④ vim postgres-app-service.yml

```
---
apiVersion : v1
```

```
kind: Service
metadata:
    name: postgres-app-service
    labels:
        ~~name~~:
        author: intelliqit
Spec:
    ports:
        - targetPort: 5432
          port: 5432
    selector:
        name: postgres-app
...
```

→ Now deploy all the 9 files and accen from the browser, via

clusterip : nodePort

→ For the above code, for all the files just create a docker compose file and convert into kompose, it will create all 9 definition files

vim myapp.yml

```
---
version : '3.8'
services :
    my-voting-app :
        image : dockersamples/example
                votingapp_vote
        ports :
          - 5050 : 80
        deploy :
          replicas : 2
    redis-app :
        image : redis
        ports :
          - 6379 : 6379
    worker-app :
        image : dockersamples/examplevoting
                app_worker
    postgres-app :
        image : postgres
        ports :
          - 5432 : 5432
        environment :
          POSTGRES_PASSWORD : sudarshan
          POSTGRES_USER : myuser
```

```
    POSTGRES_DB: mydb
result-app:
    image: dockersamples/example
           votingapp_result
    ports:
      - 6060:80
    deploy:
      replicas: 2
```

...

```
docker stack deploy -c myapp.yml voting app
```

→ To see the stack files created

```
docker stack ps
```

(or)

use that same docker compose file and convert it into kompose, it will create necessary files, take that files and deploy into cluster.

Kubernetes:-

Memory/resource allocation:-

→ We can specify that how much amount of memory and cpu can be used by the pods.

→ In this we have two components.

(i) request

(ii) Limits

Limits:- It is the maximum amount of resources that pod can consume or ask for even

request:- This is the minimum amount of resources that should be given to the pod

→ Depending on the work doing by the pod it can ask for, but if the cluster contains that hardware it can allocate order it won't.

eg:- vim requestLimits.yml

```
---
apiVersion : v1
kind : Pod
metadata :
    name : nginx-pod
    labels :
        author : intelliqit

Spec :
    containers :
      - name : myngnix
        image : nginx
        resources :
            requests :
                memory : "64Mi"
                cpu : "250m"
            Limits :
                memory : "128Mi"
                cpu : "500m"
...
```

vim requestLimits.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:
      type: proxy
Spec:
    replicas: 3
    selector:
      matchLabels:
        type: proxy
    template:
      metadata:
        name: nginx-pod
        labels:
          type: proxy
      Spec:
        containers:
          - name: mynginx
            image: nginx
            resources:
              requests:
```

```
        memory: "64Mi"
        cpu: "250m"
    limits:
        memory: "128Mi"
        cpu: "500m"
...
```

Node affinity :-

The pod which is created will be attracted towards that node only. i.e the pod is created on that particular node only.

→ To perform node affinity, first you have to label your slave

```
kubectl label nodes node_name/node_id
key = value
```

eg:- kubectl label nodes node_name/node_id

slave1 = intelliqit1

vim affinity1.yml

```
---
apiVersion: v1
kind : Pod
metadata:
    name : httpd-pod
    labels:
      type: webserver

Spec:
  containers:
    - name : myhttpd
      image : httpd
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuring
                Execution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: slave1
              operator: In
              values:
                - intelliqit1

...
```

vim affinity2.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: httpd-deployment
    Labels:
        type: webserver
Spec:
  replicas: 2
  selector:
    matchLabels:
        type: webserver
  template:
    metadata:
        name: httpd-pod
        labels:
            type: webserver
    Spec:
        containers:
            - name: myhttpd
              image: httpd
        affinity:
```

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
      - matchExpressions:
          - key: slav1
            operator: In
            values:
              - intelliqit1.
```

. . .

Taint and Tolerations :-

Taint :- It is exactly opposite of node affinity.

→ If you taint a machine, the pods which are we create are never created on that tainted machine.

→ These pods will be created on untainted machines.

To taint
kubectl taint nodes node_name ~~key=value~~
: key = value : NoSchedule

To untaint
kubectl taint nodes node_name key=value : NoSchedule -

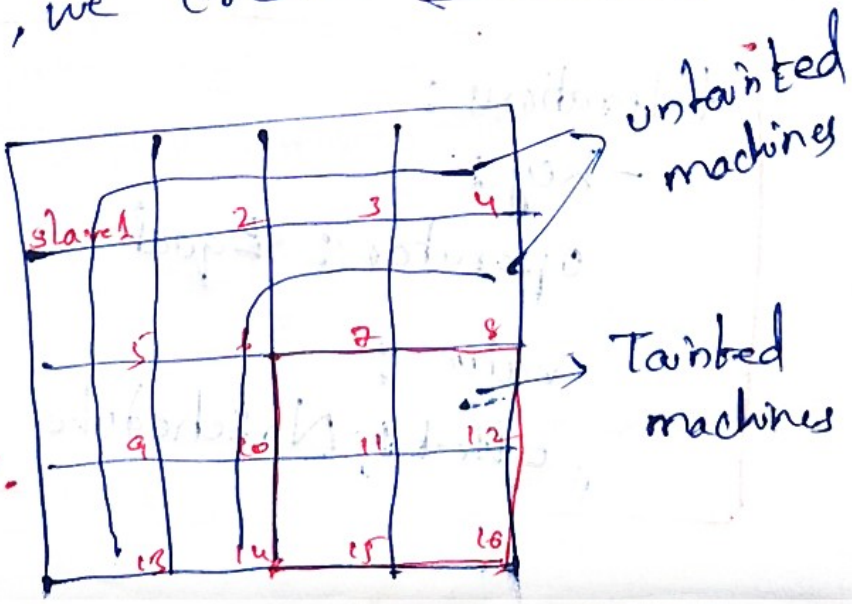
Tolerations :-

→ To run the pod on the tainted machines we use tolerations.

eg:- In realtime we have many servers and we tainted some servers for using databases. they are kept secure.

→ In this case no pods must be created on those tainted servers.

→ Then to create pods on the tainted servers, we create tolerations.



→ Regular pods will run on untainted machines and database related pods will run on tainted machines by using tolerations.

→ To taint a machine

```
kubectl taint nodes node_name key=value : NoSche
      dule
```

→ To untaint a machine

```
kubectl taint nodes node_name key=value : No
  Schedule-
```

Syntax :- for tolerations

```
tolerations :
 - key :
   operator : Equal
   value :
   effect : NoSchedule
```

(8) Create a Pod definition file to run on the tainted machine.

```
---
apiVersion : v1
kind : Pod
metadata :
    name : tomcat-pod
    labels :
        type : webserver
        auther : intelliqit

Spec :
    containers :
        - name : mytomcat
          image : tomee
    tolerations :
        - key : slave1
          operator : Equal
          value : intelliqit1
          effect : NoSchedule

...
```

(*) Create a tomcat-deployment file to run on tainted machines

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: tomcat-deployment
    labels:
        type: appserver
        author: intellejit
Spec:
  replicas: 3
  selector:
    matchLabels:
        type: appserver
  template:
    metadata:
        name: tomcat-pod
        labels:
            type: appserver
    Spec:
      containers:
          - name: mytomcat
```

```
    image: tomee
  tolerations:
    - key: slave1
      operator: Equal
      value: intelliqit1
      effect: NoSchedule
...
```

Helm :-

→ It is the package management solution of kubernetes.

→ It creates work easier

→ No need of creating definitionfiles and service files.

→ With a single command we create definitionfile and service file and exposed to the customers.

→ It is used for generic applications only

→ If the organisation created a customised

software then Helm won't work, then depend on definitionfiles only.

→ first install helm
→ For managed K8s, helm is pre installed.
→ In helm we create charts

step-I :- `mkdir helm`

`cd helm/`

step-II :- Here you create helm charts

`helm create chart_name`

eg :- helm create myngnx

step-III :

Now go inside the chart

`cd myngnx`

There you find a folder called

values.yml

Then edit the file, whichever you want like image name, Service type etc.

step-IV:-
Now to run/install the chart

```
helm install release_name chart_name
```

eg:- helm install newnginx mynginx

→ There will be helm charts available in artifacthub.io

step-V:- To uninstall the chart

```
helm uninstall release_name
```

→ To see the list of helm's installed

```
helm list -a
```

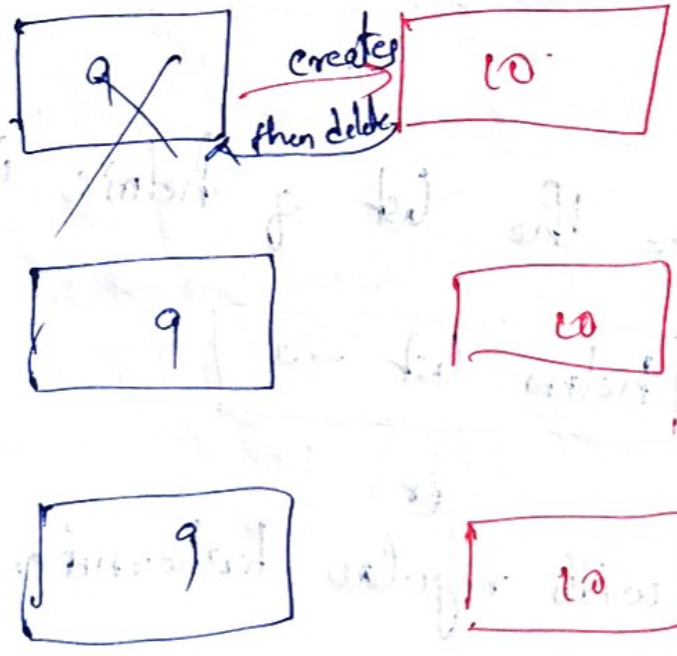
(or)

go with regular kubernetes commands

Rolling update :-

→ In docker swarm the rolling update is done like, lets say we have 3 replicas it will down the first one and update it, when it comes into running condition it performs rolling update on next replica. like wise it goes on.

→ but in kubernetes, it will create a new replicas of the version you want when it comes into running condition it deletes the older replica.



→ In this we have different types of strategies for rolling update, they are

(i) Recreate strategy

(ii) Blue-green "

(iii) Carneny "

* Create strategy:-

From the command line, update nginx

```
kubectl set image deployment/nginx-deployment
nginx = nginx:1.25
```

```
kubectl set image deployment/deployment_name
container_name = image_name:version
```

(ii) Recreate strategy:-

In this, first it deletes the older version and creates the new replicas with latest version (which version you want),

In this case customer will experience down time.

vim definitionfile.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    Labels:
      type: proxy
      author: intelliqit
Spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      type: proxy
  template:
    metadata:
      name: nginx-pod
      labels:
        type: proxy
        author: intelliqit
```

```
Spec:
  containers:
    - name: nginx
      image: nginx:1.24
      ports:
        - containerPort: 80
          hostPort: 8181
```

→ Now

```
kubectl apply -f definitionfile.yaml
```

→ Now deployment pod is created, then operate rolling update on it.

```
kubectl set image deployment/nginx-deployment
nginx = nginx:1.25
```

→ This will do update/migrate nginx:1.24 to nginx:1.25

—x—

⑧ Blue-green deployment:

→ To upgrade from one version to another version, there might be compatability issues.

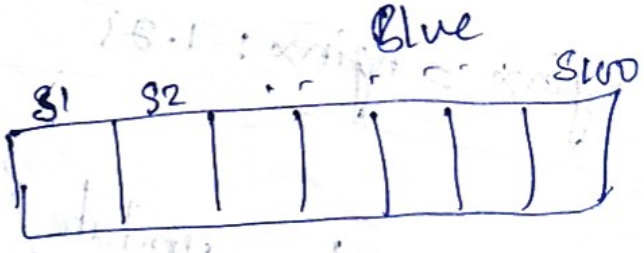
→ To avoid those issues we go with blue green deployment.

→ Let's say nginx : 1.24 and upgrading to nginx : 1.25, it doesn't make any compatability issues,

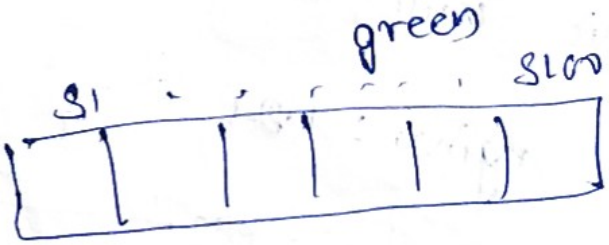
→ If you migrating from nginx : 1.24 to nginx : 2, bigger version, it may give compatability errors.

→ let's

x : version



y : version



→ They are using x:version on 100 servers and it is exposed to the customers. and they are accessing it.

→ Now they want to upgrade from x:version to y:version, without experiencing the down time of customers.

→ For this they won't disturb the older x:version which is running on 100 servers they call it as "Blue"

→ Now they create 100 servers and install with y:version, they call it as "green", and it also exposed to customers, when they see that y:version is behaving stable and when customers are accessing, then they delete the older x:version, and now they called y:version as blue version.

Disadvantages:-

→ It will increase the servers for certain time.

→ Cost gets increased.

vim blue-deployment.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: tomcat-deployment
    labels:
        type: appserver
        author: intelliqit
Spec:
    replicas: 3
    selector:
        matchLabels:
            type: appserver
    template:
        metadata:
            name: tomcat-pod
            labels:
                type: appserver
```

```
Spec:
    containers:
        - name: tomcat
          image: tomcat:9
          ports:
            - containerPort: 8080
              hostPort: 8181
```

...

→ The above file/architecture is called as blue-deployment, because it is older version which is running.

→ Now I have to upgrade to tomcat:10, Create same architecture definition file and deploy. If it is working properly then delete the older version.

vim green-deployment.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: tomcat-deployment
```

```
    labels:
      type: appserver
      author: intelliqit
Spec:
  replicas: 3
  selector:
    matchLabels:
      type: appserver
  template:
    metadata:
      name: tomcat-pod
      labels:
        type: appserver
    Spec:
      containers:
        - name: tomcat
          image: tomcat:10
          ports:
            - containerPort: 8080
              hostPort: 8181
```

...

→ After deleting the old version, now this version is called blue-deployment and coming versions in future are green deployment

Errors and debuging :-

① crashloopbackoff :-

It indicates that the container in a pod is failing repeatedly and kubernetes is apply a backoff strategy before attempting to restart the container again.

→ To troubleshoot and resolve this issue, you need to examine the container logs, pod status and any relevant configurations to identify and fix the underlying problem.

→ Common issues are :-

(i) incorrect container configurations
(ii) missing dependencies
(iii) resource limitations

```
kubectl logs pod_name
```

```
kubectl describe pod pod_name
```

⑧ ImagePullBackOff :-

It occurs when a pod is unable to pull the specified image

→ This error typically indicates a problem with fetching the container image specified in the pod-definition from the container registry.

Common reasons for this error:

① Incorrect image name or tag:

② Registry authentication issues: (username/password)

③ Image availability :- ensure that image is available or not in the registry.

④ Network issues:

⑤ Registry rate Limiting:- Some container registries have rate-limiting polices. If you exceed the limits you may encounter issues pulling images.

⑥ Proxy configurations.

⑦ Livenessprobe failure :=

It is a mechanism to determine whether a container within a pod is healthy and running as expected.

⑧ Canary deployment :

→ This is done on scaling mechanism.

Let's say we have 100 pods running in the cluster.

→ First we scale it to 95 pods, the older version is running on 95 pods and remaing 5 pods are updated to new version.

→ Then the users are allowed to new version, if they think that new version is working stable, then they scale down

5 more pods and scale up to 10 pods like wise they update all the pods running in the cluster.

old version

spec:
  replicas: 95

New version

spec:
  replicas: 5

→ first create deployment definition file with new version and scale it upto 5 replicas and scale down the older version deployment to 95

→ The older version 95 and new version 5 replicas have the desired state of 100;

→ If it is working properly then scale down older version to 90 and scale up new version to 10 ... do so on upto new version is updated in all the 100 pods

→ Software version format

x . y . z → patch update

major update minor update

Interview

→ whenever we have minor and patch updates we are going with regular rolling update process and whenever we have major updates we are going with blue-green deployment in our organisation

Volumes :-

→ In docker swarm we don't have handling volumes, i.e if a container with some data is crashed, then docker swarm will automatically creates a new container but it don't have the data where as in kubernetes the data will present.

→ The persistence of data will be present in kubernetes.

→ We have two types of volumes

(i) emptydir

(ii) Persistant Volume.

(i) emptydir

Syntax:

```
volumeMounts:
  - name: → name of the volume
    mountPath: → where it is mounted
```

(*) Create a redis pod and volume

```
---
apiVersion: v1
kind: Pod
metadata:
    name: redis-pod
    labels:
        type: db

Spec:
    containers:
        - name: myredis
          image: redis
          volumeMounts:
              - name: redis-volume
                mountPath: /data/redis

    volumes:
        - name: redis-volume
          emptyDir: {}
```

→ Now go inside the pod, in the /data/redis folder create some files

→ then make that pod to be crashed i;e kill the default process of the pod then pod will be crashed

→ To kill the pod we need PID

PID (Process ID) :

```
kill PID
```

→ To know PID → `ps -ef` → linux command

→ This ps must be installed in the pod

→ To install ps, go inside the pod and install procps software

```
sudo apt-get update
sudo apt-get update -y procps
```

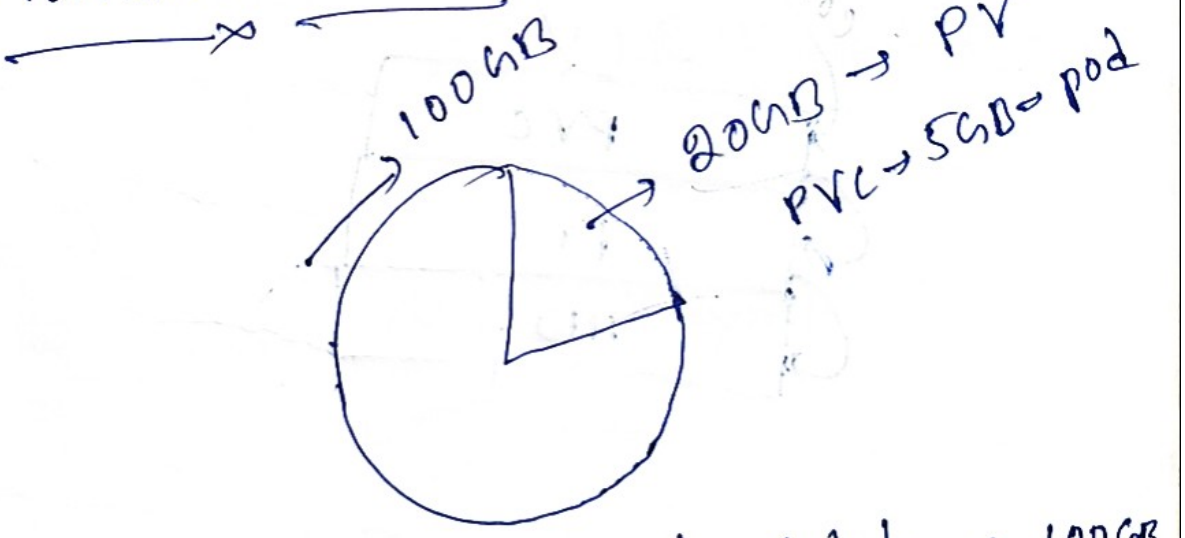
→ After installing then pickup the PID and kill it.

→ Automatically new pod is created, in that there will be our previous data.

→ In pods data will be present, but when the pod crashes, and a new pod comes but the softwares installed in it will be lost.

→ To keep softwares also persistent in the pod, create the docker file keep all the necessary softwares you want and create image out of it, then use that image in the Pod definition file.

② Persistant Volume :-

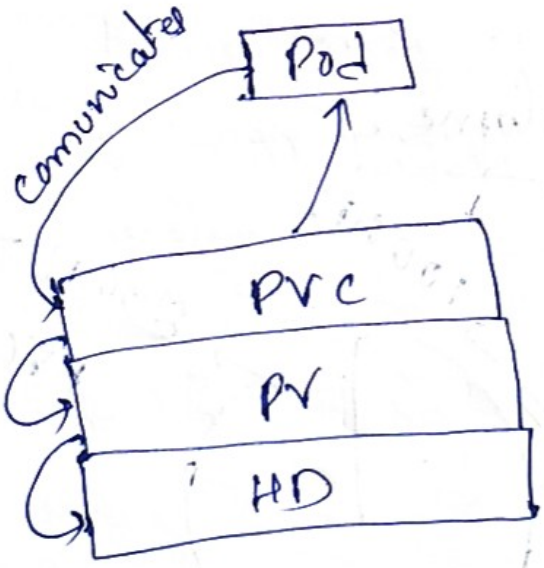


→ Lets say we have hard disk of 100GB from that 20GB is reserved for volume

it is called Persistant Volume, then when we create a pod, and it requires 5GB, then it claims the 5GB from 20GB only is called persistent volume claim (PVC)

→ For next pod it requires 2GB, it is given from PV only.

→ Pods directly can't talk to PV, so via PVC we claim the required storage.



⊛ Create a PV of 20GB and PVC definition file.

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
    name: task-pv-volume
    labels:
      type: local

spec:
    storageClassName: manual

    capacity:
      storage: 20Gi

    accessModes:
      - ReadWriteOnce

    hostPath:
      path: /mnt/data
```

...

vim persistentVolumeclaim.yml

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

name: task-pv-claim

Spec:

storageClassName: manual

accessModes:

- ReadWriteOnce

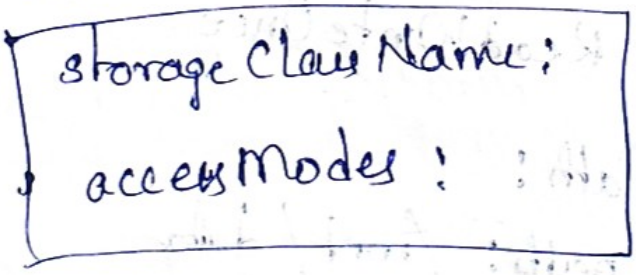
resources:

requests:

storage: 5Gi

...

→ The linking of these two files done based on



⊛ Create a pod definition file to use the pvc.

vim pod-volumes.yml

---

apiVersion: v1

```
kind: Pod
metadata:
  name: nginx-pod
  labels:
    author: intellipaat
    type: proxy
spec:
  containers:
    - name: mynginx
      image: nginx
      volumeMounts:
        - name: task-pv-storage
          mountPath: /usr/share/nginx/
                     html
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
```

. . .

→ The ~~pods~~ persistent volumes work only the Pod definition files, and it won't work on deployment, because when

pod crashes, a new pod gets created automatically with the same name, if the same name is there, with that same name it is linked with pvc.

→ But in case of deployment, pod gets created but it won't come up with same name therefore link between pvc and deployment gets unlinked and we can't recover the data.

→ To overcome this problem we go with statefullSet definition files.

StatefullSet :-

→ If I create 3 replicas with mysql

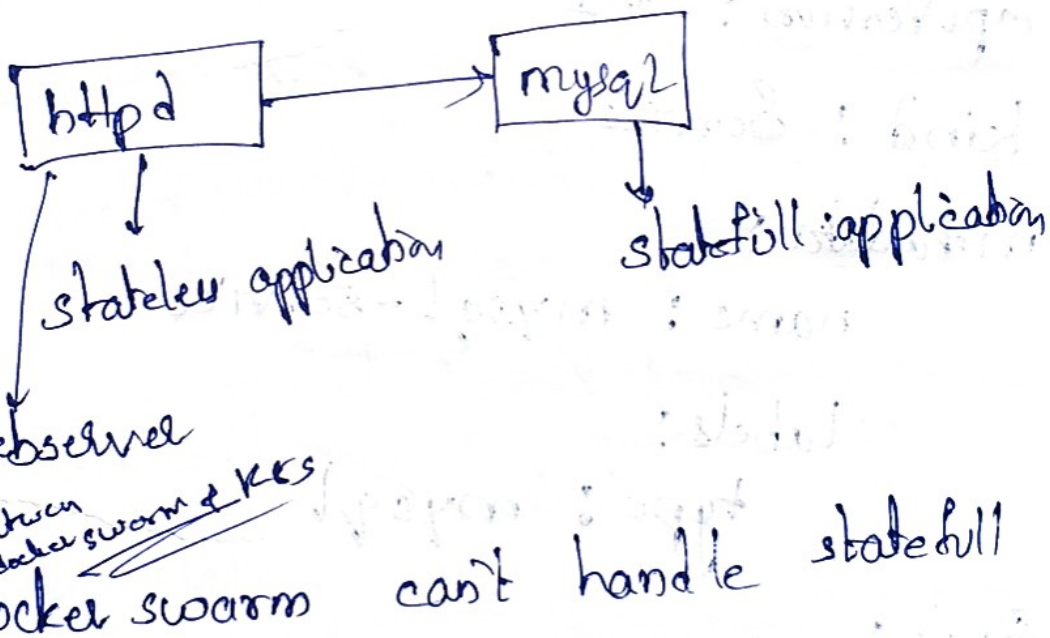
→ It creates like mysql - 0 → master
- 1 - slave
- 2 - slave

→ The master pod continously keeps on checking the slaves

→ while deleting also it deletes in reverse way.

→ Deployments are good for stateless applic - always where as StatefullSets are good for statefull applications.

→ Statefull applications are those applicati - ons which has persistence of data. (like databases)



☆☆ Difference between docker swarm & k8s

→ Docker swarm can't handle statefull applications, it is excellent on stateless applications.

→ The master pods the newly created slaves whether it is linked with pvc

→ The master pod always communicates with pods via a service object called as headless

(*) Create mysql database with 3 replicas using statefullset.

→ first create service definition file

```
---
apiVersion: v1
kind: Service
metadata:
    name: mysql-service
    labels:
      type: mysql
Spec:
  clusterIP: None  /* it means headless */
  ports:
    - name: tcp
      protocol: TCP
      port: 3306
```

```
    selector :
        type : mysql
...
---
apiVersion : apps/v1
kind : StatefullSet
metadata :
    name : mysql-statefull
    labels :
        type : db
Spec :
    replicas : 2
    selector :
        matchLabels :
            type : mysql
    serviceName : mysql-service
    template :
        metadata :
            name : mysql-pod
            labels :
                type : mysql
        Spec :
            containers :
```

```
- name : mydb
  image : mysql
  env :
    - name : MYSQL_ROOT_PASSWORD
      value : intelliqit
  volumeMounts :
    - name : task-pv-storage
      mountPath : /var/lib/
      : mysql
volumes :
  - name : task-pv-storage
    persistentVolumeClaim :
      claimName : task-pv-claim
...
```

Autoscaling :-

→ whenever cpu, memory utilization is happening we want to increase the no. of pods or decrease the no. of pods this is done based on the

object called as "Horizontal Pod Autoscaler"

→ whenever HorizontalPodAutoScaler is applied the pods automatically increase or decrease.

vim autoscaling.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: php-apache
    labels:
        author: sudarshan
Spec:
  selector:
    matchLabels:
        type: php-apache
  template:
    metadata:
        name: php-apache-pod
        labels:
            type: php-apache
    Spec:
        containers:
```

```
    - name : php-apache
      image : intelliqit/mynews
      resources :
        requests :
          cpu : 200m
        limits :
          cpu : 500m
---
apiVersion : v1
kind : Service
metadata :
    name : php-apache
    Labels :
      type :
      auther : sudarshan
Spec :
    ports :
      - targetPort : 80
        port : 80
    Selector :
      type : php-apache
...
```

The command to apply autoscaling is

```
kubectl autoscale deployment name_deploym
ent --cpu-percent=50 --min=1 --max=10
```

→ If cpu crosses the limit of 50% utilization it creates one pod, the pods will maintain the number between 1 to 10

eg:
```
kubectl autoscale deployment php-apache
--cpu-percent=50 --min=1 --max=10
```

→ To chect the horizontal Autoscaler

```
kubectl get hpa
```

→ To check the application for the autoscaling whether it is working or not.

→ Create a pod, and ping it continously. then the load increases then pods also

increases.

```
kubectl run -i --tty lg --rm --image=
busybox --restart=Never -- bin/sh -c
"while sleep 0.01; do wget -q -O-
http://php-apache; done"
```

→ Without giving the command always, it shows whenever the modification is done.

```
kubectl get hpa --watch
```

Prometheus

→ we can check the status of pods, without giving commands, we have a graphical tool called as prometheus.

→ Devops engineer knows the commands and he will check the status by using commands, but other people who develop -ed this application they are interested in seeing the status of application.

→ So just setup the monitoring tool called prometheus, its a GUI.

→ We can have no. of dashboards for prometheus in grafana google, use whichever you want.

→ Prometheus will capture the status from the ~~cluster~~ cluster, and the output is shown on the colour -full screen is done by Graphana

→ Prometheus will capture the information and grafana will display in colourful mode.

→ Now setup Prometheus with the help of helm

→ Go to artifacthub.io

→ Search for prometheus

→ And install it based on the commands shown in it

→ To check whether it is added or not

```
helm repo list
```

→ Next to update the repository

```
helm repo update
```

→ After installing the prometheus, type

```
kubectl get all
```

it will show all the pods it created, then

in the pods search for service pod
of prometheus-grafana

service/prometheus-grafana

for this pod, it is created on clusterIP,
then change it to nodePort/LoadBalancer
to expose it to external world

→ To do this we use a command as

```
kubectl patch svc prometheus-grafana
-p '{"spec": {"type": "LoadBalancer"}}'
```

→ To check it `kubectl get all`

→ It generates public ip and access it from
the browser

→ when grafana loads, the default user
name and password are

```
username : admin
password : prom-operator
```

→ Use any of the dashboards which are available in google, search for

grafana dashboards.

→ Search a dashboard for prometheus in DataSource pannel.

→ Search for a dashboard called as

kubernetes cluster monitoring (via prometheus)

or

kubernetes cluster (7249 - dashboard no)

→ In that you have copy-id, click on it and copy that id in notepad.

→ Then go to grafana dashboard and click on '+' icon and click on 'import dashboard'

→ They will ask the id, give that number, and click on load

helm

→ To add helm repositories which are coming from third parties.

```
helm repo add repository_name repository_url
```

eg:-

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

→ `helm repo update`

→ To see the list of repo's

`helm repo list`

→ To remove the repository

```
helm repo remove repository_name
```

eg:- helm repo remove bitnami

→ We have downloaded bitnami repository

→ from that repository, we can use charts whatever we required.

For example:-

* I want setup wordpress with mariadb

→ first `helm search hub wordpress`

It shows the charts already present in the repo via links/url's

To see the full url address in the repo.

`helm search hub wordpress --max-col-width=0 | less`

→ first create values files, which are required for database, like username, password etc.

vim wordpress-values.yml

---

wordpressUsername: admin

wordpressPassword: admin

wordpress Email: sudarshanswt@gmail.com

wordpress FirstName: Durgam

wordpress LastName: Sudarshan

wordpress BlogName: mywordpress.com

Service:

type: LoadBalancer

...

```
helm install wordpress bitnami/wordpress
--values=wordpress-values.yml
```

→ helm install release_name chart_name_ofrom _repo --values=file_of_your_database_credentials.ym

Modifications done in helm chart

① chart.yaml

→ In this provide metadata i.e in description :- provide some meaningful description here.

→ It is not import.

② template :-

└→ deployment.yaml

→ Edit the deployment.yaml file

→ Edit the containerPort

→ Because the port number is created for nginx, so edit it depending upon your application.

③ values.yaml

→ change image name, Service name, tags.

image name :

tage : Latest / give any version of your image.

Service : LoadBalance / nodePort

replicount :

port :

In managed kubernetes we don't want to create pv and pvc, because the cloud will create them automatically.

→ We just create a deployment file and use the required components.

→ In StatefullSet just create that file and service file,

→ No need of creating pv and pvc's

vim statefullSet.yml

```
---
apiVersion: v1
kind : Service
metadata:
    name : statefullset-service
    labels :
        type : webservice
Spec :
    clusterIP : None
    ports :
      - port: 80
        name : web
    selector:
        type : webservice
---
apiVersion : apps/v1
kind : StatefullSet
metadata :
    name : statefullset-nginx
    Labels :
        type : webserver
```

```
spec:
  replicas : 2
  serviceName : webservice
  selector :
    matchLabels :
      type : webservice
  template :
    metadata :
      name : nginx-pod
      labels :
        type : webservice
    Spec :
      containers :
        - name : mynginx
          image : nginx
          ports :
            - containerPort : 80
              name : web
          volumeMounts :
            - name : my-volume
              mountPath : /usr/share
                          /nginx/html
  volumeClaimTemplates :
    - metadata :
        name : my-volume
```

```
spec:
  accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 5Gi
...
```

(*) To install jenkins using docker container

vim dockerfile

```
FROM jenkins/jenkins:jdk11
USER root
RUN apt-get update
RUN apt-get install -y maven
RUN apt-get install -y git
USER jenkins
```

Init containers :-

A pod can have multiple containers runing apps within it, but it can also have one or more init containers, which are run before the app containers are started

→ Init containers always run to completion.

→ Each init container must complete successfully before the next one starts.

→ If a pod's init container fails, K8s repeatedly restarts the Pod until the init container succeeds.

→ However, if the Pod has a restartPolicy of Never, K8s does not restart the Pod

→ Init container dies itself after execution is successful.

✱ The files which are there in init containers you won't be able to access them (use for security purpose)

Sidecar containers! –

→ A sidecar is a utility container in the Pod and its purpose is to support the main container.

→ It must be paired with one or more containers.

→ It is reusable and can be paired with many type of main containers

Usecase:

Using init and sidecar containers deploy nginx pod. secure (https) and push nginx logs into aws S3

Create a cert file and share it to application container

Collect logs from app container and upload it to aws S3 bucket

Vim nginx-dep.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: myapp
Spec:
  replicas: 2
  selector:
    matchLabels:
      type: myapp-dep
  template:
    metadata:
      labels: myapp-dep
    Spec:
      initContainers:
      - name: ssl-cert-creation
        image: sudarkhan/openssl
        command: ["/bin/sh"]
        args: ["-c", "mkdir -p /etc/nginx/ss
              openssl req -newkey rsa:2046
             -nodes: /nginx/ssl/nginx.key -x509 -d
        volumeMounts:
          - name: ssl-cert
            mountPath: /etc/nginx/ssl
      Containers:
        -name: myapp-dep
         image: nginxhttps
```

```
command : ["/home/awb-reload-nginx.sh"]
ports :
- containerPort : 443
- containerPort : 80
livenessProbe :
    httpGet :
        path : /index.html
        port : 80
    initialDelaySeconds : 30
    timeoutSeconds : 1
volumeMounts :
    - mountPath : /etc/nginx/ssl
      name : ssl-cert
    - mountPath : /etc/nginx/conf.d
      name : nginx-file
    - mountPath : /var/log/nginx
      name : logs
- name : sidecar-logs
  image : sudarshan/awsdi
  command : ["/bin/sh"]
  args : ["-c", "cp /root/scripts/syncs3.sh
          /root/syncs3.sh; chmod 777 /root/
          syncs3.sh; while true; do sh /root
          /syncs3.sh; sleep 10;
          done"]
  volumeMounts :
    - name : logs
      mountPath : /var/log/nginx
```

```
      - name : aws-credentials
        mountPath: /root/.aws/config
        subPath: config
      - name : aws-credentials
        mountPath: /root/scripts/syncs3.sh
        subPath: syncs3.sh
  Volumes:
    - name : ssl-cert
      emptyDir : {}
    - name: logs
      emptyDir : {}
    - name: aws-credentials
      configMap:
          name: "aws-config"
    - name : nginx-files
      configMap:
          name : "nginx-config"
```

vim config-map_nginx.yml

```
apiVersion : v1
kind: ConfigMap
metadata:
    name: nginx-config
data: default.conf: |
```

Server {

Paste all the certificates here.

}

config-map-aws.yml

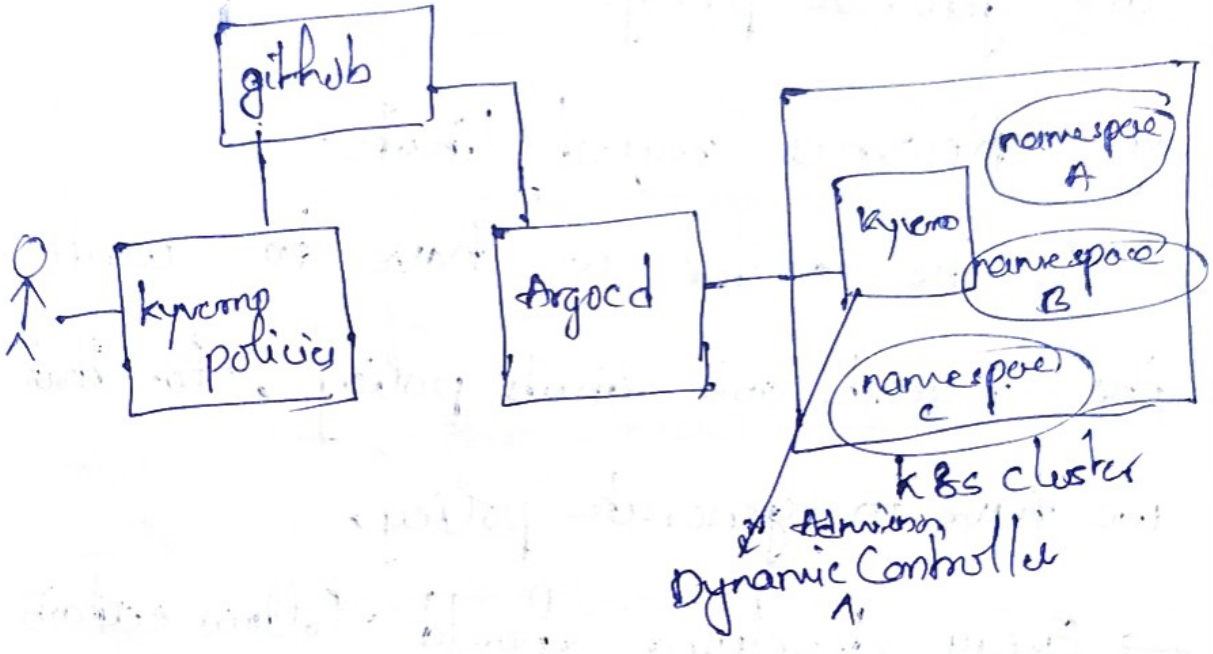
```
apiVersion: v1
kind: ConfigMap
metadata:
    name: aws-config
data:
    config: |
      [default]
      region = us-east-1
    sync3.sh: |
        export AWS_CONFIG_FILE="/root/.aws/config
        export AWS_ACCESS_KEY_ID=
        export AWS_SECRET_KEY=
        aws s3 ~~copy~~ sync /var/log/nginx s3://bucket
                                                name
```

nginx-service.yml

```
apiVersion: v1
kind: Service
metadata:
```

```
    name: nginx-svc
    labels:
      app: nginx
spec:
  type: NodePort
  ports:
    - port: 80
      protocol: TCP
      name: http
    - port: 443
      protocol: TCP
      name: https
  selector:
    type: myapp-dev
```

Enforce Automated K8s cluster security using kyverno policy generator and argocd



k8s governance :- Managing according to the compliance of organisation.

Compliance → set of rules

eg: ① In this organisation, nobody should create a pod with latest tag

② Nobody should create a pod the memory size of 500MB.

etc.

Admission Controller:-

It validate, mutate, verify images and generate policy.

eg:- Resources request/limits.

In the cluster we have to write the requests and limits policy, for that we have to generate policy.

→ Every developer should follow certain rules to create a pod., then what is the guarantee that every developer is following the rule., for that we create

→ As a devops engineer we write admission controller,

eg:- Every pod should have resources requests/limits.

→ We write some yaml files with policies that kyverno will understand, then kyverno will write required webhook configuration for us.

→ I worked on kyverno validate functionality to block the users, if it doesn't meet the standards of our organisation

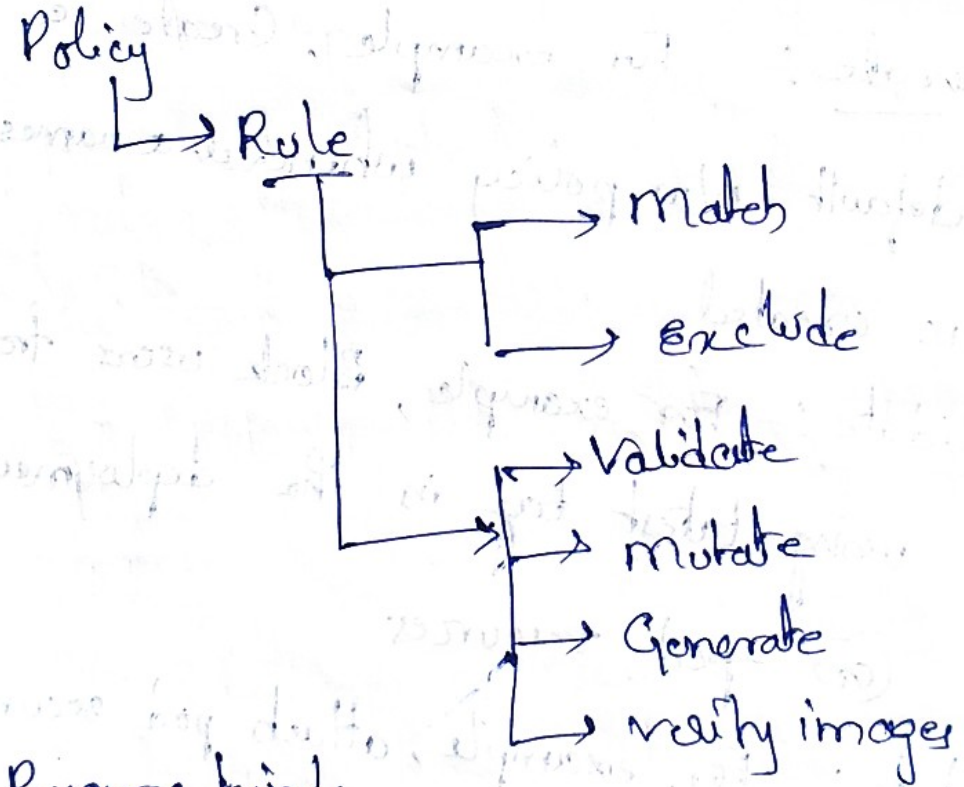
① Generate :- For example, Create a default n/w policy whenever a namespace is created

② Validate :- For example, Block users from using latest tag in the deployment (or) pod resources

③ Mutate :- For, example, attach pod security policy for a pod that is created without any security policy configuration.

④ Verify images :- if the images used in the pod resource are propely signed and valid images

→ A kyverno policy is a collection of rules. Each rule consists of a match declaration, an optional exclude declaration and one of a validate, mutate, generate or verify images declaration. Each rule can contain only a single validate, mutate, generate or verify images child declaration



→ Resource kinds
→ Resource names
→ Labels
→ Annotations
→ Namespace
→ Namespace labels

→ Roles
→ ClusterRoles
→ Users
→ Groups
→ ServiceAccounts

step-①:- first install kyverno on the cluster

→ Install with helm charts/manifest files

step-②:- Install Argocd

step-③:- Copy the link of the policy which is written in yaml file and

`kubectl apply -f "paste the link"`

→ It will install that policy in the cluster

→ This is directly installing policy on cluster without the argocd.

→ The admission controller which is in the cluster will read the policy.

→ To see the kyverno pod

`kubectl get pods -A | grep kyverno`

→ To see the logs of kyverno pod

`kubectl logs kyverno.pod id`

RBAC

Role Based Access Control

→ It gives authorization over cluster. for users, dev team, testing team, devops team. etc. like creating/deleting pods, to see the list of pods,

→ Resources on cluster are

pods, deployments, secrets, statefulset, Replicaset, ReplicationController etc

→ Verbs :-

get, list, watch, create, update, patch, delete

→ We have Role, and Rolebinding, clusterRole and clusterRolebinding.

→ We have to create a role and bind it to the users

→ Permissions can be applied to user, group or cluster

→ Forbiden error :- On the k8s says that he don't have permission to do the action.

Scenario :-

(*) Only give read access to over the pods in dev namespace.

```
vim role.yml

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: reader
rules:
- apiGroups: [""]  # indicate all api groups
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

types of access

Now bind the role to a user (dev) in dev namespace.

vim rolebinding.yml

```
---
apiVersion: rboc.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: read-pods
    namespace: dev
subjects:
- kind: User
  name: dev
  apiGroup: ""
roleRef:
    kind: Role
    name: reader
    apiGroup: ""
...
```

→ kubectl get role -n dev

→ kubectl get rolebinding -n dev

Role	Access
Developer	— create, Read, Update
monitoring	— Read
Admin	— create, read, update, delete

Creating Users on k8s

step-1: generate a private key

step-2: create Certificate signing request

step-3: Kubernetes certificate authority (CA)

private key	ca.key
certificate	ca.crt

these will be found in /etc/kubernetes/pki

→ Private key, certificate signing request; certificate found in Kube config

Scenario :-

(*) Create a namespace as finance, and create a user and attach role to him.

step-I :-

Create private to sudarshan

```
openssl genrsa -out sudarshan.key 2048
```

(2048 = no. of bits)

step-II :- Create certificate signing request

```
openssl req -new -key sudarshan.key -out
sudarshan.csr -subj "/CN=sudarshan/O=finance
```

step-III :-

copy ca.crt, ca.key to the directory where sudarshan.key is present.

```
cd /etc/kubernetes/pki
```

```
cp ca.key .
cp ca.crt .
```

step-IV: Now sign that key

```
openssl x509 -req -in sudarshan.csr -CA
ca.crt -CAkey ca.key -CAcreateserial -out
sudarshan.crt -days 365
```

step-V:- Now create kubeconfig file for sudarshan

kubectl --kubeconfig sudarshan.kubeconfig set-cluster my-cluster --server https://ip of server:6443 --certificate-authority=ca.crt

To view cluster service

```
kubectl config view
```

step VI: Add user to kubeconfig file..

kubectl --kubeconfig sudarshan.kubeconfig config set-credentials sudarshan --client-certificate path of sudarshan.crt --client-key path of sudarshan.key

→ Now set context

kubectl --kubeconfig sudarshan.kubeconfig config set-context sudarshan-kubernetes --cluster my-cluster --namespace finance --user sudarshan

→ Now go to sudarshan.kubeconfig file and change

current-context: sudarshan-kubernetes

→ Now ~~cp sudarshan.kubeconfig ~/.kube/config~~

Now open copy

cp ~/.kube/config sudarshan.kubeconfig

then edit it

vim sudarshan.kubeconfig

then edit

- context
  user: sudarshan
  name: sudarshan-kubernetes

users:
- name: sudarshan

delete the present data in sudarshan.crt

client-certificate-data: paste sudarshan certificate

client-key-data: sudarshan pvt key

sudarshan.key

Convert them to base64 and paste there.