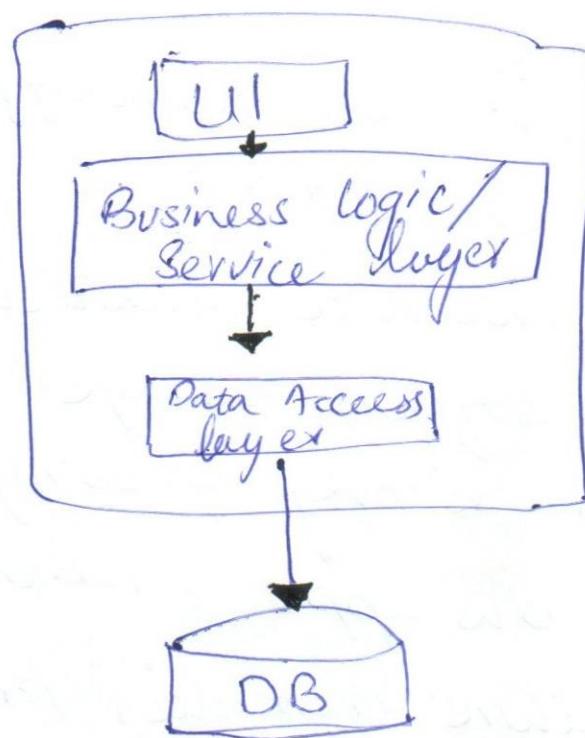


KUBERNETES

* Traditional Architecture :-

- Monolithic : Composed all in one piece



- Monolithic software is designed to be self-contained meaning having all that is needed in itself.

PROBLEM:-

- ① Slow release cycle and are updated relatively infrequently.
- ② Once new release get ready, developers package up the whole system and hand over to the ops team, who then deploy and monitor it.
- ③ In case of hardware failures, the ops team manually migrate it to the remaining healthy servers.

* MICROSERVICES :-

- It is breaking down big monolithic component application into smaller independent app ~~comp.~~
 - Decoupled
 - Scaled individually.
 - Easy to adopt new technology.
- MICROSERVICE COMPLEXITIES :-
- In bigger microservice architecture it is very difficult to configure, manage and keep the whole system running smoothly.
 - ① Data linking b/w apps. problem
 - ② hardware failure remedies problem

What is kubernetes :-

- ① Automated deployment process
- ② No ops team requirement
- ③ Help ops team to transfer/migrate from one server/hardware to another.
- ④ One or thousand servers seems as a single server. to provide ease in deploying.

What is kubernetes Continued

- ① kubernetes select from range of servers and decide how micro service talk to other micro service.
- ② Provide ease for cloud providers

KUBERNETES ORIGIN :-

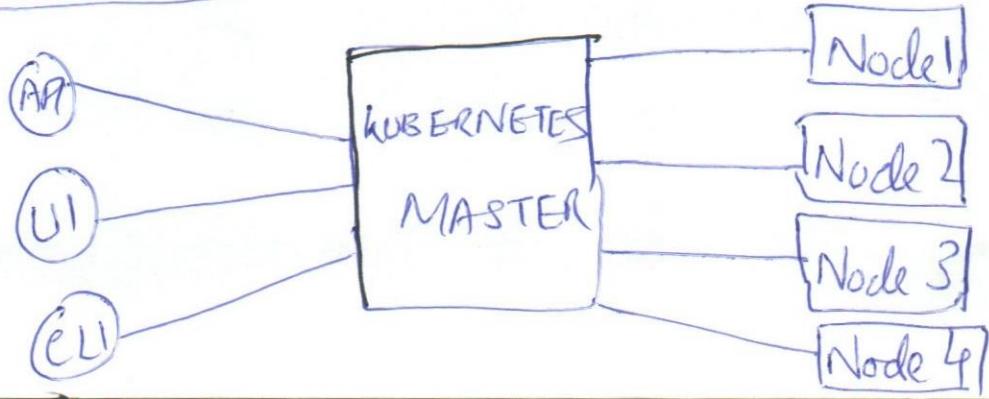
- Google created a system for their internal use with the name "Borg"
- later name changed with "Omega"
- 2014, Google introduced kubernetes an open source system based on the experience

BROADER PERSPECTIVE :-

- Simplify the infrastructure layer.
- Resource allocation.

* KUBERNETES

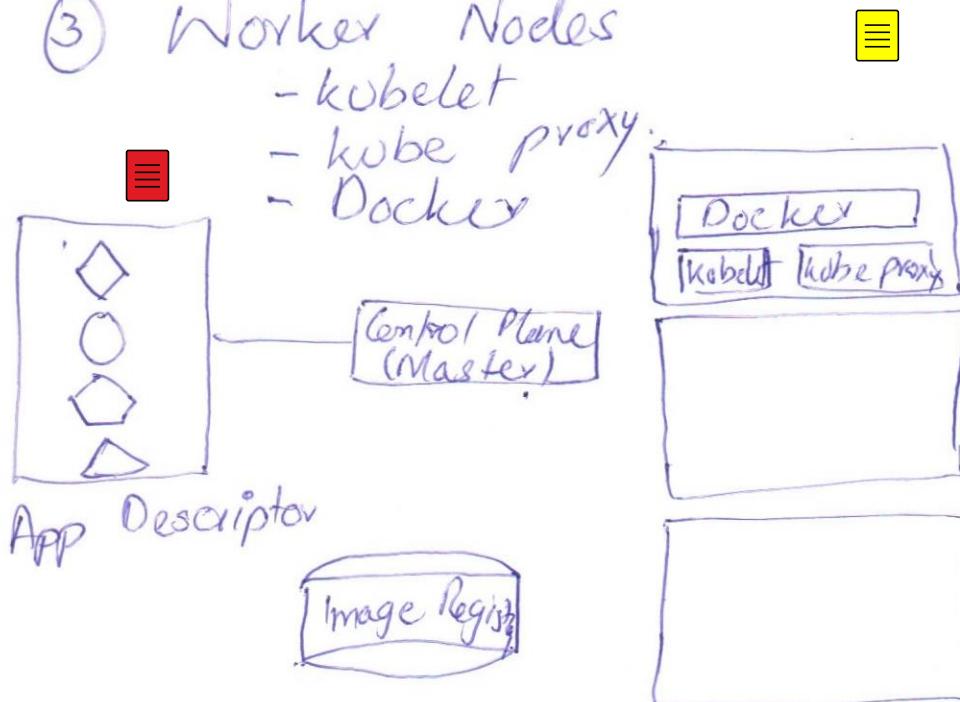
ARCHITECTURE



(2)

KUBERNETES BROADER VIEW

- ① App descriptor.
- ② Control Plane (Master Node)
- ③ Worker Nodes



PODS1) MINIKUBE E KUBECTL :-

Minikube creates single node cluster of kubernetes.

(Master + Worker node at a same time)

minikube start

① To check status:-

minikube status

② To check kubectl cluster info :-

kubectl cluster-info

2) NODES

① To check list of worker & master nodes

kubectl get nodes

② To describe nodes

kubectl describe node minikube

3) ALIAS :-

To short command kubectl get nodes

alias kgn = "kubectl get nodes"

- * Pods it is basically like a wrapper or a separate machine with its own IP, hostname, processes, and so on.
- * Pods deploy single containerised app. on single worker node.

Why Pods :-

- * Pod have capability to group supporting containers.

E.g One container is creating logs
 second container is reading & manipulating that logs:

About Pods Isolation :-

- It is necessary if we share volumes then both nodes must be on same worker node.

MULTI - TIER APPS :-

Multiple container in a single pod always runs on a single worker node.

7 24

27-JUN-2020
SATURDAY
9:58 PM

GROUPING PODS

All dependent container must be grouped to run in a particular worker node

* YAML FILE (CREATING A POD)

kind: Pod

apiVersion: v1

metadata:
name: myfirstpod

spec:

containers:

- name: container1
image: almirpinger/helloworld:latest.

ports:

- containerPort: 80

kubectl create -f myfirstpod.yaml

* POD LISTING & INSIGHT :-

kubectl get pods

(2)

POD LISTING & INSIGHT :-

- ① # kubectl get pods
- ② # kubectl get pods myfirstpod -o yaml
- ③ # kubectl get pods myfirstpod -o json
- ④ # kubectl ~~get~~ describe pods myfirstpod

* PORT FORWARDING :-

kubectl port-forward mysecondpod 6500:80

E.g localhost: 6500

| <u>CREATING POD</u> | <u>FROM COMMAND LINE</u> |
|--|--------------------------|
| # kubectl run mysecondpod --image=nginx/helloworld:latest --port=80 --restart=Never | --image= |

* Note: --restart=Never is not written then kubernetes will create other type of resource

LABELS :-

- Kubernetes organizes pod with the help of label
- key/value pair

```
# vi myfirstpodwithlabels.yaml
```

```
- kind: Pod
- apiVersion: v1
- metadata:
  name: myfirstpodwithlabels
```

```
  labels:
    type: backend
    env: production
```

```
- spec
  containers:
    - image: aamirpinger/helloworld:latest
      name: container1
    ports:
      - containerPort: 80
```

```
# kubectl create -f myfirstpodwithlabels.yaml
```

```
# kubectl run anotherpodwithlabel --image=
image aamirpinger/helloworld --port=80
--restart=Never --label=type=frontend,env=development
```

```
# kubectl get pods --show-labels
```

```
# kubectl get pods -L type cnv, type, run
```

LABELING POD AT RUNTIME :-

kubectl label pods myfirstpod app=helloworld
type=frontend

* To change label during runtime

kubectl label pods anotherpod withlabel
env=prod --overwrite

* To remove label

kubectl label pod myfirstpod app -

LABEL SELECTOR :-

kubectl get pods -l type=frontend

Not equal to

kubectl get pods -l type!=frontend

-show-labels

kubectl get pods -l type!=frontend,env=production
--show-labels

kubectl get pods -l 'env' --show-labels

kubectl get pods -l 'type in (frontend,backend)

```
# kubectl get pods -l 'typenotin  
(frontend,backend)' --show-labels
```

* Pods SCHEDULING WITH Node Selector

```
# nano podwithnodeselector.yaml
```

kind: Pod

apiVersion: v1

metadata:

name: podwithnodeselector

spec:

nodeSelector:

typeofhddisk: ssd

containers:

- image: amivpinger/helloworld:latest

ports:

- containerPort: 80

```
# kubectl create -f podwithnodeselector.yaml
```

- Assigning type of hddisk

```
# kubectl label node minikube typeofhddisk=ssd
```

ANNOTATION:

- Annotation are basically words that explanation or comment on something.

E.g creator name, contact

But it can not be used to group or filter like label

```
# nano podwithannotation.yaml
```

kind: Pod

apiVersion: v1

metadata:

name: podwithannotation

annotation:

createdBy: "Zohaib Ahmed"

email: zohaib4986@gmail.com

email:

spec:

containers:

- name: container-new

image: aamirpinger flag: latest

ports:

- containerPort: 80

```
podwithannotation.yaml
```

```
# kubectl create -f
```

```
# kubectl annotate pod myfirstpod app-describ  
= this is annotation example"
```

* DESCRIBING POD'S INSIGHTS:-

kubectl describe pod myfirstpod

* OVERLAPPING LABELS

It may be possible that different pods have same label which may create difficulty for your team while deployment or management.

* NAME SPACE :-

To solve overlapping issue name space is introduced

- it is virtual box which isolate self contain resources with other namespace

CREATING NAMESPACE :-

kubectl create namespace production

kubectl get ns

POD INSIDE NAMESPACE :-

kind: Pod
apiVersion: v1

metadata:

name: myfirstpod
namespace: production

Spec:

containers:-

- name: container

* image: almirpinger/helloworld:latest

10:12 PM

MONDAY

REPLICA SETS, JOBS AND CRONJOB :-

* REPLICA :-

- It is also a resource
- It helps managing multiple copies of application (replica) in kubernetes.

REPLICASET PARTS :-

It has three essential parts

- 1) Label Selector.
Determine what pods are in replicaset ^{Scope}
- 2) Replica Count
desire no. of pods must be running
- 3) Pod template
Replicaset uses to create new pod

* CREATING REPLICA SET :-

nano vs. yaml

kind: Pod

apiVersion: apps/v1beta2

metadata:

name: myrs

spec:
replicas: 3

selector:
matchLabels:
app: vsexample

template:

metaData:
labels:

app: vsexample

(1)

Spec:

Containers

- name: vs container
- image: amirpingers/helloworld: latest.

Ports

- containerPort: 80

kubectl create -f vs.yaml

"LISTING RS": -

kubectl get vs myrs -o yaml

kubectl describe vs myrs

kubectl label pod myrs-sq5zz
app=vexample --overwrite.

DELETING PODS AND REPLICASET:-

kubectl delete pods -l app=vexample

kubectl delete rs vs but individual pods will exist
To delete pod rs

kubectl delete vs myrs --cascade=false

• Note:- After deleting vs, if we recreate so it will function old deleted pods.

OPERATORS $\backslash\wedge$ MATCH EXPRESSIONS :-

There are 4 operators in match Expression

- In
- notIn
- Exist
- Does not Exist.

⦿ If match label & match expression

nano rs.yaml

```
kind: ReplicaSet
apiVersion: apps/v1beta2
metadata:
  name: myrs
spec:
  replicas: 3
```

```
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
```

- vs example

```
  template:
    metadata:
      labels:
```

app: vs example

spec:

containers:

- name: vs container
- image: aamirpinger/hello
- ports:
- containerPort: 80

kubectl delete vs
myrs --cascade=false

MODIFYING

REPLICA SET:-

- ReplicaSet can be modified at runtime
- Only changes to the replica count will affect existing pods.
- Label Selector and pod template in replicaSet will only affect new containers.

```
# kubectl edit vs myrs
```

SCALING

REPLICA SET:-

```
# kubectl scale vs myrs --replicas=5
```

JOB RESOURCE:-

- Job is another in Kubernetes.
- It is basically a Pod which we create under the type(kind) of Job.

JOB RESOURCE :-

kind: Job

apiVersion: batch/v1

metaData:

name: Jobexample

Spec

template:

Specs

containers:

- name: jobcontainer

image: docker/whalesay

commands: ["cowsay", "This is job resource example"]

restartPolicy: Never

backoffLimit: 4

activeDeadlineSeconds: 60

kubectl create -f job.yaml

kubectl logs jobexample-b2529

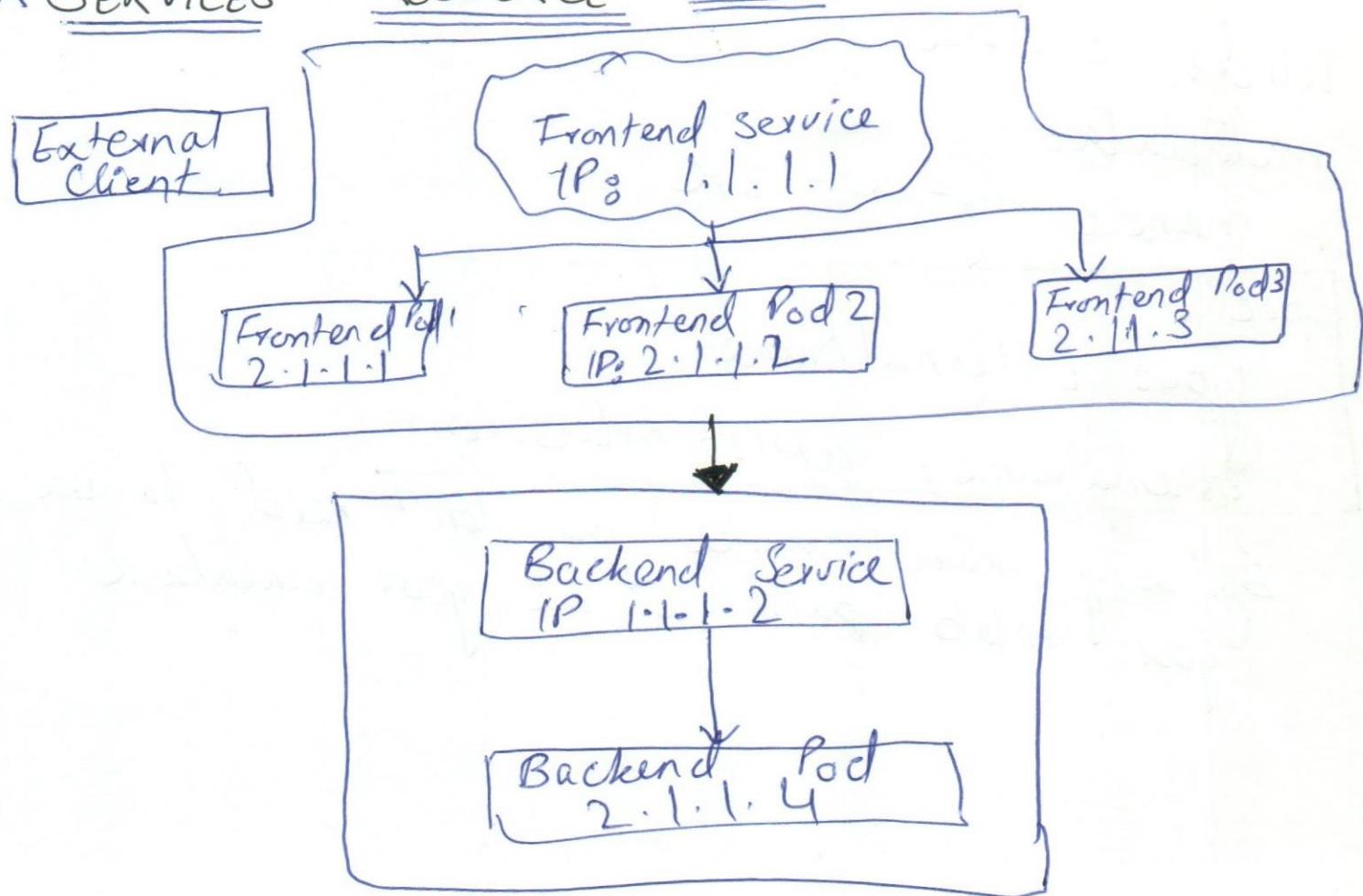
Kubernetes:

② B

"SERVICES & PROBES" :-

- Since k8s assign every pod to individual IP.
- It provides static IP to grouped of resources.
- It provides IP so that we can easily access it keeps the records of changing IPs if any changes occurs at worker node.

* SERVICES RESOURCE Flow :-



①

SERVICE RESOURCE TYPE :-

- Cluster IP
- Node Port
- Load Balancer
- External Name.

EXTERNAL NAME EXAMPLE :-

```
# apiVersion: v1
kind: Service
metadata:
  name: my-ext-svc
```

Spec :

```
  type: ExternalName
  externalName: ap12.mlab.com
```

By using external service you don't need to use your mlab path to use your database

LOADBALANCER

SERVICE

EXAMPLE (YAML)

```
# nano mysvc.yaml
```

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - port: 8080
      targetPort: 80
  type: LoadBalancer
  selector:
    app: vsexample.
```

```
# kubectl create -f mysvc.yaml
```

```
# kubectl get svc
  name   type        ClusterIP ExternalIP  Port(s)  Age
  my-service  -  8080:32468/TCP  -
```

```
# kubectl expose rs myrs --name=my-svc-lb
  --selector=app=vsexample --port=8000
  --target-port=80 --type=LoadBalancer.
```

LIVENESS

PROBE :- (HEALTH CHECK)

① HTTP GET :

If checks container IP & port and if it is not get any response then it will restart that container.

② TCP SOCKET :

If tries specified port and if it is not get any response then it will restart.

③ EXEC PROBE :

If status code is 0 then container is good otherwise upon receiving other codes it will restart the container.

```
my-in-http.yaml
kind: Pod
apiVersion: v1
metadata:
  name: myapp-in-http
spec:
```

containers:

- name: myapp
 images: aamirpingx/hi
 ports:
 - containerPort: 80

liveness Probes

```
  httpGet:
    port: 80
    path: /
```

```
my-in-tcp.yaml
kind: Pod
apiVersion: v1
metadata:
  name: myapp-in-tcp
spec:
```

containers:

- name: myapp
 image: aamirpingx/hi
 ports:
 - containerPort: 80

livenessProbes

```
  tcpSocket:
    port: 8080
```

```
my-in-exec.yaml
kind: Pod
apiVersion: v1
metadata:
  name: my-in-exec
spec:
```

containers:

- name: aamirpingx/hi
 ports:
 - containerPort: 80

livenessProbes

execs

commands

- cat: /tmp/healthy

LIVENESS

PROBE

ADDITIONAL

PROPERTIES:

- failure Threshold: 3
- period Seconds: 10
- success Threshold: 1
- timeout Seconds: 1
- Initial Delay Seconds: 15

READINESS

PROBE

The main difference b/w liveness probe & readiness probe is readiness probe check the pod whether it is ready to entertain traffic or not.
- It marks container so that

READINESS

PROBE

TYPES

- 1 Http GET
- 2 TCP Socket
- 3 Exec probe.

READINESS

PROBE

EXAMPLE

apiVersion: v1

kind: Pod

metadata:

name: vpod

spec:

containers

- name: container-vp
image: aamirpinge/xhelloworld:latest

ports:

- containerPort: 80

readinessProbe:

HttpGet

port: 80

path: /

kubectl create -f ipod.yaml

HttpGet:

port: 90

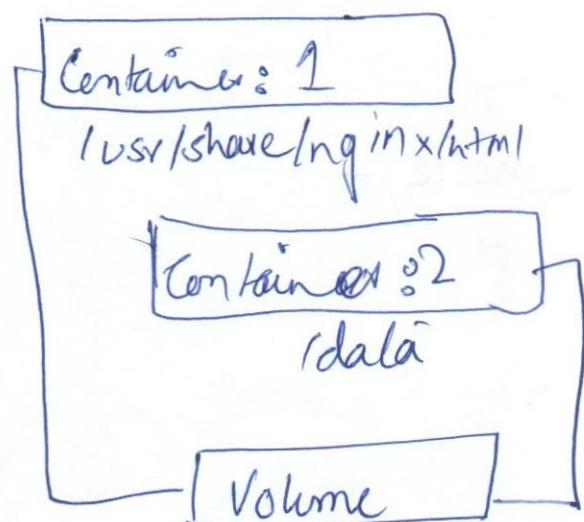
path: /

VOLUME

- It provides a shared folder type storage.
- It is basically a shared folder.

VOLUME FLOW :-

Pod 1



VOLUME TYPES

- emptyDir
- configMap, secret, downward API
- persistentVolumeClaim
- gitRepo
- gcePersistentDisk
- AWS Elastic Block Store
- azure disk

```
# nano vol.yaml
```

kind: Pod

apiVersion: v1

metadata:

name: my-pod-with-vol

Spec:

volumes

- name: shared-dir
- emptyDir: {}

containers:

- name: container-one

image: aamirpinger/logfile-nodejs

ports:

- containerPort: 2020

volumeMounts:

- name: shared-dir
- mountpoint: /data

- name: container-two

image: nginx

ports:

- containerPort: 80

volumeMounts:

- name: shared-dir

mountpath: /var/c-two

```
# kubectl create -f vol.yaml
```

PERSISTENT VOLUME :-

- In case of pod termination data in the volume will be lost.
- To solve the issue kubernetes provide us option of Persistent Volume.
- P.V add a volume at a cluster level instead pod level.

PERSISTENT VOLUME CLAIM :-

- It is a kind of request for claiming a persistent volume by pod.
- kubernetes bound persistent volume request by match persistent volume claim configuration

PERSISTENT VOLUME SPECS :-

- Access Mode
 - Read/Write Once. (RWO)
 - Read Only many. (ROX)
 - Read Write Many. (RWX)
- RECLAIM POLICY
 - Delete
 - Recycle
 - Retain (Default)

- DELETE

PVC is deleted but data will persist

- Recycle

- data / volume's contents will be deleted.
- Persistent volume will be available to be claim again.

- Retain (default)

- k8s will retain the vol. and its content after it released from its claim.
- We can manually delete and recreate the persistent volume resource.

```
# nano pv.yaml
```

```
kind: PersistentVolume
```

```
apiVersion: v1
```

```
metadata:
```

```
  name: PV
```

```
spec:
```

```
  accessModes:
```

- ReadWriteOnce.

```
  capacity:
```

```
    storage: 100M
```

```
  hostPath:
```

```
    path: /tmp/pvexample
```

```
PersistentVolume Reclaim Policy: Delete.
```

```
# kubectl create -f pv.yaml
```

```
# nano pvc.yaml
```

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: PVC

spec:

accessModes:

- ReadWriteOnce.

resources:

requests:

storage: 100M

Storage Class Name:

```
# kubectl create -f pvc.yaml
```

```
# kubectl get PV,PVC
```

```
# nano pod.yaml
```

kind: Pod

apiVersion: v1

metadata:

name: pod-PV

spec:

volumes

- name: PV-vol

persistentVolumeClaims:

claimName: PVC

containers:

- name: container1

image: almi/pingex logfile-nodejs

volumeMounts:

- name: PV-vol

mountPath: /data

```
# kubectl exec -f pod.yaml
```

PERSISTENT VOLUME EXAMPLE PART 4 :-

kubectl exec pod-pv -it sh.

CONFIGURATION APPLICATION AND MOBILE

DEPLOYMENTS :-

- ConfigMaps allow you to separate your application configuration from your application code, which helps keep your containerized applications portable.
- ConfigMaps are useful for storing and sharing non-sensitive, unencrypted configuration information which you can change at runtime.
- It helps makes their configuration easier to change and manage and prevent hardcoding configuration data to pod specification.

CONFIGMAP FROM LITERALS

```
# kubectl create configmap myfirstcm
--from-literals=name=Aamir --from-literals
=name=Pingex.
```

```
# kubectl get cm
```

CONFIG MAP FROM FILE :-

```
# nano user.txt
```

fname = Aamir

Lname = Pinger

```
# kubectl create cm myfirstfromfile  
--from-file=user.txt.
```

If we want the key to be used personalized

```
# kubectl create cm cmfromfile --  
from-file= user.txt
```

Describe:

```
# kubectl get cmfromfile -o yaml
```

```
# kubectl describe cm cmfromfile.
```

CONFIG MAP FROM ENVFILE :-

```
# nano em.env
```

CREATED BY = Zohaib

```
# kubectl create em cmfromfile  
--from-envfile=em.env
```

CONFIGMAP As VOLUME :-

nano pod with cm.yaml

kind: Pod

apiVersion: v1

metadata:

name: podwithcmvol

spec:

volumes:

- name: cmvol

configMap: myfirstcm

containers:

- name: container1

image: nginx

volumeMounts:

- name: cmvol

mountpath: /data/cm

imagePullPolicy: If Not Present

kubectl create -f podwithcm.yaml

kubectl exec podwithcmvol -c
container1 -it sh.

↳ cd /data/cm

↳ cat lname

CONFIGMAP As Env VARIABLE

```
# nano podcmenv.yaml  
kind: Pod  
apiVersion:  
metadata:  
  name: podcmenv  
  labels:  
    app: envcm example  
spec:  
  containers:  
    - name: container1  
      image: aamirpinge/node-app-image  
      imagePullPolicy: IfNotPresent  
  envFrom:  
    - configMapRef:  
        name: cm from envfile  
# kubectl create -f podcmenv.yaml  
# kubectl exec podcmenv -it sh  
For SERVICE RESOURCE  
# kubectl expose pod podcmenv  
  --port=8080 --target-port=8080 --type=loadbalancer
```

SECRET : FROM LITERALS :-

```
# kubectl create secret generic myfirstsecret  
--from-literal=username=aamirpingor --from-  
literal=api=list-db.aws.com
```

```
# kubectl describe secret myfirstsecret.
```

```
# kubectl get secret myfirstsecret -o yaml
```

```
# echo _____ | base64 -d
```

SECRET FROM FILE :-

```
# nano keyval.txt
```

```
fname: aamir
```

```
lname: pinger
```

```
# kubectl create secret generic secret  
fromfile --from-file=keyval bio=keyval.txt
```

```
# kubectl get secret
```

```
# kubectl describe secret secret fromfile
```

SECRET FROM ENV FILE

```
# kubectl create secret generic secretfrom  
envfile -from-env-file=keyval.txt
```

```
# kubectl get secret
```

```
# kubectl describe secret secretfromenvfile
```

SECRET AS VOLUME :-

```
# nano podsecret.yaml
```

```
kind: Pod
```

```
apiVersion: v1
```

```
metadata:
```

```
  name: podwithsecret
```

```
spec:
```

```
  volumes:
```

```
    - name: secvol
```

```
  secrets:
```

```
    secretName: myfirstsecret
```

```
  containers:
```

```
    - name: container1
```

```
      image: nginx
```

```
      imagePullPolicy: IfNotPresent
```

```
      volumeMounts:
```

```
        - name: secvol
```

```
          mountPath: /secretdata
```

```
# kubectl create -f podsecret.yaml
```

```
# kubectl get po
```

```
# kubectl exec podwithsecvol -it sh
```

```
# cd secretdata  
api username
```

```
# cat api  
test.db.awst
```

SECRET As Env VARIABLE :-

```
# nano secenv.yaml
```

kind : Pod

apiVersion : v1

metadata :

name : secenv

spec :

containers

- name : container1

image : nginx

imagePullPolicy : If Not Present

envFrom :

- secretRef :

name : myfirstsecret

```
# kubectl exec secenv -it sh
```

```
# env
```

ENVIRONMENT VARIABLE :-

```
# nano podwithenv.yaml
```

kind: Pod

apiVersion: v1

metadata:

name: podwithenv

SPEC :

containers:

- name: container1

image: nginx

imagePullPolicy: IfNotPresent

envs

- name: USERNAME

value: aamirpinge

- name: COURSENAME

value: kubernetes.

To Create Pod: # kubectl create -f podwithenv.yaml

```
# kubectl exec podwithenv env
```

DEPLOYMENT :-

- How to group containerized app into pods.
- Ways to provide them with temporary & permanent storage.
- How to pass both secret & non secret config data to them.
- How to run a full fledged system composed of independently running smaller components (microservices)

DEPLOYMENT USE CASE

- If you created your pod with a container image `aamirpinger/hello-world: v1`
- If you upgrade your image with `aamirpinger/hello-world: v2`
- You can't change existing pods image after the pod is created, you need to remove the old pods and replace them with new ones running the new image

DEPLOYMENT STRATEGIES PROS

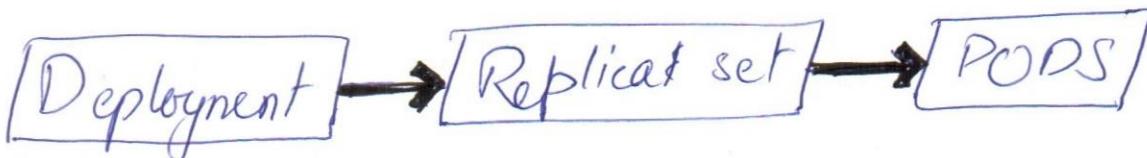
PROS

- You can delete pods manually and replica set will recreate it.

- Change service resource . but it will consume double resource meanwhile if you have deleted older pods/replica set.
(Rolling update)

WHAT STRATEGY To CHOOSE :-

- To resolve rolling update kubernetes gives us solution by providing deployment resource



PART 1

```

# nano deploy.yaml
kind: Deployment
apiVersion: apps/v1 beta
metadata:
  name: deployexample
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: frontend
  spec:
    containers:
  
```

- name: container
image: aamirpingeek/helloworld
imagePullPolicy: IfNotPresent

kubectl get deploy

kubectl get rs

kubectl expose
deploy example --port=80
--target-port=80
--type=loadbalancer

kubectl get svc

PART 2 :-

kubectl set image deploy deployexample
container1=aamirpingal/flag.

kubectl get deploy, rs, pod

PART 3 :-

- Default strategy of kubernetes is rolling update
If we define strategy in spec

nano deploy.yaml

same as above

spec:

replica: 4

—

—

strategy: Rolling Update } or
 type: Rolling Update
 type: Recreate
 maxUnavailable: 0
 maxSurge: 1

HELPER COMMAND

kubectl rollout pause deploy deployexample

" " resume " "

```
# kubectl rollout status deploy deployexample  
# kubectl rollout history deploy deployexample  
# Same as above --revision=1 or --revision=2
```

Revert back:

```
# kubectl rollout undo deploy deployexample  
--to-revision=1
```

To check pod

```
# kubectl get po
```

```
# kubectl describe po deployexample-no-no
```

BEST PRACTICES

- Use different tag policy like V1, V2, V3 rather use latest
- Image policy: (image pull policy) should be used wisely. (Always/Never)
- Use label of multi-dimension.
- Annotation as we can give bigger description.
- Application logs.