Kubernates

*) It is a orchestration engine & opensource platform for managing Containairization application.

*) Responsibilities include Container deployment, Scaling & descaling of Containers & Container load balancing

*) It is a replacement of docker-Swarm.

*) It is Born in Google, written in Go|Golang. Donated to CNCF [cloud native Computing foundation] in 2014.

features of kubernates:-

1) Automated ~~sca~~ Scheduling:- K8's Provides advanced scheduler to launch Container on cluster nodes based on their resource requirements & other Constraints.

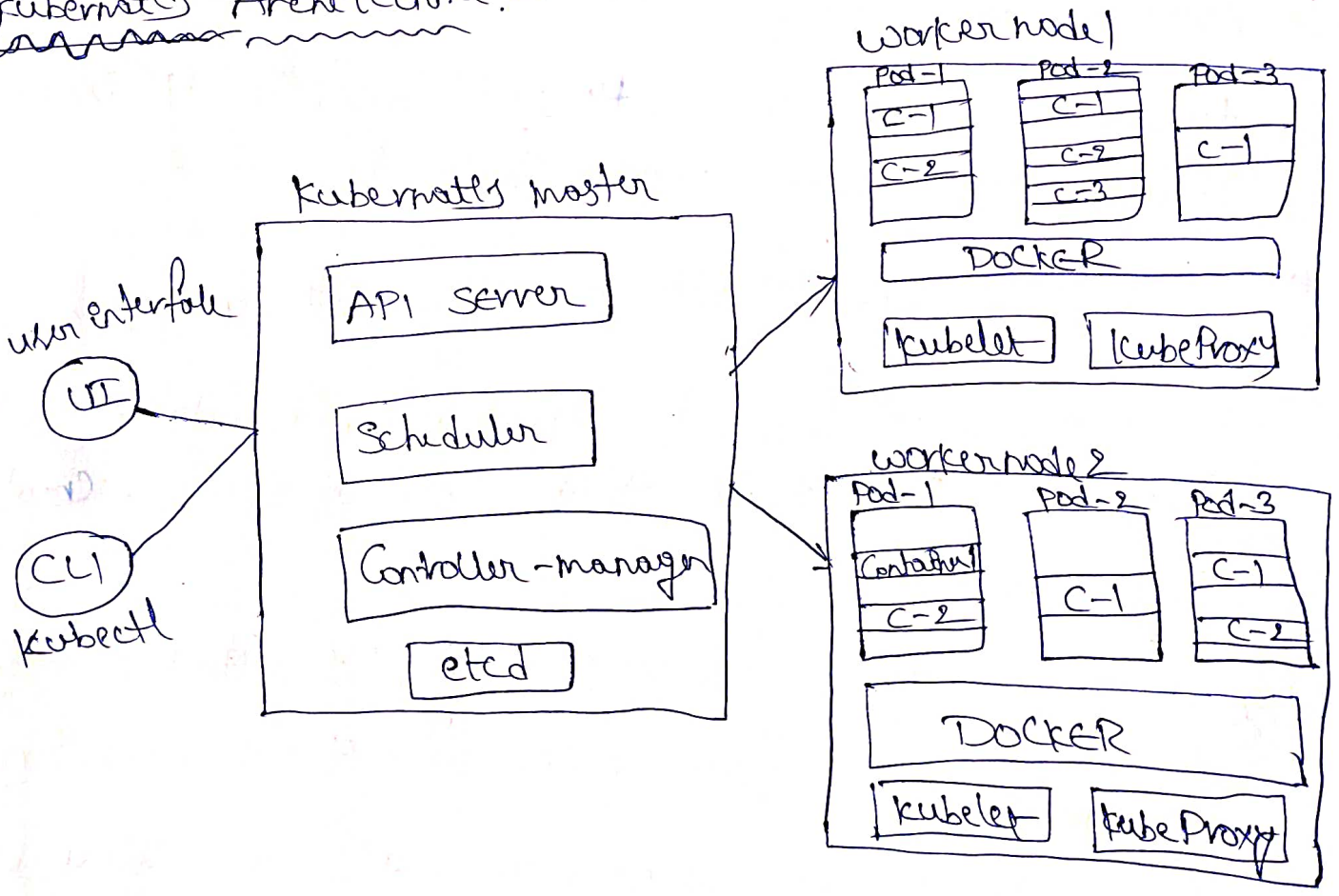
2) Self healing capabilities:- K8's allows to replaces & reschedules Containers when nodes dies, if Container don't respond kubernates kill that Containers Again recreate the Containers.

3) Automated rollout & rollback:- kubernates rollout changes the application & Configuration with the new version. if Something goes wrong, kubernates Can rollback to the Previous version.

4) Horizontal Scaling & load balancing:- K8's can scale up & scale down the applications as per requirements with a simple command, using UI, or automatically based on CPU usage.

5) Service discovery & load balancer:- ~~K8's~~ Pods having application (Containers) & these Pods assign to nodes. kubernetes assign the IP to all Pods. if single or multiple application running in Pods, kubernetes distribute the traffic with the help of service using single DNS name.

Kubernetes Architecture:-



when we are deploying any object (Pods, deployment, services, PC, PV) through the CLI then the request goes to initially API-SERVER. API-SERVER is a front end component of master-node. API-SERVER take that request & Persist the data in Etcd Component & Process the request.

Etcd :- It is a key value storage which is maintain & Store all the kubernates cluster information Pods & nodes.

Scheduler :- scheduler first talk to the Etcd & scheduler will schedule or schedule Pods in nodes with the help of kublet based on resource requirements.

kubelet :- kubelet will talk to the docker Container runtime & it makes sure that Containers managing & running.

Control manager :-

↳ ~~Pod~~ Replication Controll manager
↳ node " "
↳ Deployment " "
↳ Replicaset " "
↳ Deamon Set " "

suppose in Replication Controll manager, ⊗ of some Pods allocated to nodes. if something goes down one Pod on node then Replication Controller make sure that enough number

Pods upend running. Similarly node, deployment controllers also same.

Kube-Proxy:- It maintain the network rules on nodes. these network rules allow network communication to your Pods from inside or outside of cluster. It redirects the request from clusterIP to Pod IP.

Kubernetes Setup ways:-

1) Self managed k8 cluster
   - ↳ kubeadm → multinode k8's cluster (play with k8's)
   - ↳ minikube → single node k8's cluster (katacoda)
   - ↳ you can setup k8's in local m/c also through kubeadm.

2) managed k8's Services:
   - ↳ EKS → Elastic k8's Service (AWS)
   - ↳ AKS → Azure " " (Azure)
   - ↳ GKE → Google " Engine (GCP)
   - ↳ IKE → IBM " " (IBM)

In Aws cloud there are different ways to setup k8's:-

i) through EKSctl.
ii) through kops
iii) through kubeadm
iv) through Aws Console

[kubernetes | ravdy | kubernetes]

NOTE:- 1) If nodes (master & worker) not ready then check in
↳ kubectl get all -n kube-system [shows Daemon, Deploy & replicaset Pods]
Command ↳ kubectl get Pods -o wide -n kube-system.

if anything is not ready Configure the related to Pods. (DNS Pods, Network Pods, masternode Components Pods, worker node Component Pods). if network Pods not ready you can execute network interface Command plugin (wavenet, NSX-T Container-Plugin, kube-router). then check nodes so here all nodes become ready state.

(2) If you want access API SERVER (master node) from another EC2 instance. only here you need Configure kubectl install in EC2 instance & create one file vi ~/.kube/Config file. here Paste the master node Config file in EC2 m/c ~/.kube/Config file. then check it using Command.

↳ kubectl get nodes [∵ from EC2 m/c]
(shows here)

(3) kubectl will interact with the k8s cluster with the help of kubeConfig file.

Namespace:- It is used for (isolation) do many users Projects into seperate environments.

↳ kubectl get namespace(ns)

↳ kubectl create ns Dev for creating 'Dev' namespace.

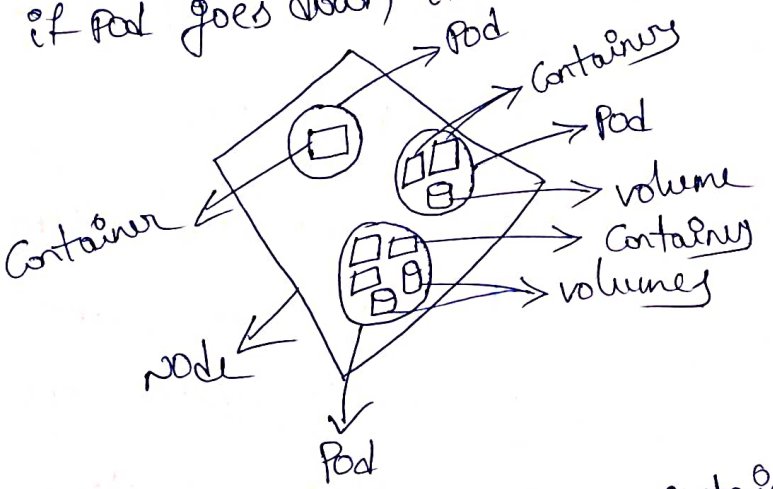
POD:-

*) A Pod always run on Node

*) A Pod is smallest building block of scheduling in Kubernetes.

*) Inside the Pod you have one or more Containers. those Containers all share unique network IP, storage etc.

*) if Pod goes down it won't create again that is defect.



there are two ways to run Containers in Pod,

1) Interactive way → It is using Commands

2) Declarative way → It's using yml file.

↓

```
apiVersion: V1
kind: Pod
metadata:
  name: <Pod name>
  labels:
    <key>: <value>
spec:
  Containers:
    - name: <Containername>
      Image: <Imagename>
      Ports:
        - ContainerPort: <ContainerPort>
```

↳ kubectl get events → shows more information & Pod scheduled to which node.

↳ kubectl get Pods -o wide → shows more information about Pod

↳ curl Pod.IP:8080 → for accessing the application within the cluster (master & worker nodes)

Service:- Service makes Pod accessable/discoverable within the cluster & outside the cluster. when we create a Service you will get one vertual IP (clusterIP) address. this IP will registered in kubernates dns with it's name (service). So other application communicate using service name.

the Group of containers ~~with~~ Pods accessable only through the service. there are diff. types of Services.

i) clusterIP → used for Expose application within cluster

ii) NodePort →
iii) LoadBalancer → } Expose application outside of world.

iv) Headless Service

Here Service will identifies the Pods based on labels & Selectors.

↳ kubectl get Pods --show-labels → for showing Pods labels

↳ kubectl describe Pod <Podname> → for more information about Pod.

Service.yml file:-

NodePort → 30000-32767 range

```
apiVersion: v1
kind: Service
metadata:
  name: <servicename>
spec:
  type: ClusterIP, NodePort, LoadBalancer
  ports:
  - port: 80 → service port
    targetPort: 8080 → Container Port in Pod.yml, Deploy.yml, replication Controller, rs.yml.
  selector:
    app: <Give here> → match label do Deploy.yml file.
```

ClusterIP (only within cluster):-

↳ kubectl apply -f service.yml

↓

kubectl get svc [all services shows here.]

↓

curl serviceIP → virtual IP → for within cluster checking.

clusterIP, NodePort, loadbalancer

↓

kubectl describe svc <svc-name> → for knowing more information about Service

↓

kubectl get endpoints → for which Pods assign to this Service.

↳ You can access one Pod of Container to another Pod of Container using Service-name.

↳ curl <Service-name> [: when you are another Container]

↳ for checking Ports listening in nodes you need to install,

→ sudo apt install net-tools

↓

Sudo netstat ~~-tun~~ -tulnp → for showing listening Ports

NodePort (outside access) :-

↓

kubectl apply -f service.yml

↓

kubectl get svc -

shows services

[default -
nodePort - 80|30005
(ServicePort | NodePort)]

↓

kubectl get Pods -o wide

[shows here Pods assigned to which node (Pt IP shows for node)]

↓

curl ServiceIP:ServicePort → for within cluster checking
(ServiceIP | 80)
(NodePortIP)

for outside of world access, you must open Port (nodeport) in Security groups for worker nodes & master nodes.

PublicIP : <nodePort>

30005

[worker nodes & master nodes]

Static Pods :- Static Pods managed by directly kubelet & API-SERVER doesn't have any control over these Pods. The kubelet is responsible to watch each static Pod & restart if it crashes.

↳ kubectl get all Pods -n kube-system

↳ kubectl get ds -n kube-system → for showing deamon Set Pods in kube-system. if something 'is' Pod goes down the deamon Set responsible for Creating new Pod. Similarly deploy & rc is same.

↳ kubectl get deployment -n kube-system

↳ kubectl get rc -n kube-system

only these Pods,
Pod | kube-apiserver
Pod | kube-Controller-manager
} responsible for kubelet because these Pods will maintain static IP.

↳ sudo ls /etc/kubernetes/manifest for showing Components manifest files.

Replication Controller:- RC which is responsible for managing the Pod life cycle. it is responsible for making sure that the specified number of Pod replicas are running at any point of time. if Pod does crash, the Replication Controller replaces it & Again Creates.

↳ kubectl get Pods <Pod-name> -o yaml → for Pods showing yaml file format.

↳ Create Replication Controller.yml file & service.yml (nodeport) file in one file,

↓

kubectl apply -f <yml-file>

↓

kubectl get Pods [shows here]

↓

kubectl get rc [replication Controller shows]

↓

kubectl get svc [default & nodeport shows here]

↓

kubectl scale rc <rc-name> --replicas 5 [for scaling rc]

↓

kubectl get all [shows all information]

↓

kubectl get Pods --show-labels

In Replication Controller file ~~rc.yml~~ ~~pod~~ selector option is a not mandtory. (match labels). not contains. under, Selector: app: <name>

Replica Set:- it's next generation of replication controller.

Replica set will also manages Pod life cycle. we can scale up scale down Pods. only difference in it's selector support. if Pod goes down replicaset will replacing that Pod & create new Pod. but here there is small downtime of application

under selector,
matchLabels:
app: <name>

↳ kubectl get rs <rs-name> -o yaml for showing rs in yaml file format

there are two types of selectors.

i) Equality based selector
```
selector:
  app: <name>
```

ii) set based selector
```
selector:
  matchExpressions:
  - key: app
    operator: in
    values
      - Javawebapp
```

↳ create Replication set.yml & Service.yml (NodePort) file in one yml file,

↓

kubectl apply -f <one.yml>

↓

kubectl get pods

↓

kubectl get rs → for showing Replicasets

↓

kubectl get svc -o wide

↓

curl serviceIP:servicePort → for checking internally

↓ ↓

nodePortIP 80

↓

for outside Paste the Public IP : Port (nodeport) (this port open on sec. group)

↓ master node & worker node.

↓

sudo netstat -tulnp → for Ports assigned & not.

↓

kubectl scale rs <rs-name> --replicas 3 → for scale the RS.

↓

kubectl exec <Pod-name> <Pod Container name> -- ls → for shows all file for that Particular Container.

If two containers runs in a single Pod then for going to Particular Container, under manifest file,

```
Container:
  name: JavawebappContainer
  - Port: 8080
Container:
  name: mavenwebappContainer
  - Port: 8080
```

↳ kubectl exec <Pod-name> -c JavawebappContainer

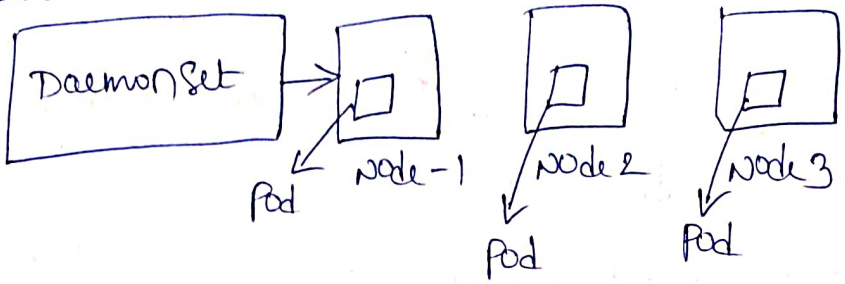
↓

kubectl logs <Pod-name> <Pod Container name> → for checking logs.

↓

kubectl exec -it <Pod container name> /bin/bash → for going to Particular Pod.

DaemonSet:-



In this DaemonSet it will ensure that one copy of Pod defined in our Configuration will always available on every worker node.

if Daemon file.yml execute then whatever worker nodes you have in all worker nodes Pod will Consisting.

↓

kubectl apply -f Demon.yml

↓

kubectl get Pods

[shows Pods support if three worker nodes then 3 Pods running here.]

if you want to do log monitoring in all the worker nodes then DaemonSet better way to monitor our application.

↓

kubectl delete all --all → for deleting all Pods.

NOTE:- if you are deployed multiple application in nodes then Some times node becomes went down State (not ready). So you Connect to that Server & check the logs (kernal logs, kublet logs). if it not Connecting then instance you able to Stop & Again Start.

what is diff b/w RC, RS & Deployments:-

RC & RS:- when we are getting new version of image by changing of code, here delete the Pod.yml file ~~old~~ with old version of file & create .yml file with new version and Again apply. we can't rollback to the Previous of image if something went wrong with new version.

Deployment:- Deployment is a recommended way to deploy the application. if new version image coming ~~s~~then here simply you can modify the file and Again apply. So here automatically changes apply with the new version of image. we can rollback to Previous image also if something went wrong with new version.

Deployment Strategies:-

i) Recreate strategy → under spec:
replicas: 2
strategy:
type: Recreate

ii) rolling update strategy (default deployment)

under spec:
replicas: 2
strategy:
type: RollingUpdate
rollingUpdate:
maxUnavailable: 1
maxSurge: 1

i) Recreate Strategy:-

↳ Create deploy.yml & service.yml file in one file.yml,

↓

kubectl apply -f <deploy.yml>

↓

kubectl get Pods -o wide for shows information

↓

kubectl get all → for shows all Pods, deploys & Services

↓

kubectl rollout history deployment <deploy-name> → for checking revision & CHANGE-CAUSE of application [1 revision shows here]

↓

kubectl rollout ~~history~~ status deployment <deploy-name> → for checking status of deployment object.

↓

kubectl rollout history deployment <deploy-name> --revision 1 → for checking particular revision of Pod template.

↓

for outside access, paste the PublicIP: Port (Port ↓ nodeport) . then

↓

master & worker nodes PublicIPs

able to see application page.

↓

If you want to update the deployment suppose you can change the version of image & Again apply the deployment.yml file. so here showing new version of image in Pods because Pods are created with new version.

↓ kubectl apply -f <deploy.yml> --record=true → for apply & save in CHANGE CHAUSE

↓ kubectl get Pods

↓

kubectl describe Pod <Pod-name> → for showing new version coming or not.

↓

kubectl rollout history deployment <deploy-name> → for showing revision & CHANGE CAUSE (2 writing shows)

↓

here there is a little bit down time of application.

↓

if your application is not working with new version then you need to rollback with previous version of image

kubectl rollout undo deployment <deploy-name> --to-revision 1

→ for going to revision 1 image working.

↓

kubectl get Pods -o wide → for shows Pods with old version.

ii) rolling update strategy (default strategy) :-

under spec :-

```
replicas: 2
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
  minReadySeconds: 60
```

maxUnavailable: 1 → if one Pod not available then another Pod we have for serve the application traffic (out of 2 replicas)

maxSurge: 1 ← one by one Pods Creates as given of 2 replicas

↓

if something update version happens, new image Pod will creates but ~~it~~ this is not immediatly serve the traffic it will take some time. so inorder to that the old Pods not deletes immediatly it will take some time i.e 60 seconds ~~of~~ after ~~reading~~ ready of new container this old container dies.

↓

kubectl apply -f <deploy.yml> --record=true

↓

kubectl get Pods -o wide → for showing all Pods

↓

kubectl rollout history deployment <deploy-name> → for checking revision & CHANGE CAUSE

↓

for outside access paste PublicIP:NodePort then able to see application page.

↓

If you want to update the version image of Pod & Again apply then Pods Created with new image

↓

kubectl get Pods -o wide

↓

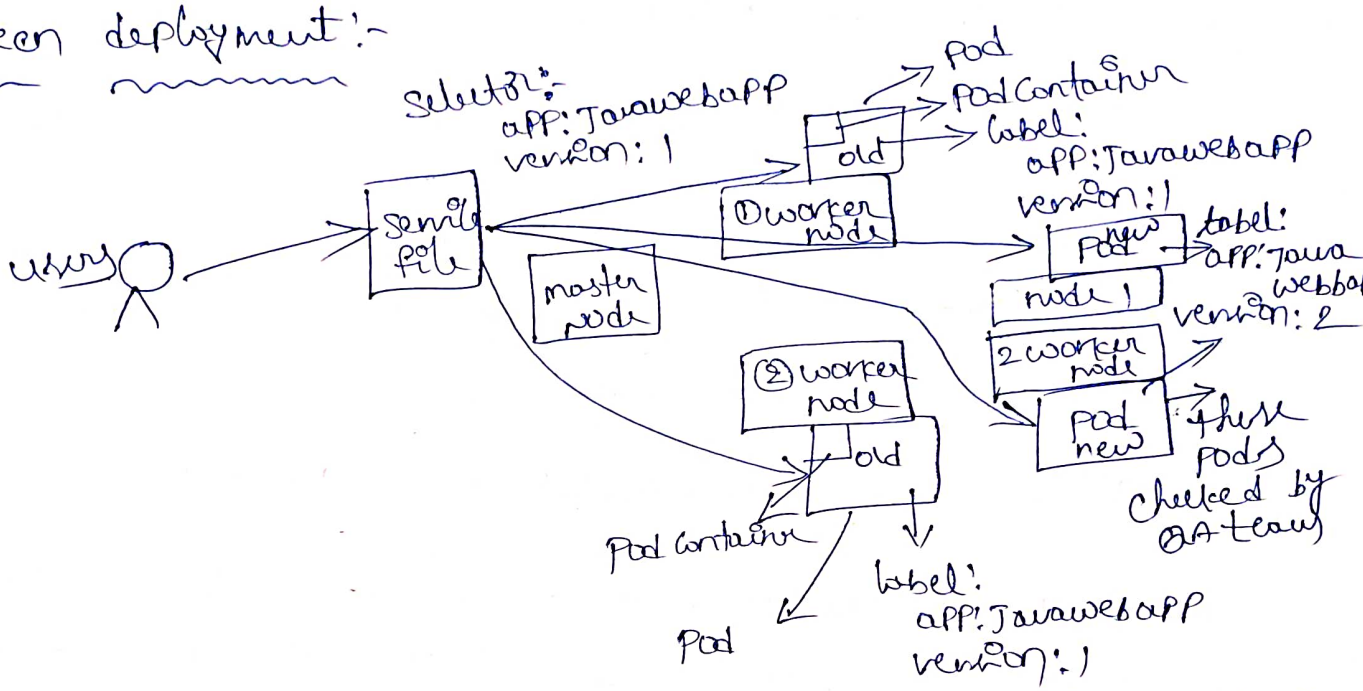
* in other tab
↳ watch kubectl get Pods
shows rolling update happening

here old Pods & new Pods shows because when new Pod creates & ready after 60 seconds the old Pod get deletes because here we done "rolling update strategy". so here not having a any downtime of application.

↓

kubectl scale deploy <deploy-name> --replicas 5 → for scaling up & down also

Bluegreen deployment:-



Initially for creating Bluegreen deployment first setup Pods on worker nodes [in the form of yml file] After checking of QA team. then edit the Service file & Correct the labels & selector for taking traffic request from users to new Pods & you can delete the old Pods or scaledown to zero.

In this Blue green deployment we need more Configuration (CPU, memory) for the nodes.

application uses this deployment in some times:-

i) flipkart
ii) amazon
iii) JCPenny
iv) tesco

Autoscaling :- there are two types

i) Vertical Pod Autoscaling (VPA)

ii) Horizontal " (HPA)

i) vertical Pod Autoscaling means we can increase the size of Pod resources like CPU, memory .. etc.

ii) HPA :- The ~~Aut~~Horizontal Pod Auto Scaler Automatically scales the number of Pods in replication Controller, rs & deployments based on observed CPU utilization & memory utilization.

HPA will interact with Metric Server to identify CPU/ memory utilization of POD.

→ Kubernetes metric Server :- It is an application that collects metrics from objects such as Pods, nodes according to the state of CPU, RAM & keeps them in time.

Install metric-Server :-

1) Install metric-server through Component.yml file is google documentation.

In Component.yml file we have,

i) Service account for metric-server

ii) Role for access of Pods & nodes .. etc

iii) Role binding [from Service account to Role]

iv) Deployment for metric-server

↳ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.4.1/components.yml

Check metric-server Pod in ~~kk~~ kube-system namespace,

↳ kubectl get pods -n kube-system → for showing Pods related to deploy, RS, metric-server & static Pods

For showing CPU & memory utilization in Pod & nodes,

↳ kubectl top pods

↳ kubectl top nodes

→ HPA.yml file :- [Create HPA yml file] :-

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: hpadeploymentautoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment/ReplicaSet → Give kind here.
    name: Javawebappdeployment → Give deployment name in deploy.yml file.
  minReplicas: 2 → Always minimum '2' replicas
  maxReplicas: 5 → Pods coming by Autoscaler but not more than 5.
  metrics:
    - resource:
      name: CPU → which resource you want
      targetAverageUtilization: 50
      type: Resource
```

All of Replicas in deployment

Pod reaches CPU utilization more than 50 then Pods are coming [Autoscaler].

1 → Create Deployment file under this file add resources here.

under SPEC:-

```
Containers:
  - name: Javawebbappcontainer
    image: Javawebappimage
    Ports:
      - name: http
        containerPort: 80
    resources:
      requests:
        cpu: "100m"
        memory: "64Mi"
      limits:
        cpu: "100m"
        memory: "256Mi"
```

requested utilization for Pod ← requests

max utilization we have ← limits

megabytes

NOTE:- 1000 millicore = 1 core

100millicore means 1/10 of core.

↳ Create Deploy.yml, Service.yml & HPA.yml files in one.yml file & apply it.

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods

↓

kubectl get hpa → for hpa details like max, min Replicas & running replicas

↓

kubectl top Pods → shows CPU & memory information how much utilized.

if you getting huge traffic for the application, hpa will takeup the Replicas & creates apods.

↓

watch kubectl get Pods → for shows Pods information when we get huge traffic.

NOTE:- In case of managed K8's we will have a cluster Autoscaler. if something goes down with node ~~we~~ will recreates by a cluster autoscaler.

Volumes:-

There are two types application. Those are

i) Stateless - [front end spring boot]

ii) Statefull [Mongo Bakend db]

↳ In Deployment file you refer enviornment variable of mongodb. & create Servicefile for deployment.

Under Spec:-

Container details { spring-~~mong~~ boot Container

resources {

env:
- name: MONGO_DB_HOSTNAME
  value: mongo
- name: MONGO_DB_USERNAME
  value: devdb
- name: MONGO_DB_PASSWORD
  value: devdb@123

↳ create mongodb (replicaset & deployment) & Service file for mongodb.

apiversion: apps/v1
kind: ReplicaSet / statefulset
metadata:
  name: mongodbrs
spec:
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      name: mongodbPod
      labels:
        app: mongo
        ~~ports: mongo~~
        ~~image: mongo~~
    spec:
      containers:
      - name: monogodb container
        image: mongo
        ports:
          - containerPort: 27017
        env:
          - name: MONGO_INITDB_ROOT_USERNAME
            value: devdb
          - name: MONGO_INITDB_ROOT_PASSWORD
            value: devdb@123
        volumemounts:
          - name: mongodbhostPath → same name
            mountPath: /data/db
      volumes:
        - name: mongodbhostPath
          hostPath:
            Path: ~~/tmp~~ /tmp/mongodbdata

If pod delete but data not deletes in this path.

Data will have in host path of workpode.

(or)

nfs:
  Server: Pvt IP → NFS server instance Pvt IP
  Path: /mnt/share → mount path in NFS server.

Service file for mongo:-

apiversion: v1
kind: Service
metadata:
  name: mongo → Give name whatever you given in deployment file under DB_HOSTNAME.
spec:
  type: ClusterIP for internal
  selector:
    app: mongo
  ports:
    - Port: 27017.

There are different types of volumes:-

1) hostPath:- It will use host (node) file system. We can mount container directory with host (node) file system. In this volumes mounts, if you are delete the Pod then data will safe in host path of worker nodes.

2) emptyDir:- it's temporary storage. if container goes down we will have a data but if Pod goes down we will not have a data that's why we called as temporary storage.

3) nfs:- (network file system):-

4) Elastic Block storage for Aws cloud

1) hostPath:- [with mongodb.yml file hostPath volume]:-

↓

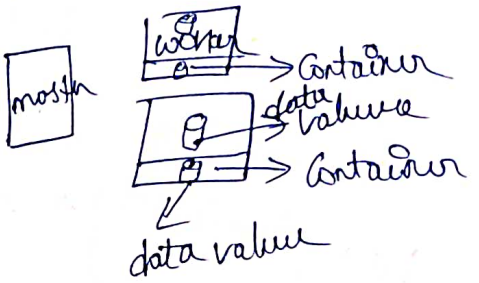
kubectl apply -f <one.yml> --dry-run → for checking errors in manifest files

↓

kubectl get Pods -o wide → shows Pods for Deploy, mongodb & check mongo on which node

↓

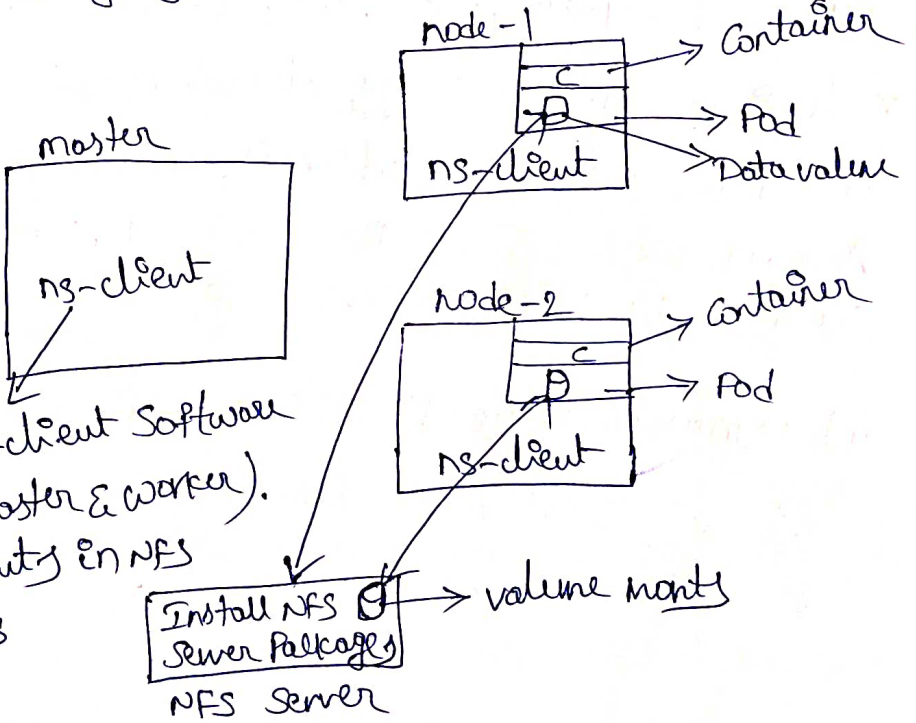
if mongo data Pods deletes & then Again automatically Creates mongo Pod but data will not loss here. for checking goto that worker node & check it in Path after deleting & recreating of Pod. if Something went wrong & goes down of node then the Scheduler changes & Scheduling the Pod to another node. here data will not consisting in another node. Incase of NFS Server we have a data for another node because it's a external Server.



4) NFS:- [with mongo.yml file NFS volume]:-

Create one EC2 instance for NFS & give Ports P-22 & P-2049 [VPC CIDR] range. this VPC_CIDR Range only for with network Communication. In this instance Install (NFS Server install on ubuntu) it NFS Server.

on google



Install ns-client Software in all nodes (master & worker). because volume mounts in NFS & Connected to NFS Server

Install ns-client Softwares in master & worker nodes.

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods

↓

for checking data coming or not ⇒ ls /mnt/share → for checking files in NFS server install

In this, if node is down & crash then the data will safe in NFS server.

PV & PVC :-

PV :- Persistent volumes are simply a piece of storage in cluster. Similar how you are disk CPU storage in server, a Persistent volume Provides storage resources for objects in cluster. PV exists independtly from the Pods. PV represents some storage which can be hostpath, nfs, ebs, azurefile, azuredisk ... etc

PV defines in two ways of volumes:

i) Static volume :- which is created manually, As k8's Admin we can create PV manually, which can be used/ claimed by PODS whatever required some storage.

ii) Dynamic volumes :- these allows storage volumes to be created on-demand.

PVC :- If Pod requires storage (volume), POD will get an access to the storage with the help of PVC. we need to to make a volume request by creating PVC by specifying size, access mode. PVC will associated with PV.

↳ Create [Deployment & service yaml files], [mongodb, service files] [PV & PVC yml file] in one.yml file,

under mongodb.yml file :-

```
spec:
  { Containers
    { env
    { volume mounts
  volumes:
    - name: mongodbhostPath
      PersistentVolumeClaim:
        claimName: mongodbPVC
```

PVC.yml file :-

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodbPVC
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce ✓
  resources:
    requests:
      storage: 1Gi ✓
```

if you don't have storage class you mention 'manual'

PV.yml file :- (PV with hostPath) :- & NFS :-

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: hostPathPV
spec:
  storageClassName: manual
  Capacity:
    storage: 1Gi ✓
  accessModes:
    - ReadWriteOnce ✓
  { hostPath:
      Path: "/kube/mongo"
  (or)
  { nfs:
      server: Pvt.IP of instance
      Path: /mnt/share
```

NOTE :-

PVC will associated with PV based on accessModes & resources required of PVC. if you have a storage class then we don't need to create PV manually. Storage class will create PV by dynamically

for Pv Hostpath:-

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide

↓

kubectl get Pv → for checking Pv info.

↓

kubectl get Pvc → for checking Pvc info

↓

for external access check in URL PublicIP4: nodeport. if you want to check data you can check in worker node do mongo in particular data.

Access Modes:-

1) ReadWriteOnce → only one Pod/nodes access read/write data

2) ReadWriteMany → ~~Any~~ multiple ~~only~~ Pod/nodes access read & write data

~~3) ReadWriteMany~~

3) ReadOnlyMany → Any Pod/nodes access read only data.

	RWO	ROM	RWM
AWS-EBS	✓	–	–
Azuredisk	✓	✓	✓
NFS	✓	✓	✓
Hostpath	✓	–	–

Reclaim Policies:- ↳ kubectl get PV → shows ~~Retain~~ Reclaim policies, status storage & bounding.

PV will have a Reclaim Policies, those are,

i) Retain:- When the PVC is deleted, PV still exists. but it is not yet available for another claim because previous claim data remains on volume

ii) Delete:- when the claim is deleted & PV also ~~deleted~~ removed.

iii) Recycle:- when the claim is deleted the volume remains but performs a basic (delete data from storage) (rm -rf /volume/*).

for PV NFS:-

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide

↓

kubectl get PV [show hostpath, NFS PVs]

↓

kubectl get PVC [shows host, NFS PVC]

↓

for checking { ls /mnt/share → for data volumes checking in NFS Server instance.

Storage class:- If you have a storage you need not to create PV. Storage class will creates PV with automatically.

NFS-Provisioner:- In this i) Create service account, Role NFS-Provisioner,

ii) create clusterRole

iii) cluster Role binding, Role binding

iv) Deployment for NFS-Provisioner

v) storage class creation

[kubernetes-manifest folder in github. change NFS-server PVC-PVC IP in manifest.]

↳ create ~~Deyd~~ [Deployment, service yaml files], [mongodb, service], [PVC] files in one file,

Under PVC file:-

AccessModes: ReadWriteMany

Storageclass: ——
↓
Empty & you can mention 'nfs-provisioner' storageclass name.

↳ this nfs-provisioner manifest files execute in master node.

↓

kubectl apply -f <nfs-provisioner.yml>

↓

kubectl get storageclass → shows here

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide

↓

kubectl get PV & PVC [shows PV & PVC]

↓

for outside access, enter in URL PublicIP:Port.

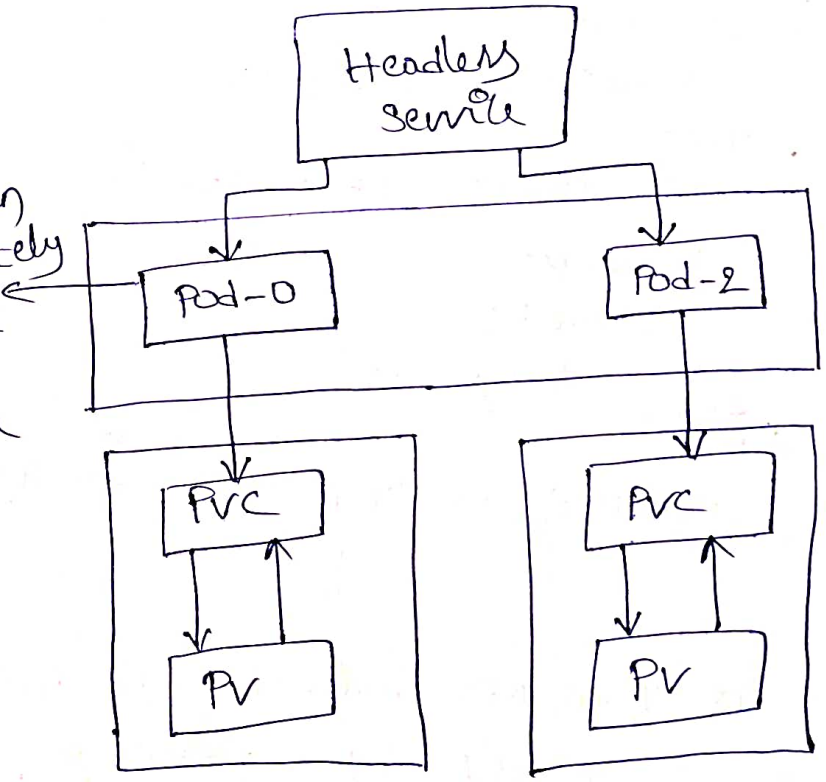
↓

for checking data in nfs-server } ls /mnt/share → check data volumes coming & not

Stateful Sets :- Statefulset used for maintaing volume for each Pod & it's manages the stateful application [Ex:- mongo].

Every replica of stateful set will have its own state, & each of the Pods will creating own ~~state~~ PVC. So a statefulset with 3 replicas will creates 3 Pods, each having its own volume so total 3 PVCs.

But in Deployments are usually used for stateless application. however, you can save the state of deployment by attaching the PV.



If Pod is deletes then volume is safe seperately for next creation of Pod with that name & uses same volume also.

In this, Pods will creates one by one. but not creates all replicas in one at a time.

↳ Create [deployment & service.yml], [statefulset for mongo & service with Headless] in one.yml file,

StatefulSet for mongo!-

---> serviceName: "mongo" → here enter Headless Service name exactly.

under spec!-

```
spec:
  terminationGracePeriodSeconds: 10
  {containers
  {env
  volumeMounts:
    - name: mongo-persistent-volume
      mountPath: /data/db
volumeClaimTemplates:
  - metadata:
      name: mongo-persistent-volume
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

In deployment, MONGO_HOSTNAME is keep it as Headless service name (mongo)

↓

NOTE!- here in this NFS Provisioner is installed so automatically PV is created by storageClass. if it not having any Provisioner you need to create PV & PVC manually.

↓

kubectl apply -f <one.yml>

↓

watch kubectl get Pods → mongo Pods Creates one by one based on replicas

↓

kubectl get PV&PVC → shows three PV&PVCs for 3 replicas

↓

for outside access use Public:Port.

↓

if you want to check data in NFS-server instance then goto that Path

for checking → ls /mnt/share → shows data for 3 replicas

ConfigMap:- It's kubernetes object using which we can create/define Configuration files or Configuration values as per key value pairs.

Here I am not hard code directly in Deployment & StatefulSet mongodb. I am refreeing values from ConfigMap files. so here I am creating configmap file.

StatefulSet mongo:-

under SPEC:-

Deployment:

```
SPEC:
  { Containers
  env:
  - name: MONGO_INITDB_ROOT_USERNAME
    valueFrom:
      ConfigMapKeyRef:
        name: SpringappConfig → Give ConfigMapname here
        key: mongousername → Give exact name key in ConfigMap file
  - name: MONGO_INITDB_ROOT_PASSWORD
    valueFrom:
      ConfigMapRef:
        name: SpringappConfig → Give ConfigMapname here
        key: mongopassword → Give exact name key in ConfigMap file.
```

} Same Deployment also.

Create ConfigMap.file :-

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: springappconfig
data:
  mongousername: devdb
  mongopassword: devdb@123
```

↳ create [Deployment & Service], [StatefulSet Mongo & Service] files in one.yml file & Configmap file,

↓

kubectl apply -f <Configmap file>

↓

kubectl get Configmap(cm) → Shows cm

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide

we alternate way also for creating Configmap. Instead of Configmap file you can create Imperative way also using Commands.

↓

kubectl describe cm <cm-name>

↓

Here shows Password & keys data. so in order to avoid those problem Secret is come.

Secret:- It is used for store the Confidential data such Password, ACCESS & secret access keys .. etc.

Create one secret file with passwords (mongo password) & refrence this secret in deployment & statefulset mongo yaml files.

Under spec:-

Deployment:-

```
spec:
  { Container
    env: MONGODB_PASSWD  USERNAME
      valueFrom:
        secretKeyRef:
          name: springappsecret  → Secret name give here
          key: mongpassword  → Give exact key name in Secret file
```

Illy Deployment also.

Create secretkey file:-

```
apiVersion: v1
kind: Secret
metadata:
  name: springappsecret
type: opaque  → for encrypting
stringData:
  mongopassword: devdb@123
```

for creating interactive (2) way,

↳ kubectl create secret generic springappsecret --from-literal=mongopass-word=devdb@123

↓

kubectl apply -f <Secret.yml>

↓

kubectl get secrets → for shows secrets

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide

↓

for access to outside PublicIP4: Port.

↓

kubectl describe Secret <Secret-name>

↓

Here not shows password information because here we are encrypted those data through 'opaque'.

Rediness Probe & Liveness Probe:-

These two Probes are used to Controll the health of an application running in Pod's Container.

Rediness Probe:- this type of Probe used to detect if a Container is ready to accept traffic.

Liveness Probe:-

Suppose our Pod is running but Container is not responding to our requests due to some reasons let's memory leak, CPU usage & application deadlock.

Liveness Probe checks Container health checks, if some reasons Liveness Probe fails, it restarts the Container.

we can Perform diffrent types of health checks,

1) httpGet
   / (or) /JavawebApp
2) execute Command
3) tcp

without Liveness & rediness:-

Suppose Created 2Pods with the help of deployment & take that node IP with Port then application access to outside. if Suppose in of Container the .war file was deleted intentially. but here Pod is running always but container application not running. if you ~~trai~~ trying access the application you will get [404 error] at that time unless until application coming to normal stage. but here we not able to see restarts of Container & EndPoints of services removed or not.

With liveness & readiness Probe:-

Goto deployment file & [HPA Autoscaling],

Under the spec:-

spec: { Containers

{ resources

readinessProbe: → healthcheck

httpGet:

Path: /javawebapp

Port: 8080

initialDelaySeconds: 15 → if pod is Scheduled & application is not up and running immediately. it will take some time i.e 15 seconds. for starting ~~container~~ stage.

timeoutSeconds: 1

PeriodSeconds: 15

livenessProbe:

httpGet:

Path: /javawebapp

Port: 8080

initialDelaySeconds: 15 →

timeoutSeconds: 1 → if 1 second application is not responding Consider as failure.

PeriodSeconds: 15 → Every 15 seconds once liveness Probe wat Perform

↓

kubectl apply -f <deploy.yml>

↓

kubectl get Pods -o wide

↓

kubectl get svc

↓

suppose if delete .war file in one of the Container. then here able see service endpoints removing & Pod Again restarting.

kubectl get Pods -o wide [Pods restarts]

↓

kubectl describe svc [endpoints removed Again Coming]

That means here readiness & liveness Probes are failed. So Pods health checking is done by these Probes watching of [restart Pods & ends Points of Svcs]. Intersuntil readiness Probe is successful we don't see an endpoints of Svc. Illy liveness Probe also same.

Node selector:- this is a simple Pod Scheduling feature that allows scheduling a Pod onto a node (particular node). node selector labels are key-value pairs that can be specified inside the PodSpec.

creating label for node,

↳ kubectl label nodes <node-name> <label-key=label-value>
↓ ↓ ↓
IP-172.10.43.73 app=workerone

↳ kubectl get nodes --show-labels → for showing node labels

Under deployment:-

```
Spec:
  nodeSelector:
    name: workerone
  {Container
   {resources
   {readiness & liveness
        Probes
```

↓

kubectl apply -f <conf.yml>

↓

kubectl get Pods -o wide → here show replicas Pods running on particular node based on labels.

Node Affinity :- It is a Advance feture of nodeSelector. In this 2 types of affinity rules. 1) Preffred rule
2) Required rule.

1) In Preffred rule, a Pod will assigned on non matching node if & only if no other node in cluster matches the specified label. In this "PreferredDuringSchedulingIgnored DuringExecution" rule Pod will assigned to any nodes. It doesn't depend on labels of node it can be assigned to non of labels of nodes also.

under deployment:-

```
spec:
  affinity:
    nodeAffinity:
      PreffredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          Preference:
            matchExpressions:
              - key: name
                operator: In
                values:
                  - workerone  ---> node label name here.
    { Containers
      { resources & Probes
```

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide → for showing

↓

kubectl scale deploy <deploy-name> --replicas 3

Pods Schedule to non matching labels of nodes

2) In Required rule, if there are no matching nodes, then the Pod won't be rescheduled. there are ways.

i) requiredDuringSchedulingIgnoredDuringExecution:- Pods will be scheduled based on labels & ~~selector~~ spec matches do Pods if non do node will not having label the Pod doesn't schedule here. In this ~~Re~~ once Pod scheduled, labels are ignored means labels are changed still the Pod will Continue to run that node.

ii) requiredDuringSchedulingRequiredDuringExecution:- Similar to upper one but here In this once Pod scheduled to node, labels are ignored means labels are changed then Pod will be removed from the node.

underDeployment:-

```
spec:
  nodeAffinity:
    ~~R~~requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "name"  ---> label key value
          operator: In
          values:
          - workerone  ---> label name here.
```

why do you need ServiceAccount:-

Some Suppose Some application need access to the k8 & APISERVER & workload & k8 resources, we will create serviceaccount, we will create a role & we will bind that role to service account. it will get an access.

Taints & Tolerates:-

The Pods are don't scheduled in master. because kubernates master node a Taints. if you want schedule the Pods in mastermode then you need to tolarate the mastermode.

If Some of the Pods don't want scheduled in worker nodes then you can do one thing i.e you need to taint the worker node. then you not able to scheduled the Pods into a particular worker node Again if you want to ~~to~~ Schedule ~~Point~~Pods to the node then you can tolarate the nodes.

Go through "Taints & tolarate" Documentation they better to understanding easily.

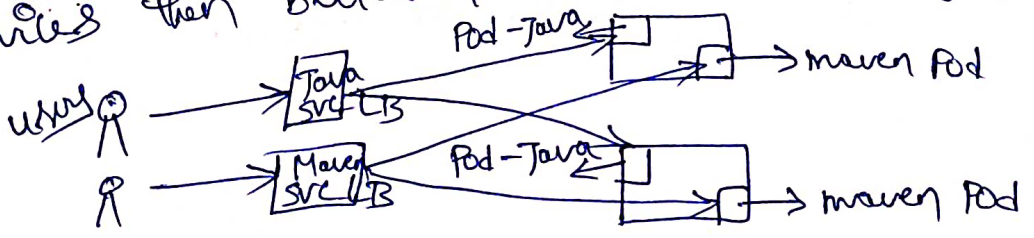
if you taint the node & then check

↳ kubectl describe nod <node-name> → for here shows taint information & all

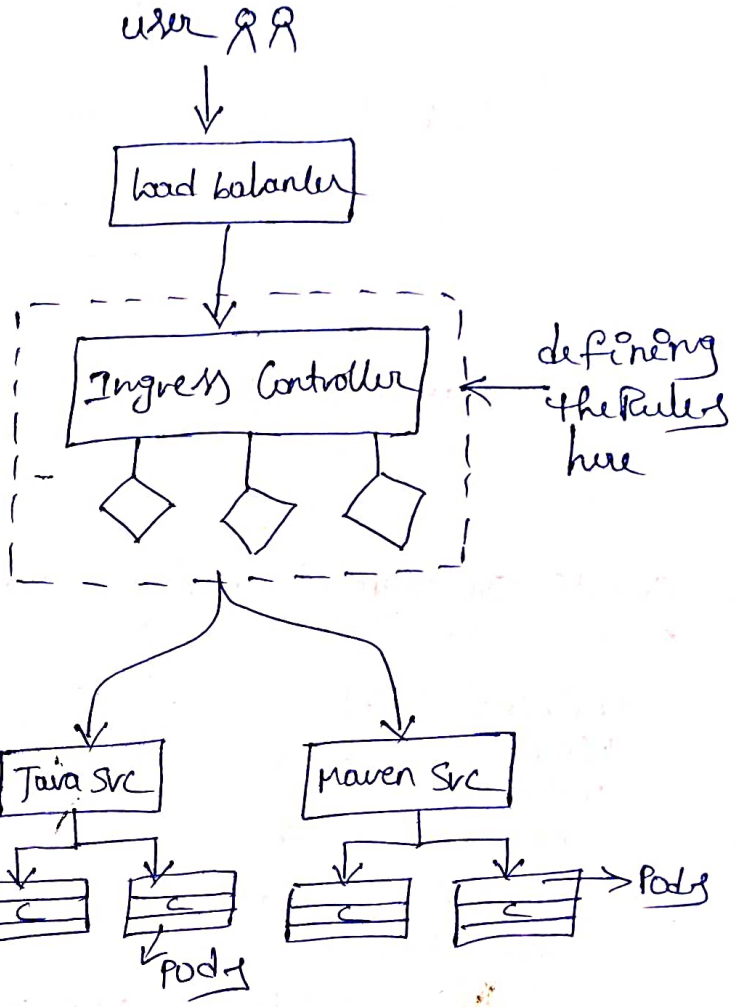
if you want to tolarate then use Some of Commands in Pods manifest file.

LoadBalancer Service:-

It is the one type of service. it is also expose the application to outside. but in this Each service ~~app~~ ~~code~~ we have a ~~Service~~ Separate loadbalancer that means Separate DNS is Provided. if you want to single DNS loadbalancer for all Services then better to Go with the Ingress. ~~this~~



Ingress :- It's kubernetes object where we can define the rules to route the traffic external sources to the Services with in k8's cluster.



Here external traffic will talk to the Ingress Controller through the ingress load balancer. The ingress controller based on the rules the traffic sending the Srcs. from the Srcs traffic passes to the application inside of the Pod.

There are different types of ingress-Controllers,

i) NGINX ingress Controller
ii) Traefik k8's ingress Provider
iii) Tyk operator
iv) HAPROXY ingress ..etc

First install ingress-Controller in AWS :-

get done "k8-ingress repository" [Github :- kubernate-ingress repository.

↓

cd k8-ingress | deployments

↓

ls [daemon-set, deployment, service, Common]

↓

For creating ns & service account for ingress → kubectl apply -f Common | ns-and-sa.yaml

↓

kubectl apply -f common/ → for creating RBAC, Default Secret & Configmap

↓

kubectl apply -f daemon-set/nginx-ingress.yaml → for deploy ingress-controller
(or deployment)

↓

kubectl get all -n nginx-ingress → show information for particular ns.

Create svc type loadbalancer for ingress in nginx-ingress ns:- [in Github]:-

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
  namespace: nginx-ingress   → Give namespace here.
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "tcp"
    "          "          "          -proxy -    "   : "*"
spec:
  type: LoadBalancer   → type of loadbalancer
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 80
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    app: nginx-ingress   → match the label here
```

↓

kubectl apply -f service/loadbalancer-aws-elb.yaml

↓

kubectl get all -n nginx-ingress → show DS, ELB, ingress Pods all information

↳ Create [Javawebapp deployment with svc clusterIP] & [mavenweb appdeployment with svc clusterIP] ~~Create~~ files & apply here.

↓

kubectl apply -f <Javawebdeploy>

kubectl apply -f <mavenwebdeploy>

↓

kubectl get Pods -owide → shows Pods for Java & maven web app

↓

kubectl get svc → show all svc.

Creat a Rule for ingress Controller for Passing traffic to Pods through svc:-

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-resource-1
spec:
  ingressclassName: nginx
  rules:
    - host: javawebapp.mituntechdevops.co.in → host based routing
      http:                                  ↑ Paste in URL
        Paths:
        - backend:
            serviceName: javawebsvc ⟵ Give svc name here for Passing traffic to Particular Pods
            servicePort: 80
    - host: mavenwebapp.mituntechdevops.co.in
      http:
        Paths:
        - backend:
            serviceName: mavenwebsvc
            servicePort: 80
```

↓

kubectl apply -f <ingress-rule>

↓

kubectl get ingress

Now here "ingress-load-Balancer DNS" name add into the main domain name with the help of ~~Sub~~ creation of records with subdomain names.

Goto main domain name → create record → Give name & Select loadbalancer DNS → Save it ⇒ for Javawebapp

Goto main domain name → Creat record → Give name & Select loadbalancer DNS → Save it ⇒ for mavenwebapp

↓

Paste in URL with Sub-domain names then application page is coming for both Javawebapp & mavenwebapp.

↓

Here traffic is sends to the Pods through the ingress-Controller based on rules from the load balancer.

Path based Routing Rule

Under ingress Controller Rule:-

```
under SPEC:
  ingressclassname: nginx
  rules:
  - host: mithuntechdevops.co.in
    http:
      paths:
      - backend:
          servicename: springapp      → Give springboot application svc name is this running on root path.
          servicePort: 80
        path: /java-web-app
        backend:
          servicename: Javawebappsvc  → Give Javawebapp svc name this is running on /java-web-app path
          servicePort: 80
      - path: /maven-web-app
        backend:
          servicename: mavenwebappsvc →
          servicePort: 80
```

↓

Create record in main domain with "mithuntechdevops.co.in" & add the ingress load balancer → Save it.

In URL
- mithuntechdevops.co.in → spring boot page coming
- mithuntechdevops.co.in/java-web-app → Javawebapp page
- mithuntechdevops.co.in/maven-web-app → maven webapp page.

Here one single load balancer traffic passes (traffic) to "3" application based on ingress-controller rules it passes through the svcs to Pods.

If you want to https connection for application you need to create 1) Self signed Certificate for that website [shows CA file & cert file] 2) Create Secret for that Certificates

Refer: kubernate-ingress

And

3) Add the tls or SSL Certificate details in ingress Controller Role yaml file under the Spec.

↓

check connection in google with that website. the website becomes to be secure storage position.

Role Based Access Control:-(RBAC):-

When a request is sent to the API-server, it first needs to be authenticated (to make sure the requestor is known by the system) before it's authorized (to make sure the requestor is allowed to perform the action requested).

Authentication methods:-

i) IAM Authentication (Token)

ii) x509 Certificate

iii) LDAP

RBAC in k8's is the mechanism that enables you to Configure specific set of Permissions that define how a given user or groups can interact with any k8's object in cluster.

1) Devops/Admins → for accessing k8's objects giving the required Permissions to users by the Giving of RBACRoles

2) Applications → for creating applications (metric Server, nfs server) in k8 cluster by the giving of RBACRoles

3) Endusers → Endusers Just access the application which is deployed in k8's via Services.

RBAC in kubernetes is based on three key Concepts which under in Role,

i) verbs — used, write, Create, update, delete

ii) API Resources — Pods, deployments, rc, rs, Services, PV, PVC

iii) Subjects — users, groups & Service account

↳ kubectl api-versions → for showing 'apiversions' of all objects.
↳ kubectl api-resources

for Creating RBAC access to user & interact with the K8 cluster:-

1) Create one user (Mithun) & Give Programatic access. ~~& attach the Policy EKS to~~

2) Create one Policy & Give Permissions (Read, execute) & Add resources names [EKS ARN & All K8's in account). And attach this policy to user & Group. that means user can access to ~~user~~ EKS cluster.

3) take one vertual machine (Ubuntu terminal) & install "awscli" in this → and Aws configure (access & Secret key & region & cluster).

↓

aws eks list-clusters → for showing all clusters.

↓

aws eks update-kubeconfig --name=EKS_Demo --region ap-south-1
[shows /.kube/config file path]

↓

kubectl get nodes

↓

error: unauthorized

4) Go to kubernetes cluster machine & check it one.

↓

kubectl get configmap -n kube-system → for shows default configmap & 'aws-auth' configmap shows.

Here map the user in "aws-auth",

kubectl edit configmap aws-auth -n kube-system

↓

Under mapRoles:-

in last,

mapUsers: |

Given user ARN in IAM we have this. ← - userarn: arn:aws:iam::935840844891:user/mithun
username: mithun

5) In kubernetes cluster create, Role/clusterRole & Binding/ ClusterBinding.

Role:- the grant permission defined to users at a namespace level.

ClusterRole:- the grant permissions defined to users or groups at irrespective namespace that means all namespaces.

↳ we are giving key concepts (verbs, resources, apigroups) & namespace & accessmodes types in Role creation.

↳ we are giving username information, namespace & accessmode information in Rolebinding creation.

↓

kubectl apply -f <Role-rolebinding>

6) Goto terminal & check it once kubectl command.

↓

kubectl get nodes → shows nodes information

Here, Based on Role permissions (Pods, deploy, rc, rs, services) we are going Perform work.

Helm:- Helm is the package manager for kubernetes. It allows you to install/deploy application on k8's cluster in similar manner to yum/apt for linux distributions. Helm fetch, deploy & manage the lifecycle of applications both own & 3rd party application.

Helm introduces several familiar concepts such as,

i) 'charts' → Package containing k8's manifest (templates)

ii) Helm repository → It is a central repository which maintains the helm charts.

iii) ACLI → with this install/upgrade/remove the application by using commands.

If you want deploy the application in k8's you need to create all the objects [Deploy, Services, Configmap & Secrets ..etc] with yaml files. But Helm manages all of this for you. Helm greatly simplifies the process of creating, managing & deploying applications using Helm charts.

In Addition, Helm also maintained versioned history of chart (application). if something goes wrong, you can simply call "helm rollback". similarly if you want upgrade chart, you can simply call "helm upgrade".

In Helm chart there are two main components,

i) templates

ii) values

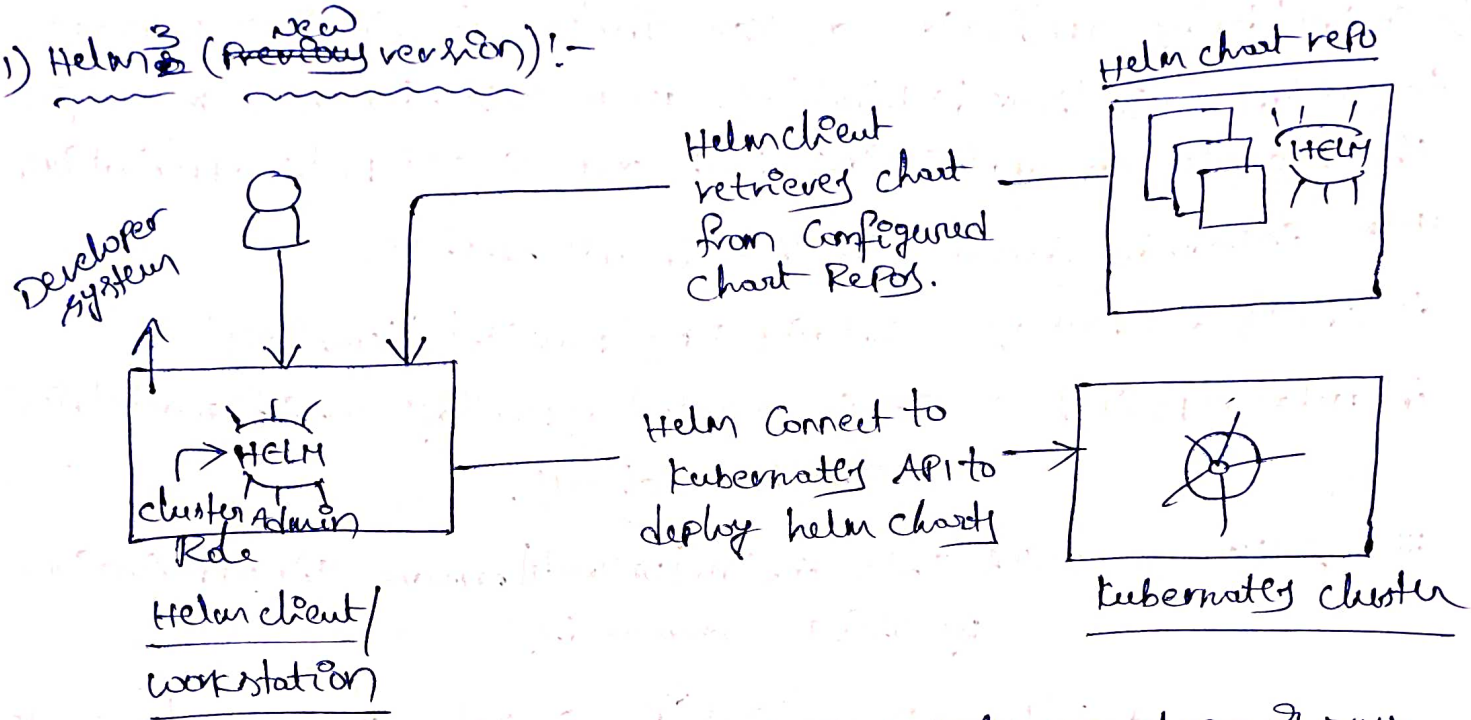
Javawebapp

- chart.yaml # metadata information
- charts # define this application will depend another application
- templates chart name.
  - deployment.yml
  - service.yml
  - Configmap, Secret.yml files
  - serviceaccount.yml
- values.yml # define values

we are not directly hardcode the values here in manifest files. here referring variables from values.yml file.

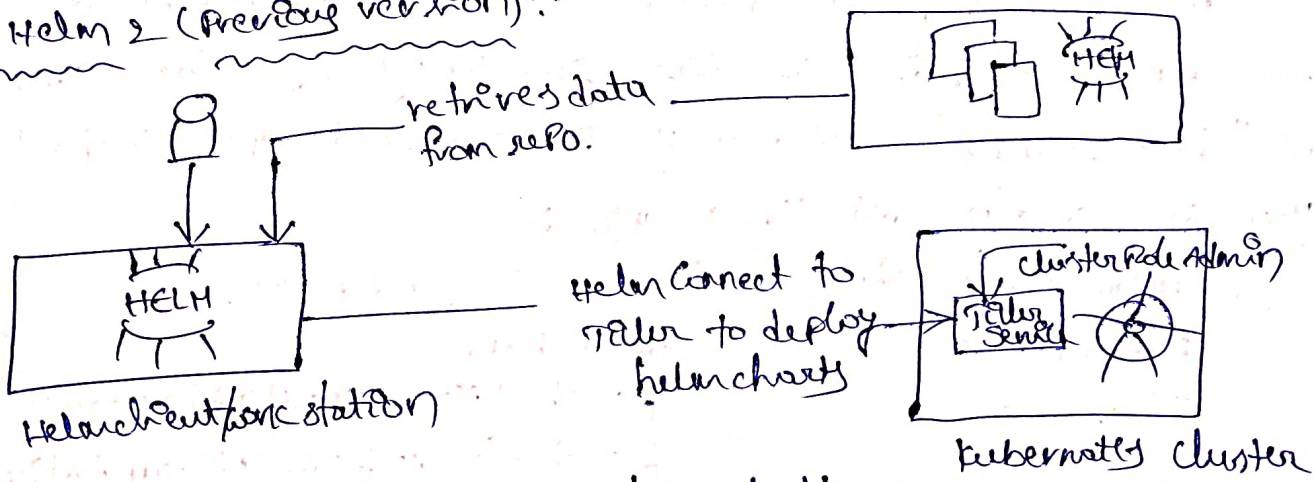
Helm architecture:-

1) Helm 3 (new version):-



Here Helm client system operated by either Developer or any. this Helm client retrives the data from Helm chart & it's Connected to kubernetes API for deploy the this helmchart. this is done because of helm client will having a kube/config file in system.

2) Helm 2 (Previous version):-



Similar to Helm3, But helmcharts deployed in Tiller Server first. not directly access to API SERVER of kubernetes.

Helm3 install on machine!-

Google! Helm3 installation (script way or Binary way).

i) Script way!- Download the file → ls (shows list of file) →
through link

Give permissions to file → run that file as script file.

↓

↳ helm → (shows all information about helm).

ii) Binary way!- Download the zip file (tar) based on flavour →

extract it → and move to /usr/local/bin/helm.

↓

↳ helm → (shows all information about helm)

If you want to install the application you need a "Helmchart".

applications are like metric-sever, nfs Provisioner, Prometheus, nginx-ingress ...etc.

here I want to deploy Metric-server,

↓

kubectl get pods
kubectl get nodes
} Initially, metric API not available

↓

helm repo ls → for showing repositories

↓

for adding repository → helm repo add stable https://charts.helm.sh/stable

Give name for repo

↓

helm repo ls → shows here repository

↓

↳ helm search repo stable <reponame> → for showing list of charts in particular repository.

In this repository so many charts having some example charts are metric server, nfs provisioner ...etc.

↓

↳ helm search repo stable | grep "stable/metrics-server" → for checking particular chart in repo.

↓

I want install metric server chart application so

↳ helm template stable/metrics-server → showing for templates in the do yaml files
(stable/metrics-server: chart name)

↳ helm show values stable/metric-server → for showing values information file.

↳ helm show values stable/metrics-server >> metricservervalues.yml → here values of stable/metrics-server file passing to our creation of file (metricservervalues.yml). if you want customise or edit the data in values files you can edit in this file or you can pass through the commands (indirect way) also. now I am installing.

↓

↳ helm install metricserver stable/metrics-server -n kube-system -f metricservervalues.yml
(metricserver: give name)

for installing metric-server

↓

↳ helm ls -n kube-system → for showing application which is deployed to helm.

↓

↳ kubectl get all -n kubesystem → metrics server shows here.

↓

↳ kubectl get serviceAccount -n kube-system → service account, Roles shows here.
clusterRole
clusterRolebinding

↓

kubectl get top pods
kubectl top nodes } → metrics shows here.

↓

helm ls -n kubesystem → for showing application & versions.

↓

if you want update anything in this values. you can update.

↓

helm upgrade metricserver stable/metrics-server -f metricsservervalues.yml -n kube-system → for upgrading of application.

↓

helm ls -n kube-system → for showing application versions.

↓

kubectl get all -n kube-system → for shows all updated metric server pods

↓

if Something goes wrong with new version then you can Rollback to previous version.

↓

helm rollback metric-server -n kube-system → for rollback to Previous version

↓

helm ls -n kube-system → show application versions

↓

helm uninstall metrics-server -n kube-system → for deleting the metric-server application Completely.

Cluster Autoscaler :- the cluster Autoscaler automatically adds or removes nodes in a cluster based on resource request from Pods.

initially check,

kubectl get all -n kube-system → here shows default DS, RS, Deployments, kube-Proxy, aws-node, coredns Controller Pods.

1 → create a [one deployment & Service type load balancer] & apply it.

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -o wide

↓

kubectl get svc [shows, 1) load balancer Dns 80:30491
2) Default svc

this instance Port open in all the k8s nodes because, the request sends the traffic to svc & svc use this

ServicePort

InstancePort

Port & Pass this traffic to nodes (pods in Containers). if you want to access through Domain name, make sure you need to create one record with this load balancer.

(or) ↓

~~nslookup~~
nslookup <loadbalancerDNS>

↓

Google!
nslookup download for which flavour you want

Here load balancer IP's shows. take this IP & keep in vi /etc/hosts file. then able to access in URL Externally i.e [only for me].

vi etc/hosts
↓
10.3.40.12 Java.com
↓ ↓
load balancer IP | Give website name

↓

for your website getting more traffic. at that time HPA Creates more Pods (replicas) based on traffic that means we can scaleup & down Pods with the HPA. if one of node is down due to some Problem. the k8 will reschedule the Pod to another node. After some time Again another node is coming to ready state based on desired node members.

Based on request resources (CPU, memory) the replicas are created in nodes. if no. of replicas are increased some of the Pods are not scheduled to any node because that nodes CPU & memory was full so we not able to schedules those pods to any node.

So this Problem you can overcome with this "cluster AutoScaler". with this clusterAutoScaler you can increases the nodes based on increasing of no of replicas & automatically scale down also based on no. of Pods. But max nodes will comes only based on your given number. the main ~~Advante~~ Advantage of this managed k8's it's automatically scale up & down Pods & nodes. But in self managed k8's we need to configure Prometheus & grafana for monitoring these states if you getting high traffic something Pods & nodes goes down then notification will sends to team & team can fix this issue.

↳ create clusterAutoScale service account, Role, Role binding, Deployment for clusterautoScaler. in one.yml file.

↓

kubectl apply -f <one.yml>

↓

kubectl get Pods -n kube-system

[shows clusterautoScale Pods, DS, Deployments, aws-node, kubeProxy, coredns.]

↓

kubectl logs <clusterautoscalerPod> → for showing more information about logs.

[authorized Perform, Pods pending not scheduling.]

↓

↳ Create one Policy [Json file]. In this file we have a all permissions of Autoscaling. so this policy you can attach to Iam Role which is using workernode group in kubernetes (eks) cluster in Console.

↓

kubectl get Pods -n kube-system

↓

kubectl get nodes → { here nodes increases based on no. of replicas }

↓

kubectl get Pods → { Pods Creating to nodes now there is no Pending Pods }

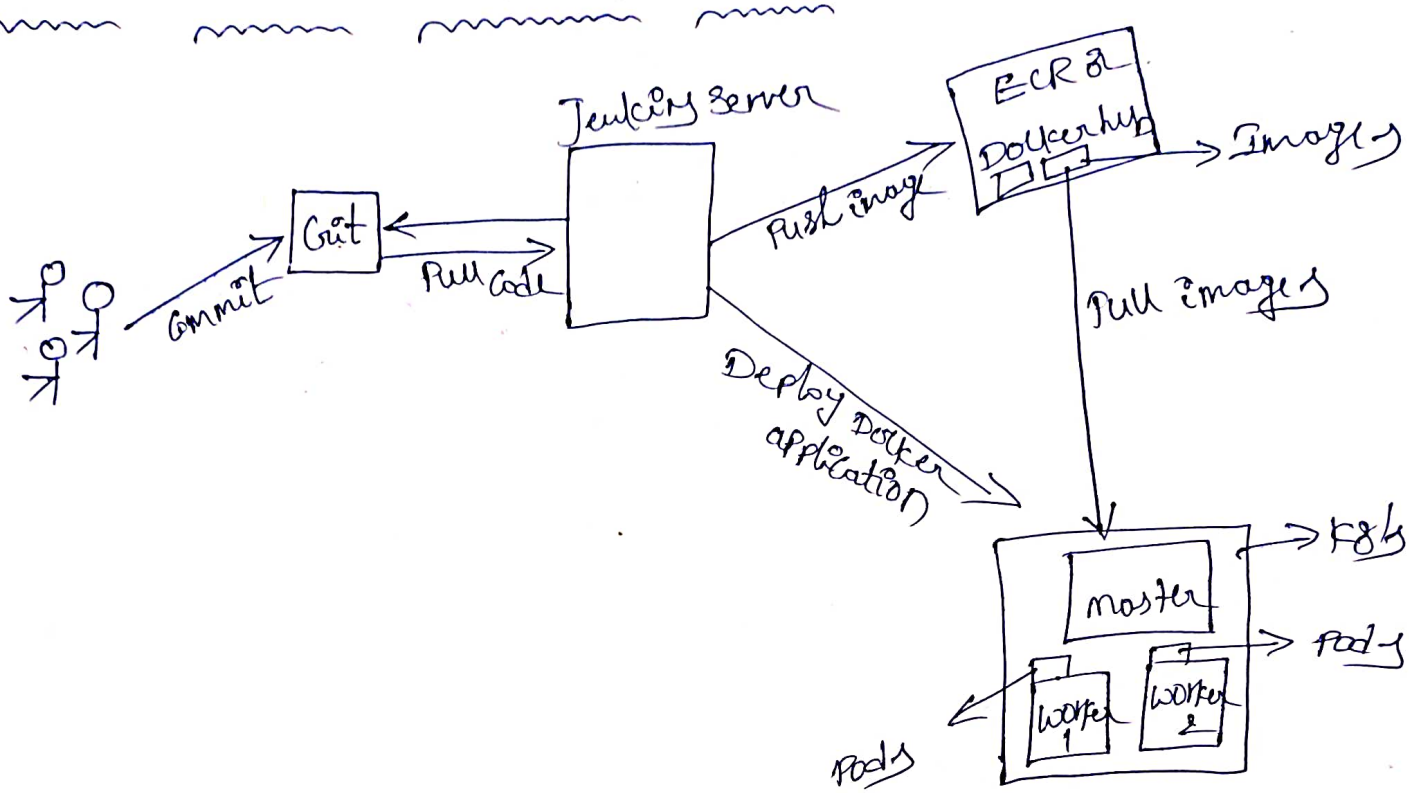
↓

here Autoscaling of cluster work done based on no. of replicas

↓

kubectl get storageclass
[default storage aws-ebs in managed k8s]

Jenkins - Kubernetes Integration Project:-



First create one Jenkins server & one K8's Server. Install all those Configuration (binary, keys .. etc) in both Jenkins & kubernetes (kubectl, eksctl .. etc).

Step①:- Create one instance & Install Jenkins & check status of Jenkins, And install Docker in Jenkins server & add permission Jenkins into a Docker, Install maven also.

↳ Sudo usermod -aG docker Jenkins
↳ chmod 777 /var/run/docker.sock
↳ Docker pipeline plugin install
↳ 'kubernetes Deploy' plugin install
↳ 'kubernetes CLI' plugin install

Step②:- Create kubernetes cluster & setup this. execute the command 'kubectl' interacting or not.

Git-hub repository :- (spring-boot-mongo-docker) :-

src/main → source code we have
.gitignore
Dockerfile → for creating Docker image
Jenkins file → Pipeline entire script
docker-compose file
Pom.xml → with the help of this maven download dependencies & compile it & then creating Artifact.
SpringBootmongo.yml → Deployment to k8's

[Deployment, svc file
mong, svc file,
Secret, configmap]

Jenkins Pipline Scripts :-

node { }

1st → Git clone → so copy the URL & create script through

Git clone URL.

Goto Pipeline Syntax → Sample step = Git (Selecture)
Branch = master/main)
Credentials = Github username & password add here with name of "Id". this Id is directly Add into the "Credentials" option in Jenkins Dashboard

stage("Git clone") {
"Pastehere script"
}

↓

Generate Pipeline Script
(Script shows here).

2nd → Creating Compile & Package

if you are not install maven in Server then install in Dashboard.

Manage Jenkins → Global tool Configuration → Add maven →
Install automatically [name – maven-3.6.1]

↓

apply & Save

```
stage("maven clean Build") {
    def mavenHome = tool name: "maven-3.6.1", type: "maven"
    def mavenCMD = "${mavenHome}/bin/mvn"
    sh "${mavenCMD} clean Package"
}
```

maven where is it.

Creating Package

3rd → Build Docker image

```
stage("Build Docker Image") {
    sh "docker build -t dockerhandson/spring-boot-mongo ."
}
```

dockerhandson/ ← username of Dockerhub (or) Give username of ECR registry

4th → Pushing Docker image

Groovy Pipeline Syntax → Sample Step = withCredentials: Bind Credentials

Bindings = Secret text

Credentials = Add (Secret text)

Here enter Secret is dockerhub or ecr login password, Create Give ID name. this ID Adds into Jenkins Credentials.

↓

Generate Pipeline Script.

Stage ("Docker Push") {

"Paste Pipeline Script here" → Use password entry script, not human readable.

sh "docker login -u dockerhandson -p ${GiveIDname}"

}

sh "docker push dockerhandson/springg-boot-mongo"

↓

dockerhub username or ECR username Give here.

}

Instead of dockerhub password & username, you can enter AWS ecr login credentials username & Password.

↳ If you want run if image as a container then create stage & Give portmapping. (or)

5th → Deploy application into kubernetes cluster Deploy in k8s cluster

Install plugin - kubernetes Continuous Deploy

Jenkins will talk to k8s in two ways :-

first way → Credentials → Add Credentials → select kind (kubernetes Configuration kube.config) → ID = Give here

Description :

kubeConfig { o Enter directly ✓
o from file on Jenkins
o from a file on the k8s master node.

here enter kub/config file Content.

↓

OK

Second way:- Install kubectl & add kube config in Jenkins Server.

vi .kube/config

only you can install be in when Jenkins user.

first way

```
/**
stage("Deploy Application in k8s cluster") {
    kubernatesDeploy(
        configs: "springBootmongo.yml",
        kubeconfigId: "k8Id.enterhere"
                      in credentials ,
        enableConfigSubstitution: true
    )
}
**/
```

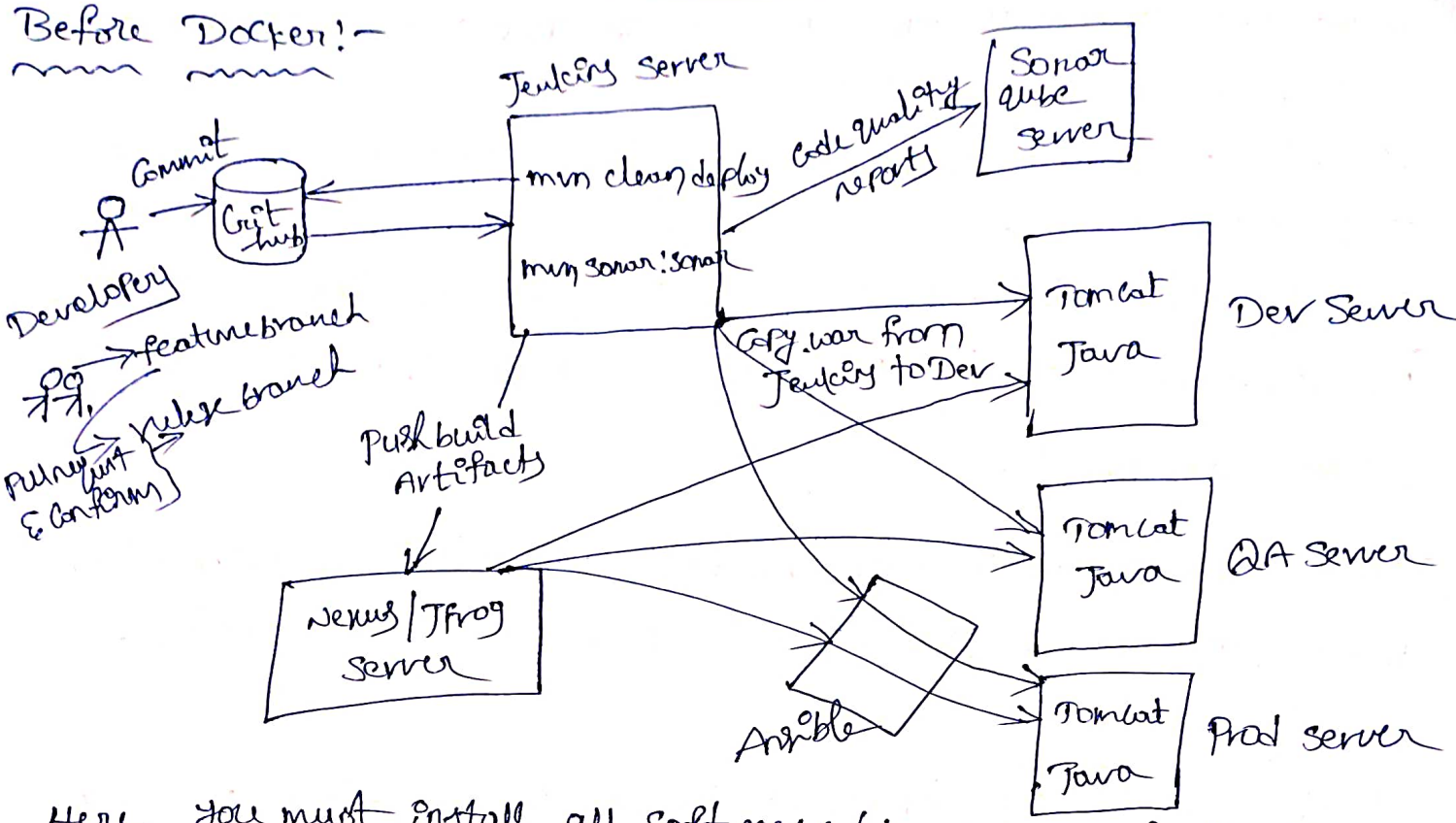
(or)

2nd way

```
stage("Deploy TO kubernates cluster") {
  sh "kubectl apply -f springBootmongo.yml"
}
```

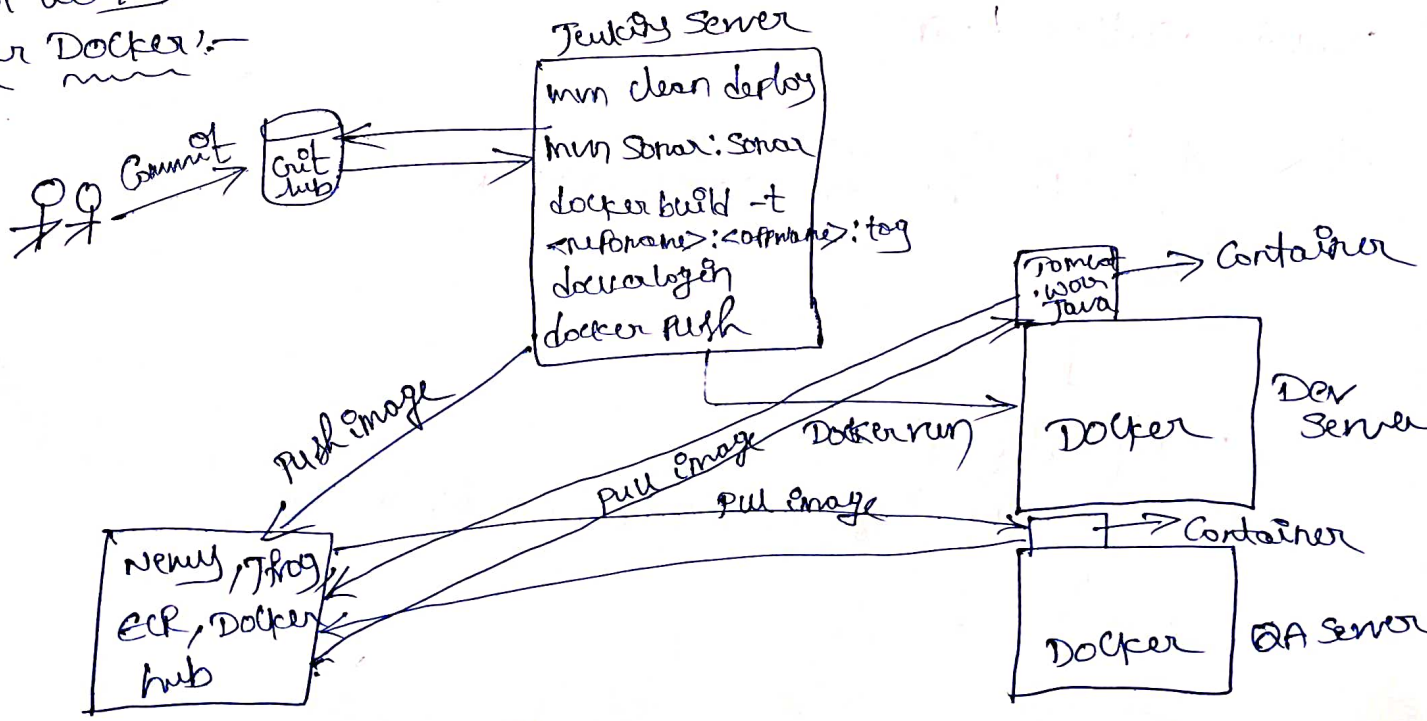
Goto kubernates master & check Pods Containg running or not. & you accessable also through Jenkins Server because here copied the .kube/config file.

Before Docker:-



Here you must install all softwares (tomcat, Java) in Dev, Stg & Production servers because for the deploying of Artifact. then able to access through outside. all ~~the~~ require dependencies keep it as a same in all environments ~~for~~ deploying the Artifacts if something goes wrong then the application not works.

After Docker:-



you not able to install any software in Dev, Stg, Production environment only you need to install docker that's it. whatever build artifact is coming that is converted into a form of docker image. this docker image having required dependencies & application code. that means docker image consisting of [Tomcat, .war & Java]. Here deployment is fast compared to normal deployment.

Step①:- Create one instance (Jenkins) & install Java, Jenkins, maven, Docker & give require permissions to Jenkins.

Step②:- Create one deployment server for installing the Docker & give Permissions.

↳ apt install docker.io -y

↳ sudo usermod -aG docker ubuntu → for ubuntu user added to docker group to execute docker commands

Jenkins pipeline Script:-

```
1st → Git clone the repository.
      def buildNumber = BUILD_NUMBER
  node {
      }
  stage("Git clone") {
      git url: 'Past URL of Git', branch: 'master'
      }
```

2nd :- Install mvn automatically in Global tool Configuration.

```
stage ("maven Clean package") {
    def mavenHome= tool name: "maven-3.6.1", type: "maven"
    sh "${mavenHome}/bin/mvn clean package"
```

3rd → Build docker image

```
stage("Build Docker image") {
    sh "docker build -t dockerhandson/Java-web-app-docker:${buildNumber} ."
```

(dockerhandson → Give username of dockerhub & ECR URL; . → cd)

Check image coming or not in Jenkins server not in deployment server,
↳ Sudo docker image → shows here with tags also.

4th → Docker login & Push

Goto Pipeline Syntax, → Sample step = withCredentials: Bind Cred
Bindings = Secret text
Credentials = kind (Secret text)
here enter Password of docker hub & ECR & Give ID name.
these credential will added in Jenkins credentials.

↓

Generate Pipeline Script

```
stage ("Docker login & Push") {
    Paste Generate Pipeline script {
    sh "docker login -u dockerhandson -P ${ID name here}"
    }
    sh "docker Push dockerhandson/Java-web-app-docker:${buildNumber}"
}
```

5th → Deploy application in Docker

L> Install 'SSH Agent' plugin here.

Groovy Pipeline Syntax → Sample Step = sshagent = SSH Agent

↓

Add Credentials → ssh username & Pvt key

ID =
Username = ubuntu
Pvt key = "Paste DockerServer Pemkey".

↓

Generate pipeline script

```
stage ("Deploy application in Docker server") {
    Paste Generate Pipeline Script {
        sh "ssh -o StrictHostKeyChecking=no ubuntu@PvtIPof DockerServer docker rm -f javaweb-app Container || true"
        sh "ssh -o StrictHostKeyChecking=no ubuntu@PvtIP docker run -d -P 8080:8080 javawebapp-docker : ${buildNumber}"
    }
}
```

delete container if already exists.

Image name

↓↓

Paste PublicIPv4 of Docker Server then able to see tomcat application.

https://20.17.179.44:8080/java-web-app - for Going to particular option.

Deploy springboot microservice into EKS cluster using Jenkins Pipeline:

Step①:- Create one Jenkins Server & setup Jenkins in Server.

↳ install Jenkins (Binary, keys, Jenkins)

↳ install Java, maven, Docker

↳ sudo usermod -aG docker ubuntu & Jenkins

In this Server install all utilities in ubuntu user,

1st → Awscli install

↳ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"

↳ sudo apt install unzip

↳ sudo unzip awscliv2.zip

↳ sudo ./aws/install

↳ aws --version.

2nd → eksctl install

3rd → kubectl install

Step②:- Create one Role & user & add this Role to Jenkins Server.

Step③:- switch to Jenkins user because deployment will done in this user through a pipeline.

↳ sudo su - jenkins

~~↳ sudo usermod -aG docker jenkins~~

1st:- Create cluster & nodes here through Command.

Step④:- Create one ECR Repository

Step⑤:- Install some plugins in Jenkins Dashboard & add maven,

↳ Docker

↳ DockerPipeline

↳ kubernetes CLI

Perform docker Commands

STEP⑥:- Create credentials connecting to cluster which present in Jenkins user using kube/config file.

↳ GOTO Jenkins user → copy .kube/config file → And save into a local system any dir name → ok

↳ GOTO Jenkins Dashboard → Add credentials → kind (secret file) → select file from local system → Give ID → okay

↳ GOTO Pipeline syntax → Sample step (withKubeConfig: Configure kubernetes CLI) → Select credential Upper creation → Generate Pipeline Script

Jenkins Pipeline Script:-

~~git → git clone stage~~

```
pipeline {
    agent any
    environment {
        registry = "ECRURL"
    }
    stages {
        stage('gitclone'){
            steps {
                Paste gitcheckout URL
            }
        }
        stage('Build') {
            steps {
                echo 'Build the Code',
                sh './mvnw Package'
            }
        }
    }
```

```
stage('Building image') {
  steps {
    script {
      dockerImage = docker.build registry  → refree here
      }
    }
  }
stage('Pushing to ECR') {
  steps {
    script {
      sh 'aws login ECR view Push command'
      sh 'docker push ECRusrname/imagename:latest'
        }
    }
  }
stage('k8s deploy') {
  steps {
    script {
      Past generated pipe Script in 6th step
      sh ("kubectl apply -f ____.yml")
      }
    }
  }
}
```

finally check in Jenkins user, it will accessed.

kubectl get Pods

kubectl get nodes

kubectl get svc