EC2 > OS > docker engine > container > kubernetes

Kubernetes is the most popular container orchestration tool.All the containers we are running in one single docker engine.But what if that docker engine fails? Obviously all the containers running inside that will be down and users will not be able to access them. How about doing high availability on that, having multiple docker engine. we should be doing a clustering of a docker engine if you are running it for production.So not just one single docker

node, multiple docker nodes, but we will also need a master node that is going to control all these docker nodes. The master node will give instruction to the docker node about containers to run.It's going to distribute containers across your docker nodes in case any of the docker node fails. You can run containers on the live docker engines. Or

how about they get started automatically on the docker engine. Containers on the third node failed or the node itself failed, the containers migrated to the healthy node. This itself is called as container orchestration. We will have a master node, which we call it as orchestrator and you will have cluster of docker nodes or worker nodes where orchestrator will distribute the container. So all your docker node will be one

single pool of resource which is fault tolerant. Container orchestration is done mostly for production environment, but you can run normal containers also on them. And today's time we have few orchestration tools in the market, we have docker swarm, it's from docker, we have Kubernetes the most famous one, Mesosphere Marathon from Apache, we have cloud based AWS ECS & AWS EKS (Elastic Kubernetes service), we have Azure

Container service, We have Google container engine, CorOs fleet, OpenShift. So you have cloudbased also. You have inhouse

solution also to build your own orchestration platform. Now, among all this, Kubernetes is the most famous one and other technologies like EKS is actually Kubernetes. And on Mesosphere Marathon you run Kubernetes cluster. So most of the places Kubernetes is used, why? A news from past from Google? Back in 2014, Google announced that everything in Google Gmail or everything runs on Linux containers. And each week we launch more than 2 billion containers. This was in 2014. Now, when I saw this first time when I was learning containers and when I was learning Kubernetes, it was really a jaw dropping news for me. I did not hear it in 2014, I got to know it later. Jaw dropping because not of container, but because of the number of containers, 2 billion containers. But when I use the technology, I understood why is that. Containers are disposable, So any changes you want to make, we replace the containers and imagine Google data center across the world. Of course, there'll be managing billions of containers. Now, That's a big news. Everyone wanted to know how they are doing it. So Kubernetes was created by Google. Actually they created a tool known as Borg to manage Linux container LXC There was no Kubernetes worked back then. But then in 2014, mid 2014, Google introduced

Kubernetes as an open source project, which is just a version of Borg. Then in mid 2015, Kubernetes stable version was released V1.0. And also Google partnered with CNCF Cloud Native Computing Foundation. Now the project is managed by CNCF. CNCF also has certification on Kubernetes and training programs also In 2016, Kubernetes goes mainstream. There are tools that started getting developed for it, like Kops, mini kube. These are tools to create or set up the Kubernetes cluster. Then in late 2016, a case study was released by Pokemon Go, which gave people more confidence on running Kubernetes for production. Then in 2017, the enterprise adoption came in Google. IBM announced istio controller its ingress controls more like application load balancer. GitHub started running on Kubernetes, oracle joined CNCF, and the rest is history. Now, I'm talking as if it happened long, long time back. It's just 2017. It's just three years now. Not even complete three years now. Along with its mature platform, Google is using it. Google was using it for decades now. So,apart from all that Google power, there's so many amazing feature that Kubernetes really provides.

First of all, Kubernetes really not a replacement for docker engine. Kubernetes manages the cluster of docker engine. And not only docker engine, it can manage cluster of other container runtime environment like rocket.

Amazing features with Kubernetes :

1.you have service discovery, load balancing. You create a container, which we

call it a pod here, which gets automatically discovered by the load balancer, and it gets updated in the load balancer. Also we'll see how cool is that?

2.Storage orchestration, kubernetes provide integration with lots of storage, SAN, NAS,

even EBS volume, ceph storage. And there's a huge list that goes on. So people got more confident on running stateful containers.

3.Automated Rollout & Rollback it's very easy to roll out a new image version and also roll back very easily if it's not working. Just like we do in beanstalk, but faster than that.

4.Automatic bin packing. So it's going to place your container on the right note where it gets the right resource based on the requirement. And because of that, your resource is best utilized your computer resource.

5.Self Healing on Node we already discussed orchestration tools. If the node goes down, it brings your containers to life on the live node. Apart from that, your containers are

also monitored. You can set that just like Auto Scaling group. When an instance goes down, Auto scaling group will launch a replacement like that The self healing capability, it's much faster than Auto scaling group.

6.You can manage the configuration in form of variables and volumes and also secrets which are encoded values. There are actually many more things we can go on and on but kind of highlighting cool features of Kubernetes.

Kubernetes architecture :

Two main components, master node and worker node. Worker nodes are the one where docker engine are running.Master node is the one that is managing these worker nodes. So you don't log into the worker node and run the containers.You tell it to the master node. You don't even log into master node. You connect by using some client.You give information to the master node that I want to run so on so containers. And it is going to take the action based on the requirement. Master node is also called as control plane.

Master node have 4 services : ASCE Ascending

API server

Scheduler

controller manager

etcd

Worker node have 3 services : KPD

Kublet

proxy

docker engine.

Master Node :

1.Kube API server:

It handles all the incoming and outgoing communication. This makes the communication possible to and fro Kubernetes cluster. So when you want to send instructions to Kubernetes, kube-API-server is going to receive that. And then it's going to pass the information to other services like scheduler etcd and worker nodes.It exposes Kubernetes API, if you want, you can build your own tool that gets integrated with Kubernetes API. There are so many third party tools to be available to integrate with your Kubernetes API which you can use like monitoring agent, logging agent, web dashboards. It is the front end of the control plane or for the whole Kubernetes cluster, that's the front end.

Being an admin or even DevOps in general admin, we can use kubectl command line interface to connect to the API server. So we should have this kubeCTL installed in our machine. We're going to use that and connect it to the Kubernetes cluster. There's actually a lot of commands and you really need to master the art of Kubectl, if you're managing Kubernetes cluster.There's also web dashboard which you can integrate with the API server. And there are many more integrations.

2.etcd Storage :

etcd is a key value store. It stores the information of your Kubernetes cluster.

The kube API server going to store or retrieve information from this etcd. It will have all the runtime information and it should be backed up regularly. Because if this fails, you lose the current data. You will not know what pod is running, where the containers will be still running, but you'll lose the information.It stores the current state of everything in the cluster.

3.Scheduler: Scheduler is going to schedule your container on the right node. So it's going to watch for the request, when it receives request it will pick up the right worker node and send the information to the worker node saying that, hey, you need to run this container. And there are various factors based on which it is going to decide, like 1.based on the resource requirement 2.or the hardware software or any policy that you have given. Like you have said, I want to run it on a worker node that has XYZ hardware or XYZ software. So those factors will be considered 3.Affinity and anti affinity rules. You can say I want to run my container on this particular node or just the opposite, I don't want to run my container on this particular node. Okay, that is also a factor 4.data locality and 5.Inter-workload interference & deadlines So mostly automatically it will decide,

but you can also give the policy affinity or anti affinity rules.

4.Controller Manager: Actually it's group of multiple things that are running. So to reduce the complexity. We just call it as Controller Manager one single thing, but it actually does multiple things.

Node controller : It's a node controller. It's going to monitor your worker node. If it goes down, it's going to take some actions.

Replication controller : it has Replication controller which is going to monitor your pods. Pods as of now, understand as container. We'll get into what is pods. Think of them as just container for now. So replication controller is going to monitor your containers and if it goes down, it's going to do the auto hailing.

Endpoint controller : It's going to populate the endpoint object which we're going to see there is service object

Service account and Token controller : manages the authentication authorization.

Worker node components :

1.Kubelet : that is the agent. It will be running on every node. And this is going to listen to your Kubernetes master request or commands. So when Scheduler decides that this worker node is going to run the container, it's going to assign the responsibility to kubelet. Now kubelet is going to fetch your image and run the container from it. So it's going to do the heavy lifting. So as we run the commands right, docker run -p -v Now kubelet will be doing it.

2.Kubeproxy: This actually is a network proxy that is going to run on every node in the cluster. You can set network rules also like security group rules, we have allow this or deny that.

3.Container runtime environment : Most important one is Container runtime environment. Now, Kubernetes is quite flexible in this. You can have docker engine in that, you can have container D, you can have cri-o, rktlet Rocket or you can have Kubernetes CRI(Container runtime interface).

So if you go with docker swarm, then you can only use docker engine. But with Kubernetes you can use other runtime environment also.

Addons :

So, along with all these components, if you want, you can do some add ons like DNS, web UI, or container resource monitoring, or cluster level logging. Most of the time these components are taken by some third party vendors who have some specialization in that area, like better logging tool, better monitoring tool, or web user interface, or even DNS service.

Kuberntes Architecture flow :

In master node we have **kubectl** is our tool, which we are going to use to connect to the Kubernetes master node. In master node you have **API server, Scheduler, controller manager, ETCD**. ETCD Stores the current information, API server enables the communication, Scheduler decides where your container will be running on which node, Controller manager responsible for monitoring worker node, your containers, and also the authentication authorization.

In worker node, you have kubelet, which is the agent. Now, don't get confused between kubelet and kubectl. It's quite easy to confuse between them. kubectl is our tool to connect to master node, and kubelet is an agent running in the worker node.

So, kubelet will do all the heavy lifting on the container. It's going to fetch the image, run the containers, map the volumes, do all those stuff.

kubeproxy is a network proxy. If you want to expose a Pod to the outside world, you can do it through Kube proxy, or you can even set the network rules. And then the docker engine, of course, where your containers will be running. But you see the container enclosed in Pod. Okay, Pod, we'll understand what really is this Pod.

What is the relation between Pod and the container? It's the same relation a VM has with the process running inside it. So let's say a Tomcat process is running in a VM. So the VM is going to provide all the resource to the process, which is running network, Ram, CPU, storage, everything, and the process just uses it.

Similarly, Pod is going to provide all the resource to a container. The container will be running inside the Pod. So container will be like the process and Pod will be like the VM. And I'm just giving you this example so you can relate again. There is no virtualization here it's again isolation.

Why does Kubernetes use pod Why not directly run containers?

Well, it's because Kubernetes can use different container runtime environment like docker, rocket, CRI. If you don't have the Pod, there will be no abstraction. Now, we have Pod, it's a standard set of commands, standard set of configuration that we do, it doesn't matter what technology we are using behind the scene. So Pod gives us an abstraction. So we give information to the Pod, what to do, and Pod is going to do it, that for the container which is running inside it. So if you're running a tomcat process in the Pod, the Tomcat will be the container which will be running on port 8080 and the Pod will give the IP address. So you can access it by giving the Pod IP and the port number of the container. We'll see how that works. As of now, you can go with the idea containers are inside the Pod. And in a Pod you can have one or many container. Pod gives the resources to the container.

Insert image kb1.png from desktop

So first you see Pod One, there's a Pod and there's one container running inside that. Second example you see Pod,there's a container and there is a volume attached to it. Third example you see two containers and a volume.Now in this case, both the containers will have access to this volume. So you can have one or many containers running inside the Pod. But should you run multiple containers inside the Pod? Well, it really depends.

Ideally you will see one container running inside the Pod. The other container will be the helper containers.

Insert image kb2.png from desktop

So you see here in node one, you have a Pod and a main container running inside that. So one Pod, one container.

In the second example you see two containers. One is called a sidecar. The other one is init. Now in its container will be short lived container. It's going to start does some command execution and then it will be dead. Then when it is dead, the main container will start with the sidecar container. If you have sidecar container, work will be helping the main container, like for example, streaming the log. So it could be a logging agent or a monitoring agent to help your main container. But at any given point of time, you should have one main container only running in the Pod. The other containers will be helper containers. So my point is, if you have tomcat and MySQL, you are not going to run both in the same pod. You'll run it on different different pods. Containers will be distributed across multiple and now we will use the word Pod.

Pod will be distributed across multiple worker nodes.

Insert image kb3.png from desktop

So let's say you have a Pod One, which is in node One and you have Pod Six, which is in node three, and they want to interact. Maybe Pod One is Tomcat, pod Six is MySQL. So how willthey interact? Well, there is overlay network. Think of this as the VPC that we have seen in AWS. So you have a joint network, a virtual network that connects all the node and every node, you'll have a subnet, like a local area network, or a private network running inside the node. And you see their bridge zero. This will act like a switch. So all the Pod running in this node, one will be able to communicate with each other with the help of this bridge zero. But when it wants to connect to a container or to a Pod running in another node, then bridge zero is going to forward the request to this WG zero. You see, that acts like a router. So it's going to route by looking at the IP address, it's going to route it to the right node router. So it receives by the other router in the node that forwards it to the switch. And then the switch sends it to the Pod. Now I'm using the word switch and router for understanding. So there will be a joint network, virtual network. All your Pod doesn't matter in what node it is, will be in that network. Every node will have a small private network. And all these private network will be connected in one bigger network. This is overly network.

How to Set up a Kubernetes Cluster now?

So Kubernetes Cluster can be set up manually, which is the hardest way. And frankly, you really don't need to do that if you want to do it for understanding and understanding only. If you really want to set up Kubernetes Cluster, there are nice tools to do it.Tools to create kubernetes cluster - Minikube, Kubeadm, kOps

1.Minikube :

You have mini kube to start with, which is for testing and learning purpose. It is just going to set one node Kubernetes Cluster mostly in your computer. So it's going to launch a virtual machine using VirtualBox and on that one VM, the master node and worker node components will be running. Only for testing and learning purpose, not really production.

2.Kubeadm :

kubeadm is a very popular tool to set up Kubernetes Cluster for production. Multi node, Kubernetes cluster. So you can have as many as worker node. Let's say you're saying I need four worker node and one master node. So you need to log into master mode, run some command, get into worker node, run some command, and then they are finally connected together. So you can use any platform if you're using kubeadm, you want to do for EC2. physical machines, virtual machines anywhere. There are a lot of manual steps over here that you need to execute

3.Kops: Multi node kubernetes cluster on AWS

I find this is the most stable way of running Kubernetes Cluster for production. So it came initially only for AWS, but now it has support also for Google Cloud, Digital Ocean, and Open Stack. So if you want to run your own cluster, kubernetes Cluster the most stable way you can use kops.

So we're going to see minikube and kops both way.

We send instructions to kube-api-server using kubectl. kube-api-server pass information to scheduler, etcd & worker nodes.kube-API-server store & retrieve all information on etcd storage.etcd stores current state of everything in cluster

002

1.Steps setup Kubernetes with minikube :

Install Oracle VM

Open powershell as admin

Setup Chocolaty

Install Minikube with Chocolaty

Close powershell & open again & run minikube start

2.Steps setup Kubernetes with minikube :

Purchase Domain for kubernetes DNS records

Create linux VM & setup kops, kubectl, ssh keys, awscli

On AWS setup S3 bucket, IAM user for awscli, Route53 hosted zone

mkdir f:Kubernetes

git clone https://github.com/devopshydclub/vprofile-project.git

cd vprofile-project

git checkout kubernetes

cd minikube

open Minikube-commands.txt

1.Setup kubernetes with Monikube :

Install chocolatey

powershell as admin > Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

Restart powershell with administrator

choco install minikube kubernetes-cli -y

https://minikube.sigs.k8s.io/docs/start/

minikube start : launch VM in virtualbox which executes minikube cluster

minikube --help

minikube start : start local kubernetes cluster

kubectl get nodes :list nodes which contains master node. kubectl uses file cat .kube/config which contains cluster details like name of cluster, users & context.Context marries the user with the cluster.kubectl get nodes uses .kube/config file to access our cluster

Deployments & pods :

STEPS :

We create a Deployment resource that defines how your application should run, including the container image, desired number of replicas, labels etc

IMP : The Deployment controller creates and manages the specified number of Pods based on the Deployment's template.To expose the Pods to the network,we create a Service resource. The Service acts as a load balancer that directs traffic to the Pods behind it. When you create the Service, you use label selectors to target the Pods you want to expose.

Deployment controller > Pods > Service > label selectors

Deployment manages the creation and scaling of Pods. We create a Deployment and that deployment creates and manages the underlying Pods for us.Below command create deployment name hello-minikube & through which pod name hello-minikube-5d9b964bfb-7zqx4 is created :

kubectl create deployment hello-minikube --image=kicbase/echo-server:1.0

kubectl get pod : list of pods along with their current status, names

kubectl get deploy : lists all the deployments currently running in the cluster

To access hello-minikube deployment, expose it as a service by below command :

kubectl expose deployment hello-minikube --type=NodePort --port=8080 : making it accessible from outside the cluster

To get URL of our exposed deployment use below command, copy that URL & paste in it browser :

minikube service hello-minikube --url : check url of deployment/service. We can access our application with this url

kubectl get svc : list of services running in kubernetes cluster along with their current status, names

kubectl delete svc hello-minikube

kubectl get deploy : list of deployments running in kubernetes cluster along with their current status, names

kubectl delete deploy hello-minikube

minikube stop

minikube delete : delete VM also

2.Setup kubernetes with Kops :

1.Purchase domain godaddy

2.Create VM & setup kops, kubectl, ssh keys, awscli

3.AWS : s3 bucket, IAM user for cli, create Route53 hosted zones give entry in godaddy & create NameServer records

Domain on godaddy is subdomain on route 53 :

- Launch EC2 with ubuntu > Tags - Name : Kops > kops-sg SSH MyIP > kops-key

- Create IAM user kops-admin > Programmatic acces > AdministratorAccess > Download csv

- Create S3 for maintaining kops state, so we can run kops command from anywhere > vprofile-kops-state

- Route53 > Create hosted zones > kubevpro.groophy.in (this is going to be our subdomain) > create > copy NS server list > Go to godaddy > Domain settings > ADD > Type : Nameserver > Host : kubevpro > 1st NS entry > Do this for all

4 NS (use kube.cloudwisdom.co.in)

SSH EC2 > ssh-keygen > apt update && apt install awscli -y > aws configure > Enter access & secret key of kops-admin > us-east-1 >

INSTALLING KUBECTL - CLI tool used to interact with Kubernetes clusters

https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-binary-with-curl-on-linux

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

ls : we will see kubectl directory

chmod +x ./kubectl

mv kubectl /usr/local/bin

kubectl --help

INSTALLING KOPS - open-source tool used for managing production Kubernetes clusters

https://kubernetes.io/docs/setup/production-environment/tools/kops/

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

ls : we will see kops-linux-amd64 directory

chmod +x kops-linux-amd64

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

kops --help

nslookup -type=ns kubevpro.groophy.in : you will get 4 nameservers

CREATE CLUSTER WITH KOPS :

#kops create cluster --name=kubevpro.groophy.in \

--state=s3://vprofile-kops-states --zones=us-east-1a,us-east-1b \

--node-count=2 --node-size=t3.small --master-size=t3.medium --dns-zone=kubevpro.groophy.in \

: 2 worker nodes will be distributed in 2 zones us-east-1a,us-east-1b DONT USE t3.micro

--node-volume-size=8 --master-volume-size=8 : imp give volume size for nodes & master Default it takes 120GB

kops create cluster --name=kube.cloudwisdom.co.in \

--state=s3://my-kops --zones=us-east-1a,us-east-1b \

--node-count=2 --node-size=t3.small --master-size=t3.medium --dns-zone=kube.cloudwisdom.co.in \

--node-volume-size=8 --master-volume-size=8

This command does not create cluster, it create configuration for cluster & store it in S3 bucket

#kops update cluster --name kubevpro.groophy.in --state=s3://vprofile-kops-states --yes --admin

kops update cluster --name kube.cloudwisdom.co.in --state=s3://my-kops --yes --admin

EVERYTIME YOU RUN kops COMMAND ALWAYS SPECIFY BUCKET

#kops validate cluster --state=s3://vprofile-kops-states : Your cluster is ready

kops validate cluster --state=s3://my-kops

cat ~./kube/config When we create cluster with kops, kops will create file for kubectl command

This file used by kubectl to connect to cluster

kubectl get nodes : we will see master node with 2 worker nodes. Check from EC2 also

These instances are created by ASG. Check ASG we see 3 ASG 1 for master nodes 2 for worker nodes

We will see VPC created. In Route53 hosted zones we see new records : api.kubevpro.groophy.in points to master node

public IP and api.internal.kubevpro.groophy.in points to master node private IP

kops delete cluster --name=kubevpro.groophy.in --state=s3://vprofile-kops-states --yes

This command deletes cluster, VPC, ASG, DNS records, IAM role.

sudo poweroff : turn off kops VM

004

KUBERNETES OBJECTS :

**Pod** : we manage pods & pod manage container for us

**Service** : to have static endpoint for pod

**Replica set** : create cluster of pods OR replica of same pod

**Deployment** : most used object. We can deploy new image tags using deployment

**Config map** : store our variables & configurations

**Secret** : store our variables & configurations secretly stored

**Volumes** : different volumes attached to pod

GO TO KUBERNETES DOCUMENTATION & search above objects https://kubernetes.io/docs/home/

005

Kubeconfig file : How kubectl connecting to kubernetes cluster.How kubectl know where is master ? Ans Kubeconfig file.

When we create kubernetes cluster using kops we get file Kubeconfig. This file contains Clusters : cluster information, Users : users used to access the cluster,Namespaces, Authentication mechanism

When we do ssh we need IP, username,password like that kubectl needs cluster information, user information, authenticate mechanism & namespace.

Start kops vm >

less ~/.kube/config : It contains clusters info, users info, context & current context information

apiVersion: v1

1.clusters:

- cluster:

1.certificate-authority-data: certificate for this cluster

2.server: https://api.kube.cloudwisdom.co.in

"url of master node API server.API server lives in the master node & kubectl connects to the master node"

3.name: name of the cluster

2.contexts:

- context:

"context contains user login info & clusters information & authentication it basically tells for this cluster use this user

cluster: <cluster\_name> kube.cloudwisdom.co.in

user: <user\_name> kube.cloudwisdom.co.in

name: <context\_name> kube.cloudwisdom.co.in

3.current-context: <context\_name> kube.cloudwisdom.co.in

"kubectl command by default uses this context, we can change the context while running kubectl command"

kind: Config

preferences: {}

4.users: kube.cloudwisdom.co.in

- name: <user\_name>

user:

client-certificate-data: <client\_certificate\_data> certificate for the user

client-key-data: <client\_key\_data> key to authenticate user

kubectl config use-context : use current context from config file

kubectl config view : certificate are hidden here

We can use following arguments with kubectl command :

--kubeconfig : to specify another location for kubeconfig file

--context : we can mention more than one context

--user : we can use another user for cluster

--cluster : we can use another cluster

If your cluster is behind proxy server we can mention it.

mkdir ~/.kube in windows home directory

vim ~/.kube/config > paste all config file in this file from kops VM

kubectl get nodes : we can use kubectl from our windows now

If we have kube/config file we can run kubectl command from anywhere.

We can use this kubeconfig file for deployment in jenkins, also we can use it in ansible.

006

**NAMESPACES :** **Grouping & isolating our resources within single cluster**

Defn : provides mechanism of isolating groups of resources within single cluster.

Names of namespaces needs to be unique within namespace but not across the namepsaces.

K8 clusters can have multiple namespaces. These **namespaces used to set securities, quotas to our resources**.When we create the cluster it creates 3 namespaces automatically **default, kube-system & kube-public**.We can also create different namespaces like for developement & production.Also we can create different namespace for projects.

You can create sperate namespaces for developement & production enviornments

when you **delete namespace** **created by you, it will delete everything all resources** in one command

**kubectl get ns** : list existing namespace

ALL THESE COMMANDS ON Windows (Git bash)

**kubectl get ns** : list all namespaces

**kubectl get all** : list resources from default namespace, Pods & Services

**kubectl get all --all-namespaces** : list all the resources from all the namespaces, In kube-system namespace you have all master/control plane resources & they run as a pod. It also contains pods, replicaset, deployments, daemonset, service.

**kubectl get svc -n kube-system** : list all the resources from kube-system namespace , namespace > pod

**\*kubectl create ns kubekart** : create namespace name kubekart

**\*kubectl run <container\_name> --image=<contai ner\_image> -n <namespace\_name>**

**kubectl run nginx --image=nginx -n kubekart** : **we can create pod with same name but with different namespace.**If we run this command again it shows nginx already exists, but if we specify different namespace instead of kubekart it will run the command.

**Specify namespace in definition file** : pod.yml > metadata > namespace: kubekart

**appVersion**, **kind**, **metadata** (name, namespace), **spec**(containers(name, image, ports(containerPort)))

Login to kops VM :

vim pod1.yaml

apiVersion: v1

kind: Pod

metadata:

name: nginx12

**namespace: kubekart**

spec:

containers:

-name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**kubectl apply -f pod1.yml** : create resource here pod is created from pod1.yaml

**kubectl get pod -n kubekart** : list pod from kubekart namespace

**kubectl delete ns kubekart** : delete all resources from namespace & the namespace

we can specify namespace in kubeconfig(/.kube/config) file, everytime we use kubectl command it uses that namespace.

**kubectl config current-context : Copy context-name i.e cluster name & paste in below command**

**kubectl config set-context <cluster-name> --namespace=<namespace-name>**

**kubectl config set-context my-cluster --namespace=my-namespace**

by default it uses default namespace if you do not specify namspace in the kubectl command

**kubectl config set-context --current --namespace=<namespace-name>**

006

**PODS :** Container runs inside pod. It represents process running in your cluster

**1.Pods with single container :**

1.One container per pod model - To use multi-container in pod use other container as helper container(sidecar or init container or both)

2.Pod is a wrapper around the container

3.Kubernetes manages pods rather than managing containers directly. we are going to give commands for the pod

**2.Multi-container pod :**

If you want to run multiple container in pod they must be tightly coupled & they need to share same resources

One main container & other **helper containers like sidecar or init container or both**

Use single pod for each service or application - Create pod for every application like tomcat, mysql, rabbitMQ.

For high availability scale multiple pods horizontally. e,g multiple tomcat pods horizontally scaled

We can create pod : 1.direct pod run on kubernetes cluster

2.pod run by definition file pod-setup.yml (best way)

e.g pod-setup.yml

**kubectl api-resources | grep pod : lists version, shortname for pod resource**

**kubectl explain pod : gives all information about pod resource**

apiVersion: v1 / v1 / apps/v1 / networking.../v1beta1 **(type is string)**

kind: Pod / Service / Deployment / Ingress (**(type is string)**above are versions for kind check documentation also)

metadata: **(metadata type - dictionary)** **information about the pod / resource**

name: webapp-pod

labels:

app: frontend

project: infinity

spec:

containers: **(type is dictionary & value is in list format coz we can have multiple containers)**

- name: httpd-container

image: httpd

ports:

- name: http-port

containerPort: 80

Login to kOps VM :

Create cluster in Kops VM

**kops create cluster --name=**kube.cloudwisdom.co.in \

**--state=**s3://mybucket-kube **--zones=**us-east-1a,us-east-1b \

**--node-count=**2 **--node-size=**t2.micro **--master-size=**t2.micro **--dns-zone=**kube.cloudwisdom.co.in \

(**--node-volume-size=**8 **--master-volume-size=**8)

**kops update cluster --name** kube.cloudwisdom.co.in **--yes --state=**s3://mybucket-kube

**kops validate cluster --name** kube.cloudwisdom.co.in **--state=**s3://mybucket-kube : Your cluster is ready

**kubectl get nodes** : kubectl uses kubeconfig file to get connected to API server of kubernetes cluster

**ls –a : we see .kube directory which contains config file**

**cat .kube/config** : **kubectl use this file information to connect to kubernetes cluster**

**This file has information like cluste info, url of cluster, cluster name, username, key, certificate, context**

To use kubectl from windows powershell copy this file & put It in home directory

**Kubectl get nodes : copt node name from here**

**kubectl describe node** <nodename\_from\_kubectl\_get\_nodes>

**kubectl describe node** <nodename\_from\_kubectl\_get\_nodes> **-o yaml** : in yaml format good for troubleshooting

**kubeclt edit pod <pod-name> : edit only few things**

**RUN POD THROUGH DEFINITION FILE :**

**mkdir definitions && cd definitions**

**mkdir pod && cd pod**

vim vproapppod.yaml **Creating tomcat pod from dockerhub image**

---

apiVersion: v1

kind: Pod

metadata:

name: vproapp

labels:

app: vproapp

spec:

containers:

- name: appcontainer

image: imranvisualpath/freshtomapp:v7 <url of dockerhub image with tag> (from containerization project)

ports:

- name: vproapp-port

containerPort: 8080

**vproapp is pod name from metadata & appcontainer is name of container from spec**

**kubectl create -f vproapppod.yaml**

**kubectl get pod** : 1/1 means how\_many\_containers\_running/ how\_many\_you\_containers\_specified

**kubectl describe pod** vproapp : **Events** > Scheduled(default scheduler - pod running on which node) > pulling > pulled > created > started Container : MUST CHECK EVENTS, IP : ip address of pod, container id, port after running this command

008 DIFFERENT LEVELS OF LOGGINGS : **Finding issues with our pod (Troubleshooting)**.Always check things in local environment then in the test or dev environment & then finally in production environment

**1.Error in pulling image :** [solved by **kubectl describe nginx12** command]

vim pod2.yaml

apiVersion: v1

kind: Pod

metadata:

name: nginx12

spec:

containers:

- name: nginx

image: **nginox:1.14.2 “MISTAKE HERE”**

ports:

- containerPort: 80

**kubectl apply -f pod2.yaml**

**kubectl get pod : if status of container is ImagePullBackOff or CrashLoopBackOff**

**TRY ALL BELOW COMMANDD FOR TROUBLESHOOTING : find errors**

**kubectl get pod -o wide**

**kubectl get pod nginx12 -o yaml** : check containerStatuses > state > waiting > message : Back-off pulling image "nginox:1.14.2" here we need nginx instead of nginox - Back-off means error

**kubectl describe pod nginx12** : check out events which has errors > in error message - pull access denied repository does not exist or require authorization, this error message come for private registry also if we did not gave credentials. The error is due to image: **nginox:1.14.2 we made mistake here**

**kubectl get pod**

**kubectl delete pod nginx12**

Edit file pod2.yaml & add nginx insted of nginox in image: **nginox:1.14.2**

**kubectl apply -f pod2.yaml**

**kubectl get pod : now we see pod is running status**

2.**Error create (run) container command** with command inside container [solved by **kubectl logs web2** command]

**kubectl run web2 --image=nginx test47 : create container & run test47 command inside container but test47 is not command THAT’S THE ERROR**

**kubectl run web2 --image=nginx test47** : run test47 on nginx container which is wrong

**kubectl get pod**  : web2 CrashLoopBackOff Restarts 8 times

**kubectl get pod web2 -o yaml** : does not give anything

**kubectl describe pod web2** : in events warning Back-off restarting failed container

**kubectl logs web2** : /docker-entrypoint.sh: 38 exec test47: not found

pod is running in container & container execute commands, commands could be a script it could be a command that run the process like nginx. Whatever process runs in container, the output of that process we can see using logs

/docker-entrypoint.sh: 38 exec test47: not found means executing command test47 failed

**history | grep test47**

**kubectl run web2 --image=nginx test47**

**kubectl delete pod web2**

**kubectl run web2 --image=nginx**

**kubectl get pod** : we will see web2 is running

**PROCESS TROUBLESHOOTING POD :**

1.Check status from **kubectl get pod** if any error

2.**kubectl describe pod web2** check EVENTS from here

3.**kubectl get pod web2 -o yaml**

4.**kubectl logs web2**

5.Delete pod & recreate it Coz we cant edit the pod

009

**SERVICE :**

If you want to **expose your pod or the application running inside pod as a network service**, we use service with your pod.**Service provides static endpoint to the pod**.Service is similar to load balancer.To expose the pod & establish communication between pods we use service in front of your pod.

Why service NOT simple port mapping ?

Like containers pods are disposable.If we want to change a pod we need to create a different pod. Pods are mortal & they dont have static IP address, so we need something static like and endpoint (behind the endpoint we can change things). Service gives that static endpoint like ELB does to EC2 instances it gives static endpoints so we can access load balancer. Behind the load balancer we can keep deleting & creating EC2 instances. Similar service does it for pods in kubernetes.

**3 Types of Service :**

**1.NodePort 2.ClusterIP 3.Load balancer** Insert image 1 2 3 from F:\Notepad++\_Images

NodePort is simillar to port mapping in docker(host port mapping with container port). NodePort is not for Production purpose.It exposes pod to outside network

ClusterIP used when we don’t want to expose pod to outside network but **for internal network like Tomcat connecting MySQL.It is for internal communication**

Loadbalancer to expose pod to outside network for production used cases.

**Every pod has its service to communicate with other pods.**

Let's say we have two pods running in a node. Blue color square or rectangle is our worker node. And you have two pods running over here. Pod will have label and IP address. And of course, the container running inside that. The nginx container is running on both the pods, one container per Pod and it's exposed on port 80. Pod has an IP address and a port number to access and it has a label. Service will be like a load balancer, in front of pods. Service will have a static IP address, which will not change until unless you don't delete it. It will have a frontend port like a load balancer has a frontend port, this is internal frontend port for internal communication. And it has back end port which will be the port number of the container. So Service like a load balancer, has a frontend port and a back end port. But **how does it know to which Pod it has to route the request to?** There could be hundreds of pods running like this. Well, **it is going to match the label selector**. **When we create a service, we give a label name** that is label selector here App:Frontend. Any Pod that has this label App:Frontend, service is going to forward the request to that Pod on port 80. So I have two part. If I run exactly third part, exactly same with same label, my third part will be automatically included under the Service auto discovery.

**When we run pod with label selector that pod will automatically runs under that service**

When you say node port, you have to mention a node port number. So let's say 30,001. So a node port or host port 30,001 will be mapped to your service.

**When you access the node by giving its IP address and the node port, the request will be forwarded to the Service. The Service is going to forward the request to the Pod and Pod will send it to the container.** And the same way it comes back.

There are multiple port over here. 30,001 is the node port for the communication to outside network. It sends it to the Service. Service has an internal frontend port 80(upper). It's going to send the request on the back end port 80, which is the container port.So your **node port and front-end port of the service can be same or can be different.But** **back-end port of the service and the container port should be exactly same**. **And label selector should match with the label of the Pod.**

When you're creating a service, two things are very important : **1.Matching the label selector** **2.Back-end port of the service & container port must be same**

Sample definition file :

apiVersion:

kind: Service

metadata:

name: webapp-service

spec:

type: **NodePort : service of type nodeport**

ports:

* targetPort: 80 **Back-end port of service where pods interact**

port: 80 **internal** **front-end port of service we cant access it**

nodePort: 30005 **for outside network**

**“if we access node on nodePort 30005 then its going to forward the request to any pod that has label app: frontend in selector”**

protocol: TCP

**selector:**

**app: frontend**

**IMP : container > pod > worker node(nodePort) > service (targetPort (Back-end), port(Front-end)) >**

kind service API version v1, metadata name spec type Nodeport. You are specifically saying that I want to create a service of type node port. And then you give the port number target port that is the back end port 80. Port 80 means it's a frontend port, but internal frontend port. You cannot access it from the outside network. Node port is for the outside network. So if I access this node on port 30,005, it's going to forward the request to any Pod that has this label. Okay, app frontend on port 80. So if you know your Pod, you know its label, you know sorry, it's container port. Then you can very easily write a service definition file.

Similar to port mapping in docker, host port is mapped with container port. Used for non prod purpose to expose your pod to the outside network.

cd definitions/pod & cd ..

mv pod app

cd app

cat vproapppod.yaml > we need **label & port number**

**vi vproapppod.yaml**

---

apiVersion: v1

kind: Pod

metadata:

name: vproapp

**labels:**

**app: vproapp**

spec:

containers:

- name: appcontainer

image: imranvisualpath/freshtomapp:v7 <url of dockerhub image with tag>

ports:

- name: vproapp-port

**containerPort: 8080**

**vi vproapp-nodeport.yaml**

apiVersion: v1

kind: Service

metadata:

name: helloworld-service

spec:

ports:

- port: 8090 **internal service port**

nodeport: 30001 **external front end port for outside communication its value start from 30000**

targetPort: vproapp-port : **8080 which is from vproapppod.yaml as we have given name to it we can use it here**

**8080 used for pods communication with service**

protocol: TCP

selector:

app: vproapp : **from vproapppod.yaml**

type: NodePort

**kubectl create -f vproapp-nodeport.yaml**

**kubectl get svc : we see helloworld-service Cluster IP PORTS 8090:30001**

**(8090:30001 internalFrontEndPortofService:externalFrontEndPort)**

**kubectl describe svc helloworld-service** : IP is static IP of service & Endpoints is IP address of pod which is automatically mapped

**kubectl get pod**

**kubectl describe pod | grep IP** : we will see **pod IP mapped at port 8080** with endpoint we just mentioned the label selector

change SG of worker node : All traffic MyIP > copy public IP of worker node > in browser publicIP\_worker:30001

Our tomcat application of vprofile is running on kubernetes cluster through node port

**kubectl delete svc helloworld-service**

2. ClusterIP : **If we dont want to expose a pod to outside network but for internal communication like Tomcat connecting to MySQL.ClusterIP dont a have nodePort.Mostly used for front-end & back-end service interating with each other.Nginx interacting with backend service tomcat.**MySQL needs a static endpoint so we can create ClusterIP service. **No port mapping here.**

**Nginx pod > backend > Tomcat pod > databases**

Sample tom-svc-clusterip.yaml **We just give targetPort & internal frontend port for service No nodePort No loadbalcer.Internally if any other pod wants to interact with Tomcat pod its gonna access the port 8080**

apiVersion: v1

kind: Service

metadata:

name: app-service

spec:

type: ClusterIP

ports:

- targetPort: 8080

Port: 8080

protocol: TCP

selector:

app: backend

**Sample tom-app.yaml**

apiVersion: v1

kind: Pod

metadata:

name: app-pod

labels:

app: backend

project: infinity

spec:

containers:

- name: tomcat-container

image: tomcat

ports:

- name: app-port

containerPort: 8080

label “app: backend” from tom-svc-clusterip.yaml used in tom-app.yaml

Insert image 6 7 8 from F:\Notepad++\_Images

Cluster IP is really without any external port. No node port will be there. It will be mostly for the frontend service to refer to the back end service or back end service interacting with each other, they will need service of type cluster IP. So for nginx referring back to the Tomcat Pod, you need to create a service of type cluster IP.

You just mentioned type cluster IP, instead of node port or load balancer, just cluster IP and you give the target port and the frontend port. That's all. No node port, no load balancer.

Now internally, if any other Pod wants to interact with your Tomcat Pod, it's going to access the service on port 8080. Okay? So on the frontend, you can have type of node port or load balancer in the back end services you'll have service of type cluster IP and load balancer for production use cases

**3. LoadBalancer : loadbalancer > nodePort > Service > pods**

Used for production to expose pod to outside network. E.g Users from internet to access Tomcat pod we need to create service of type load balancer. It actually create load balancer in AWS & map our pod to it.

For every pod or cluster of pod we need service infront of pod if its providing a network service.

If we did not specify nodePort it will pick random nodePort

**Load balancer routes the request to the service & service forward that request to the pod. Service is across the cluster its not on a pod or container. It is like rule that get created across all the nodes.**

Insert image 4 5 from F:\Notepad++\_Images

You access the load balancer. It's going to route the request to any of the worker node on the node port.And we did not specify the node port. So it's going to pick up a random node port. It has a range. It's going to pick up from that random port number and assign it to the worker node. Okay?

Like that load balancer, we access it throughout the request to the internal service and that's going to forward the request to your Pod.

Now, service is across your cluster. It's not on a worker node. It's not a Pod, it's not a container. Service is rule or some rules, proxy rules. So it gets created across all your nodes. Okay? All the worker node, even the master node will have the rules that if the request comes on, this port is going to route it to the right Pod.

**cp vproapp-node-port.yaml vproapp-loadbalancer.yaml**

**vim vproapp-loadbalancer.yaml**

apiVersion: v1

kind: Service

metadata:

name: helloworld-service

spec:

ports:

- port: 80

targetPort: vproapp-port

protocol: TCP

selector:

app: vproapp

**type: LoadBalancer**

nodeport will be created automatically, port 80 is of loadbalancer

**kubectl create -f vproapp-loadbalancer.yaml**

**kubectl get svc** : under EXTERNAL-IP we see endpoint of loadbalancer

AWS > EC2 > load balancer > instances > we see 2 worker nodes

AWS > EC2 > load balancer > Description : above 2 worker nodes are mapped on Ports where frontend port is 80 & 32654 is nodeport Instances

AWS > EC2 > load balancer > Description > **copy DNS name & paste it in browser**

**kubectl get pod**

**kubectl get all**

**kubectl delete pod/vproapp**

**kubectl delete pod/helloworld-service** : it will delete loadbalancer also

**kubectl get all**

**Replicaset :** Insert image 10 11 12 13 from F:\Notepad++\_Images

**To maintain replica of our pods for high availability we use replicaset.It ensures a specified number of replica pods are running all the time.If any of pod goes down replicaset will recreate it on healthy node.It also ensures high availability and fault tolerance for application.** **If replicas option set to 3 it will create & maintain 3 pods always.**

ReplicaSet maintains a replica of your pod. What does that mean? And why do we really need it?

Okay, so you have a pod running on a node and users are accessing it. let's say the pod is running, some web app lication. And for some reason the pod goes down, the users won't be able to access the service. And that's it, end of the story. Someone needs to login, delete the pod, recreate it, fix the problem.

But if the pod is running with ReplicaSet.In ReplicaSet object you mention, these are the pods, i want to run. These many Replicas I want to run off this pod. ReplicaSet will do that for you. **Any pod crashes, it can recreate a new pod for you. And you can mention Replicas for scaling.** You can add more pods into the ReplicaSet. **You can remove pods from the ReplicaSet.**

And when I say Replica, it doesn't really need to be really Replicas. You can just say one also. Replica One, you will see. If I have to mention one, why will I use ReplicaSet? Well, for the health checks. If your pod goes down, you don't need to manually recreate it. ReplicaSet will do it for you automatically. And the best part is mentioning more replicas scheduler will distribute your pods across multiple worker nodes. So in an event of the entire node crashing, pods running on that node can get recreated on healthy worker nodes.If you have pod running on a node without ReplicaSet, just you created a pod and that node goes down. Pod goes down. That's it. End of the story. But if it is created with a Replica Set, it will recreate that pod for you.

Definition file : login to kOps VM > vi replset.yaml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: frontend

labels: #we use lables for filtering

app: guestbook

tier: frontend

spec:

# modify replicas according to your case

replicas: 3 #number of replica we need

selector:

matchLabels:

tier: frontend

template: # pod information

metadata:

labels:

tier: frontend

spec:

containers:

- name: php-redis

image: gcr.io/google\_samples/gb-frontend:v3

**Matchlabels & labels are same here “tier: frontend” that means if you have existing pod running with the same label.**

**matchLabels helps the ReplicaSet “find and manage existing Pods with specific labels”, while the labels in the template dictate “what labels new Pods should get when they're created”.**

**If there are fewer than 3 Pods with the label tier: frontend, the ReplicaSet will create new Pods until it reaches the desired count of 3.**

**If there are more than 3 Pods with the label tier: frontend, the ReplicaSet will not delete any of them. It only ensures that there are at least 3 replicas.**

**Top of Form**

**kubectl create -f replset.yaml**

**kubectl get rs** : DESIRED is replicaset we want, CURRENT means how many pods got created, READY out of those pods how many are ready

**kubectl get pod**  : we see 3 pods here, 1/1 means 1 container running out of 1 container

**kubectl delete pod <name of pod2> <name of pod3>** : take names from above command

kubectl get pod : **even if we deleted 2 pods, replicaset create it again quickly**

**Scale up & down replicaset :**

1st method : Editing definition file

vim replset.yaml > replicas > 5 : this scales up/out replicaset

**kubectl apply -f replset.yaml**

**kubectl get pod** : we will see 5 pods

vim replset.yaml > replicas > 2 : this scales down replicaset

**kubectl apply -f replset.yaml**

**kubectl get pod** : we will see 2 pods

2nd method : by command - NOT RECOMMEDED FOR PRODUCTION

scale down/in replicaset command :

**kubectl scale --replicas=1 rs/frontend** : frontend is replicaset name

**kubectl get pod** : we will see 1 pod

scale up replicaset command :

**kubectl scale --replicas=5 rs/frontend** : frontend is replicaset name

OR

scale up/down replicaset command :

**kubectl edit rs frontend** > replicas : 2 > wq

**kubectl get pod** : we will see 2 pods

**IMP : ALWAYS USE EDITING DEFINITION FILE METHOD 1 IN PRODUCTION TO SCALE UP/DOWN REPLICASET DON’T USE COMMANDS**

kubectl cheat sheet :

https://kubernetes.io/docs/reference/kubectl/cheatsheet/ always use this documentation

**kubectl delete rs** : dont delete pod coz they gets recreated again delete replicaset

**kubectl get pod**

011

**DEPLOYMENT : Most used object by DevOps**

**Deployment manages the creation, scaling, and updating of Pods and ReplicaSets. You define your application's desired state in the Deployment file, and Kubernetes takes care of creating and managing the Pods and ReplicaSets based on that specification.(For most use cases, you don't need separate Pod or ReplicaSet definition files when using Deployments)**

**Deployment controller provides declarative (by writing deinition files) updates for pods & replicasets.**

**We define the desired state in the deployment** & **the deployment controller will change actual state(older image tag) to the desired state(newer image tag) at a controlled rate**(in rolling update type-one at a time).

**Deployment creates replica set to manage number of pods**

insert image 14 15 16

Currently image tag is v1 & we mentioned in deployment we want to upgrade it to v2 tag coz we have new tag v2 in our registry. Deployment will upgrade it one by one. And if something goes wrong it rollbacks it. It works like ASG in AWS.

Login kOps vm > vi deployment.yaml

apiVersion: apps/v1

kind: **Deployment**

metadata: **data about deployment name labels etc**

name: nginx-deployment

labels:

app: nginx

spec: **information about replicaset**

replicas: 3

selector:

matchLabels:

app: nginx # label of pod

template: # **pod information**

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**Deployment (metadata: name, labels) > replicaset (spec: replicas, selector(matchLabels)) > pods (template: metadata(labels), spec(container: name, image, ports (containerPort)))**

**kubectl applu -f deployment.yaml**

**kubectl get deploy** : we see deployment name as nginx-deployment

**kubectl describe deployment <deployment-name>**

**kubectl get rs** : we will see 3 replicaset for 3 pods

**kubectl get pod** : we will see 3 pods

**kubectl describe pod <pod\_name>** : take pod name from above, container > nginx > Image

**Update deployment: Best way change deployment files**

kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1

kubectl set image deployment.v1.apps/<deployment\_name> <container\_name>:<image\_with\_new\_tag>

BEST WAY TO CHANGE DEPLOYMENT FILE deployment.yaml

kubectl get deploy

kubectl get pod

kubectl describe pod <takePodNameFromAboveCommand> : see container > nginx > Image version changed 1.16.1

kubectl get rs : **we see 2 replicasets** when we created deployment it created 1st replicaset & 2nd replicaset is created when we updated deployment. It is done by rolling update (one by one pod)

kubectl rollout status deployment/nginx-deployment

**ROLLING BACK DEPLOYMENT :**

**kubectl get deployments -n <namespace>**

**Kubectl get deployments**

**1.Restart deployment : To restart all Pods controlled by a Deployment**

**kubectl rollout restart deployment <deployment-name>**

**2.Pause and Resume a Deployment : You can pause a Deployment to prevent further rolling updates. This is useful if you need to investigate issues or temporarily halt updates.**

**kubectl rollout pause deployment <deployment-name>**

**kubectl rollout resume deployment <deployment-name>**

**3.History of Rollouts: list of revisions and allows you to see when each revision was deployed and its status.**

**kubectl rollout history deployment <deployment-name>**

**4.Undo a Rollout (Rollback): If an update causes issues, you can roll back to a previous revision**

**kubectl rollout undo deployment/deployment-name : roleback to previos deployment**

**kubectl rollout undo deployment <deployment-name> --to-revision=<revision-number>**

**We get revision number from history command**

kubectl get rs : we see new replicaset gone to zero & old replicaset got 3 new pods

kubectl get pod

kubectl describe pod <takePodNameFromAboveCommand> | grep Image : we see image version changed to 1.14.2

kubectl rollout history deployment/nginx-deployment : we see revision numbers

rollout with revision number :

kubectl rollout undo deployment/nginx-deployment --to-revision=2

Scale up/down deployment :

kubectl scale deployment/nginx-deployment --replica=10

kubectl get deploy

kubectl delete deploy nginx-deployment

WE SHOULD DO EVERYTHING THROGH DEFINITION FILE IN PRODUCTION

012

**COMMANDS & ARGUMENTS : pass commands & arguments to a container in the pod**. pod does not execute command, container which is inside the pod execute the commands

In dockerfile Entrypoint has higher priority over CMD. Entrypoint contains commands & CMD contains arguments.

FROM ubuntu : docker run printer - **will output hi**

CMD ["echo hi"]

FROM ubuntu : docker run printer hi - **we need to pass the argument**

ENTRPOINT ["echo"]

FROM ubuntu : docker run printer - if we didnt pass argument it prints hi

ENTRPOINT [echo]

CMD [“hi”] : docker run printer but if we pass hello as a argument it will print hello by CMD["hi"] overriding hi

**We mention commands & arguments in pod definition file but actually container runs the commands NOT the pod**

Login to kOps VM :

vim coms&args.yaml

apiVersion: v1

kind: Pod

metadata:

name: command-demo

labels:

purpose: demonstrate-command

spec:

containers:

- name: command-demo-container

image: debian

**command: ["printenv"]**

**args: ["HOSTNAME", "KUBERNETES\_PORT"]**

restartPolicy: OnFailure

**kubectl apply -f coms&args.yaml**

**kubectl get pod** : we see status as completed NOT running, container is dead here, it just executed printenv & destroyed automatically. **"Completed" status** of a Pod indicates that the **main container within the Pod has completed its execution and terminated.We need to manually delete these containers**

**kubectl logs <pastePodNameFromAboveCommand>** : it shows hostname of container & kubernetes port

These kind of containers used when we want to run something which return some output

env:

- name: MESSAGE

value: "hello world"

command: ["/bin/echo"]

args: ["$(MESSAGE)"]

TRY THIS IN ABOVE FILE

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-deployment

spec:

replicas: 1

template:

metadata:

labels:

app: my-app

spec:

containers:

- name: my-container

image: my-image

env:

- name: MESSAGE

value: "hello world"

command: ["/bin/echo"]

args: ["$(MESSAGE)"]

**VOLUMES : Map volume to a pod**

**Google persistant volume topic & backend storage**

Insert image

Kubernetes supports EBS, azurdisk, cephfs, cinder, fibre channel, gcepersistantdisk(google cloud), flocker, glusterfs(redhat), iscsi, local, nfs, portworxvolume, vpsherevolume(vmware), hostpath

**We are doing hostpath.** **We pick up a directory on a worker node & we map that as a volume to pod**

**hostPath allows a container running inside Pod to access files or directories on the physical or virtual machine.**

Example file :

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: registry.k8s.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-pd **# where do we want to mount the volume in the container**

name: test-volume **# it is going to mount a volume name as test-volume in the container at the path /test-pd**

volumes: # host machine

- name: test-volume # test-volume coming from

hostPath: # test-volume is of type hostpath

# directory location on host

path: /data # on worker node it picks path /data

# this field is optional

type: Directory

**test-volume is mapped to the /data directory on the host machine (worker node created from kops), and its contents can be accessed within the container at the mount path /test-pd**

whatever data container stores at “/test-pd” this directory will get automatically stored in /data

Best pratice : volume created seperately & then in pod definition file it will be mounted

Login kops VM :

vi mysqlpod.yaml

apiVersion: v1

kind: Pod

metadata:

name: **dbpod**

spec:

containers:

- image: mysql:5.7

name: mysql

volumeMounts: **# for container path**

- mountPath: /var/lib/mysql **# mysql container store all data in /var/lib/mysql so we're mounting here**

name: dbvol **# volume name**

volumes: **# for worker node path**

- name: dbvol

hostPath: # test-volume is of type hostpath

# directory location on host

path: /data **# we are mapping /var/lib/mysql from container to /data on worker node.**

type: Directory / (DirectoryOrCreate) **# if directory doesn’t exists on worker node create it**

**kubectl apply -f mysqlpod.yaml**

**kubectl get pod**

**kubectl describe pod dbpod** : In **Events** we see error host path type check failed /data. /data would be an external storage mapped to your worker node.We change type in hostpath to DirectoryOrCreate in mysqlpod.yaml. DirectoryOrCreate means if directory does not exists create it

kubectl delete pod

**kubectl apply -f mysqlpod.yaml**

kubectl describe pod dbpod : We see in Mounts /var/lib/mysql is coming from dbvol /data from Volumes section

**Whatever data container stores in /var/lib/mysql is going to come to the worker node in /data.** **Even if pod is deleted we will have our data saved in /data directory**

**THIS IS NOT FOR PRODUCTION. IN PRODUCTION WE HAVE PROPER VOLUME OF STORAGE MECHANISM SET & WE HAVE TO MAP THAT STORAGE VOLUME TO THE POD.**

**kubectl delete pod dbpod**

013

**Enviornment variables & CONFIG MAP : How to inject data in the pod ?**

**Config maps used to store all your variables & configuration at one place & whenever we want we can injet these variables & configuration files.**

Pods are disposable. If we want to make change we have to delete them & create a new pod with a new container image. But we can inject variable & configuration in the pod while its running.

containers:

env:

- name: MYSQL\_DATABASE

value: accounts

- name: MYSQL\_ROOT\_PASSWORD

value: admin123

Enviornment variables :

We add variables in env section of definition file with name value pair.For eg name MYSQL\_DATABASE & value accounts, MYSQL\_ROOT\_PASSWORD & value admin123. Definition file exports these variables in the container.

We can use configMaps to store variables & configurations at one place. We can later inject these variables & configuration files.

**Set variables in configMap:**

1.Imperative way : NOT BEST WAY

**Kubectl create configmap <config\_map\_name> --from-literal=<variable\_name>=<value> …**

kubectl create configmap db-config --from-literal=DB\_HOST=example.com --from-literal=DB\_PORT=5432 --from-literal=DB\_USER=myuser --from-literal=DB\_PASSWORD=mypassword

kubectl create configmap db-config --from-literal=MYSQL\_DATABASE=accounts --from-literal=MYSQL\_ROOT\_PASSWORD=admin123 configmap/db-config created

**kubectl get cm : see configmaps**

**kubectl get cm db-config -o yaml** : in yaml format. We see 2 keys MYSQL\_DATABASE & MYSQL\_ROOT\_PASSWORD

**kubectl describe cm db-config**

**2.Declarative way : BEST WAY**

**Vi configmap.yaml**

apiVersion: v1

kind: **ConfigMap**

metadata:

name: data-config

data:

MYSQL\_ROOT\_PASSWORD: aadmin123

MYSQL\_DATABASE=accounts

**We have 2 variables stored in configmap MYSQL\_ROOT\_PASSWORD & MYSQL\_DATABASE**

**Kubectl create –f configmap.yaml**

**We have variables stored in configmap How do you inject in it a pod ?**

**1.Export all variables in container:**

**envfrom:**

* **configMapRef: #all variables from db-config will be exported into this container**

**name: db-config**

**2.Export selected variable only in the container :**

**Env:**

* **name: DB\_HOST # DB\_HOST varible to store value of key DB\_HOST (below) from configMap db-config**

**valueFrom:**

**configMapKeyRef:**

**name: db\_config # name of configMap**

**key: DB\_HOST # key of variable that we want to export from configMap db-config into the container**

vi samplecm.yaml:

apiVersion: v1

kind: **ConfigMap**

metadata:

name: **game-demo**

**data:**

# property-like keys; each key maps to a simple value

player\_initial\_lives: "3" #key:value

ui\_properties\_file\_name: "user-interface.properties" #key:value

# file-like keys

game.properties: | # **game.properties is key here & value is some multiline content.** We can

enemy.types=aliens,monsters #store this multiline content as a file in the container

player.maximum-lives=5

user-interface.properties: |

color.good=purple

color.bad=yellow

allow.textmode=true

**ConfigMap has 4 keys in above example player\_initial\_lives, ui\_properties\_file\_name, game.properties, user-interface.properties**

**kubectl apply -f samplecm.yaml**

**kubectl get cm**

**kubectl get cm game-demo -o yaml** : we see 4 keys in data section

**4 ways to get this configMap data into the container**

1.Inside container commands & args

2.Environment variables for a container

3.Add a file in read-only volume, for the application to read

**4.Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap**

**IMP : Injecting configuration as a volume : To put configuration inside container**

**Steps : 1.Create configMap 2.Mention configMap as a volume 3.Map that volume at directory inside container**

Create configMap & we mention it as a volume & that volume we are going to map at some directory (/config)

**vi readcm-pod.yaml**

apiVersion: v1

kind: Pod

metadata:

name: configmap-demo-pod

spec:

containers:

- name: demo

image: alpine

command: ["sleep", "3600"]

env:

# Define the environment variable

- name: PLAYER\_INITIAL\_LIVES # Notice that the case is different here

# from the key name in the ConfigMap.

valueFrom:# value from game-demo-configmap-key player\_initial\_lives will be stored in

configMapKeyRef: #PLAYER\_INITIAL\_LIVES same for UI\_PROPERTIES\_FILE\_NAME

name: game-demo # The ConfigMap this value comes from.

key: player\_initial\_lives # The key to fetch.

- name: UI\_PROPERTIES\_FILE\_NAME

valueFrom:

configMapKeyRef:

name: game-demo

key: ui\_properties\_file\_name

volumeMounts: **# configmap can be mounted as volumes**

- name: config **# config is a volume name & it will be mounted at /config in the container**

mountPath: "/config"

readOnly: true

volumes:

# You set volumes at the Pod level, then mount them into containers inside that Pod

- name: config

configMap: **# volume can be also of type configMap we have seen hostpath volume type before**

# Provide the name of the ConfigMap you want to mount.

name: **game-demo #configMap name from readcm-pod.yaml**

# An array of keys from the ConfigMap to create as files

items:

- key: "game.properties" **# multiline content from samplecm.yaml we are storing it in file path:game.properties**

path: "game.properties" **# and user-interface.properties file. These 2 files will be created in**

- key: "user-interface.properties" **# /config directory inside container**

path: "user-interface.properties"

**When we run the pod then the created container will have 2 variables PLAYER\_INITIAL\_LIVES & UI\_PROPERTIES\_FILE\_NAME and 2 files inside directory /config in container**

**kubectl apply -f readcm-pod.yaml**

**kubectl get pod**

**kubectl exec --stdin --tty configmap-demo-pod -- /bin/sh : we login inside container**

**kubectl exec --stdin --tty <pod\_name> -- /bin/sh**

**we created container with alpine image and alpine image has sh shell not bash shell**

**ls /config/**

**cd /config**

**cat game.properties**

**cat user-interface.properties**

**echo $PLAYER\_INITIAL\_LIVES** : we get 3

**echo $UI\_PROPERTIES\_FILE\_NAME** : we get user-interface.properties

015

**SECRETS :**

Configmap stores variables or files in clear text.we cant store passwords or docker registry credentials which we inject in pod definition file as a clear text coz it is sensitive information.

**We can store variables encoded or encrypted by using secret & then inject it safely in a pod**

**1.Imperative way : NOT GOOD**

**kubectl create secret generic db-secret --from-literal=MYSQL\_ROOT\_PASSWORD=admin123**

**kubectl create secret generic <secret\_name> --from-literal=<variable>=<value>**

**kubectl get secret db-secret -o yaml : we will see MYSQL\_ROOT\_PASSWORD value is enocoded**

**Encode any text in shell or gitbash :**

**echo –n “<string\_to\_encode>” | base64**

**echo –n “<encoded\_value\_from\_above\_command>” | base64 --decode**

echo -n "secretpass" | base64 : we will see encoded value

echo 'output\_of\_above\_command' | base64 --decode : we see secretpass as output

Declarative way :

echo -n "admin123" | base64 : we see encoded value

pass this encoded value in db-secret.yaml inside data section

**Sample file : db-secret.yaml**

apiVersion: v1

kind: Secret

metadata:

name: mysecret

type: Opaque

data:

my\_root\_password: **paste encoded value here**

**1.To export Multiple secret into pod.yaml**

**In pod definition file under containers section :**

**envFrom:**

* **secretRef:**

**name: db-secret**

**2.To export Multiple secret into pod.yaml**

**In pod definition file under containers section :**

**env:**

* **name: MY\_ROOT\_PASSWORD**

**valueFrom:**

**secretKeyRef:**

**name: db-secret**

**key: my\_root\_pass # value of this key will be stored in variable above MY\_ROOT\_PASSWORD**

Insert image 20 21 22 from F:\Notepad++\_Images

In pod definition file envFrom > secretRef for multiple secrets & env for single secrets

**secret types :**

**opaque (user defined data),**

**service-account-token,**

**dockercfg(~/.dockercfg file),**

**dockerconfigjson(~/.dockercfg),**

**basic-auth : credentials for basic authentication**

**ssh-auth, : credentials for SSH authentication**

**tls : data for tls client or server**

**token : bootstrap token data**

**IMP To store privarte docker registry credentials : when you store docker image in private registry, to pull that image in kubernetes we need docker-registry credentials**

**kubectl create secret docker-registry <docker\_registry\_name(regcred)> \**

**--docker-email=<emailID> \**

**--docker-username=<username> \**

**--docker-password=<password> \**

**--docker-server=<for\_custom\_registries\_url:PortNumber> \ # many company use their custom registry**

**--namespace=<namespace\_name>**

**This will create a secret regcred**

**We can decode this secret using below command :**

**kubectl get secret my-registry-secret -o jsonpath='{.data.\*}' | base64 -d**

**Using regcred secret in the pod.yaml :**

apiVersion: v1

kind: Pod

metadata:

name: private-reg

spec:

containers:

- name: private-registry-container

image: <registry-url>/<image-name>:<tag>

**imagePullSecrets:**

**- name: regcred** **# we are pulling secret created from above command here**

Exercise :

Login kOps VM :

**echo -n "admin" | base64**

**echo -n "mysecretpass" | base64**

**vim mysecret.yaml**

apiVersion: v1

kind: Secret

metadata:

name: mysecret

**data:**

**username: paste\_output\_of\_echo -n "admin" | base64**

**password: paste\_output\_of\_echo -n "mysecretpass" | base64**

type: Opaque

**kubectl create -f mysecret.yaml**

**vim readsecret.yaml # pod file to access secrets created from above mysecret.yaml**

apiVersion: v1

kind: Pod

metadata:

name: secret-env-pod

spec:

containers:

- name: mycontainer

image: redis

env:

**- name: SECRET\_USERNAME**

**valueFrom:**

**secretKeyRef:**

**name: mysecret**

**key: username**

**optional: false**

**- name: SECRET\_PASSWORD**

**valueFrom:**

**secretKeyRef:**

**name: mysecret**

**key: password**

**optional: false**

**restartPolicy: Never**

**kubectl create -f readsecret.yaml**

**kubectl get pod**

**kubectl exec --stdin --tty secret-env-pod -- /bin/bash** : connect to pod

we are inside container/pod

**echo $SECRET\_USERNAME : we see decoded secret**

**echo $SECRET\_PASSWORD**

IMP : pull image from private registry kubernetes

016

INGRESS : **Ingress exposes Http and Https routes from outside the cluster to services within the cluster**

**Ingress is basically rules that we create for the service.ClusterIP creates network load balancer.**

Insert image 23 24 from F:\Notepad++\_Images

**Client > Ingres-managed loadbalancer > Ingress > Routing rule > Service > Pods**

**An API object that manages external access to the services in a cluster, typically HTTP.**

Ingress is specifically for the external access that comes from outside, from the Internet. You are running an application in the Pod inside the Kubernetes cluster. Let's say it is a website, web application,So how will the user access it? Yes, we have seen service and we have seen service of type node port and load balancer. But that is just not enough. Because when you have many applications, especially microservices, then controlling different access from externally, like access based on a URL path or a port number, there are many kinds of rules that goes in. All those rules can be managed inside ingress. Ingress may provide load balancing, SSL termination. So the Https based access, you can terminate the SSL connection on the ingress. So internally in Kubernetes, you don't need to handle it.

So, again, what is ingress? Ingress exposes Http and Https routes from outside the cluster. Think of this as a load balancer, where you can set various kinds of rules. Like if someone access on /videos, it goes to this service. As it says, exposes Http, Https routes from outside the cluster to services within the cluster.

Client > Ingress > Service > pod

So you will have service on top of your Pod, and on top of the service there will be ingress. Look at this diagram. User accesses ingress, which will basically a load balancer. And there'll be some routing rule based on how you are accessing it, it's going to route to a service that intern will route to the Pod.So here the service will be of type clusterIP, internal service, ingress exposes it to the outside world.

**In AWS we use newtwork load balancer to expose the nginx ingress controller.Basically nginx ingress controller is a pod & in front of it we use newtwork load balancer as a service.**

Login KOPS VM :

**Below command creates ingress nginx controller :**

**kubectl apply –f** [**https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller**](https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller)**-v1.8.1/deploy/static/provider/aws/deploy.yml**

**kubectl get ns**

**kubectl get all –n paste\_ingress\_ns\_from\_above\_command**

**kubectl get all -n ingress-nginx : we have 3 pods, a service of type loadbalancer which is network load balancer.**

**Check in AWS we see a load balancer.Out of 3 pods we see one pod of ingress-nginx-controller (Running) where we set rules so based on what the request comes its gonna route it to the correct service.Also we see deployment, replicaset & jobs.**

**In AWS console we see network loadbalancer & target groups with 2 worker nodes which are the target with nodeport.** pod/nginx-controller - where we set routing rules

PATH BASED ROUTING WITH INGRESS : FLOW :

Create Controller > Create Deployment > Service > Create DNS CNAME record for loadbalancer > Create Ingress

mkdir vprofile && cd vprofile

vi vprodep.yaml # deployment definition file

apiVersion: apps/v1

kind: **Deployment**

metadata:

name: **my-app**

labels:

app: nginx

spec:

selector:

matchLabels:

run: my-app # label of pod

replicas: 1

template: # pod information

metadata:

labels:

run: my-app

spec:

containers:

- name: my-app

image: **imranvisualpath/vproappfix # tomcat image**

ports:

- containerPort: 8080

**Above file create a pod**

**vi vprosvc.yaml # it’s an internal service which cant be accessed from outside**

apiVersion: v1

kind: Service

metadata:

name: my-app

spec:

ports:

- port: 8080 **#frontend port for clusterIP**

targetPort: 8080 **#container port from above vprodep.yaml**

protocol: TCP

selector:

run: **my-app # for the pod which has this label**

type: **ClusterIP**

**This will create service of type clusterIP for internal access only**

**Ingress basically means rules, that we create for above created service.**

vi vproingress.yaml **#ingress definition file**

apiVersion: networking.k8s.io/v1

kind: Ingress **# rules created for service**

metadata:

name: vpro-ingress

annotations:

nginx.ingress.kubernetes.io/use-regex: "true" /

spec:

ingressClassName: nginx

**rules:**

**- host: vprofile.groophy.in**

http:

paths:

**- path: /login**

pathType: Prefix

backend:

service:

**name: my-app**

port:

number: 8080

**This will create network load balancer from nginx-controller which is used to expose your Ingress resource (pod**

**created from vprodep.yaml) to the external network**

**If we access vprofile.groophy.in/login from browser it comes to network loadbalancer & it comes to ingress controller & it is going to route this request to the service my-app (service type clusterIP) on port 8080**

**kubectl apply -f vprodep.yaml**

**kubectl apply -f vprosvc.yaml**

**Copy dns name from aws(delete last full stop if it has) > EC2 > loadbalancer > Description**

**goDaddy > Domain settings > Add > Type: CNAME, vprofile, Value: Paste dns name from aws > Add record**

**Wait for sometime it will update the CNAME record**

**kubectl get svc**

**kubectl describe svc my-app : Endpoint- is the pod IP & port number of the container 8080 & we are creating ingress rules for this service**

**vprofile.groophy.in from vproingress.yaml points to our network load balancer & with /login (vprofile.groophy.in/**

**login) NLB is going to route the request to the service : name: my-app and vprofile.groophy.in we mention this in GoDaddy**

**FLOW : Create controller > Create deployment > Create Service(working) > Create DNS CNAME record for load balancer > Create Ingress**

**On real time project controller will already be there & we will be adding the app so all we need to do it create deployment & create ingress.**

**kubectl apply -f vproingress.yaml**

**kubectl get ingress : copy & paste the ADDRESS in browser vprofile.groophy.in/login**

**kubectl delete ingress**

**How routing works based on the path ? HOST BASED ROUTING WITH INGRESS :**

**change path from /login to / in vproingress.yaml**

**kubectl apply -f vproingress.yaml # wait load balancer URL will update**

**kubectl get ingress –watch # for waiting loadbalancer URL will update**

in browser vprofile.groophy.in/

**IMP : Check Host based, Path based, Port based**

DELETE :

**kubectl get ns**

**kubectl delete ns ingress-nginx**

OR

**kubectl delete -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.1/deploy/static/provider/aws/deploy.yaml**

**IMP : WE CREATE INGRESS FOR THE SERVICE. In micro services we will have requirement that if I access it at /api it goes somewhere. This means we have to create ingress rule**

Kubernetes cheat sheet : read all

go from imperative way to declarative way :

create pod definition from command :

kubectl run nginxpod --image=nginx --dry-run=client -o yaml > ngpod.yaml

cat ngpod.yaml

create deployment definition file from command

kubectl create deployment ngdep --image=nginx --dry-run=client -o yaml > ngdep.yaml

cat ngdep.yaml

kubectl logs my-pod -c my-container

kubectl cordon my-node : no new pod running on this node, used when bring down for maintenance

kubectl drain my-node : remove all work load from that node

kubectl uncordon my-node : once done maintenance activity

EXTRA :

Tains & Toleration :

A pod that can tolerate that taint, can only go inside

kubectl taint nodes node1 key1=value1:NoSchedule : we tainted a node here

pod will run on node1, if it has this information key1=value1

In pod definition file add :

tolerations:

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoSchedule"

if pod does not have this toleration it will not run on that node

Limit : if you want to run pod with specific ram memory & CPU

Only ff node have memory: "64Mi" & cpu: "250m" then pod will run on that node.IF no node exists with this much resource then pod will be in pending state

request is reserving & limit is restricting

---

apiVersion: v1

kind: Pod

metadata:

name: frontend

spec:

containers:

- name: app

image: images.my-company.example/app:v4

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

- name: log-aggregator

image: images.my-company.example/log-aggregator:v6

resources:

requests:

memory: "64Mi"

cpu: "250m"

limits:

memory: "128Mi"

cpu: "500m"

Jobs : container with sepcific job. Runs the job and return some output. Job may be commands/script

apiVersion: batch/v1

kind: Job

metadata:

name: pi

spec:

template:

spec:

containers:

- name: pi

image: perl:5.34.0

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

backoffLimit: 4

CRONJOB :

CronJob is meant for performing regular scheduled actions such as backups, report generation

apiVersion: batch/v1

kind: CronJob

metadata:

name: hello

spec:

schedule: "\* \* \* \* \*" min hour date month dayofweek

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: busybox:1.28

imagePullPolicy: IfNotPresent

command:

- /bin/sh

- -c

- date; echo Hello from the Kubernetes cluster

restartPolicy: OnFailure

Job runs once & returns information. Cronjob runs on specific time interval like alarm

DaemonSet : If you have 4 worker nodes & if you run a pod with daemonset you will have 4 pods on each worker node. DaemonSet used to collect logs & do monitoring of your nodes

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

kubectl get ds -A

vi sampleds.yaml

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-elasticsearch

namespace: kube-system

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: fluentd-elasticsearch

template:

metadata:

labels:

name: fluentd-elasticsearch

spec:

tolerations:

# these tolerations are to have the daemonset runnable on control plane nodes

# remove them if your control plane nodes should not run pods

- key: node-role.kubernetes.io/control-plane

operator: Exists

effect: NoSchedule

- key: node-role.kubernetes.io/master

operator: Exists

effect: NoSchedule

containers:

- name: fluentd-elasticsearch

image: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

volumeMounts:

- name: varlog

mountPath: /var/log

terminationGracePeriodSeconds: 30

volumes:

- name: varlog

hostPath:

path: /var/log

kubectl apply -f sampleds.yaml

kubectl get ds -A

kubectl get pod -n kube-system

delete node

kubectl delete node <node-name>

Lens : Dashboard of central view of all your kubernetes cluster

Download lens install it & check options in it

Kubernetes is a complex container orchestration platform with its own set of terminologies. Here's a list of some key Kubernetes terminologies and their brief explanations:

1. **Cluster**:
   * A Kubernetes cluster consists of a set of physical or virtual machines (nodes) that run containerized applications. It includes both the control plane (master nodes) and worker nodes.
2. **Node**:
   * A node is a single machine within a Kubernetes cluster. It can be a physical server or a virtual machine and is responsible for running containers.
3. **Control Plane**:
   * The control plane (also known as the master nodes) is responsible for managing the cluster's overall state. It includes components like the API server, controller manager, and scheduler.
4. **Pod**:
   * The smallest deployable unit in Kubernetes. A pod can contain one or more containers that share network and storage namespaces.
5. **Container**:
   * Containers are lightweight, standalone executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools.
6. **Namespace**:
   * Namespaces provide a way to logically divide a Kubernetes cluster into multiple virtual clusters. They help isolate resources and applications.
7. **Service**:
   * A Kubernetes Service is an abstraction that defines a logical set of Pods and a policy for accessing them. It provides network connectivity to a group of Pods.
8. **Deployment**:
   * A Deployment is a higher-level abstraction for managing and scaling a set of identical Pods. It ensures that a specified number of replicas are running at all times.
9. **ReplicaSet**:
   * A ReplicaSet is an older way to manage a set of identical Pods. Deployments are recommended over ReplicaSets for most use cases.
10. **StatefulSet**:
    * A StatefulSet is used to manage stateful applications that require unique, stable network identifiers and persistent storage.
11. **DaemonSet**:
    * A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. It's often used for running system daemons or agents on every node.
12. **ConfigMap and Secret**:
    * ConfigMaps and Secrets are used for managing configuration data and sensitive information (e.g., passwords) that can be mounted into containers.
13. **Ingress**:
    * An Ingress is a Kubernetes resource that manages external access to services within the cluster, typically HTTP.
14. **Volume**:
    * A Volume is a directory that can be mounted into a container. It can be used for sharing data between containers or persisting data.
15. **Persistent Volume (PV) and Persistent Volume Claim (PVC)**:
    * PVs and PVCs provide a way to manage and access persistent storage in a cluster.
16. **RBAC (Role-Based Access Control)**:
    * RBAC is a security mechanism for controlling access to Kubernetes resources based on roles and permissions.
17. **Secrets**:
    * Secrets are used for securely storing and managing sensitive information, such as API tokens or database passwords.
18. **Kubelet**:
    * The Kubelet is an agent running on each node in the cluster. It ensures that containers are running in a Pod.
19. **Kube-proxy**:
    * Kube-proxy is a network proxy that runs on each node. It maintains network rules on the host and forwards traffic to the appropriate Pod.
20. **Kubeconfig**:
    * A kubeconfig file is used to configure access to a Kubernetes cluster, including the cluster's API server, user credentials, and context.

These are just some of the key terminologies in Kubernetes. Kubernetes has a rich ecosystem with many additional components and concepts, making it a powerful platform for container orchestration and management.

Top of Form

=================================================================================================================

**ADD jdk : OracleJDK & maven : MAVEN3 in JENKINS TOOLS**

**INSTALL PLUGINS : Nexus Artifact Uploader, SonarQube Scanner, Build Timestamp, Pipeline Maven Integration, Pipeline Utility Steps**

**ADD SonarQube Scanner : sonar4.7 in JENKINS TOOLS**

**SonarQube generate token & COPY it (My account > Security > Generate token)**

**CONFIGURE SonarQube servers > SonarQube > sonar > URL : publicIP of SonarServer EC2 > Server authentication token > secret text > secret : paste above token**

Pipeline > agent > tools (maven, jdk) > stages(Fetch code) > steps > git repo url with branchname > stage(build) > sh mvn install > post > success (archiveArtifacts) > stage(Test) > steps > mvn test > stage(checkstyle analysis) > steps > mvn checkstyle:checkstyle

**Add one more stage of Sonar Analysis > environment > scannerHome = tool 'sonar4.7' > steps > withSonarQubeEnv('sonar') in pipeline**

**Add one more stage for Quality Gate with timeout & waitForQualityGate in pipeline**

**Create own quality gate in sonarqube with condition & select created quality gate in project**

**Create hosted repository on nexus server**

**Add Credentials of nexus in jenkins Security Give username password of nexus & ID as nexuslogin**

**Configure build timestamp & SonarQube servers with URL and Server authentication token as MySonarToken**

**Sonarqube Add webhooks from project settings & URL jenkinsIP:8080/sonarqube-webhook**

**Add one more stage UploadArtifact with credetialsId as 'nexuslogin' & version for build stamp**

**Configure slack & for notifications – Create workspace, Add email of team mates, Add channel Add emails to channel**

**Add jenkins ci app to slack (google add app to slack) > select channel > copy token > save**

**Add plugin slack notification to the Jenkins & integrate configure system with paste token and channel name. Test connection**

**IMP WHENEVER WE ADD PLUGIN TO JENKINS INTEGRATE/CONFIGURE THAT PLUGIN TO JENKINS**

**Add stage print error at the top with steps sh fake comment**

**Add post installation step after all stages post > always > slackSend > channel name Also define color\_map function at the top of pipeline**

**Install docker engine & awscli on jenkins server, add jenkins user to docker group, IAM user with AmazonEC2registryFullAccess, AmazonECS\_FullAccess**

**Create rfepository on ECR**