Kubernetes Ubuntu Pod Creation

# **Docker image:** **Docker** image is an executable package of software that includes everything needed to run an application.

Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments

# **Docker File:** Docker file is a text document that contains all the commands a user could call on the command line to assemble an image.

**Docker File Commands:**

1. **From** - A FROM statement defines which image to download and start from. It must be the first command in your Dockerfile.

Ex: FROM java: 8

1. **Maintainer -** This statement is a kind of documentation, which defines the author who is creating this Dockerfile or who should you contact if it has bugs.

Ex: MAINTAINER Firstname Lastname [example@geeksforgeeks.com](mailto:example@geeksforgeeks.com)

1. **Run -** The RUN statement defines running a command through the shell, waiting for it to finish, and saving the result. It tells what process will be running inside the container at the run time.

Ex: RUN unzip install.zip /opt/install

RUN echo hello

1. **Add -** It basically gives instructions to copy new files, directories, or remote file URLs and then adds them to the file system of the image. To sum up it can add local files, contents of tar archives as well as URLs.

**Ex:** Local Files: ADD run.sh /run.sh

Tar Archives: ADD project.tar.gz /install/

1. **Env** – This statement sets the environment variables both during the build and when running the result. It can be used in the Dockerfile and any scripts it calls. It can be used in the Dockerfile as well as any scripts that the Dockerfile calls. These are also persistent with the container and can be referred to at any moment.

**Ex:** ENV URL\_POST=production.example-gfg.com

1. **Entrypoint -** It specifies the starting of the expression to use when starting your container. It specifies the start of the command to run. If your container acts as a command-line program, you can use entrypoint.

**Ex:** ENTRYPOINT ["/start.sh"]

1. **Cmd -** It specifies the whole command to run. We can say CMD is the default argument passed into the ENTRYPOINT. The main purpose of the CMD command is to launch the software required in a container.

**Ex:** CMD ["program-foreground"]

CMD ["executable", "program1", "program2"]

1. **Expose –** It statement maps a port into the container. The ports can be TCP or UDP but by default, it is TCP.

**Ex:** EXPOSE 3030

1. **Volume-** The VOLUME statement defines shared volumes or ephemeral volumes depending upon whether you have one or two arguments.

If you have two arguments, it maps a host path into a container path.

VOLUME ["/host/path" "/container/path/"]

If you have one arguments, it creates a volume that can be inherited by the later containers.

VOLUME ["/shared-data"]

### **Workdir-** Its main purpose is to set the working directory for all future Dockerfile commands.

**Ex:** WORKDIR /directory-name

1. **User**- It sets which user container will runs as. This can be useful if you have shared network directories involved that assume a fixed username or a fixed user number.

**Ex:** USER geeksforgeeks USER 4000

1. **Arg-** A variable that can be provided at build time is defined by an ARG Instruction. Once it has been specified in the Dockerfile, you can specify it using the –build-arg switch when creating the image. The Dockerfile supports multiple ARG instructions. The only instruction in the Dockerfile that can come before the FROM instruction is ARG.After the image is created, ARG values are not accessible. An ARG variable value won’t be accessible to a running container.

### Ex:

ARG image\_name=latest

FROM centos:$image\_name

docker build -t <image-name>:<tag> --build-arg image\_name=centos8

**Dockerfile for Ubuntu:**

First, create a new project directory and create an empty Dockerfile.

mkdir –p test

cd test/

touch dockerfile

Now edit the 'Dockerfile' script using your own editor (for this example we're using nano)

nano Dockerfile

# Ubuntu Dockerfile

# <https://github.com/dockerfile/ubuntu>

# Pull base image.

FROM ubuntu: 18.04

# Install.

apt-get update && \

apt-get -y upgrade && \

apt-get install -y build-essential && \

apt-get install -y software-properties-common && \

# Set environment variables.

ENV HOME /root

# Define working directory.

WORKDIR /root

# Define default command.

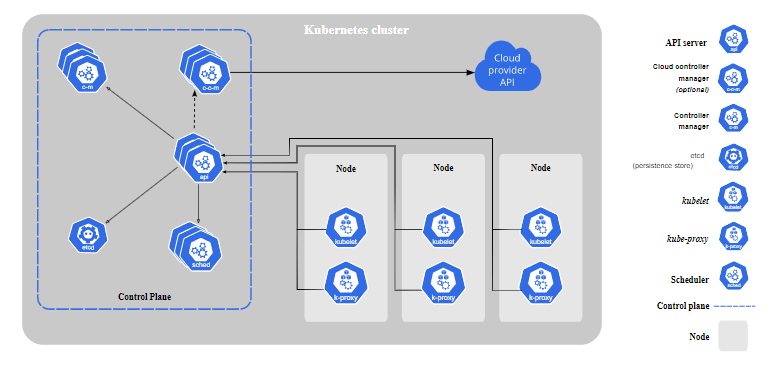
CMD ["bash"]

**Kubernetes-K8**

[Kubernetes](https://kubernetes.io/docs/concepts/overview/), also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, which facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

# **Kubernetes cluster:**

A Kubernetes cluster consists of a set of worker machines, called [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/), which run containerized applications. Every cluster has at least one worker node. The worker node(s) host the [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) that are the components of the application workload. The [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

**Fig.no.1 Kubernetes Cluster**

### **kube-scheduler**

Control plane component that watches for newly created [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) with no assigned [node](https://kubernetes.io/docs/concepts/architecture/nodes/), and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

### **kube-controller-manager**

Control plane component that runs [controller](https://kubernetes.io/docs/concepts/architecture/controller/) processes.

Logically, each [controller](https://kubernetes.io/docs/concepts/architecture/controller/) is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

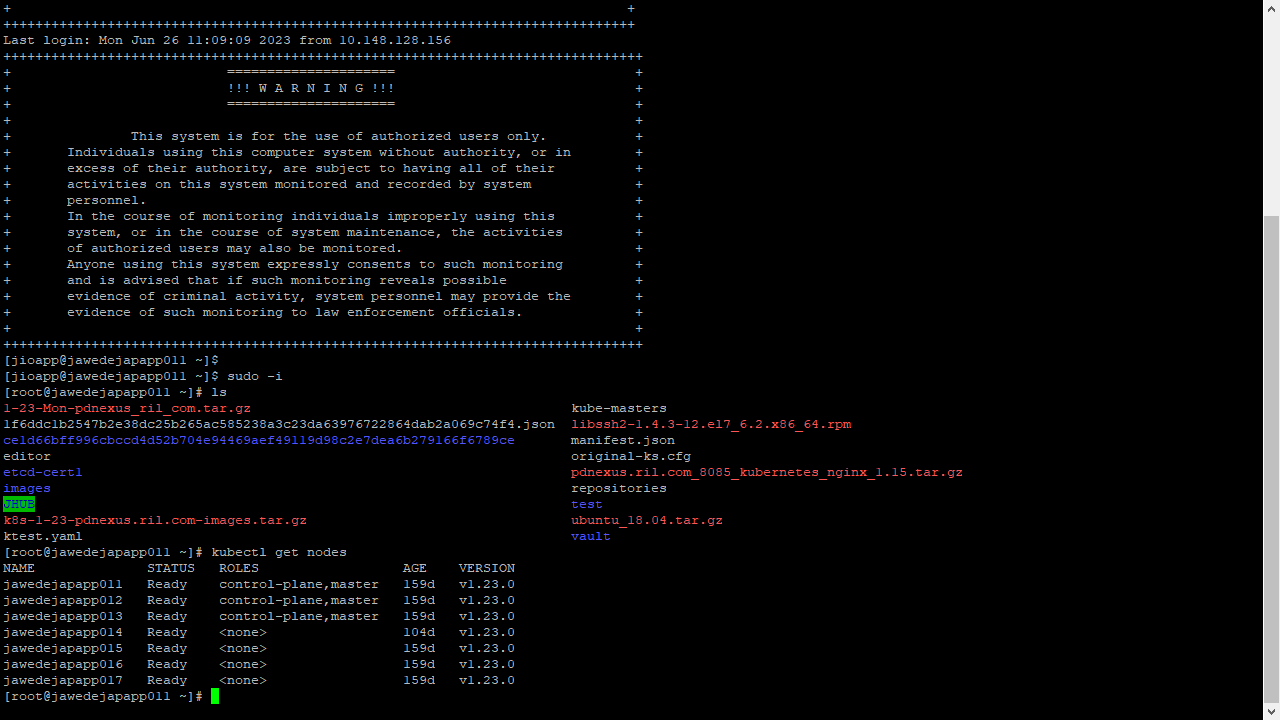
Some types of these controllers are:

* Node controller: Responsible for noticing and responding when nodes go down.
* Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
* EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
* ServiceAccount controller: Create default ServiceAccounts for new namespaces.

## **YAML File:**

YAML Ain't Markup Language is a data serialization language that matches user’s expectations about data. It designed to be human friendly and works perfectly with other programming languages. It is useful to manage data and includes Unicode printable characters.

## **Kubernetes master-slave:**



**Fig.no.2 Kubernetes master-slave nodes**

**Kubernetes Namespaces**

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.

## **Initial namespaces**

Kubernetes starts with four initial namespaces:

**default**

Kubernetes includes this namespace so that you can start using your new cluster without first creating a namespace.

**kube-node-lease**

This namespace holds [Lease](https://kubernetes.io/docs/concepts/architecture/leases/) objects associated with each node. Node leases allow the kubelet to send [heartbeats](https://kubernetes.io/docs/concepts/architecture/nodes/#heartbeats) so that the control plane can detect node failure.

**kube-public**

This namespace is readable by all clients (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

**kube-system**

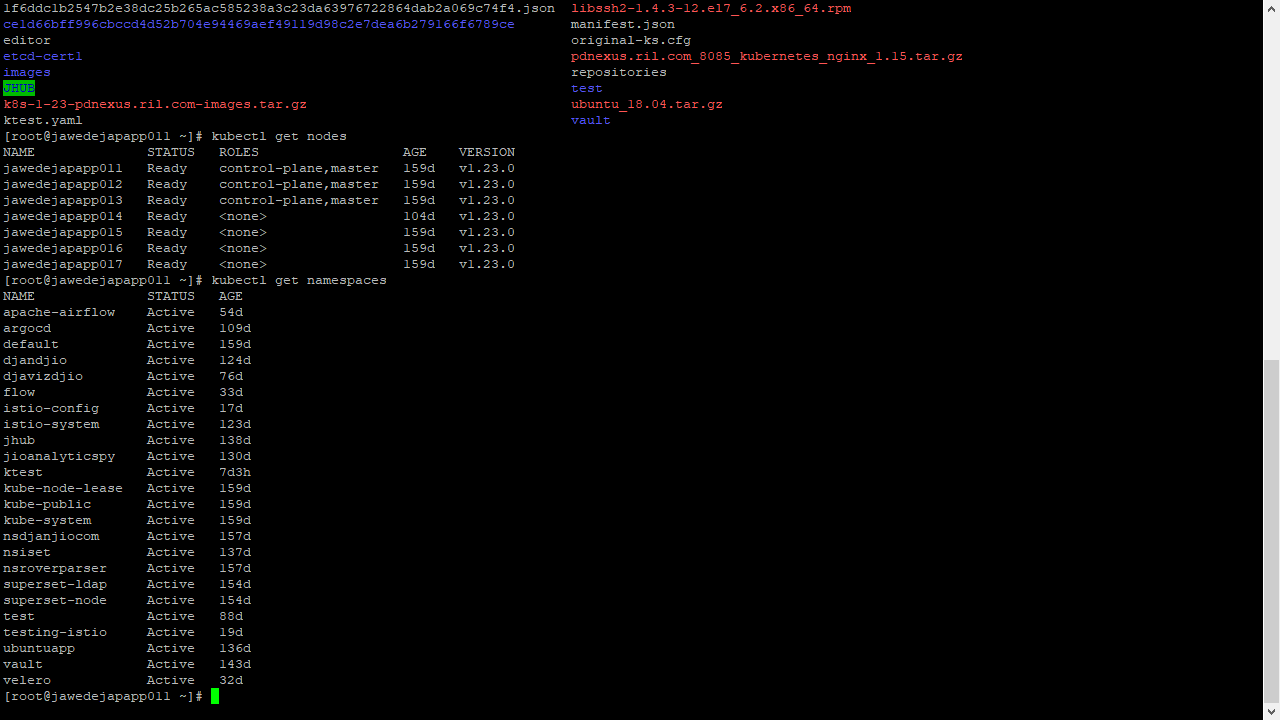
The namespace for objects created by the Kubernetes system.

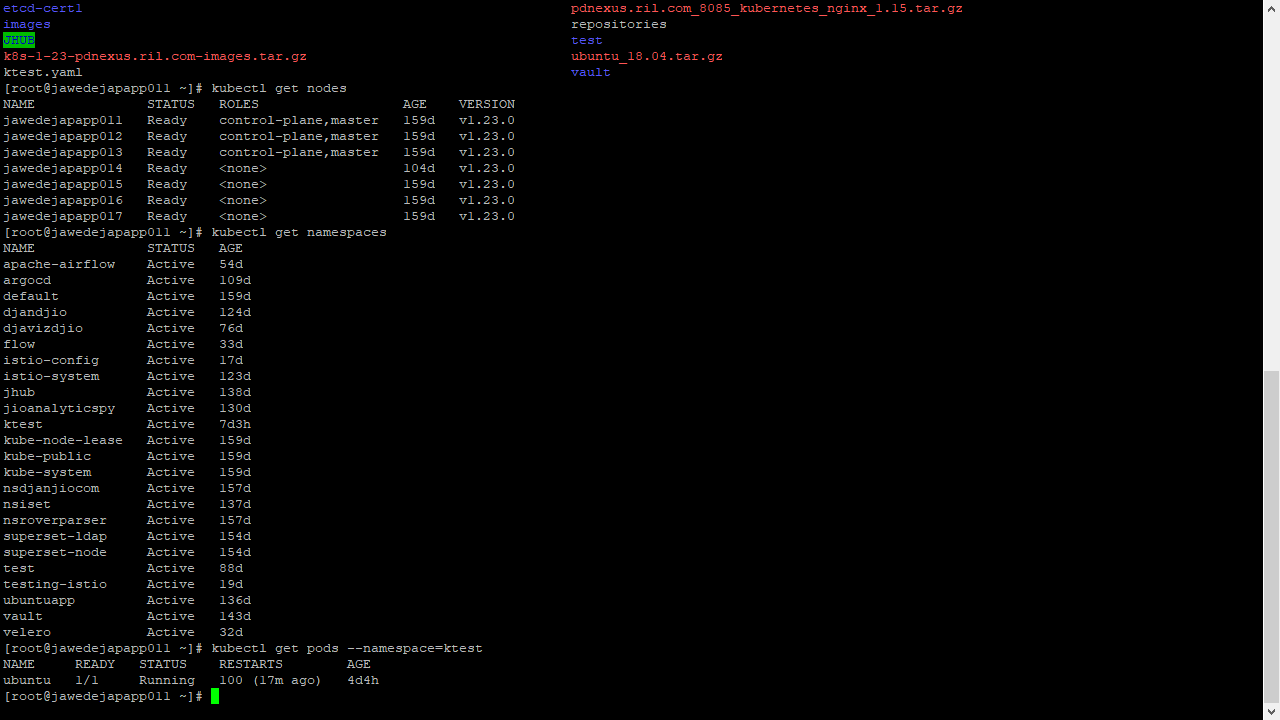
**Kubernetes Pod**

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more [containers](https://kubernetes.io/docs/concepts/containers/), with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

**Following are the screenshots for Ubuntu pod:**





**Ubuntu Image Errors Faced:**

**CrashLoopBackOff:**

CrashLoopBackOff is a Kubernetes state representing a restart loop happening in a Pod: a container in the Pod is started, but crashes and is then repeatedly restarted. CrashLoopBackOff is not an error in itself but indicates that there’s an error happening that prevents a Pod from starting correctly. By default, a pod’s restart policy is Always, meaning it should always restart on failure (other options are Never or OnFailure). Depending on the restart policy defined in the pod template, Kubernetes might try to restart the pod multiple times. Every time the pod is restarted, Kubernetes waits for a longer and longer time, known as a “**backoff delay**”. During this process, Kubernetes displays the CrashLoopBackOff error.

# **How to Detect the Error and its Status:**

Typically, when we do any deployment in Kubernetes and to check our pod’s condition, we use the below command which displays the output

Kubectl get pods

1) Image is Not ready 0/1.

2) Column **status** displays CrashLoopBackOff.

3) Column **restarts** display **one or more** restarts.

**ImagePullBackoff:**

The ImagePull part of the ImagePullBackOff error primarily relates to your Kubernetes container runtime unable to pull the image from a private or public container registry. The Backoff part indicates that Kubernetes will continuously pull the image with an increasing backoff delay. Kubernetes will keep on increasing the delay with each attempt until it reaches the limit of five minutes. Here are some of the possible causes behind your pod getting stuck in the ImagePullBackOff state:

* Image is not defined properly
* Tag may changed/missing/incorrect
* Image name/path not correct
* Image missing/incorrect
* Image may private, and there is a auth failure
* Check pull secret defined or not
* Secret name Is correct?
* Secrets are correct?
* Network issue
* Container registry Rate Limits

## **ErrImagePull / ImagePullBackOff?**

Kubernetes pods sometimes experience issues when trying to pull container images from a container registry. If an error occurs, the pod goes into the ImagePullBackOff state.

When a Kubernetes cluster creates a new deployment, or updates an existing deployment, it typically needs to pull an image. This is done by the kubelet process on each worker node. For the kubelet to successfully pull the images, they need to be accessible from all nodes in the cluster that match the scheduling request.

## **How Does Kubernetes Work with Container Images?**

A container image includes the binary data of an application and its software dependencies. This executable software bundle can run independently and makes well-defined assumptions about its runtime environment. You can create an application’s container image and push it to a registry before you refer to it in a pod.

## **Image Pull Policy**

A container’s imagePullPolicy and its tag determine when the kubelet tries to download (pull) the image. You can set various values for an imagePullPolicy, each achieving a different effect. The IfNotPresent value tells the kubelet to pull the image only if it is not present locally, Never tells the kubelet not to try fetching this image, and Always means the kubelet queries the container registry to resolve the name to an image digest when launching a container.

**Kubernetes Master Nodes:**

10.169.176.4, 10.169.176.5, 10.169.176.6

**Kubernetes Slave Nodes:**

10.169.176.7, 10.169.176.8, 10.169.176.9, 10.169.176.10

**Step 1:** **Ubuntu image is pulled on development node (10.160.136.178) using command:**

docker image pull ubuntu:18.04

**Step 2:** **Identify master and slave nodes in K8s using command:**

kubectl get nodes

**Step 3:** **Save image into .tar.gz file extention using command:**

docker save ubuntu:18.04 | gzip > ubuntu\_18.04.tar.gz

**Step 4: Copy ubuntu image on all nodes using command:**

scp analyticspoc@10.160.136.178:ubuntu\_18.04.tar.gz .

**Step 5: Create namespace on master node (10.169.176.4) using command:**

kubectl create namespace ktest

**Step 6: Create ktest.yaml for pod creation on master node using command:**

vi ktest.yaml :

apiVersion: v1

kind: Pod

metadata:

name: ubuntu

namespace: ktest

spec:

containers:

- name: ubuntu

image: ubuntu:18.04

command:

- sleep

- "3600"

**Step 7: Create pod on master node using command:**

kubectl apply –f ktest.yaml --namespace=ktest

**Step 8: Check all pods using command:**

kubectl get pods --namespace=ktest

**To get into Ubuntu terminal using following commands:**

kubectl exec -it ubuntu --namespace=ktest -- whoami

kubectl exec -it ubuntu --namespace=ktest -- bash

**K8 and Docker used commands:**

docker images

docker image rm image name

docker images | grep ubuntu

docker load < ubuntu\_18.04.tar.gz

kubectl delete ubuntu --namespace=ktest

kubectl get namespaces

kubectl describe pod pod name

**References:**

<http://10.169.176.10:30566/#/pod?namespace=ktest>

<https://gcore.com/learning/how-to-transfer-move-a-docker-image-to-another-system/>

<https://docs.docker.com/engine/reference/commandline/image_rm/>

**Kubernetes Basics and Complete Notes ----- Core Course**

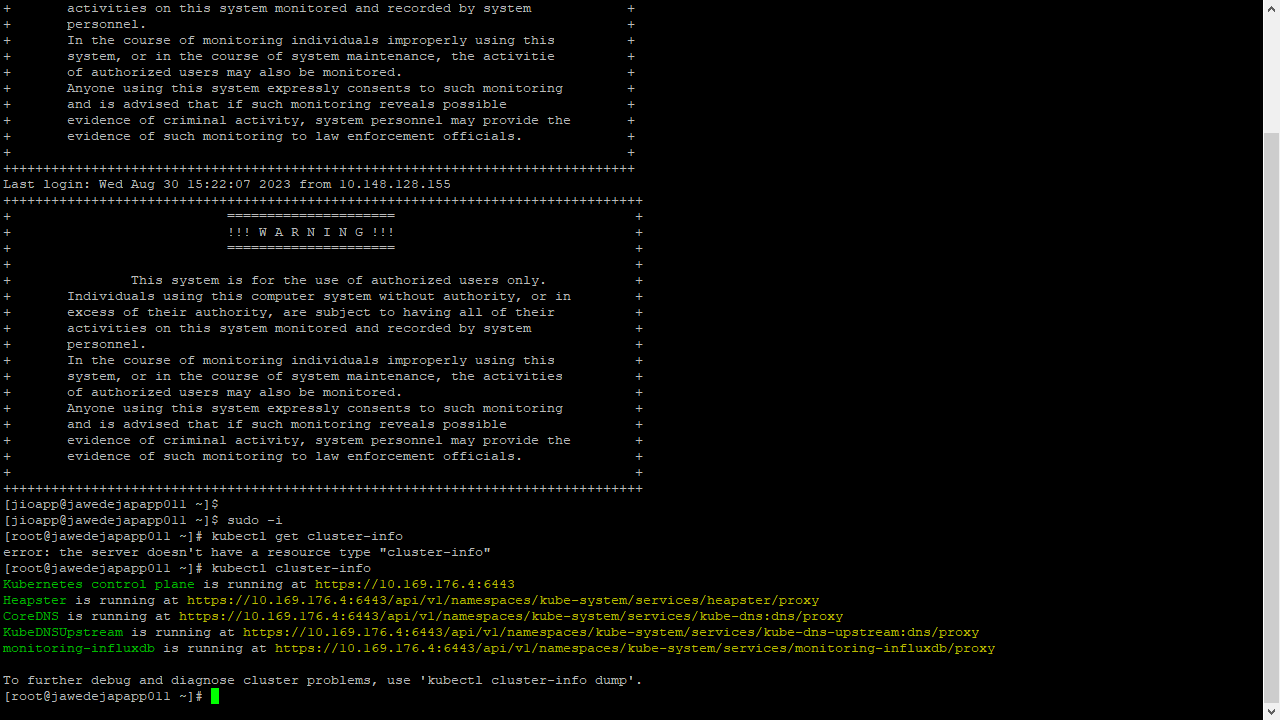
Local k8 cluster setup

Docker desktop makes easy installation for kubernetes

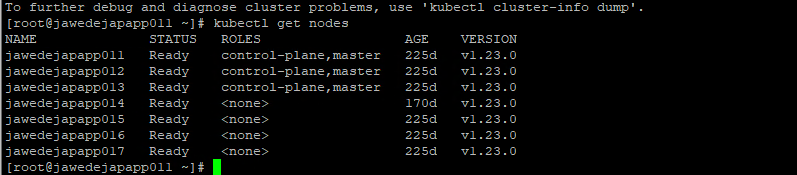
Docker desktop for OS – Windows /Linux /Mac-os

Docker desktop ---- Kubernetes – enable k8 – cluster installation- k8 images and setup will be done

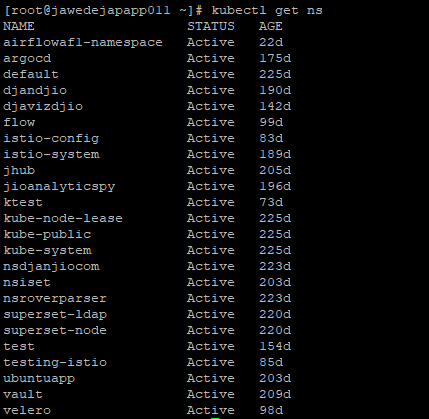
kubectl cluster- info will show setup info if installed



kubectl get nodes



kubectl get na / ns



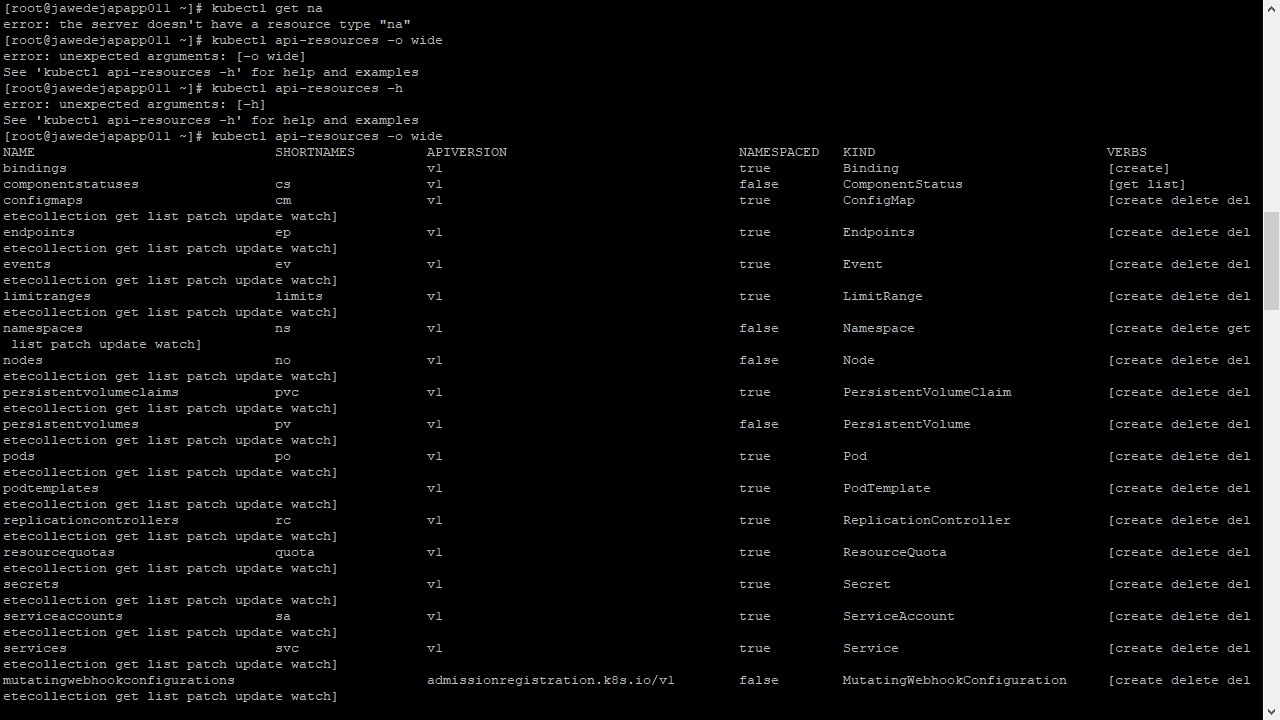
**Core concepts**

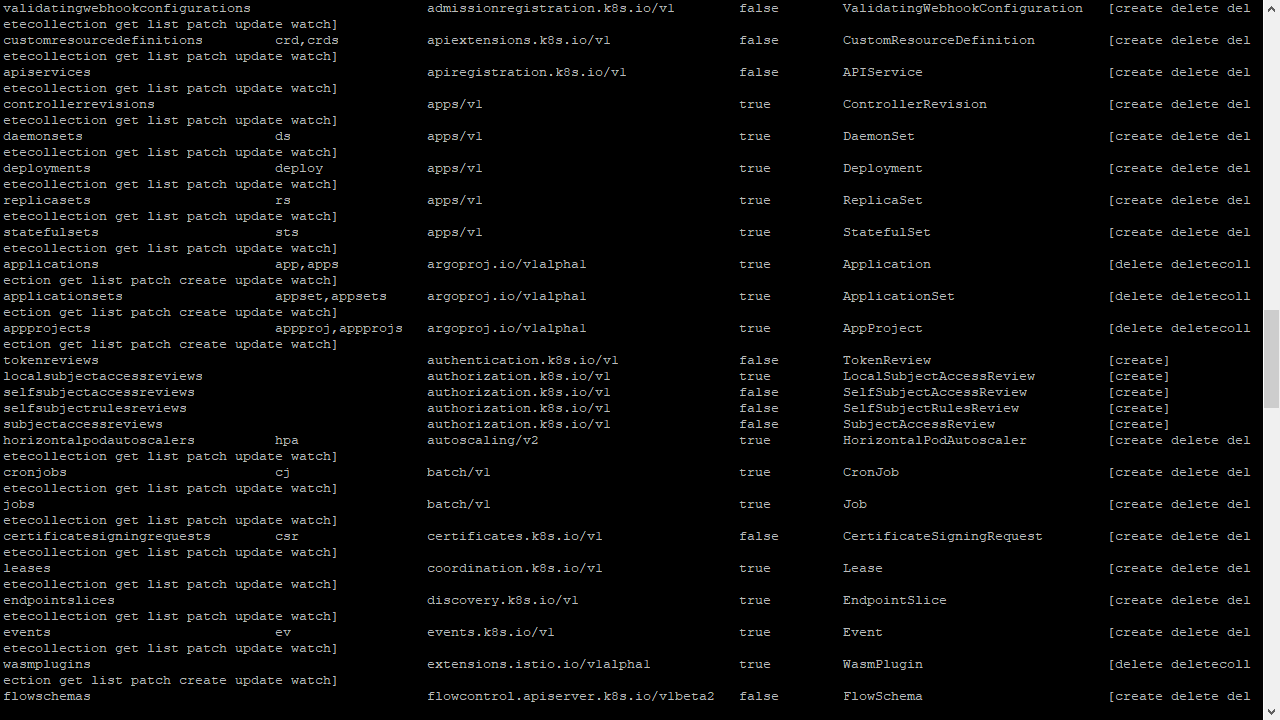
Control plane manages all nodes and pods with the different components

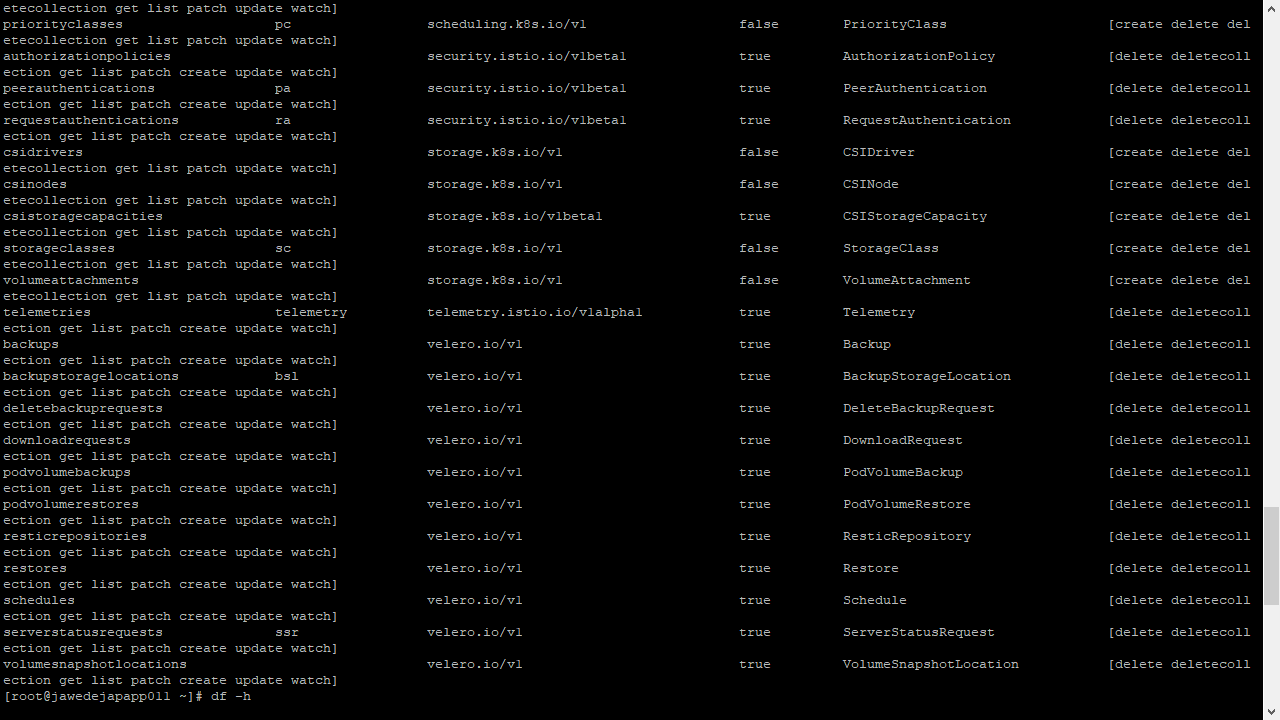
Kubescheduler decides when schedule and on which node will pod deployed

Each worker node contains Docker, Kubelet and Kube proxy

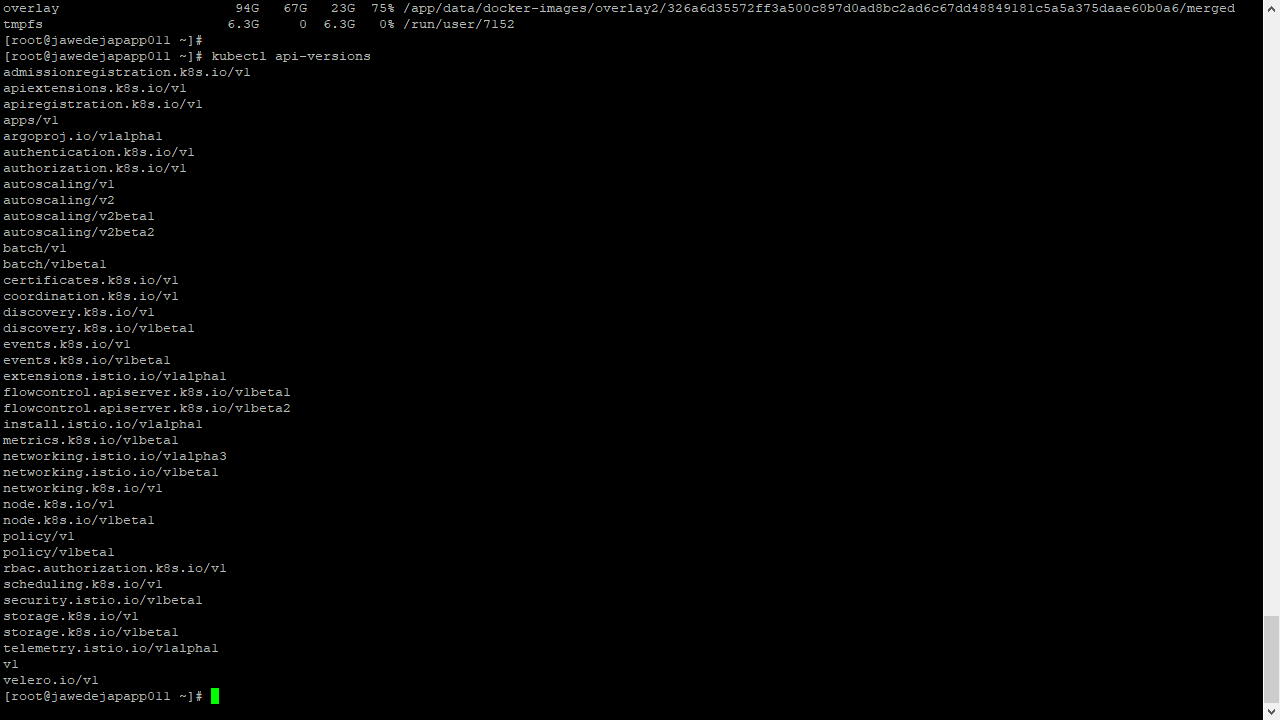
kubectl api-resources –o wide







kubectl api-versions



kubectl proxy commands and URLs

K8 namespaces, labels and commands

kubectl get ns

Kube config – we can create multiple cluster connection, context

kubectl run pod name – creates a pod

Labels with pods ---Can pass arguments with pods and attributes

kubectl get pods –A

--overwrite flag with pods name

kubectl run –it busybox - -image=busybox –restart=Never –rm –

**K8 Pods info commands are following:**

kubectl describe pods/nginx

kubectl describe po nginx

kubectl describe po –l name=mylabel

kubectl explain pods

kubectl explain pods.spec.conatiners

kubectl exec podname -- date

kubectl exec podname -- env

**To get inside into the conatiners on K8:**

kubectl exec --it podname -- sh

kubectl exec --it podname --bash

kubectl exec --it podname -- /bin/sh

kubectl exec --it podname -- /bin/bash

kubectl port –help

**To set the image with existing one in pod:**

Syntax: kubectl set image POD/PODNAME CONTAINER\_NAME=IMAGE\_NAME:TAG

Example: kubectl set image pod/nginx nginx=nginx:1.7.1

kubectl logs podname

kubectl logs –f podname container name

kubectl logs –f podname –c container name

**Practical scene 1:**

**Create namespace and a pod – nginx with image ‘nginx’ in that namespace**

**Commands:** kubectl create ns demo

kubectl get ns demo

kubectl –n demo run nginx --image=nginx

**Practical scene 2:**

**Create following pod yaml file in - ‘/tmp/redis1.yaml’.**

**YAML File contents:**

**Pod name: ‘redis1’**

**Image: ‘redis’**

**Namespace: ‘demo’**

**Expose on port 6379**

**Commands:**

kubectl –n run redis1 –port=6379 –dry-run=client -o yaml > /tmp/redis1.yaml

cat /tmp/redis1.yaml – to see redis1.yaml file

kubectl create –f /tmp/redis1.yaml – will create pod

**Practical Scene 3:**

**Create a busybox pod in ‘demo’ namespace that runs command ‘env’**

**Write the pod log to ‘/tmp/busybox.txt’**

**Command:** kubectl –n demo run busybox --image=busybox --restart=Never –command – env

kubectl –n demo logs busybox > /tmp/busybox.txt - will write logs

cat /tmp/busybox.txt – will show logs

**Practical Scene 4:**

**Export the busybox pod in demo namespace yaml definition to ‘/tmp/busybox.yaml’ and delete pod**

**Commands:** kubectl –n demo get po busybox –o yaml > /tmp/busybox.yaml

cat /tmp/busybox.yaml

kubectl –n demo delete po busybox

**Practical Scene 5:**

**Create yaml in /tmp/myrq.yaml for new ResourceQuota called ‘myrq’ with hard limits of 1 CPU , 1G memory and 2 pods without creating it.**

**Commands:** kubectl create quota –h -- for resource quota info

kubectl create quota myrq --hard=cpu=1, memory=1G,pods=2 --dry-run=client -o yaml > /tmp/myrq.yaml

**Practical Scene 6:**

**Create a pod with image nginx called nginx2 and expose traffic on port and set an env value as “NGINX\_PORT=80”.**

**Change pod’s image to nginx: 1.19.8 and record the change.**

**Commands:**

kubectl –n demo run nginx2 --image=nginx --port=80 –env=NGINX\_PORT=80

kubectl –n demo set image pod/nginx2 nginx2=nginx:1.19.8 --record

**Practical Scene 7:**

**Get ‘nginx2’ pod’s IP created in previous step, use a temp busybox image to wget of it’s ‘/’ and save to ‘/tmp/nginx-index.txt’**

**Commands:** kubectl –n demo po nginx2 -o=jsonpath=’{.status.podIP}’

**Defining a variable:**

Export NGINX\_IP=$( kubectl –n demo po nginx2 -o=jsonpath=’{.status.podIP}’)

Echo $NGINX\_IP

kubectl –n demo run busybox --image=busybox –restart=Never –rm -it – wget -o http:// $NGINX\_IP/ > /tmp/nginx-index.txt

**Practical Scene 8:**

**Create a myfrontend pod in demo ns with nginx img and set env value as ‘var1=var1’.**

**Check the env value existence within pod**

**Commands:** kubectl –n demo run myfrontend –image=nginx –env=var1=var1

**Practical Scene 9:**

**Update myfrontend pod with following:**

**Labels, annotations and environment variables**

**Commands:** kubectl –n demo po/myfrontend environment=demo owner=finance

kubectl –n demo get po –show-labels

kubectl –n demo annotate po/myfrontend description=’Frontend Application’

kubectl –n demo edit po myfrontend

**Practical Scene 10:**

**Create a busyboxgoogle pod with img ‘busybox’ in demo ns which executes ‘sleep 3600’,**

**Connect interactive shell pod and save current date to ‘/tmp/date.txt’ inside the container**

**Commands:**

kubectl –n demo run busyboxgoogle --image=busybox --restart=Never –command -- /bin/sh –c ‘sleep 3600; ‘

kubectl -n demo exec –it busyboxgoogle – sh

date

date > /tmp/date.txt – save the current date

**Kubernetes (K8) Sub Domains:**

**Configmaps** = API object used to store non-confidential data in key-value pairs

Pods consume configmaps as env variable, command line arguments or config files in volume

Configmap allows you to decouple env – specific configuration from your container images , so that ur applications are easily portable.

kubectl create configmap configmap-name

kubectl get configmaps

kubectl get cm

kubectl edit configmap configmap\_name

**Secrets:** store and manage sensitive info – passwords, OAuth tokens, and ssh keys

kubectl create secret generic app-secret \ --from-literal=DB\_HOST=mysql

kubectl create secret generic app-secret2 \ --from-env-file=app\_secret.properties \ --dry-run=client -o yaml > app-secret2.yaml

kubectl edit secret secret\_name

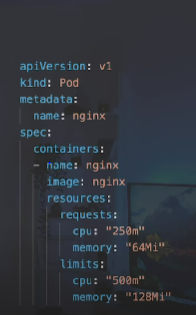
kubectl create secret docker-registry regcred \ --docker-server=<ur-registry-server> \ --docker-username=<ur-name> \ --docker-passwprd=<ur password> \ --docker-email=<ur-email>

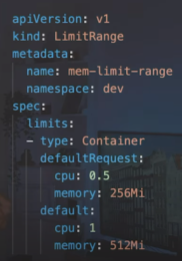
kubectl create secret tls my-tls-secret \ --cert=path/tc/cert/file \ --key=path/to/key/file

**Security Context** – defines privileges and access control settings for pod or container

Security context settings include, but not limited to:

Resource Requirements:





**Service Account:** used by container to communicate with the API server of the k8 cluster

Linked to the authentication, authorization and RBAC in k8 and are namespace

kubectl create serviceaccount Jenkins-sa

kubectl get serviceaccounts

kubectl describe serviceaccount Jenkins-sa

**Practice Session**

**Practical Scene 1:**

Create a ns called ‘demo’ and create configmap named ‘config1’ with values shape= square and color=red

Create a new nginx pod with nginx img and load configmap as env variable into a pod.

**Commands:**

Kubectl –n demo create cm config1 –from –literal= shape=square –from –literal= color=red

Kubecl –n demo run nginx –image=nginx –dry-run=client –o yaml > /tmp/nginx.yaml

Vi nginx.yaml file

Add

envFrom:

* configmapRef:

name: nginx

**Practical Scene 2:**

Create /tmp/application.properties file with below content port=8000 profile=dev

In demo ns, create cm named appconfig, create pod as app with nginx img and mount the cm to /opt/path

Make sure /opt/application.properties file exists in the app pod

**Commands:**

vi /tmp/application.properties – add contents

port=8000

profile=dev

kubectl –n demo create cm appconfig –from-file= application.properties=/ tmp/application.properties

kubectl –n demo run app –image=nginx –dry-run=client –o yaml > /tmp/app.yaml

vi /tmp/app.yaml – edit file

add following contents

spec:

values:

-name: appconfig

configMap:

name: appconfig

containers:

-image: nginx

Name: app

volumeMounts:

-name: appconfig

mountPath: /opt/path

kubectl create –f /tmp/app.yaml

kubectl –n demo get po

**Practical Scene 3:**

**In demo ns , create cm appconfig2 with loading values from /tmp/application.properties file**

**Create app2 pod with tomcat img in ns and mount the cm enteries as env variables**

**ENV name: SERVER\_PORT**

**Configmap key: port**

**ENV name: SPRING\_PROFILES\_ACTIVE**

**Configmap key: profile**

**Commands:**

kubectl –n demo create cm appconfig2 --from-env-file= /tmp/application.properties

kubectl –n demo run app2 --image=tomcat –dry-run=client –o yaml > /tmp/app2.yaml

vi /tmp/app2.yaml – edit file add following contents :

env:

-name: SERVER\_PORT

valueFrom:

configMapKeyRef:

name: appconfig2

key: port

-name: SPRING\_PROFILES\_ACTIVE

valueFrom:

configMapKeyRef:

name: appconfig2

key: profile

kubectl create –f /tmp/app2.yaml

kubectl –n demo get po

**Practical Scene 4:**

**In demo ns, create secret DB with DB\_USERNAME=root, DB\_PASSWORD=mypass**

**Create app3 pod with img tomcat in ns with request**

**CPU=100m, memory=256Mi and limits CPU=200m, memory=512Mi**

**Mount the database credentials to the same pod as env variable**

**Commands:**

kubectl –n demo create secret generic database --from-literal= DB\_USERNAME=root --from-literal= DB\_PASSWORD=mypass

kubectl –n demo run app3 --image=tomcat –requests=’cpu=100m,memory=256Mi’ –limits=’ CPU=200m,memory=512Mi ‘ –dry-run=client –o yaml > /tmp/app3.yaml

vi /tmp/app3.yaml – edit file add following contents:

envFrom:

-secretRef:

Name:database

-name: SERVER\_PORT

**Practical Scene 5:**

**Update DB secret created in the previous step with new DB password:mynewpass. And make sure app3 pod has the new DB credentials in env variables**

**Commands:**

echo –n mynewpass | base64

kubectl –n demo edit secret database

Practical Scene 6:

Create a docker-registery secret in demo ns using Docker repository details

Docker Server : myregistery.azurecr.io

Docker Username : myregistery

Docker Password : 1qaz2wsx

Docker Email : [myregistry@company.com](mailto:myregistry@company.com)

**Create yaml of a pod named custompod in /tmp/custompod .yaml in the same namespace with the image myregistery.azurecr.io/customimage using docker credentials in docker-registry secret**

**Commands:**

kubectl create secret docker-registry docker-registry –h

kubectl create secret docker-registry docker-registry

**K8 domain – Multi Container Pods**

Kubectl run nginx –image nginx –port 80 –dry-run=client –o yaml > nginx.yaml

Multi-container-pods -- kubectl get po nginx

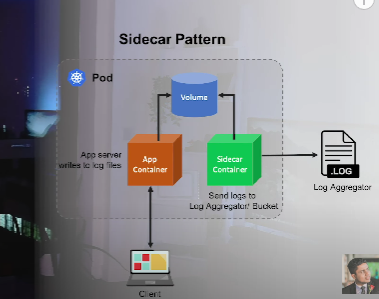
Multi-container-pods

Multi-container-pods --kubectl logs nginx –c busybox

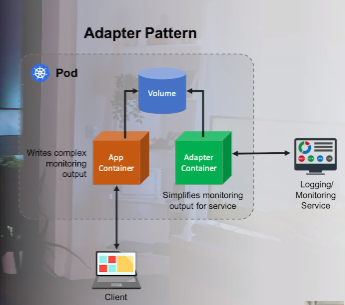
**Multi-container design patterns**

**Sidecar**: It is main application plus helper conatiner with responsibility that is essential to your application.

Ex - logging utilities, sync services, watchers, montoring agents



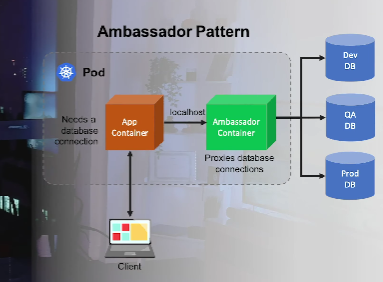
**Adapter:** It is used to standardize and normalize application. Output or monitoring data for aggregation.



**Ambassador:** It is useful to connect containers with the outside world .Proxy connections to different environments depending upon cluster’s needs.

Provides access to the database, while deploying locally you will use local database

To test and prod deployments wants different databases



**Init Containers**

Pod can have multiple containers running apps with it, but also have one or more init Containers, which are run before the app containers are started. Init containers are exactly like regular containers:

**Kubernetes Domain: Pod design – rc, rs, deployments, stateful sets, jobs, cron jobs**

**Replication Controller** (RC): It ensures that specified no of pod replicas are running at any time. Rc makes pod is always up and available.

If there are too many pods, then rc terminates the extra pods. If there are too few, then RC starts more pods. Reduce manual work to create pod

**Replica Sets** (RS): It is used to maintain a stable set of replica pods running at given time. It is used to guarantee the availabity of a specified no of identical pods.

Rs uses pod template to create new pods

**To scale replicas sets following commands used:**

kubectl scale –replicas=6 replicaset nginx

kubectl scale –replicas=6 –f rc.yaml

kubectl get rs

kubectl get replicaset

kubectl replace –f nginx.yaml

**Deployments:** It is provides declarative updates for pods and replicasets. We can create a deployment to rollout an rs. Rollback to an earlier deployment revision.

To scale up the deployment to facilitate more load. We can pause the deployment, use the status of the deployment and cleanup old replicasets

kubectl create deployment nginx –image=nginx –replicas=3 –port=80 –dry-run=client –o yaml > deployment.yaml

kubectl get deployments

kubectl get deploy

kubectl rollout status deployment/nginx

kubectl set image deployment/nginx-deployment nginx=nginx:1.17 --record

kubectl edit deployment nginx –record

kubectl create –f deployment.yaml --record

kubectl rollout history deploy/nginx

kubectl rollout history deploy/nginx --revision=2

kubectl rollout undo deploy/nginx

kubectl rollout undo deploy/nginx --to–revision=1

kubectl rollout pause deploy nginx

kubectl rollout resume deploy nginx

**StatefulSets:**

Workload API object used to manage stateful applications. It manages the deployment and scaling of set pods, provides guarantees about the ordering and uniqueness of these pods

Like deployment – StatefulaSet manages pods that are based on an identical conatiner spec.

Statefulsets maintains a sticky identity for each of their pods

**Kubernetes jobs:**

Job creates one or more pods and will continue to retry execution of the pods until a specified no of them successfully terminate. As pods successfully complete, the job tracks the successful completions.

Simple case to create one job object in order to reliably run one pod to completion.

The job object will start a new pod if the pod fails or deleted. You can also use a job to run multiple pods in parallel.

kubectl create job math-add-job --image=Ubuntu – expr 3 + 2

job kubectl create –f job.yaml

job kubectl get jobs

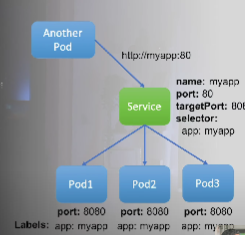
job kubectl get po

job kubectl logs job-name

kubectl create cronjob busybox –image =busybox –schedule=”\*/1 \* \* \* \*” -- /bin/sh -c ‘date; echo Hello from the Kubernetes cluster ’

**Kubernetes (K8) Domain – Services and Networking**

**Service:** It is logical abstraction in a cluster, it enables a grp of pods, which provide specific functions like web services, and image processing assigned a name and unique IP address.

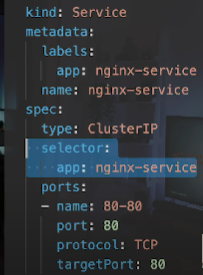


**K8 DNS**: It schedules a DNS pod and service on the cluster and configure the kubelet to tell individual container to use DNS service to resolve DNS names

<svc-name>.<namespace>.svc.cluster.local – to run the services in another namespaces

**ClusterIP Service**

kubectl create service clusterip nginx-service –tcp=80:80 –dry-run=client –o yaml > nginx-servic.yaml



kubectl create –f clusterip-nginx-service.yaml

kubectl get endpoints

kubectl get ep

kubectl run busybox –image=busybox –rm –it –restart=Never -- wget -O- http://nginx-service

kubectl run redis --image-redis --port=6379 –expose

kubectl expose deployment nginx –name=nginx-service --type=ClusterIP –port=80 --target-port=80

**Nodeport** is service built on the top of the clusterip. Used to connect internal services to the external services pod

**Create nodeport service using command:**

kubectl create service nodeport nginx –tcp=80:80 --node-port=30080

Kubernetes LoadBalancer is on top of the nodeport service

kubectl create service loadbalancer my-lbs --tcp=5678:8080

**Kubernetes K8 ExternalName service – Database service**

It will give access to database server which runs or executes outside the k8 cluster

**K8 Ingress and Egress**

It that allows access to your k8 services from outside the k8 cluster



kubectl create –f my-ingress.yaml

kubectl get ingress

kubectl get ing

**Kubernetes (K8) Network Policy:**

kubectl create –f networkpolicy.yaml # create network policy

kubectl get networkpolicy

Host path volume mounts a file or directory from the host node’s file system into your pod

**Kubernetes (K8) Persistent Volumes** are administrator provisioned volumes. They created with a particular file system, size, and identifying characteristics such as volume IDs and names.

kubectl create –f my-pv.yaml

kubectl apply –f my-pv.yaml

It has access modes: readwriteonce (), readonlymany (), readwritemany ()

**Reclaim Policy**

**PersistentVolumeClaim (PVC):** It is a request and claim to persistent volume resource

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for the pod

kubectl create –f my-pvc.yaml

# get persistent volume claims

kubectl get persistentvolumeclaim

kubectl get pvc

**Storage classes**: It provides a way for administrator to describe the “classes” of storage they offer.

kubectl create –f my scs.yaml

#Get storage class

kubectl get storageclass

kubectl get sc

**Kubernetes (K8) Observability domain and Container Probes**

**Liveness probe** = indicates the container is running. The policy is subjected to its restart policy

Service ----ready ---- liveness pass

Pod restarting --- liveness fail

httpGet

kubelet sends an http request to the specified path and port to perform the check.

HTTP probes have additional fields that can be set on httpset.

Tcpsocket

Exec

**Readiness probe:** Indicates whether the container is ready to respond to requests

Additional configurations in readiness probe

**Startup Probe:** Indicates whether the application within the container is started. All other probes are disabled if a startup probe is provided, until it succeeds.

**Kubernetes Logging**

kubectl logs nginx ----logs of pods

kubectl logs nginx –c nginx-1 ----- check logs of pod with multiple containers

kubectl logs –f nginx --- you can stream the logs with –f flag

kubectl logs –p nginx – return logs of previous terminated pod

kubectl logs –since=1m nginx ---- return the logs from pod written in last minute

kubectl logs job/hello ---- check logs of a job

kubectl logs deployment/nginx ---- check logs of a random pod of a deployment

kubectl top node

kubectl top pod

kubectl get hpa nginx

kubectl delete hpa nginx

Kubernetes Troubleshooting

<https://learnk8s.io/a/a-visual-guide->..

<https://kubernetes.io/docs/tasks/debug/>

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

<https://medium.com/@talhakhalid101/common-kubernetes-errors-made-by-beginners-274b50e18a01>

