1.Kubernetes minimum requirement

2 Core

4 GB RAM

2.kubernetes not recommend to on the swap so begining itself we will off the swap

#swapoff -a

install the kubeadm,kublet,kubectl and docker

start the docker service

#systemctl enable docker.service

#systemctl start docker.service

refer the below link

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#check-required-ports

3.kubeadm initialization

sudo kubeadm init --apiserver-advertise-address <server address> --pod-network-cidr-172.16.0.0/16(pod network range)

List of nodes through kubectl

#kubectl get nodes

Deploying Flannel with kubectl

#kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml

Describe the kubernetes node

kubectl descrie node <node\_name>

Master IP - 172.16.10.14

Node IP - 172.16.10.29

List the kubernets resources

#kubectl get nodes

#kubectl get deploy

#kubectl get svc

#kubectl get pods

#kubectl help

Application deployment:

#kubectl apply -f <filename.yaml>

#kubectl delete pod <pod\_name>

#kubectl edit deploy <deployment\_name>

#kubectl get deploy <deployment\_name> -o yaml

#kubectl get pods -o wide --> pod running in which server

directly run the kubernetes without yaml file

#kubectl run httpd --image=httpd --replicas=1 --port=80 #but not roleback the option

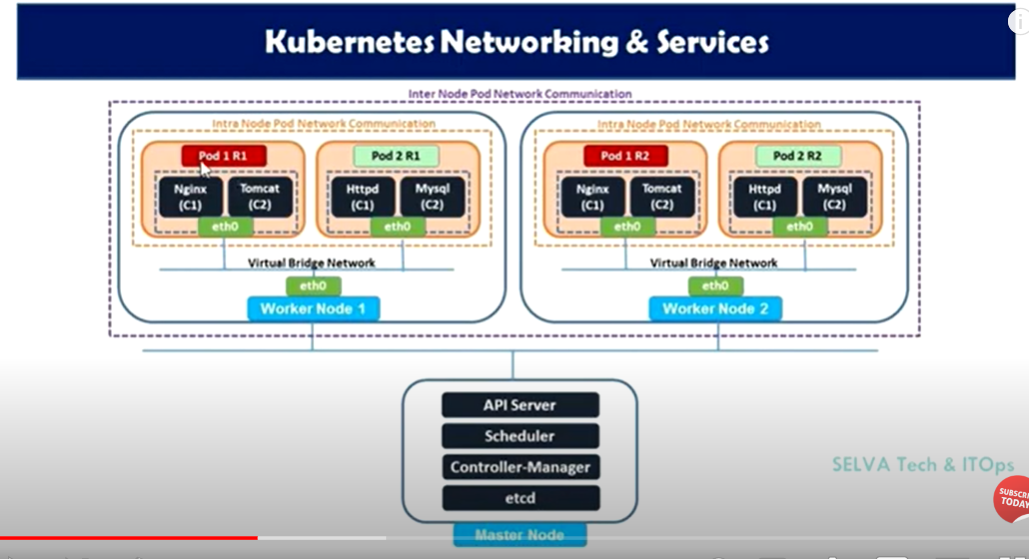
How to go to pod

#kubectl exec -it <pod\_name> /bin/bash

Server side validation of any errors

#kubectl apply -f <your-deployment-file>.yaml --validate=true

Kubernetes networking types



1.Container to container communication

2.Pod to Pod communication

1.Intra-node pod network

2.Inter-node pod network

3.Pod to service communication

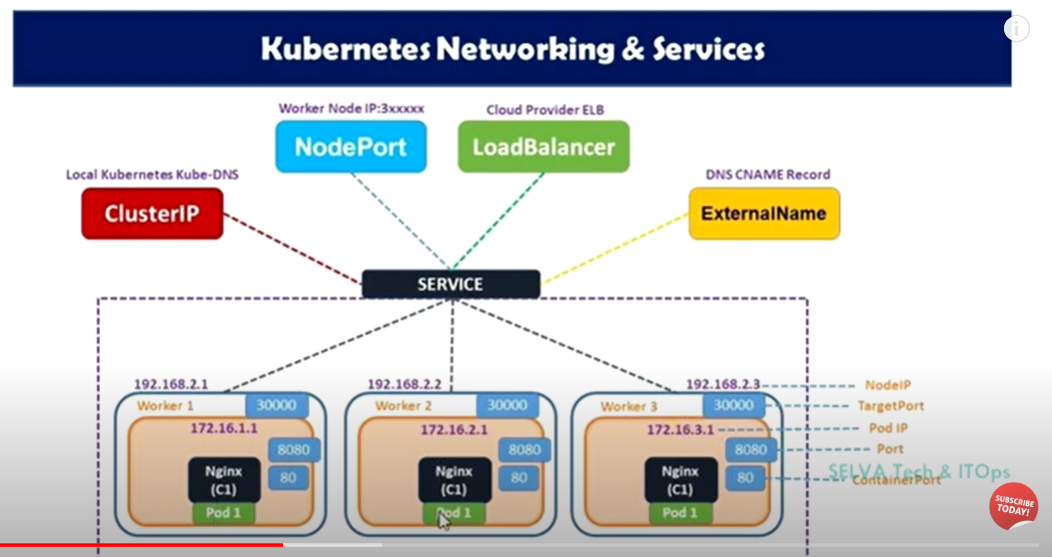
4.External to Service communication

1.Cluster ip

2.NodePort

3.LoadBalancer

4.Extername



Namespace:

Kubernetes namespaces are a way to logically partition a Kubernetes cluster into virtual sub-clusters. This allows for better organization, isolation, and security within a shared cluster environment

Commands

List out the Kubernetes namespace

#kubectl get ns

#Kubectl -n kube-system get pods 🡪 for pods

#kubectl -n default get svc 🡪 for service

#kuectl -n default get deploy

How to create the namespace in Kubernetes

Ns.yaml

apiVersion: v1

Kind: Namespace

Metadata:

name: develop

#kubectl apply -f Ns.yaml

#kubectl get ns

Kubernetes secrets:

Reference link: https://kubernetes.io/docs/concepts/configuration/secret/

Maintain the credential,ssl certificates,tokens in external resources that called as Kubernetes secrets.

It will create two types

1.create the secretes through yaml file(recommended also maintain through versoning)

2.directly through kubectl run command

List out the secrets in kuberentes:

#kubectl get secret

Examples of Kubernetes secretes:

apiVersion: v1

Kind: Secret

metadata:

name: mysql-cred

type: Opaque

data:

MYSQL\_ROOT\_PASSWORD: <password>

MYSQL\_USER: <mysql\_username>

MYSQL\_PASSWORD: <mysql\_password>

It wont support plain text format so for username and password need to change the base64 format

#echo -n “<mysql\_username> | base64

Then generate the value then placed in MYSQL\_USER

#echo -n “<mysql\_password> | base64

Then generate the value then placed in MYSQL\_PASSWORD

**Decrypt the value in secrets:**

#kubectl get secret <secret name> -o jsonpath=’{.data.MYSQL\_USER}’ | base64 –decode

How to call the secrets in kubernets deployment

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app: mariadb

name: mariadb-deployment

spec:

replicas: 1

selector:

matchLabels:

app: mariadb

template:

metadata:

labels:

app: mariadb

spec:

containers:

- name: mariadb

image: docker.io/mariadb:10.4

**env: 🡪 define single variable from secret**

**- name: MYSQ\_ROOT\_PASSWORD**

**valueFrom:**

**secretKeyRef:**

**name: mysql-cred(secret name)**

**key: MYSQL\_ROOT\_PASSWORD**

**or**

**envFrom: 🡪 define secrets from secret key**

**- secretRef:**

**name: mysql-cred**

**Sometimes we will maintain the secrets file using volume**

**spec:**

**replicas: 1**

**selector:**

**matchLabels:**

**app: mariadb**

**template:**

**metadata:**

**labels:**

**app: mariadb**

**spec:**

**containers:**

**- name: mariadb**

**image: docker.io/mariadb:10.4**

**env:**

**- name: MYSQ\_ROOT\_PASSWORD**

**valueFrom:**

**secretKeyRef:**

**name: mysql-cred(secret name)**

**key: MYSQL\_ROOT\_PASSWORD**

**volumeMounts:**

**- name: newsecret**

**mountPath: "/etc/newsecret"**

**readOnly: true**

**volumes:**

**- name: newsecret**

**secret:**

**secretName: mysql-cred**

ConfigMap:

Reference link:https://kubernetes.io/docs/concepts/configuration/configmap/

List out the configmap

#kubectl get cm or kubectl get configmap

A ConfigMap is an API object that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a spec, a ConfigMap has data and binaryData fields. These fields accept key-value pairs as their values. Both the data field and the binaryData are optional. The data field is designed to contain UTF-8 strings while the binaryData field is designed to contain binary data as base64-encoded strings.

Nano mysql-extra.conf

[mysqld]

Max\_allowed\_packets = 64M

Then create configmap and map the configuration file

#kubectl create cm mysql-extra –from-file =mysql-extra.conf

#kubectl get cm

#kubectl describe cm mysql-extra

Configuration file:

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app: mariadb

name: mariadb-deployment

spec:

replicas: 1

selector:

matchLabels:

app: mariadb

template:

metadata:

labels:

app: mariadb

spec:

containers:

- name: mariadb

image: docker.io/mariadb:10.4

env:

- name: MYSQ\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-cred(secret name)

key: MYSQL\_ROOT\_PASSWORD

**volumeMounts:**

**- name: newcm**

**mountPath: /etc/mysql/conf.d**

**readOnly: true**

**volumes:**

**- name: newcm**

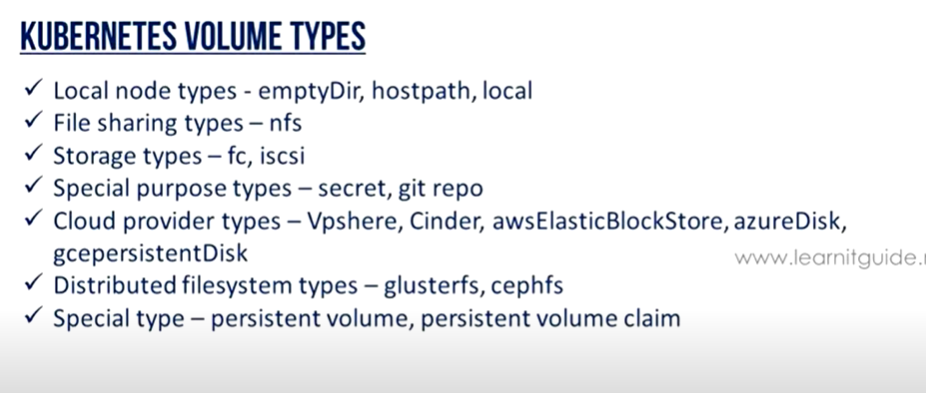
**configMap:**

**name: mysql-extra**

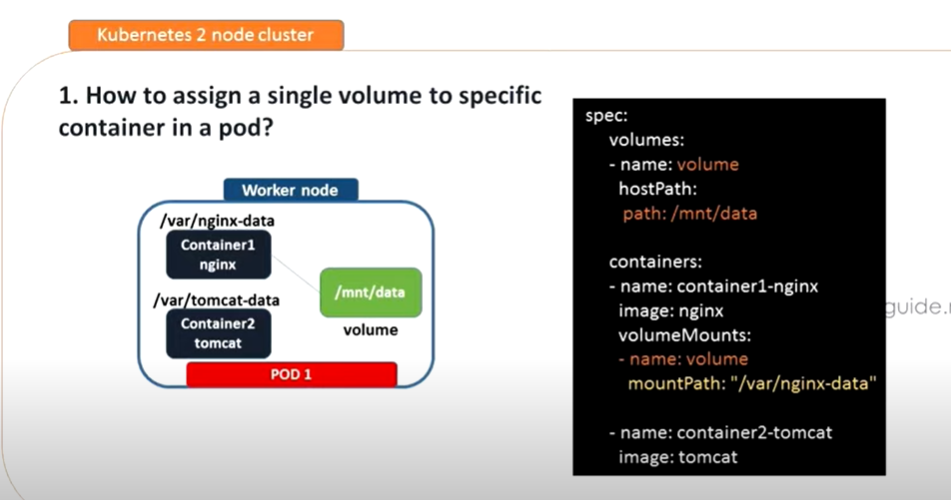
**Kubernetes volumes:**

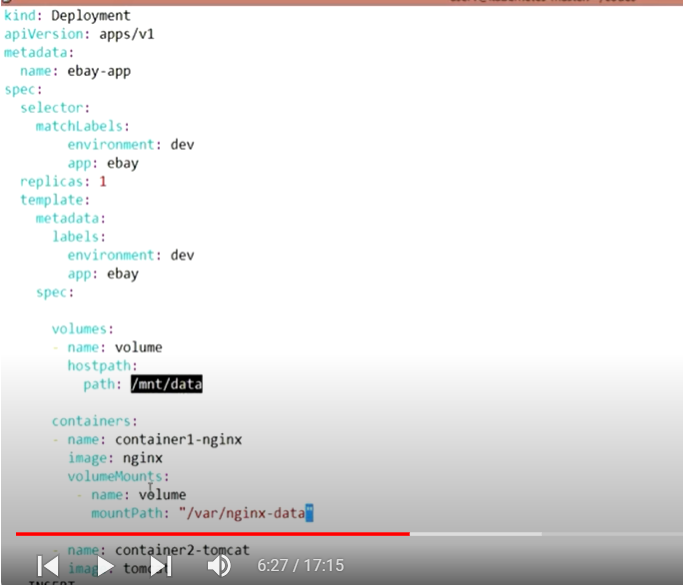
Reference link: https://kubernetes.io/docs/concepts/storage/volumes/

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a Pod and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested.

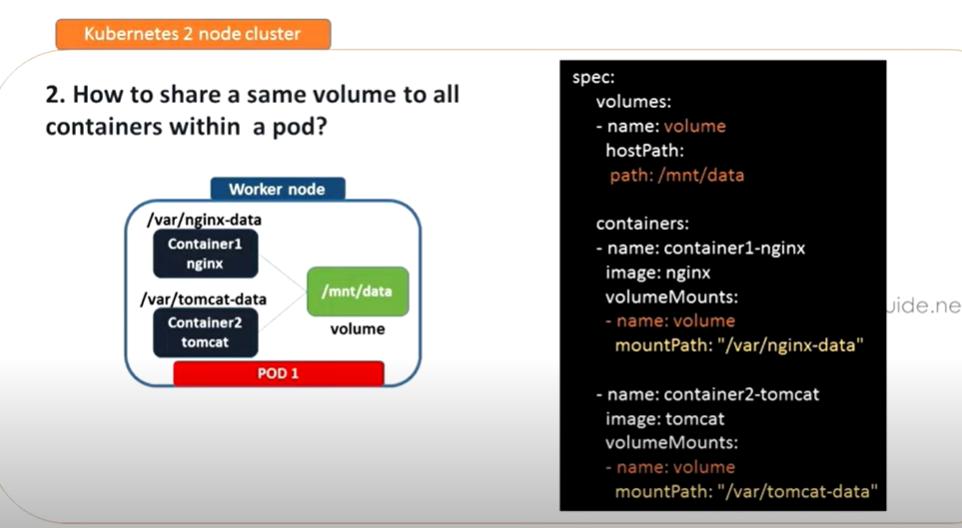


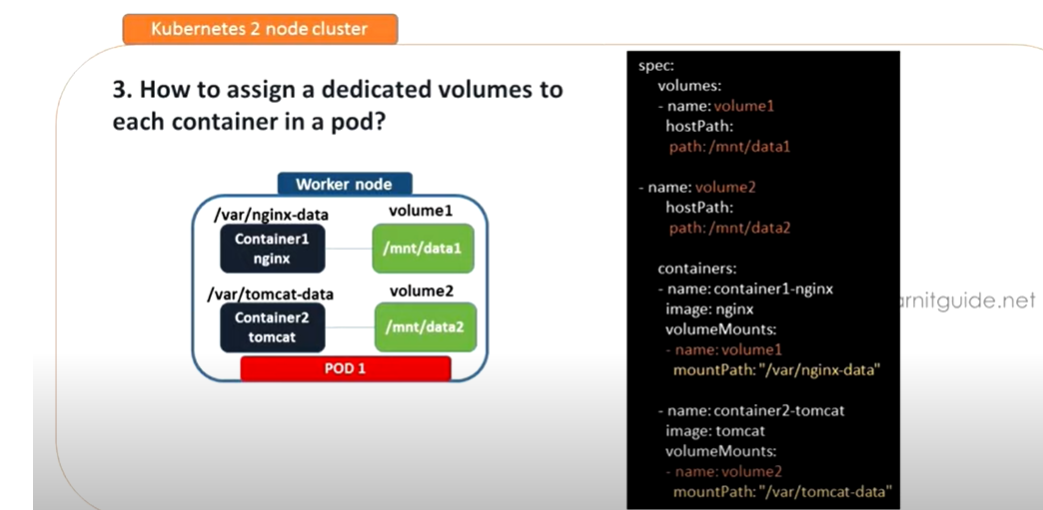
**Single volume to specific contianer**

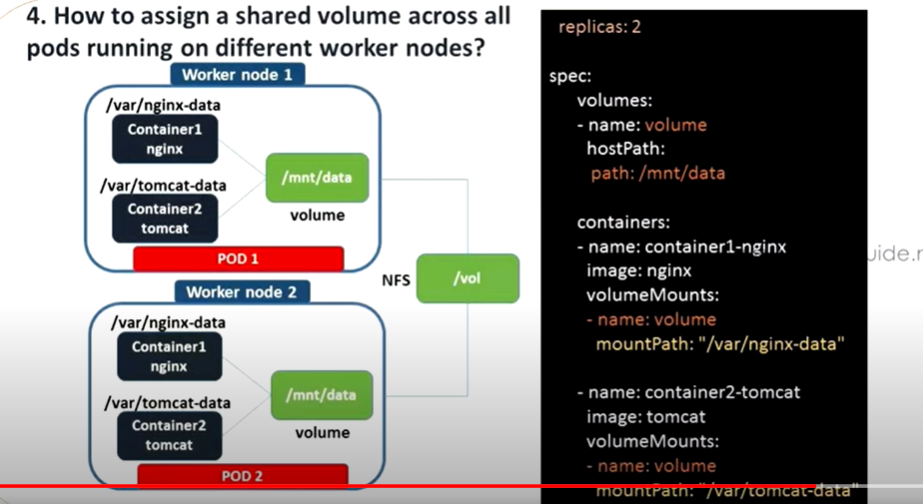




**Share volume to multiple container with pod:**







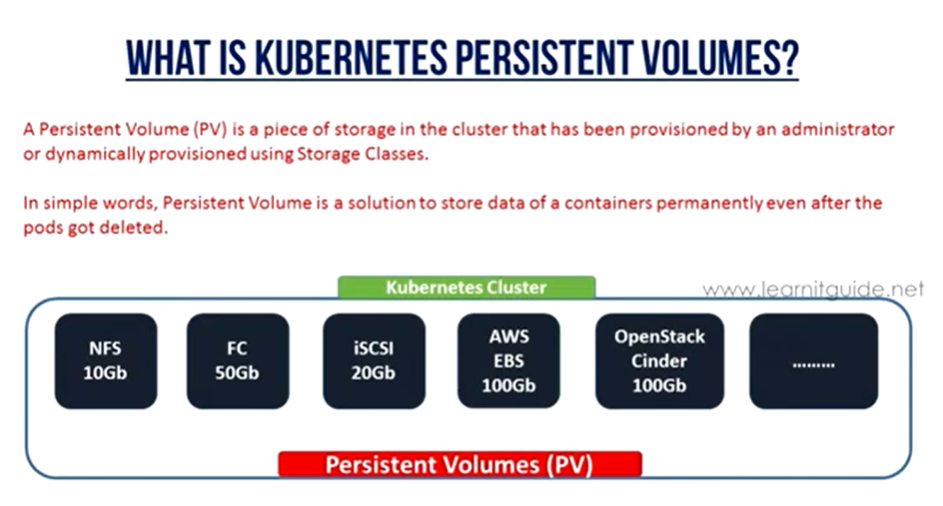
**Kubernetes persistent volume and persistent volume claim**

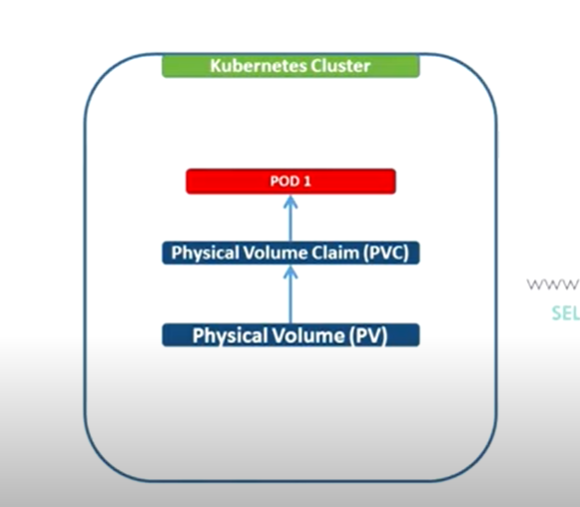
**Reference link:** **https://kubernetes.io/docs/concepts/storage/persistent-volumes/**

Managing storage is a distinct problem from managing compute instances. The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: PersistentVolume and PersistentVolumeClaim.

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany, ReadWriteMany, or ReadWriteOncePod, see AccessModes).





Ebay\_pv.yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: ebay-pv

spec:

capacity:

storage: 20Gi

volumeMode:

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Recycle

storageClassName: **ebaystorage**

mountOptions:

- nfsvers=4.1

nfs:

path: /nfsdata

server: <nfs\_server\_ip>

once deploy the yaml file it will create the pv in Kubernetes

#kubectl apply -f Ebay\_pv.yaml

#kubectl get pv

Then create pvc with below examples:

Pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name:myclaim

spec:

storageClassName: **ebaystorage**

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 2G

#kubectl apply -f Pvc.yaml

#kubectl get pvc

While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the StorageClass resource.