# What is Kubernetes ?

It is a tool for container Orchestration. That means managing & Controlling multiple docker containers as a single service.

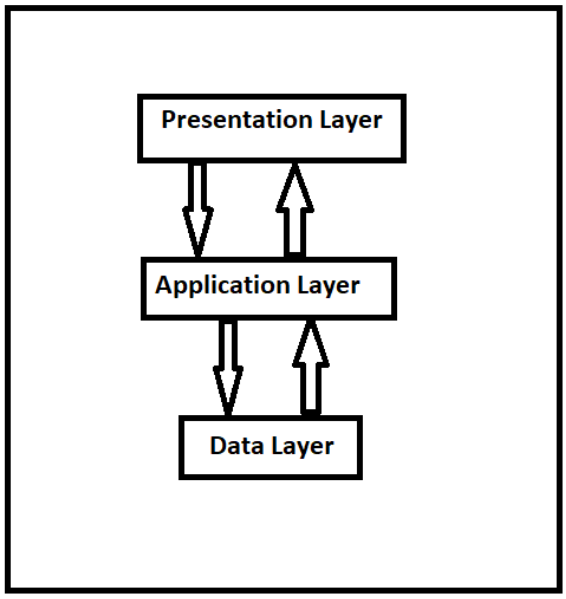
Also it is a tool that automates the deployment, Management, Scaling, Networking and availability of container based applications.

In other words you can cluster together groups of hosts running linux containers and kubernetes helps you easily and efficiently manage those clusters. Kubernetes clusters can span across on-premise, Public Cloud, Private or Hybrid Clouds. For this reason Kubernetes is an ideal platform to host cloud-native applications that required rapid scaling etc.

- Kubernetes was developed by google labs and later donated to CNCF ( Cloud native computing Foundation.
- Kubernetes is the greek word for captian of ship.
- Kubernetes is also refered as K8'S as there are 8 characters between K & S.

In Real world Applications are develop using Monolithic Architecture & Microservices Architecture.

**Monolithic Architecture :**



It Consists of 3 Layers, SO its called as 3 tier Architecture. A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means composed all in one piece, Which means all these layers are packaged & deployed together as single unit and that's the reason its called as Monolithic Architecture.

**Advantages :**

1)Easy to Develop, Test and Deploy
2)Greater Compatability

**Disadvantages :**

1)Very Large ( If one component has issues it will impact everything )
2)Locked in with initial decisions and Technology.
3)Implemented using single development stack ( We have to use the same Technology to develop all the other components )
4)Frequent deployments are not possible
5)Need to scale an entire application stack

**What are microservices?**

Microservices (or microservices architecture) are a cloud native architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services.

These services typically have their own technology stack, inclusive of the database and data management model; communicate with one another over a combination of REST APIs, event streaming, and message brokers; and are organized by business capability, with the line separating services often referred to as a bounded context.

While much of the discussion about microservices has revolved around architectural definitions and characteristics, their value can be more commonly understood through fairly simple business and organizational benefits:

Code can be updated more easily - new features or functionality can be added without touching the entire application
Teams can use different stacks and different programming languages for different components.
Components can be scaled independently of one another, reducing the waste and cost associated with having to scale entire applications because a single feature might be facing too much load.
Microservices might also be understood by what they are not. The two comparisons drawn most frequently with microservices architecture are monolithic architecture and service-oriented architecture (SOA).

The difference between microservices and monolithic architecture is that microservices compose a single application from many smaller, loosely coupled services as opposed to the monolithic approach of a large, tightly coupled application.

We can run these microservices on Physical Machines | Virtual Machines | Containers. It is always recommended to host the containers on Virtual Environment. Containers virtualize at the O/S Level where as Virtual Machines virtualize at the hard ware level. So, If we host our containers on Virtual environment we get the best of both worlds like virtualizing at the hardware level and O/S Level.

**What is container orchestration?**

Container orchestration automates the deployment, management, scaling, and networking of containers. Enterprises that need to deploy and manage hundreds or thousands of Linux® containers and hosts can benefit from container orchestration.

Container orchestration can be used in any environment where you use containers. It can help you to deploy the same application across different environments without needing to redesign it. And microservices in containers make it easier to orchestrate services, including storage, networking, and security.

Containers give your microservice-based apps an ideal application deployment unit and self-contained execution environment. They make it possible to run multiple parts of an app independently in microservices, on the same hardware, with much greater control over individual pieces and life cycles.

Managing the lifecycle of containers with orchestration also supports DevOps teams who integrate it into CI/CD workflows. Along with application programming interfaces (APIs) and DevOps teams, containerized microservices are the foundation for cloud-native applications.

**What is container orchestration used for?**

Use container orchestration to automate and manage tasks such as:

- Provisioning and deployment
- Configuration and scheduling
- Resource allocation
- Container availability
- Scaling or removing containers based on balancing workloads across your infrastructure
- Load balancing and traffic routing
- Monitoring container health

- Configuring applications based on the container in which they will run
- Keeping interactions between containers secure

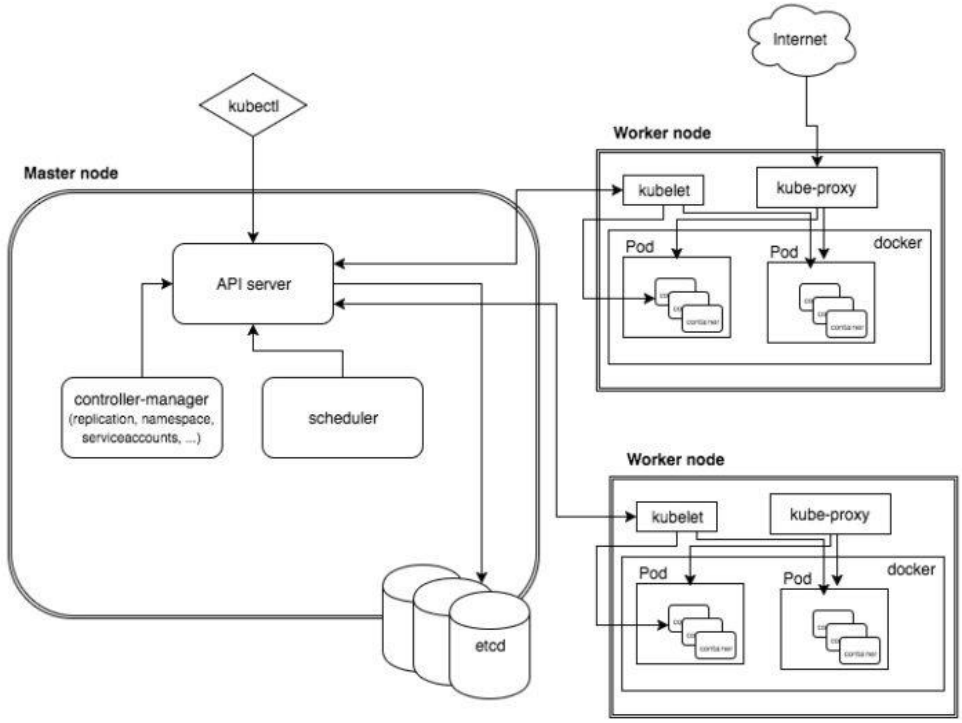
**Container orchestration tools :**

Container orchestration tools provide a framework for managing containers and microservices architecture at scale. There are many container orchestration tools that can be used for container lifecycle management. Some popular options are Kubernetes, Docker Swarm, and Apache Mesos.

Kubernetes is an open source container orchestration tool that was originally developed and designed by engineers at Google. Google donated the Kubernetes project to the newly formed Cloud Native Computing Foundation in 2015.

**Kubernetes Components and Architecture :**

Kubernetes follows a client-server architecture. It's possible to have a multi-master setup (for high availability), but by default there is a single master server which acts as a controlling node and point of contact. The master server consists of various components including a kube-apiserver, an etcd storage, a kube-controller-manager, a kube-scheduler, and a DNS server for Kubernetes services. Worker Node components include kubelet and kube-proxy on top of Docker.



# Master Node Components

Below are the main components found on the master node:

**kube-apiserver** – Kubernetes API server is the central management entity that receives all REST requests for modifications (to pods, services, replication sets/controllers and others), serving as frontend to the cluster. Also, this is the only component that communicates with the etcd cluster, making sure data is stored in etcd and is in agreement with the service details of the deployed pods.

**kube-controller-manager**

Control Plane component that runs controller processes.
Logically, each 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 | Replication Controller :**

Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.

**Endpoints controller :**

Populates the Endpoints object (that is, joins Services & Pods).

**Service Account & Token controllers :**

Create default accounts and API access tokens for new namespaces

**kube-scheduler** – helps schedule the pods (a co-located group of containers inside which our application processes are running) on the various nodes based on resource utilization. It reads the service's operational requirements and schedules it on the best fit node. For example, if the application needs 1GB of memory and 2 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. The scheduler runs each time there is a need to schedule pods. The scheduler must know the total resources available as well as resources allocated to existing workloads on each node.

**etcd cluster** – a simple, distributed key value storage which is used to store the Kubernetes cluster data (such as number of pods, their state, namespace, etc), API objects and service discovery details. It is only accessible from the API server for security reasons. etcd enables notifications to the cluster about configuration changes with the help of watchers. Notifications are API requests on each etcd cluster node to trigger the update of information in the node's storage.

**Considerations for large clusters :**

A cluster is a set of nodes (physical or virtual machines) running Kubernetes agents, managed by the control plane. Kubernetes v1.21 supports clusters with up to 5000 nodes. More specifically, Kubernetes is designed to accommodate configurations that meet all of the following criteria:

No more than 100 pods per node
No more than 5000 nodes
No more than 150000 total pods
No more than 300000 total containers
You can scale your cluster by adding or removing nodes. The way you do this depends on how your cluster is deployed

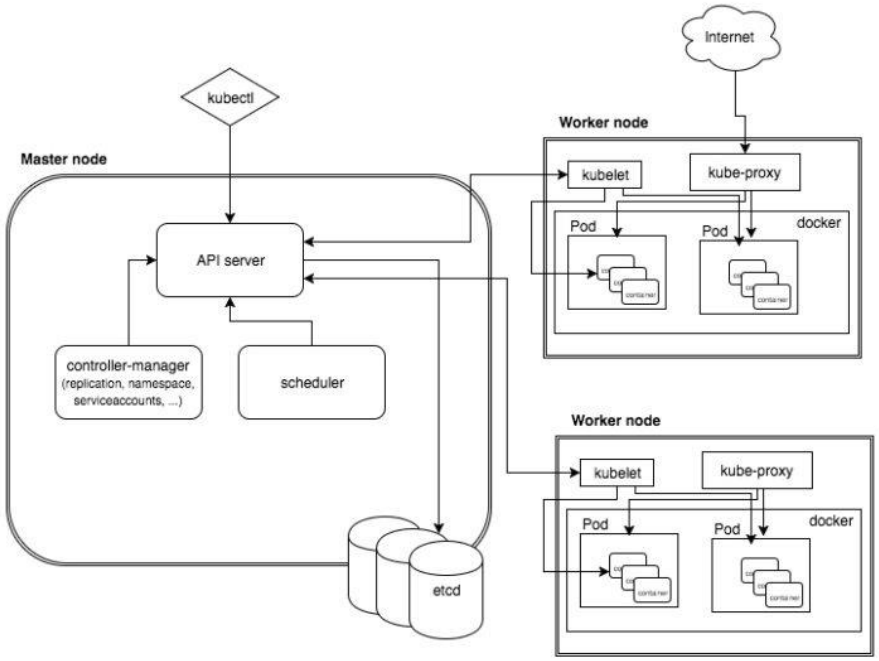
**What is a Pod?**

Pods are the smallest, most basic deployable objects in Kubernetes. A Pod represents a single instance of a running process in your cluster.

Pods contain one or more containers, such as Docker containers. When a Pod runs multiple containers, the containers are managed as a single entity and share the Pod's resources. Generally, running multiple containers in a single Pod is an advanced use case.

Pods also contain shared networking and storage resources for their containers.

# Worker Node Components




**Kubelet :**

The Kubelet is one of the most important components in Kubernetes. Basically, it's an agent that runs on each node and is responsible for watching the API Server for pods that are bound to its node and making sure those pods are running (it talks to the Docker daemon using the API over the Docker socket to manipulate containers lifecycle). It then reports back to the API Server the status of changes regarding those pods.

The main Kubelet responsibilities include:

- Run the pods containers.
- Report the status of the node and each pod to the API Server.
- Run container probes.
- Retrieve container metrics from cAdvisor, aggregate and expose them through the Kubelet Summary API for components (such as Heapster) to consume.

**Kube Proxy :**

The Kube Proxy runs on each node and is responsible for watching the API Server for changes on services and pods definitions to maintain the entire network configuration up to date, ensuring that one pod can talk to another pod, one node can talk to another node, one container can talk to another container, and so on. Besides, it exposes Kubernetes services and manipulates iptables rules to trap access to services IPs and

redirect them to the correct backends (that's why you can access a NodePort service using any node IP; even if the node you hit is not the one you are looking for, this node will already be set up with the appropriate iptables rules to redirect your request to the correct backend). This provides a highly-available load-balancing solution with low performance overhead.

# What is Minikub

Minikube is a utility you can use to run Kubernetes (k8s) on your local machine. It creates a single node cluster contained in a virtual machine (VM). This cluster lets you demo Kubernetes operations without requiring the time and resource-consuming installation of full-blown K8s.

All you need is Docker (or similarly compatible) container or a Virtual Machine environment, and Kubernetes is a single command away: minikube start.

**Recommended Min Hardware Config Required :**
2 CPUs or more
2GB of free memory
20GB of free disk space
Internet connection
Container or virtual machine manager, such as: **Docker, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox, or VMWare**

**Kubectl :**

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For more information including a complete list of kubectl operations, see the kubectl reference documentation.

**kubectl is installable on a variety of Linux platforms, macOS and Windows**.

**Steps to Config Kubernetes Cluster using MINIKUBE :**

1)Install Virtual Box

2)Install KUBECTL

3)Install MINIKUBE

4) Start MINIKUBE as Non Root User

**5)** Ensure these all we are doing on Base Operating system **( Note on Guest OS )**

**Installing Virtual Box on Base OS :**

#yum install wget vim –y **( Installing wget & Vim softwares )**
#wget https://download.virtualbox.org/virtualbox/rpm/rhel/virtualbox.repo -P /etc/yum.repos.d/ **(Configuring Virtual Box Repository )**

#yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm **( Configuring EPEL Repository )**

#yum update

#yum install binutils kernel-devel kernel-headers libgomp make patch gcc glibc-headers glibc-devel dkms -y

#Yum install VirtualBox-6.1 **( Installing Virtual Box )**
#which virtualbox
#reboot
#/sbin/vboxconfig **( Stopping & STarting the Virtual Box Services )**
#systemctl status vboxconfig **( Verify Virtual box Service is Runnin )**

**Installing KUBECTL :**

#curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl

#chmod +x ./kubectl

#mv ./kubectl /usr/local/bin/kubectl **(Move the binary in to your PATH )**

**Best to ensure the version you installed is up-to-date:**
#kubectl version --client

**Installing MINIKUBE :**
#curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube
#mkdir -p /usr/local/bin/
#Install minikube /usr/local/bin/
$minikube version

$minikube status
**Confirm Installation : (From here you need to do with non root user)**
$minikube start --driver=<driver_name>
virtualbox | vmware ( As per your Hypervisor )

**NOTE :**
1)By default Minikube used 2GB of Ram, 2 V CPU & 20GB of Disk space by default. It is possible to create MiniKube VM with our custom required Hardware configurations
$minikube --memory 8192 --cpus 6 start

**After Installing Test it :**
$minikube status **( Displays the status of MiniKube )**
$kubectl config view **( Displays Kubernetes Cluster configuration Information )**
$kubectl get no **( Displays the nodes available in the Cluster along with its state )**
$kubectl get po **( Displaying the pods Created )**
$kubectl delete po <POD NAME> **( Deleting the Specific POD )**
$minikube stop **( Stopping the Minikube VM )**
$minikube delete **( Deleting the MiniKube VM )**
$minikube status **( Verify the VM is Deleted or Not )**

## Setting up Kubernetes Cluster using "KUBEADM" – Version 1.24

Docker as an underlying runtime is being deprecated in favor of runtimes that use the Container Runtime Interface (CRI) created for Kubernetes. Docker-produced images will continue to work in your cluster with all runtimes, as they always have.

If you're an end-user of Kubernetes, not a whole lot will be changing for you. This doesn't mean the death of Docker, and it doesn't mean you can't, or shouldn't, use Docker as a development tool anymore. Docker is still a useful tool for building containers, and the images that result from running docker build can still run in your Kubernetes cluster.

If you're using a managed Kubernetes service like AKS, EkS or GKE, you will need to make sure your worker nodes are using a supported container runtime before Docker support is removed in a future version of Kubernetes. If you have node customizations you may need to update them based on your environment and runtime requirements. Please work with your service provider to ensure proper upgrade testing and planning.

If you're rolling your own clusters, you will also need to make changes to avoid your clusters breaking. At v1.20, you will get a deprecation warning for Docker. When Docker runtime support is removed in a future release (currently planned for the 1.22 release in late 2021) of

Kubernetes it will no longer be supported and you will need to switch to one of the other compliant container runtimes, like containerd or CRI-O.

Why CRI-O ?
CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes.

It is a lightweight alternative to using Docker, Moby or rkt as the runtime for Kubernetes.

CRI-O is a CRI runtime mainly developed by Red Hat folks. In fact, this runtime is used in Red Hat OpenShift now. Yes, they do not depend on Docker anymore.

Interestingly, RHEL 7 does not officially support Docker either. Instead, they provide Podman, Buildah and CRI-O for container environment.

Why CRI-O ?
CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes.

It is a lightweight alternative to using Docker, Moby or rkt as the runtime for Kubernetes.

CRI-O is a CRI runtime mainly developed by Red Hat folks. In fact, this runtime is used in Red Hat OpenShift now. Yes, they do not depend on Docker anymore.

Interestingly, RHEL 7 does not officially support Docker either. Instead, they provide Podman, Buildah and CRI-O for container environment.

## Steps to Install Kubernetes CLuster using KUBEADM

we will have 1 Master node and 3 Worker Nodes ( Total Nodes : 4 )

**On all Nodes :**

1) Ensure to have 4 VM's installed with Centos 7 OS.

2) Recommended to have all virtual machines with min 2 CPU & more than 2GB of RAM

3) Ensure swap, SELinux, firewall must be disabled on all the nodes which are a part of k8's cluster .

4) Ensure to install CRI-O runtime on all the nodes which are part of k8s cluster.

5) Ensure to install KUBEADM, KUBECTL, KUBELET on all the nodes which are a part of K8's cluster.

6) Ensure CRI-O & Kubelet service must be started on all the nodes which are part of K8's cluster.

8) Need to configure IP forwarding on all the nodes which are part of K8's cluster.

**Need to be done on Only Master node :**

9) Select any one node out of 4, Initialize as Master node

10) We need to configure the POD networking.

**Only on Worker Nodes :**

10) We need to configure the remaining 3 as worker nodes by joining them to the cluster

## ON ALL Nodes (Manager & Worker Nodes )

**Disabling SWAP :**
#swapoff -a
#swapon -s
**Note : Go to /etc/fstab and add a comment to swap partition for permanently disabling**

**Disable SE Linux :**
#sestatus
#getenforce 0 **( DIsabling Selinux )**

**To permanently disable SE Linux :**
#vi /etc/selinux/config
SELinux=disabled
:wq!

**Disabling Firewall :**
#systemctl disable firewalld
#systemctl stop firewalld
#reboot

**NOTE : If you don't disable SWAP, SELINUX & Firewall it leads to KUBELET Service failure.**

**Create a enviroment variable for OS.**
#export OS=CentOS_7
**Note: If you are using CentOS 8 then give variable name as CentOS_8**
#export VERSION=1.17

**NOTE : To make it permanent update the variables in /etc/environment , .profile and .bashrc files in the root users home directory.**

**Configuring REPO for cri-o :**

#curl -L -o /etc/yum.repos.d/devel:kubic:libcontainers:stable.repo https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/devel:kubic:libcontainers:stable.repo

#curl -L -o /etc/yum.repos.d/devel:kubic:libcontainers:stable:cri-o:$VERSION.repo https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cri-o:$VERSION/$OS/devel:kubic:libcontainers:stable:cri-o:$VERSION.repo

#cd /etc/yum.repos.d
#ls

#cd

**Installing CRI-O Runtime :**
#yum install cri-o -y

**Start and enable the cri-o service :**

#systemctl start cri-o
#systemctl enable cri-o
#systemctl status cri-o

**Enable Kernel Modules which are prerequisites for Kubernetes:**

#modprobe overlay
#modprobe br_netfilter

**To make it permanent :**
#cat /etc/modules-load.d/k8s.conf
overlay
br_netfilter
:wq!

**Configuring IP forwarding :**

#vi /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1

:wq!

**# Apply sysctl params without reboot#**
#sysctl --system

**Setting up Kubernetes Repo :**

#vi /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
:wq!

#yum install kubeadm kubectl kubelet -y
#systemctl start kubelet
#systemctl enable kubelet

**Only on the node you choose to config as Manager :**

#kubeadm init **( Initializes this node as Manager node and Cluster will be setup)**

**###To start using your cluster, you need to run the following as a regular user##**

```
#mkdir -p $HOME/.kube
#cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
#chown $(id -u):$(id -g) $HOME/.kube/config
```

Alternatively, if you are the root user, you can run:

```
#export KUBECONFIG=/etc/kubernetes/admin.conf
```

**Configuring POD Network :**

We need to install CNI plugin. So that pods can communicate with each other.
There are different CNI plugins for different purpose. Here we are using Weave Net CNI plugin.

#kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

#kubectl get nodes **( Lists all the available nodes in cluster )**

#kubectl get pods --all-namespaces **( Verify the POD network we just deployed is RUNNING or not )**

**Setting up Kubernetes Worker Node ( Only on Worker Nodes ) :**

Disabling SWAP :
#swapoff -a
#swapon -s
Note : Go to /etc/fstab and add a comment to swap partition for permanently disabling

**Disable SE Linux :**
#sestatus
#getenforce 0 **( DIsabling Selinux )**
**To permanently disable SE Linux :**
#vi /etc/selinux/config
SELinux=disabled
:wq!

**Disabling Firewall :**
#systemctl disable firewalld
#systemctl stop firewalld
#reboot

**NOTE : If you don't disable SWAP, SELINUX & Firewall it leads to KUBELET Service failure.**

**Create a enviroment variable for OS.**
#export OS=CentOS_7
**Note: If you are using CentOS 8 then give variable name as CentOS_8**
#export VERSION=1.17

**NOTE : To make it permanent update the variables in /etc/environment , .profile and .bashrc files in the root users home directory.**

**Configuring REPO for cri-o :**
#curl -L -o /etc/yum.repos.d/devel:kubic:libcontainers:stable.repo https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/devel:kubic:libcontainers:stable.repo

#curl -L -o /etc/yum.repos.d/devel:kubic:libcontainers:stable:cri-o:$VERSION.repo https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cri-o:$VERSION/$OS/devel:kubic:libcontainers:stable:cri-o:$VERSION.repo

#cd /etc/yum.repos.d
#ls
#cd
**Installing CRI-O RUntime :**
#yum install cri-o -y

**Start and enable the cri-o service :**
#systemctl start cri-o
#systemctl enable cri-o
#systemctl status cri-o

**Enable Kernel Modules which are prerequesites for Kubernetes:**
#modprobe overlay
#modprobe br_netfilter

**To make it permanent :**
#cat /etc/modules-load.d/k8s.conf
overlay
br_netfilter
:wq!

**Configuring IP forwarding :**

#vi /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
:wq!

**# Apply sysctl params without reboot**
#sysctl --system

**Setting up Kubernetes Repo :**
#vi /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
:wq!

#yum install kubeadm kubectl kubelet -y
#systemctl start kubelet
#systemctl enable kubelet

**Now this node is ready to join the cluster as Worker Node**

We need to copy the token from the manager node while we got from cluster initialization time, using the same token we can join this node to cluster

#kubeadm join 192.168.0.152:6443 --token 1we89d.de34tr34098de345 \
--discovery-token-ca-cert-hash sha256:d5f47rr6vb4h6y+6fc1g6y7fr14dd879 ( This is just for reference, You have to use the token you have got on your manager node )

**Go to Manager Node :**

Verify the Worker node has joined cluster or not
#kubectl get nodes

# Networking in Kubernetes

Kubernetes networking is an integral part of managing and making communication easier within a Kubernetes cluster. It manages a wide range of operations, such as:

- Handling internal container communication
- Exposing the containers to the internet

First, let's review the basic networking types in a Kubernetes cluster:

- Container-to-container networking
- Pod-to-pod networking
- Pod-to-service networking
- Internet-to-service networking

## Container-to-container networking :

In the most basic configurations, container to container communication within a single pod takes place via the localhost and port numbers. This is possible because the containers in the pod are located in the same network namespace.

A network namespace is a logical networking stack that includes routes, firewall rules, and network devices which provides a complete network stack for processes within the namespace. There can be multiple namespaces on the same virtual machine.

In Kubernetes, there is a hidden container called a pause container that runs on all the pods in Kubernetes. This container keeps the namespace open even if all the other containers in the pod are nonfunctional.

A new networking namespace is assigned to each pod created. Containers within the pod can use this networking namespace via localhost. However, a user must be aware of the port conflicts: If the containers use the same port, networking issues will arise within the containers.

**Pod-to-pod networking :**

Each pod in a Kubernetes node gets assigned a dedicated IP address with a dedicated namespace. Due to an automatic assignment, users do not need to explicitly create links between pods. This allows us to tread pod networking operations such as port allocation, load balancing, and naming similar to a virtual machine or a physical host.

Kubernetes imposes three fundamental requirements on any network.

1) Pods on a node can communicate with all pods on all nodes without NAT.
2) Agents on a node (system daemons, kubelet) can communicate with all the pods on that specific node.
3) Pods in a host network of a node can communicate with all pods on all nodes without NAT (Only for platforms such as Linux, which supports pods running on the host network)

**Communication between pods in the same node**

Kubernetes creates a virtual ethernet adapter for each pod, and it is connected to the network adaptor of the node.

When a network request is made, the pod connects through the virtual ethernet device associated with the pod and tunnels the traffic to the ethernet device of the node.

In a Kubernetes node, there is a network bridge called cbr0, which facilitates the communication between pods in a node. All the pods are a part of this network bridge. When a network request is made, the bridge checks for the correct destination (pod) and directs the traffic.

**Communication between pods in different nodes**

When the requested IP address cannot be found within the pod, the network bridge directs the traffic to the default gateway. This would then look for the IP within the cluster level.

Kubernetes keeps a record of all the IP ranges associated with each node. Then an IP address is assigned to the pods within the nodes from the range assigned to the node. When a request is made to an IP address in the cluster, it will:

1) Look for the IP range for the requested IP.
2) Then direct it to the specific node and then to the correct pod.

**Pod-to-service networking :**

A service is an abstraction that routes traffic to a set of pods. In other words, a service will map a single IP address to a set of pods.

When a network request is made, the service proxies the request to the necessary pod. This proxy service happens via a process called Kube-proxy that runs inside each node.

The service would get its IP within the cluster when a request first reaches the service IP before being forwarded to the actual IP of the pod. This is an important feature in Kubernetes—it decouples the dependency of networking directly to each pod. The pods can be created and destroyed without worrying about network connectivity. This is because the service will automatically update its endpoints when the pods change with the IP of the pod.

The service would get its IP within the cluster. Any request first reaches the service IP before being forwarded to the actual IP of the Pod. This is an important feature in Kubernetes as it decouples the dependency of networking directly to each Pod. Thus, pods can get created and destroyed without worrying about network connectivity as the service will automatically update its endpoints with the IP of the pod that it targets as the Pods changes.

A service knows which Pods to target using the label selector. The label selector is the core grouping primitive in Kubernetes; via a label selector, the client or user can identify a set of objects.

- The label selector will look for the specified labels in each Pod and match them with the service accordingly.

- Without the selector, properly configured services cannot keep track of the Pods and will lead to communication issues within the cluster.

**DNS for internal routing in a Kubernetes cluster**

Each cluster comes with inbuilt service for DNS resolution.

- A domain name is assigned to each service in the cluster
- A DNS name is assigned to the Pods.

(These can be manually configured via the YAML file using the hostname and subdomain fields.)
When a request is made via the domain name, the Kubernetes DNS service will resolve the request and point it to the necessary internal service from the service. The Kube-proxy process will point it to the endpoint pod.

**Internet-to-service networking :**

In the above sections, we have covered the internal networking basics of a Kubernetes cluster. The next step is to expose the containerized application to the internet.

Unlike the previous networking types, exposing the Kubernetes cluster to the internet depends on both the:

- Underlying cluster infrastructure
- Network configurations

The most common method used to handle traffic is by using a load balancer. In Kubernetes, we can configure load balancers. When a service is created, users can:

- Specify a load balancer that will expose the IP address of the load balancer.
- Direct traffic to the load balancer and communicate with the service through it.

**Tools for Kubernetes networking :**

There are plenty of tools that we can use to configure networking within Kubernetes while adhering to all the guidelines and requirements imposed by the cluster.

- **Cilium :**
  It is an open-source software for providing and securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies from L3-L7 using identity-based security models.

- **Flannel :**
  It is a simple network overlay that meets Kubernetes requirements.

- **Kube-router :**
  It is a turnkey solution for Kubernetes networking. Kube-router provides a lean and powerful alternative to default Kubernetes network components from a single DaemonSet/Binary.

- **Antrea :**
  It is a Kubernetes-native open source networking solution that leverages Open vSwitch as the networking data plane.

And here are cloud vendor-specific container network interfaces:

**AWS VPC CNI for Kubernetes is the Amazon Web Services specific CNI.**
**Azure CNI for Kubernetes is the Microsoft Azure specific CNI.**

**Creating a Manifest for creating Nginx Pod :**

```
#vim nginx-pod.yml
apiVersion : v1
kind : Pod
metadata :
  name : nginx-pod
  labels :
    app : nginx
    tier : dev
spec :
  containers :
  - name : nginx-container
    image : nginx
:wq!
```

#kubectl create -f nginx-pod.yml **( Deploying POD using the manifest file )**

#kubectl get pods **( Lists all the pods in K8's Cluster )**

#kubectl get pod <Pod Name> **( Displays information of a specific Pod )**

#kubectl get pod -o wide **( To fetch on which worker node the pod is scheduled )**

#kubectl get pod nginx-pod -o yaml **( Printing pod configuration information in YML Format )**
#kubectl get pod nginx-pod -o json **( Printing pod configuration information in JSON Format )**

#kubectl describe pod nginx-pod **( Displaying the detailed information of a specific pod )**

#kubectl delete pod nginx-pod **( Deleting Pod )**

#kubectl get pod

# ReplicationController

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

ReplicationController is often abbreviated to "rc" in discussion, and as a shortcut in kubectl commands.

A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers

- Replication controllers and pods are associated with **"LABELS"**

**Creating Manifest for Replication Controller :**

#vim nginx-rc.yml

```
apiVersion : v1
kind : ReplicationController
metadata :
  name : nginx-rc
spec :
  replicas : 3
  template :
    metadata :
      name : nginx-pod
      labels :
        app : nginx-app
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    app : nginx-app
:wq!
```

#kubectl get rc **( Lists all the available Replication Controllers on K8's Cluster )**

#kubectl get pods

#kubectl get po -l app=nginx-app **( Displaying pods having a spcific labels )**

#kubectl describe rc nginx-rc **( Displaying complete information of Replication Controller )**

#kubectl get po -o wide**( Displaying complete information of Replication Controller )**

**NOTE : Manually turn of the worker node which has POD for Testing the availability**

#kubectl get nodes

#kubectl get pods

#kubectl get po -o wide

**Now we will scale up the nginx App :**

#kubectl scale rc nginx-rc --replicas=6 **( Scaling up to total 6 Replicas )**

#kubectl get rc nginx-rc

#kubectl get pods

#kubectl get po -o wide

**Now we will scale down the nginx App :**

#kubectl scale rc nginx-rc --replicas=2 **( Scaling down to 2 Replicas )**

#kubectl get rc nginx-rc

#kubectl get pods

#kubectl get po -o wide

**Deleting Replication Controller :**

#kubectl delete -f nginx-rc.yml

#kubectl get rc

#kubectl get pods

# REPLICA SET

A ReplicaSet (RS) is one of the Kubernetes controllers that ensures a specified number of pod replicas are running at a given time. Without ReplicaSets, we would have to create multiple manifests for the number of pods needed which is very tedious.

In previous versions of Kubernetes, this was called Replication Controller. The main difference between the two is that ReplicaSets allow us to us something called Label Selector. Labels are key value pairs used to specify attributes of objects that are meaningful and useful to users, so keep in mind that It doesn't change the way the core system works. Label Selectors is used to identify a set of objects in Kubernetes.

**Creating Manifest for replica Set :**

```
#vim nginx-rs.yml
apiVersion : apps/v1
kind : ReplicaSet
metadata :
  name : nginx-rs
spec :
  replicas : 3
  template :
    metadata :
      name : nginx-pod
      labels :
        app : nginx-app
        tier : frontend
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    matchExpressions :
    - {key: tier,app, operator: In, values: [frontend, nginx-app]}
:wq!
```

#kubectl create -f nginx-rs.yml **( Deploying the Manifest file for creating Replica Set )**
#kubectl get pods -o wide
#kubectl get pods -l app=nginx-app
#kubectl get rs
#kubectl get rs nginx-rs
#kubectl get rs nginx-rs -o wide
#kubectl describe rs nginx-rs

**NOTE :**

To perform how Replicaset is providing the availability to the pods, Do manually power of your worker node and test Whether Replica Set is recreating the new POD as per the desired number mentioned in the manifest file.

**Scale up :**
#kubectl scale rs nginx-rs --replicas=7
#kubectl get pods -o wide
#kubectl get rs nginx-rs -o wide

**Scale Down :**
#kubectl scale rs nginx-rs --replicas=3
#kubectl get pods -o wide
#kubectl get rs nginx-rs -o wide

**Cleanup :**
#kubectl delete -f nginx-rs.yml
#kubectl get rs
#kubectl get pods -l app=nginx-app

# Kubernetes Deployment

A Kubernetes deployment is a resource object in Kubernetes that provides declarative updates to applications. A deployment allows you to describe an application's life cycle, such as which images to use for the app, the number of pods there should be, and the way in which they should be updated.

A Kubernetes object is a way to tell the Kubernetes system how you want your cluster's workload to look. After an object has been created, the cluster works to ensure that the object exists, maintaining the desired state of your Kubernetes cluster.

The process of manually updating containerized applications can be time consuming and tedious. Upgrading a service to the next version requires starting the new version of the pod, stopping the old version of a pod, waiting and verifying that the new version has launched successfully, and sometimes rolling it all back to a previous version in the case of failure.

Performing these steps manually can lead to human errors, and scripting properly can require a significant amount of effort, both of which can turn the release process into a bottleneck.

A Kubernetes deployment makes this process automated and repeatable. Deployments are entirely managed by the Kubernetes backend, and the whole update process is performed on the server side without client interaction.

A deployment ensures the desired number of pods are running and available at all times. The update process is also wholly recorded, and versioned with options to pause, continue, and roll back to previous versions.

**Automate deployments with pre-made, repeatable Kubernetes patterns**

**The Kubernetes deployment object lets you:**

- Deploy a replica set or pod
- Update pods and replica sets
- Rollback to previous deployment versions
- Scale a deployment
- Pause or continue a deployment

**Application management strategies using Kubernetes deployment**

Managing your applications with a Kubernetes deployment includes the way in which an application should be updated. A major benefit of a deployment is the ability to start and stop a set of pods predictably.

**Rolling update strategy**

A rolling update strategy provides a controlled, phased replacement of the application's pods, ensuring that there are always a minimum number available.

The deployment makes sure that, by default, a maximum of only 25% of pods are unavailable at any time, and it also won’t over provision more than 25% of the number of pods specified in the desired state.

The deployment won’t kill old pods until there are enough new pods available to maintain the availability threshold, and it won’t create new pods until enough old pods are removed.

The deployment object allows you to control the range of available and excess pods through maxSurge and maxUnavailable fields.

With a rolling update strategy there is no downtime during the update process, however the application must be architected to ensure that it can tolerate the pod destroy and create operations.

During the update process 2 versions of the container are running at the same time, which may cause issues for the service consumers.

**Recreate strategy**

A recreate strategy removes all existing pods before new ones are created. Kubernetes first terminates all containers from the current version and then starts all new containers simultaneously when the old containers are gone.

With a recreate deployment strategy there is some downtime while all containers with old versions are stopped and no new containers are ready to handle incoming requests.

However, there won't be 2 versions of the containers running at the same time, which may make it simpler for service consumers.

**Declarative deployment pattern for Kubernetes**

Deployments are created by writing a manifest. The manifest is then applied to the Kubernetes cluster using kubectl apply, or you can use a declarative deployment pattern. Configuration files for Kubernetes can be written using YAML or JSON.

When creating a deployment, you'll describe the desired state and Kubernetes will implement it using either a rolling or recreate deployment strategy.

Using a declarative deployment pattern allows you to use a Kubernetes deployment to automate the execution of upgrade and rollback processes for a group of pods. Kubernetes patterns are reusable design patterns for container-based applications and services.

You can update a deployment by making changes to the pod template specification. When a change is made to the specification field, it triggers an update rollout automatically.

The rollout lifecycle consists of progressing, complete, and failed states. A deployment is progressing while it is performing update tasks, such as updating or scaling pods.

Complete indicates that all tasks were completed successfully and the system is in the desired state. A failed state is the result of some error that keeps the deployment from completing its tasks.

You can check or monitor the state of a deployment using the kubectl rollout status command.

**Manifest for creating Deployment:**

```
#vim nginx-deploy.yml
apiVersion : apps/v1
kind : Deployment
metadata :
  name : nginx-deploy
  labels :
    app : nginx-app
spec :
  replicas : 3
  template :
    metadata :
      labels :
        app : nginx-app
    spec :
      containers :
      - name : nginx-container
        image : nginx:1.7.9
        ports :
        - containerPort : 80
  selector :
    matchLabels :
      app : nginx-app
:wq!

#kubectl create -f nginx-deploy.yml
#kubectl get deploy -l app=nginx-app
#kubectl get rs -l app=nginx-app
#kubectl get pods -l app=nginx-app
#kubectl describe deploy nginx-deploy
#kubectl set image deploy nginx-deploy nginx-container=nginx:1.9.1 --record
#kubectl rollout history deployment nginx-deploy
#kubectl rollout undo deployment nginx-deploy
#kubectl rollout status deployment nginx-deploy
```

#kubectl set image deploy nginx-deploy nginx-container=nginx:latest
#kubectl rollout status deployment nginx-deploy
#kubectl get deploy

**Scaleup :**

#kubectl scale deployment nginx-deploy --replicas=5
#kubectl get deploy
#kubectl get pods -o wide

**Scaledown :**

#kubectl scale deployment nginx-deploy --replicas=2
#kubectl get deploy
#kubectl get pods -o wide

**Cleanup :**

#kubectl delete -f nginx-deploy.yml
#kubectl get deploy
#kubectl get rs
#kubectl get pods

# Kubernetes DaemonSet

The DaemonSet feature is used to ensure that some or all of your pods are scheduled and running on every single available node. This essentially runs a copy of the desired pod across all nodes.

When a new node is added to a Kubernetes cluster, a new pod will be added to that newly attached node.

When a node is removed, the DaemonSet controller ensures that the pod associated with that node is garbage collected. Deleting a DaemonSet will clean up all the pods that DaemonSet has created.

DaemonSets are an integral part of the Kubernetes cluster facilitating administrators to easily configure services (pods) across all or a subset of nodes.

**DaemonSet use cases :**

DaemonSets can improve the performance of a Kubernetes cluster by distributing maintenance tasks and support services via deploying Pods across all nodes. They are well suited for long-running services like monitoring or log collection. Following are some example use cases of DaemonSets:

- To run a daemon for cluster storage on each node, such as glusterd and ceph.
- To run a daemon for logs collection on each node, such as Fluentd and logstash.
- To run a daemon for node monitoring on every note, such as Prometheus Node Exporter, collectd, or Datadog agent.

Depending on the requirement, you can set up multiple DaemonSets for a single type of daemon, with different flags, memory, CPU, etc. that supports multiple configurations and hardware types.

**Scheduling DaemonSet pods :**

By default, the node that a pod runs on is decided by the Kubernetes scheduler. However, DaemonSet pods are created and scheduled by the DaemonSet controller. Using the DaemonSet controller can lead to Inconsistent Pod behavior and issues in Pod priority preemption.

To mitigate these issues, Kubernetes (ScheduleDaemonSetPods) allows users to schedule DaemonSets using the default scheduler instead of the DaemonSet controller. This is done by adding the NodeAffinity term to the DaemonSet pods instead of the .spec.nodeName term. The default scheduler is then used to bind the Pod to the target host.

**Daemon Set Lab Demo 1 (This Example is going to deploy One Pod Per Worker Node) :**

```
#vim fluentds.yml
apiVersion : apps/v1
kind : DaemonSet
metadata :
  name : fluentd-ds
spec :
  template :
    metadata :
    labels :
        name : fluentd
    spec :
```

```
      containers :
      - name : fluentd
        image : gcr.io/google-containers/fluentd-elasticsearch:1.20
  selector :
    matchLabels :
      name : fluentd
:wq!

#kubectl get nodes
#kubectl create -f fluentds.yml
#kubectl get ds
#kubectl get pods
#kubectl get pods -o wide
#kubectl describe ds fluentd-ds
```

**Cleanup :**

```
#kubectl delete ds fluentd-ds
#kubectl get ds
#kubectl get pods
```

**Lab Demo 2 : ( In this example lets deploy pod with Nginx container on selected worker nodes)**

```
#vim nginx-ds-subsetnodes.yml
apiVersion : apps/v1
kind : DaemonSet
metadata :
  name : nginx-ds
spec :
  template :
    metadata :
      labels :
        name : nginx
    spec :
    containers :
      - name : nginx-container
      image : nginx
      nodeSelector :
        node : server
  selector :
  matchLabels :
      name : nginx
```

:wq!

**First we need to label the Workernodes inside the cluster :**
#kubectl get nodes
#kubectl label nodes wn1 wn2 node=server **( Creating labels to worker nodes )**
#kubectl get nodes --show-labels **( Lists all the nodes with the labels created )**

#kubectl create -f nginx-ds-subsetnodes.yml
#kubectl get pods
#kubectl get pods -o wide
#kubectl get ds
#kubectl describe ds nginx-ds

Cleanup :
#kubectl delete ds nginx-ds
#kubectl get ds
#kubectl get pods

## Kubernetes Volumes

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

**Types of Kubernetes Volume :**
Here is a list of some popular Kubernetes Volumes −

**emptyDir** − It is a type of volume that is created when a Pod is first assigned to a Node. It remains active as long as the Pod is running on that node. The volume is initially empty and the containers in the pod can read and write the files in the emptyDir volume. Once the Pod is removed from the node, the data in the emptyDir is erased.

**hostPath** − This type of volume mounts a file or directory from the host node's filesystem into your pod.

**gcePersistentDisk** − This type of volume mounts a Google Compute Engine (GCE) Persistent Disk into your Pod. The data in a gcePersistentDisk remains intact when the Pod is removed from the node.

**awsElasticBlockStore** − This type of volume mounts an Amazon Web Services (AWS) Elastic Block Store into your Pod. Just like gcePersistentDisk, the data in an awsElasticBlockStore remains intact when the Pod is removed from the node.

**nfs** − An nfs volume allows an existing NFS (Network File System) to be mounted into your pod. The data in an nfs volume is not erased when the Pod is removed from the node. The volume is only unmounted.

**iscsi** − An iscsi volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your pod.

**flocker** − It is an open-source clustered container data volume manager. It is used for managing data volumes. A flocker volume allows a Flocker dataset to be mounted into a pod. If the dataset does not exist in Flocker, then you first need to create it by using the Flocker API.

**glusterfs** − Glusterfs is an open-source networked filesystem. A glusterfs volume allows a glusterfs volume to be mounted into your pod.

**rbd** − RBD stands for Rados Block Device. An rbd volume allows a Rados Block Device volume to be mounted into your pod. Data remains preserved after the Pod is removed from the node.

**cephfs** − A cephfs volume allows an existing CephFS volume to be mounted into your pod. Data remains intact after the Pod is removed from the node.

**gitRepo** − A gitRepo volume mounts an empty directory and clones a git repository into it for your pod to use.

**secret** − A secret volume is used to pass sensitive information, such as passwords, to pods.

**persistentVolumeClaim** − A persistentVolumeClaim volume is used to mount a PersistentVolume into a pod. PersistentVolumes are a way for users to "claim" durable

storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

**downwardAPI** − A downwardAPI volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

**azureDiskVolume − An AzureDiskVolume is used to mount a Microsoft Azure Data Disk into a Pod.**

**Demo on EMPTY DIRECTORY :**

```
#vim nginx-empty.yml
apiVersion : v1
kind : Pod
metadata :
  name : nginx-emptydir
spec :
  containers :
  - name : nginx-container
    image : nginx
    volumeMounts :
    - name : test-vol
      mountPath : /test-mnt
  volumes :
  - name : test-vol
    emptyDir : {}
:wq!

#kubectl create -f nginx-empty.yml
#kubectl get pods -o wide
#kubectl exec nginx-emptydir -it -- /bin/sh
#df ( Inside the Container )
#kubectl describe pod <POD-NAME>
#kubectl delete po <POD-NAME>
```

**Host Path Volume DEMO :**

```
#vim hostpath-nginx.yml
apiVersion : v1
kind : Pod
metadata :
```

```
  name : nginx-hostpath
spec :
  containers :
    - name : nginx-container
      image : nginx
      volumeMounts :
      - mountPath : /test-mnt
        name : test-vol
  volumes :
  - name : test-vol
    hostPath :
      path : /test-vol
:wq!

#kubectl create -f hostpath-nginx.yml
#kubectl get pods
#kubectl exec nginx-hostpath df /test-mnt
```

Testing from the Worker Node **( Ensure to go to correct worker node where the POD is scheduled )**

```
#cd /test-vol


#cat >testfile1.txt
11111
222222
ctl+d
```

**Now login to POD from Master Node :**

```
#kubectl exec nginx-hostpath cat /test-mnt/testfile1.txt
```

**Now create test file from the POD :**

```
#kubectl exec nginx-hostpath -it -- /bin/sh
#cd /test-mnt

#cat >pod.txt
aaaa
bbbbb
```

ctl+d

**To test, go to Worker node where the POD is Scheduled :**

#cd /test-vol
#ls
#cat pod.txt
**Now delete the POD, Verify the /test-vol still exists with the data :**

#kubectl delete po nginx-hostpath
#kubectl get po

#cd /test-vol ( On Worker Node )
#ls

## What are Kubernetes Services?

A Kubernetes service is a logical abstraction for a deployed group of pods in a cluster (which all perform the same function).

Since pods are ephemeral, a service enables a group of pods, which provide specific functions (web services, image processing, etc.) to be assigned a name and unique IP address (clusterIP). As long as the service is running that IP address, it will not change. Services also define policies for their access.

In Kubernetes, a deployment is a method of launching a pod with containerized applications and ensuring that the necessary number of replicas is always running on the cluster.

On the other hand, a service is responsible for exposing an interface to those pods, which enables network access from either within the cluster or between external processes and the service.

**What are the components of a Kubernetes services?**
Kubernetes services connect a set of pods to an abstracted service name and IP address. Services provide discovery and routing between pods. For example, services connect an application front-end to its backend, each of which running in separate deployments in a cluster. Services use labels and selectors to match pods with other applications. The core attributes of a Kubernetes service are:

- A label selector that locates pods
- The clusterIP IP address and assigned port number
- Port definitions
- Optional mapping of incoming ports to a targetPort

Services can be defined without pod selectors. For example, to point a service to another service in a different namespace or cluster.

**What are the types of Kubernetes services?**
**ClusterIP.** Exposes a service which is only accessible from within the cluster.
**NodePort.** Exposes a service via a static port on each node's IP.
**LoadBalancer.** Exposes the service via the cloud provider's load balancer.

**What is the Kubernetes ClusterIP service?**
ClusterIP is the default type of service, which is used to expose a service on an IP address internal to the cluster. Access is only permitted from within the cluster.

**What is a Kubernetes NodePort service?**
NodePorts are open ports on every cluster node. Kubernetes will route traffic that comes into a NodePort to the service, even if the service is not running on that node. NodePort is intended as a foundation for other higher-level methods of ingress such as load balancers and are useful in development.

**What is a Kubernetes Load Balancer service?**
For clusters running onpublic cloud providers like AWS or Azure, creating a load LoadBalancer service provides an equivalent to a clusterIP service, extending it to an external load balancer that is specific to the cloud provider. Kubernetes will automatically create the load balancer, provide firewall rules if needed, and populate the service with the external IP address assigned by the cloud provider.

**How do Kubernetes services work?**
Services simply point to pods using labels. Since services are not node-specific, a service can point to a pod regardless of where it runs in the cluster at any given moment in time. By exposing a service IP address as well as a DNS service name, the application can be reached by either method as long as the service exists.

**NODE PORT Service LAB DEMO 1 ( Running Nginx App with one Replicas in one of the worker node and exposing its traffic from Node Port Service )**

```
#vim nginx-deployment.yml
apiVersion : apps/v1
kind : Deployment
metadata :
  name : nginx-deploy
  labels :
    app : nginx-app
spec :
  replicas : 1
  template :
    metadata :
      labels :
        app : nginx-app
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    matchLabels :
      app : nginx-app

:wq!
```

**Creating Manifest file for Node Port :**

```
#vim nginx-svc-np.yml
apiVersion : v1
kind : Service
metadata :
  name : my-service-np
  labels :
    app : nginx-app
spec :
  selector :
    app : nginx-app
  type : NodePort
  ports :
```

```
  - nodePort : 30101
    port : 80
    targetPort : 80

:wq!

#kubectl create -f nginx-deployment.yml
#kubectl get deploy
#kubectl get pods
#kubectl get pods -o wide
#kubectl create -f nginx-svc-np.yml
#kubectl get svc
#kubectl describe svc my-service-np
```

**Testing : From the base O/S Browser http://Ip of Worker node where the nginx pod running:30101**

**Lab Demo 2 : ( Multiple number of Nginx pods running in side the same worker node and exposing its traffic to outside world usinf Node port Service )**

```
#vim nginx-deployment.yml
apiVersion : apps/v1
kind : Deployment
metadata :
  name : nginx-deploy
  labels :
    app : nginx-app
spec :
  replicas : 3
  template :
  metadata :
      labels :
        app : nginx-app
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    matchLabels :
      app : nginx-app
```

:wq!

**NOTE : We are using the same Nodeport service we created for the previous demonstration for this scenario.**

#kubectl get nodes
#kubectl uncordon wn2 wn3
#kubectl get nodes
#kubectl create -f nginx-deployment.yml
#kubectl get deploy
#kubectl get pods
#kubectl get pods -o wide
#kubectl create -f nginx-svc-np.yml
#kubectl get svc
#kubectl describe svc my-service-np

**Testing : From the base O/S Browser http://Ip of Worker node where the nginx pod running:30101**

**Lab Demo 3 : ( Multiple no of nginx pods running on multiple worker nodes )**

#kubectl get nodes
#kubectl cordon wn2 wn3
#kubectl get nodes

#vim nginx-deployment.yml
apiVersion : apps/v1
kind : Deployment
metadata :
  name : nginx-deploy
  labels :
    app : nginx-app
spec :
  replicas : 3
  template :
  metadata :
      labels :
        app : nginx-app
    spec :
      containers :

```
    - name : nginx-container
      image : nginx
      ports :
      - containerPort : 80
  selector :
    matchLabels :
      app : nginx-app
:wq!
```

**NOTE : We are using the same Nodeport service we created for the previous demonstration for this scenario.**

```
#kubectl get nodes
#kubectl uncordon wn2 wn3
#kubectl get nodes
#kubectl create -f nginx-deployment.yml
#kubectl get deploy
#kubectl get pods
#kubectl get pods -o wide
#kubectl create -f nginx-svc-np.yml
#kubectl get svc
#kubectl describe svc my-service-np
```

Testing : From the base O/S Browser http://Ip of Worker node where the nginx pod running:30101

**NOTE :**

In this case we are accessing every instance separately using the Node IP.Here there is no concept of Load Balancing. This is the main disadvantage of Node Port Service,It cannot be used in Production Environments, It is only best suited for testing environments.

**To overcome this problem thats the reason we are going for Load balaner Service .**

**Load Balancer Demo :**

```
#vim nginx-deployment.yml
apiVersion : apps/v1
kind : Deployment
metadata :
```

```
  name : nginx-deploy
  labels :
    app : nginx-app
spec :
  replicas : 3
  template :
    metadata :
      labels :
        app : nginx-app
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    matchLabels :
      app : nginx-app
:wq!
```

**Now create Manifest file for Load balancer Service :**

```
#vim loadbalancer.yml
apiVersion : v1
kind : Service
metadata :
  name : my-service
  labels :
    app : nginx-app
spec :
  selector :
    app : nginx-app
  type : LoadBalancer
  externalIPs :
  - X.X.X.X ( Enter the Public IP Here )
  ports :
  - nodePort : 31000
    port : 80
    targetPort : 80
:wq!
#kubectl create -f nginx-deployment.yml
#kubectl create -f loadbalancer.yml
```

#kubectl get deploy
#kubectl get pods -o wide
#kubectl get svc
#kubectl descrive svc my-service

**Testing :**

Go to Base O/S Browser ---> http://Loadbalancerip//  (OR)  http://Nodeip:3100( incase just fortesting when we dont have public ip )