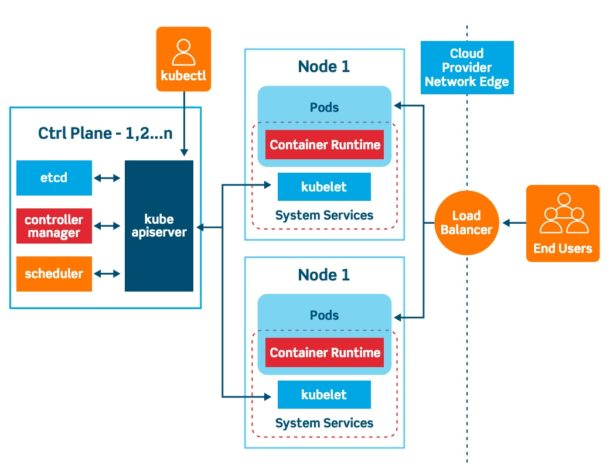
**Kubernetes**

Kubernetes (also known as k8s or “kube”) is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.

## **Features of Kubernetes**

* Continues development, integration and deployment
* Containerized infrastructure
* Application-centric management
* Auto-scalable infrastructure
* Environment consistency across development testing and production
* Loosely coupled infrastructure, where each component can act as a separate unit
* Higher density of resource utilization
* Predictable infrastructure which is going to be created

Kubernetes Architecture



Master Nodes

This Master node which control the cluster state and nodes

**API Server** - The **API Server** provides APIs to support lifecycle orchestration scaling,updates and so on for different types of applications. It also acts as the gateway to the cluster, which get the initial request to update the cluster , query to cluster so the API server must be accessible by clients from outside the cluster. Acts as a gatekeeper to authenticate the request to cluster .Clients authenticate via the API Server, and also use it as a proxy/tunnel to nodes and pods (and services).

Whenever pods to be created or new service ,deploy new application , initially user should interact with master node api server further request will be sent to other processes.It is the entrypoint to the cluster.

**The Scheduler**

It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

schedule new pod---> api server---->scheduler ---->where to put the pod.

**Controller Manager**

This controller manager detects the cluster state changes when the pods die.In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server. It works toward getting the shared state of cluster and then make changes to bring the current status of the server to the desired state. The key controllers are replication controller, endpoint controller, namespace controller, and service account controller. The controller manager runs different kind of controllers to handle nodes, endpoints

Controller manager--> scheduler --> kubelet

**ETCD**

Which is a key value store of cluster state . Cluster changes will be stored in etc in form of key value store. It stores the configuration information which can be used by each of the nodes in the cluster. t is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key value Store which is accessible to all.

**Node process/Worker Node**

Following are the key components of Node server which are necessary to communicate with Kubernetes master.

Each node has multiple pods on it. 3 processes must be installed on every node

**1.Container runtime**

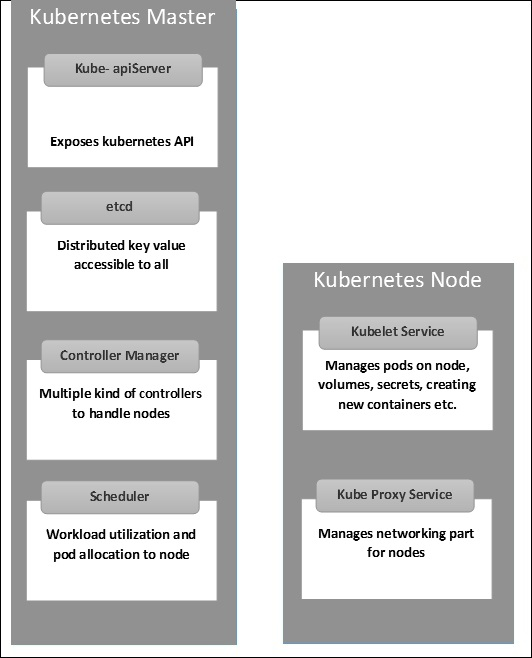
Docker which is used as container runtime which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment. Container runtime should be installed on every node.

**2. kubelet**

Kubelet which schedules the pod and interacts with both container and node.Kubelet is responsible for taking the configuration and start the pod with a container inside it. Kubelet is kubernetes process which assign reaources such as ram,cpu from node to container.

**3.Kube-proxy**

This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers’ health checkup, etc.



**Kubernetes setup in centos 7**

**Prerequisites**

* Multiple Linux servers running CentOS 7 (1 Master Node, Multiple Worker Nodes)
* A user account on every system with **sudo** or root privileges
* The **yum** package manager, included by default
* Command-line/terminal window

### **Step 1: Configure Kubernetes Repository**

Kubernetes packages are not available from official CentOS 7 repositories. This step needs to be performed on the Master Node, and each Worker Node you plan on utilizing for your container setup. Enter the following command to retrieve the Kubernetes repositories.

These 3 basic packages are required to be able to use Kubernetes. Install the following package(s) on each node:

cat &lt;&lt;EOF &gt; /etc/yum.repos.d/kubernetes.repo

[kubernetes]

name=Kubernetes

baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86\_64

enabled=1

gpgcheck=1

repo\_gpgcheck=1

gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg

EOF

### **Step 2: Install *kubelet*, *kubeadm***

### **, and *kubectl***

These 3 basic packages are required to be able to use Kubernetes. Install the following package(s) on each node:

sudo yum install -y kubelet kubeadm kubectl

systemctl enable kubelet

systemctl start kubelet

### **Step 3: Set Hostname on Nodes**

To give a unique hostname to each of your nodes, use this command:

sudo hostnamectl set-hostname master-node

Or

sudo hostnamectl set-hostname worker-node1

Example:

sudo vi /etc/hosts

192.168.1.10 master.phoenixnap.com master-node

192.168.1.20 node1. phoenixnap.com node1 worker-node

### **Step 4: Configure Firewall**

The nodes, containers, and pods need to be able to communicate across the cluster to perform their functions. Firewalld is enabled in CentOS by default on the front-end. Add the following ports by entering the listed commands.

On the Master Node enter:

sudo firewall-cmd --permanent --add-port=6443/tcp

sudo firewall-cmd --permanent --add-port=2379-2380/tcp

sudo firewall-cmd --permanent --add-port=10250/tcp

sudo firewall-cmd --permanent --add-port=10251/tcp

sudo firewall-cmd --permanent --add-port=10252/tcp

sudo firewall-cmd --permanent --add-port=10255/tcp

sudo firewall-cmd –reload

Enter the following command on worker node:

sudo firewall-cmd --permanent --add-port=10251/tcp

sudo firewall-cmd --permanent --add-port=10255/tcp

firewall-cmd –reload

### **Step 5: Update Iptables Settings**

Set the **net.bridge.bridge-nf-call-iptables** to '1' in your sysctl config file. This ensures that packets are properly processed by IP tables during filtering and port forwarding.

cat &lt;&lt;EOF &gt; /etc/sysctl.d/k8s.conf

net.bridge.bridge-nf-call-ip6tables = 1

net.bridge.bridge-nf-call-iptables = 1

EOF

sysctl –system

### **Step 6: Disable SELinux**

The containers need to access the host filesystem. SELinux needs to be set to permissive mode, which effectively disables its security functions.

sudo setenforce 0

sudo sed -i ‘s/^SELINUX=enforcing$/SELINUX=permissive/’ /etc/selinux/config

### **Step 7: Disable SWAP**

The containers need to access the host filesystem. SELinux needs to be set to permissive mode, which effectively disables its security functions.

sudo sed -i '/swap/d' /etc/fstab sudo swapoff –a

**How to deploy kubernetes cluster**

### **Step 1: Create Cluster with kubeadm**

sudo kubeadm init --pod-network-cidr=10.244.0.0/16

Initialize a cluster by executing the command

The process might take several minutes to complete based on network speed. Once this command finishes, it displays a kubeadm join message. Make a note of the entry and use it to join worker nodes to the cluster at a later stage.

### **Step 2: Manage Cluster as Regular User**

To start using the cluster you need to run it as a regular user by typing:

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

### **Step 3: Set Up Pod Network**

A Pod Network allows nodes within the cluster to communicate. There are several available [Kubernetes networking options.](https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model) Use the following command to install the **flannel** pod network add-on:

sudo kubectl apply -f <https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml>

If you decide to use flannel, edit your firewall rules to allow traffic for the flannel default port **8285**.

### **Step 4: Check Status of Cluster**

Check the status of the nodes by entering the following command on the master server:

sudo kubectl get nodes

Once a pod network has been installed, you can confirm that it is working by checking that the CoreDNS pod is running by typing:

sudo kubectl get pods --all-namespaces

### **Step 5: Join Worker Node to Cluster**

As indicated in **Step 1**, you can use the **kubeadm join** command on each worker node to connect it to the cluster.

kubeadm join --discovery-token cfgrty.1234567890jyrfgd --discovery-token-ca-cert-hash sha256:1234..cdef 1.2.3.4:6443

Replace the codes with the ones from your master server. Repeat this action for each worker node on your cluster.

Main k8s components

1. **Node** - Nodes are the physical servers or VMs
2. **Pod** – smallest unit of k8s, abstraction over container.Usually one application per pod.Each pod get its own ip address(internal ip address).whenever the pod dies new pod with new ip gets created
3. **Service** – Assigns a permannent ip address to a pod. Lifecycle of pod and application is not connected. Whenever you want to access the app through browser so external service can be used and if you wantto access the database with public access then we can create internal service.
4. If the url is https://domain-name:port then the request is first forwarded to ingress and then to service

**5.ConfigMap**

If the application need connection to database endpoint, if the database url changes then it would be difficult task to rebuild the image again. So for this purpose configMap component is used in which we configure database url, db username, db password so that any changes can be applied without the need to rebuild the image.It provides external configuratin for your application.

**6.secret**

Which used to store secret data such as username and password I.e in base64 encoded format.

Note: If pods deleted automatically associated resources also deleted.

If the node fails automatically the pod will be deleted.

Once the resources being terminated there is no chance to restart them same resources a gain, we can start like same specs other resources from our scripts and commands.

J The volumes are mapped with pods are available until pod available , if pod or node dies those will be deleted.

**VOLUMES**

Kubernetes cluster does not take care of data storage . Whenever the pod is restarted the older data is lost .Attaches physical hardware on a harddrive to your pod that could be on local machine I.e same server node where the pod is running or remotei.e outside the kubernetes cluster.

**Types of 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.

**Persistent Volume (PV)** − It’s a piece of network storage that has been provisioned by the administrator. It’s a resource in the cluster which is independent of any individual pod that uses the PV.

**Persistent Volume Claim (PVC)** − The storage requested by Kubernetes for its pods is known as PVC. The user does not need to know the underlying provisioning. The claims must be created in the same namespace where the pod is created.

kind: PersistentVolume ---------> 1  
apiVersion: v1  
metadata:  
 name: pv0001 ------------------> 2  
 labels:  
 type: local  
spec:  
 capacity: -----------------------> 3  
 storage: 10Gi ----------------------> 4  
 accessModes:  
 - ReadWriteOnce -------------------> 5  
 hostPath:  
 path: "/tmp/data01" --------------------------> 6

* **kind: PersistentVolume** → We have defined the kind as PersistentVolume which tells kubernetes that the yaml file being used is to create the Persistent Volume.
* **name: pv0001** → Name of PersistentVolume that we are creating.
* **capacity:** → This spec will define the capacity of PV that we are trying to create.
* **storage: 10Gi** → This tells the underlying infrastructure that we are trying to claim 10Gi space on the defined path.
* **ReadWriteOnce** → This tells the access rights of the volume that we are creating.
* **path: "/tmp/data01"** → This definition tells the machine that we are trying to create volume under this path on the underlying infrastructure.

### Creating PV

$ kubectl create –f local-01.yaml  
persistentvolume "pv0001" created

**Persistent Volume Claim (PVC)** − The storage requested by Kubernetes for its pods is known as PVC. The user does not need to know the underlying provisioning. The claims must be created in the same namespace where the pod is created.

kind: PersistentVolume ---------> 1  
apiVersion: v1  
metadata:  
 name: pv0001 ------------------> 2  
 labels:  
 type: local  
spec:  
 capacity: -----------------------> 3  
 storage: 10Gi ----------------------> 4  
 accessModes:  
 - ReadWriteOnce -------------------> 5  
 hostPath:  
 path: "/tmp/data01" --------------------------> 6

* **kind: PersistentVolume** → We have defined the kind as PersistentVolume which tells kubernetes that the yaml file being used is to create the Persistent Volume.
* **name: pv0001** → Name of PersistentVolume that we are creating.
* **capacity:** → This spec will define the capacity of PV that we are trying to create.
* **storage: 10Gi** → This tells the underlying infrastructure that we are trying to claim 10Gi space on the defined path.
* **ReadWriteOnce** → This tells the access rights of the volume that we are creating.
* **path: "/tmp/data01"** → This definition tells the machine that we are trying to create volume under this path on the underlying infrastructure.
* **volumeMounts:** → This is the path in the container on which the mounting will take place.
* **Volume:** → This definition defines the volume definition that we are going to claim.
* **persistentVolumeClaim:** → Under this, we define the volume name which we are going to use in the defined pod.

**7. Deployment**

In order to create the replicas of pod we use deployment component which provide blue print of a pod.You create deployment where you can scale up or down the no of pods required.

Db can’t be replicated via deployment .so this replica set can be done by statefulset for databases such as mysql, elasticsearch, mongo db.

8.Statefulset

This take care of databases replucation such as mongodb,elasticsearch, mysql etc this approach of deploying database using statefulset is tedious so it is always recommended to use databases outside kubernetes cluster.

**Services:**

To access the application from browser,we need to create a NodePort Service.

Expose pod with a service (NodePort Service) to access the application externally (from internet)

**port:** Port on which node port service listens in Kubernetes cluster internally

**targetPort:** We define container port here on which our application is running.

**NodePort:** Worker Node port on which we can access our application.

**# Create a Pod**

kubectl run <desired-pod-name> --image <Container-Image> --generator=run-pod/v1

kubectl run my-first-pod –image stacksimplify/kubenginx:1.0.0 --generator=run-pod/v1

**# Expose Pod as a Service**

kubectl expose pod <Pod-Name> --type=NodePort --port=80 --name=<Service-Name>

kubectl expose pod my-first-pod --type=NodePort --port=80 --name=my-first-service

**# Get Service Info**

kubectl get service

kubectl get svc

**# Get Public IP of Worker Nodes**

kubectl get nodes -o wide

kubectl logs <pod-name> kubectl logs my-first-pod

**#Connect to a container inside a pod**

kubectl exec -it <pod-name> -- /bin/bash

kubectl exec -it mypod -- /bin/bash

**#Running individual commands in a Container**

kubectl exec -it my-first-pod env

kubectl exec -it my-first-pod ls

**#Get YAML Output of Pod & Service**

**# Get pod definition YAML output**

kubectl get pod my-first-pod -o yaml

**# Get service definition YAML output**

kubectl get service my-first-service -o yaml

**# Get all Objects in default namespace**

kubectl get all

**# Delete Services**

kubectl delete svc my-first-service

kubectl delete svc my-first-service2

kubectl delete svc my-first-service3

**# Delete Pod**

kubectl delete pod my-first-pod

**# Get all Objects in default namespace**

kubectl get all

**#Get list of ReplicaSets**

kubectl get replicaset

kubectl get rs

**#Desc replica set**

kubectl describe rs/<replicaset-name>

**#Expose ReplicaSet as a Service**

kubectl expose rs <ReplicaSet-Name> --type=NodePort --port=80 --target-port=8080 --name=<Service-Name-To-Be-Created>

kubectl expose rs my-helloworld-rs --type=NodePort --port=80 --target-port=8080 --name=my-helloworld-rs-service

**# Get Service Info**

kubectl get service kubectl get svc

**# Get Public IP of Worker Nodes**

kubectl get nodes -o wide

**# ReplicaSet Scalability feature**

Test how scalability is going to seamless & quick

Update the replicas field in replicaset-demo.yml from 3 to 6.

Before change spec: replicas: 3 # After change spec: replicas: 6

Update the ReplicaSet

**# Apply latest changes to ReplicaSet**

kubectl replace -f replicaset-demo.yml

**# Verify if new pods got created**

kubectl get pods -o wide

**# Delete ReplicaSet**

kubectl delete rs <ReplicaSet-Name>

**#Delete Service created for ReplicaSet**

**# Delete Service**

kubectl delete svc <service-name>

**# Sample Commands**

kubectl delete svc my-helloworld-rs-service

[or] kubectl delete svc/my-helloworld-rs-service

# Verify if Service got deleted

kubectl get svc

**Minikube setup**

To Test on local machine minikube can be installed. Kubectl which will interact with kubernetes cluster . Minikube which has master process and worker proces running in same node

Kubectl get nodes- displays the nodes

Kubectl get services- displays all the services .

Pod is the samleest unit ,but we are creating deployment abstraction over pods.

Yaml configuration files

There are 3 parts in these configuration

[root@localhost centos]# cat mongo-configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: mongodb-configmap

data:

database\_url: mongodb-service

--------------------

[root@localhost centos]# cat mongo-configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: mongodb-configmap

data:

database\_url: mongodb-service

[root@localhost centos]# cat mongodb-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongodb-deployment

labels:

app: mongodb

spec:

replicas: 1

selector:

matchLabels:

app: mongodb

template:

metadata:

labels:

app: mongodb

spec:

containers:

- name: mongodb

image: mongo

ports:

- containerPort: 27017

env:

- name: MONGO\_INITDB\_ROOT\_USERNAME

valueFrom:

secretKeyRef:

name: mongodb-secret

key: mongo-root-username

- name: MONGO\_INITDB\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: mongodb-secret

key: mongo-root-password

---

apiVersion: v1

kind: Service

metadata:

name: mongodb-service

spec:

selector:

app: mongodb

ports:

- protocol: TCP

port: 27017

targetPort: 27017

[root@localhost centos]# cat mongo-express.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongo-express

labels:

app: mongo-express

spec:

replicas: 1

selector:

matchLabels:

app: mongo-express

template:

metadata:

labels:

app: mongo-express

spec:

containers:

- name: mongo-express

image: mongo-express

ports:

- containerPort: 8081

env:

- name: ME\_CONFIG\_MONGODB\_ADMINUSERNAME

valueFrom:

secretKeyRef:

name: mongodb-secret

key: mongo-root-username

- name: ME\_CONFIG\_MONGODB\_ADMINPASSWORD

valueFrom:

secretKeyRef:

name: mongodb-secret

key: mongo-root-password

- name: ME\_CONFIG\_MONGODB\_SERVER

valueFrom:

configMapKeyRef:

name: mongodb-configmap

key: mongodb-service

---

apiVersion: v1

kind: Service

metadata:

name: mongo-express-service

spec:

selector:

app:mongo-express

type: LoadBalancer

ports:

- protocol: TCP

port: 8081

targetPort: 8081

nodePort: 30000

[root@localhost centos]# cat mongo-secret.yaml

apiVersion: v1

kind: Secret

metadata:

name: mongodb-secret

type: Opaque

data:

mongo-root-username: dXNlcm5hbWU=

mongo-root-password: cGFzc3dvcmQ=

----------------------

[root@localhost centos]# cat nginx-deployment.yaml

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx

spec:

strategy:

type: Recreate

selector:

matchLabels:

app: nginx

replicas: 1 # tells deployment to run 1 pods matching the template

template: # create pods using pod definition in this template

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

1) Metadata -- name of the component

2)specification - evry configuartion we want to apply.

each deployment will have their own attribuets

3)status- desired vs actual . the information will be received from etcd.

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

spec:

replicas: 2

selector:

template

---------------------------------------------------------------------------

replicaset-demo.yml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: my-helloworld-rs

labels:

app: my-helloworld

spec:

replicas: 3

selector:

matchLabels:

app: my-helloworld

template:

metadata:

labels:

app: my-helloworld

spec:

containers:

- name: my-helloworld-app

image: webapp:v9

Kubectl apply –f replicaset-demo.yml

Kubectl expose rs my-helloworld-rs –-type=NodePort –port=80

Kubectl get svc

Template has its own metadata.and spec section

applies to a pod,blueprint for a pod

Format of configuration file

labels and selectors

metadata par contains labels and spec contains selectors

**--------**

Deployment manages pods,Replicatset manages pod,pod is an abstraction of container,Demo Project:

Mongo express application from browser ---> Mongo express external service--->Mongo pod-->Mongo db internal service--->Mongodb pod

What is Namespace?

What are the usecases?

How namespace work and how to use it?

What is Namespace?

Organise resource in namespaces,Virtual cluster inside a cluster.It Provides 4 namespace by default.

Kubernetes-dashboard – This namspace is specific to mini-kube.

Kube-system –Do not create or modify in kube system, kubectl , manages master process.

Kube-public – It contains publicly accessible data, configmap which contains clusterinfo.

Kube-node-lease –heartbeats of node,each node has associated lease object in namespace,determines the availability of a node.

Default-namespace – resources you created are located here.

Kubectl create namespace <nameof namespace>

Why should we use namespace.

* To group the resources in namespace.In order to avoid all the resources in same place.
* In order to avoid the conflicts between the teams and application. In order to differentiate the different application.
* Resource sharing and development.
* Blue/green deployment
* Access and resource limits on namespace.
* Each namespace must define own configMap.
* You can’t access most resources from another namespace.
* Access service in another namespace.

Command to get all the namespace

Kubectl get all –n my-namespace.(To get all the namespace )

Kubectl apply –f mysql-config.yaml --namespace=my-namespace.

Kubectl get configmap –n my-namespace.

We can also specify namespace in configuration file other than command line.

To change the active namespace we must install kubectx tool – Kubens namespace

**Ingress vs External service**

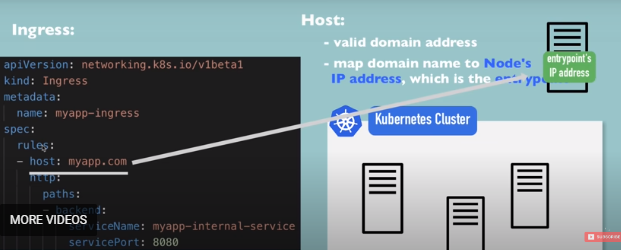
Whenever we want to access the application via domain name instead of port then we can use ingress and make external service as internal service.

## **Ingress**

*You can also use* [*Ingress*](https://kubernetes.io/docs/concepts/services-networking/ingress/) *to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. It lets you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address.*

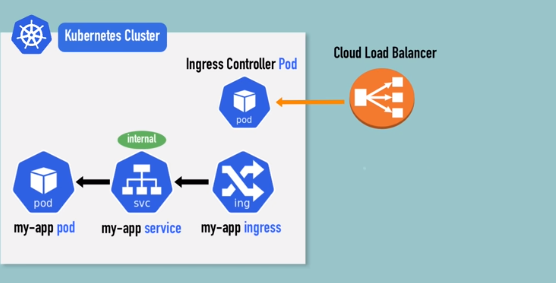
Usually the request from browser first reaches to ingress component and then to external service and then to pod in case of http://<domain-name>:ip

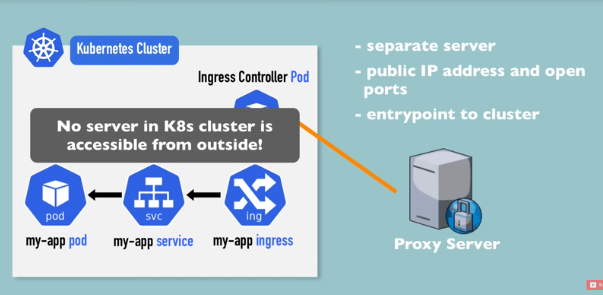
Ingress and internal service configuration



**Ingress Controller**

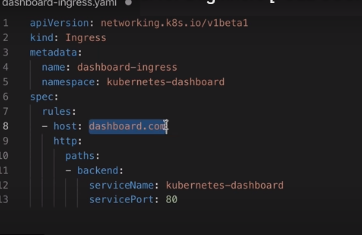
Evaluates all the rules,manages redirections,entrypoint to cluster.





Ingress yaml file configuration

Kubectl get all –n kubernetes-dashboard



Kubectl apply –f dashboard-ingress.yaml

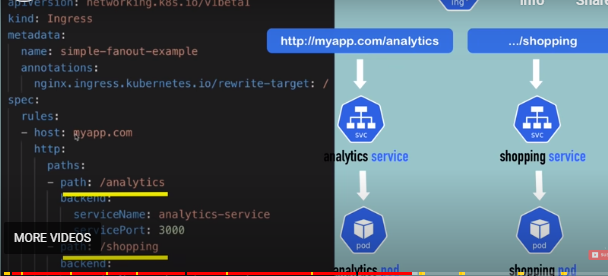
Kubectl get ingress –n kubernetes-dashboard –watch

We need to mention the ip in /etc/hosts

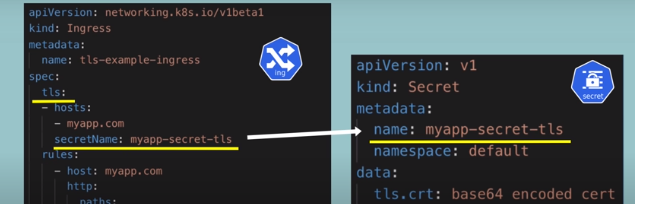
Ingress also has backend

Kubectl describe ingress dashboard-ingress –n kubernetes-dashboard

Forwarding request to multiplr path of same host



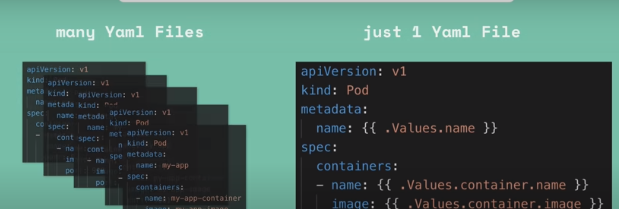
Configuring tls certificate where we need to mention the tils value in spec section.



Helm package manager

Helm which is a package manager of kubernetes and helm charts which is a bundle of yaml files,create your own helm charts with helm and push them to repository download and use existing ones.

Helm charts can be shared to public or private registeries and which can be used later.



Kubectl describe ingress dashboard-ingress –n kubernetes-dashboard

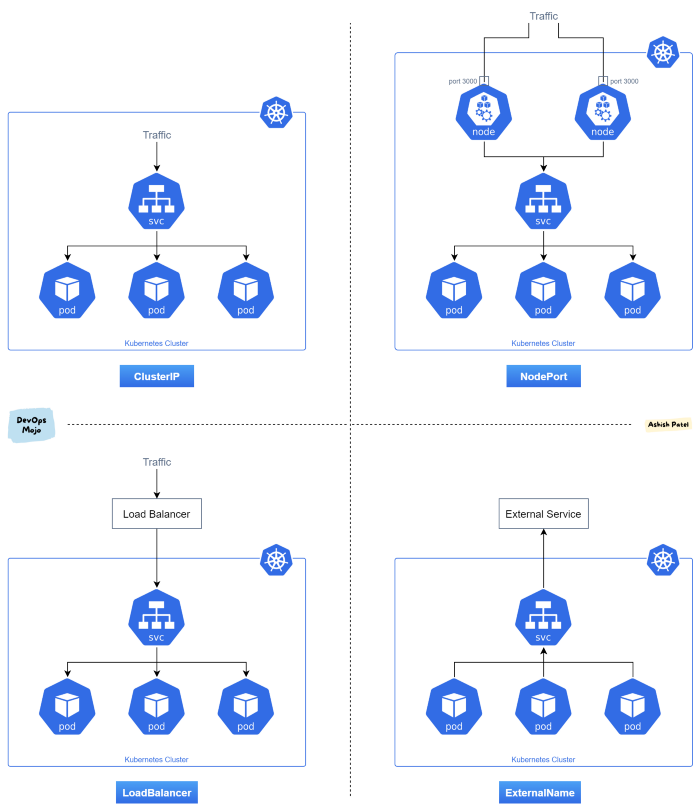
**K8s services**

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.

* **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.
* **ExternalName**. Maps a service to a predefined externalName field by returning a value for the CNAME record.



Kubernetes — Service Types

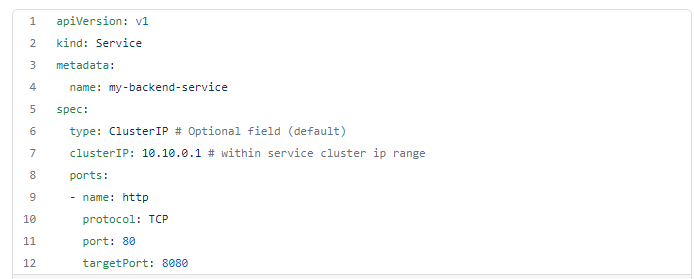
# **1. ClusterIP**

* ClusterIP is the default and most common service type.
* Kubernetes will assign a cluster-internal IP address to ClusterIP service. This makes the service only reachable within the cluster.
* You cannot make requests to service (pods) from outside the cluster.
* You can optionally set cluster IP in the service definition file.

## **Use Cases**

* Inter service communication within the cluster. For example, communication between the front-end and back-end components of your app.

Example



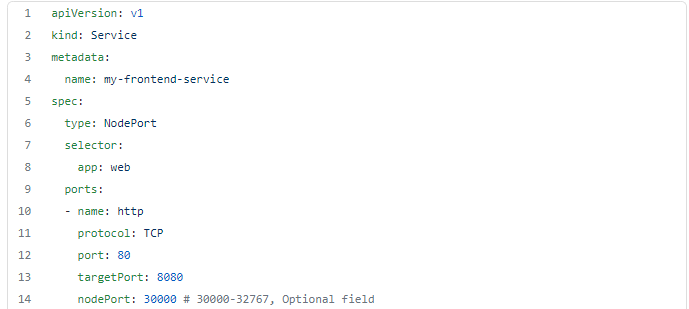
# **2. NodePort**

* NodePort service is an extension of ClusterIP service. A ClusterIP Service, to which the NodePort Service routes, is automatically created.
* It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP.
* NodePort exposes the service on each Node’s IP at a static port (the NodePort). Each node proxies that port into your Service. So, external traffic has access to fixed port on each Node. It means any request to your cluster on that port gets forwarded to the service.
* You can contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
* Node port must be in the range of 30000–32767. Manually allocating a port to the service is optional. If it is undefined, Kubernetes will automatically assign one.
* If you are going to choose node port explicitly, ensure that the port was not already used by another service.

## **Use Cases**

* When you want to enable external connectivity to your service.
* Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes’ IPs directly.
* Prefer to place a load balancer above your nodes to avoid node failure.

## **Example**



# **3. LoadBalancer**

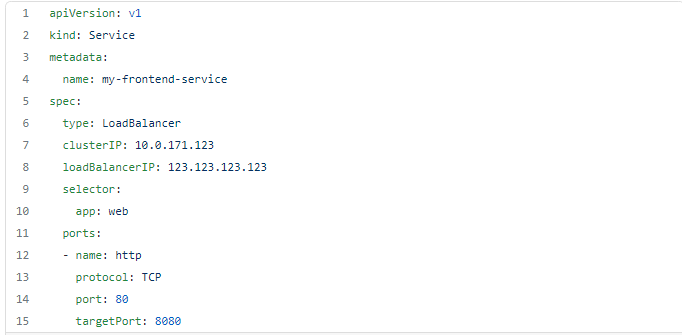
* LoadBalancer service is an extension of NodePort service. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
* It integrates NodePort with cloud-based load balancers.
* It exposes the Service externally using a cloud provider’s load balancer.
* Each cloud provider (AWS, Azure, GCP, etc) has its own native load balancer implementation. The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service.
* Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.
* The actual creation of the load balancer happens asynchronously.
* Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.

## **Use Cases**

* When you are using a cloud provider to host your Kubernetes cluster.

*This type of service is typically heavily dependent on the cloud provider.*

## **Example**



# **4. ExternalName**

* Services of type ExternalName map a Service to a DNS name, not to a typical selector such as my-service.
* You specify these Services with the `spec.externalName` parameter.
* It maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value.
* No proxying of any kind is established.

## **Use Cases**

* This is commonly used to create a service within Kubernetes to represent an external datastore like a database that runs externally to Kubernetes.
* You can use that ExternalName service (as a local service) when Pods from one namespace to talk to a service in another namespace.

## **Example**

