**Kubernetes**

Kubernetes in an open source container management tool hosted by Cloud Native Computing Foundation (CNCF). This is also known as the enhanced version of Borg which was developed at Google to manage both long running processes and batch jobs, which was earlier handled by separate systems.

Kubernetes comes with a capability of automating deployment, scaling of application, and operations of application containers across clusters. It is capable of creating container centric infrastructure.

Features of Kubernetes

Following are some of the important 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

One of the key components of Kubernetes is, it can run application on clusters of physical and virtual machine infrastructure. It also has the capability to run applications on cloud. It helps in moving from host-centric infrastructure to container-centric infrastructure.

**Kubernetes - Cluster Architecture**

As seen in the following diagram, Kubernetes follows client-server architecture. Wherein, we have master installed on one machine and the node on separate Linux machines.



Kubernetes - Master Machine Components

Following are the components of Kubernetes Master Machine.

**etcd**

It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes. It 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.

**API Server**

Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

**Controller Manager**

This component is responsible for most of the collectors that regulates the state of cluster and performs a task. 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, etc.

**Scheduler**

This is one of the key components of Kubernetes master. 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.

**Kubernetes - Node Components**

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

**Docker**

The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

**Kubelet Service**

This is a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and wright values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

**Kubernetes Proxy Service**

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 - Master and Node Structure

The following illustrations show the structure of Kubernetes Master and Node.



Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure. As of now, we are only supporting Kubernetes to support Docker images. Each container in a pod has its Docker image running inside it.

When we are configuring a pod, the image property in the configuration file has the same syntax as the Docker command does. The configuration file has a field to define the image name, which we are planning to pull from the registry.

Following is the common configuration structure which will pull image from Docker registry and deploy in to Kubernetes container.

apiVersion: v1

kind: pod

metadata:

name: Tesing\_for\_Image\_pull -----------> 1

spec:

containers:

- name: neo4j-server ------------------------> 2

image: <Name of the Docker image>----------> 3

imagePullPolicy: Always ------------->4

command: ["echo", "SUCCESS"] ------------------->

In the above code, we have defined −

name: Tesing\_for\_Image\_pull − This name is given to identify and check what is the name of the container that would get created after pulling the images from Docker registry.

name: neo4j-server − This is the name given to the container that we are trying to create. Like we have given neo4j-server.

image: <Name of the Docker image> − This is the name of the image which we are trying to pull from the Docker or internal registry of images. We need to define a complete registry path along with the image name that we are trying to pull.

imagePullPolicy − Always - This image pull policy defines that whenever we run this file to create the container, it will pull the same name again.

command: [“echo”, “SUCCESS”] − With this, when we create the container and if everything goes fine, it will display a message when we will access the container.

In order to pull the image and create a container, we will run the following command.

$ kubectl create –f Tesing\_for\_Image\_pull

Once we fetch the log, we will get the output as successful.

$ kubectl log Tesing\_for\_Image\_pull

The above command will produce an output of success or we will get an output as failure.

**Creating a Job**

Use the following command to create a job −

apiVersion: v1

kind: Job ------------------------> 1

metadata:

name: py

spec:

template:

metadata

name: py -------> 2

spec:

containers:

- name: py ------------------------> 3

image: python----------> 4

command: ["python", "SUCCESS"]

restartPocliy: Never --------> 5

In the above code, we have defined −

kind: Job → We have defined the kind as Job which will tell kubectl that the yaml file being used is to create a job type pod.

Name:py → This is the name of the template that we are using and the spec defines the template.

name: py → we have given a name as py under container spec which helps to identify the Pod which is going to be created out of it.

Image: python → the image which we are going to pull to create the container which will run inside the pod.

restartPolicy: Never →This condition of image restart is given as never which means that if the container is killed or if it is false, then it will not restart itself.

We will create the job using the following command with yaml which is saved with the name py.yaml.

$ kubectl create –f py.yaml

The above command will create a job. If you want to check the status of a job, use the following command.

$ kubectl describe jobs/py

The above command will create a job. If you want to check the status of a job, use the following command.

**Scheduled Job**

Scheduled job in Kubernetes uses Cronetes, which takes Kubernetes job and launches them in Kubernetes cluster.

Scheduling a job will run a pod at a specified point of time.

A parodic job is created for it which invokes itself automatically.

Note − The feature of a scheduled job is supported by version 1.4 and the betch/v2alpha 1 API is turned on by passing the –runtime-config=batch/v2alpha1 while bringing up the API server.

We will use the same yaml which we used to create the job and make it a scheduled job.

apiVersion: v1

kind: Job

metadata:

name: py

spec:

schedule: h/30 \* \* \* \* ? -------------------> 1

template:

metadata

name: py

spec:

containers:

- name: py

image: python

args:

/bin/sh -------> 2

-c

ps –eaf ------------> 3

restartPocliy: OnFailure

In the above code, we have defined −

schedule: h/30 \* \* \* \* ? → To schedule the job to run in every 30 minutes.

/bin/sh: This will enter in the container with /bin/sh

ps –eaf → Will run ps -eaf command on the machine and list all the running process inside a container.

Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.

**Functionality of Namespace**

Following are some of the important functionalities of a Namespace in Kubernetes −

Namespaces help pod-to-pod communication using the same namespace.

Namespaces are virtual clusters that can sit on top of the same physical cluster.

They provide logical separation between the teams and their environments.

**Create a Namespace**

The following command is used to create a namespace.

apiVersion: v1

kind: Namespce

metadata

name: elk

Control the Namespace

The following command is used to control the namespace.

$ kubectl create –f namespace.yml ---------> 1

$ kubectl get namespace -----------------> 2

$ kubectl get namespace <Namespace name> ------->3

$ kubectl describe namespace <Namespace name> ---->4

$ kubectl delete namespace <Namespace name>

In the above code,

We are using the command to create a namespace.

This will list all the available namespace.

This will get a particular namespace whose name is specified in the command.

This will describe the complete details about the service.

This will delete a particular namespace present in the cluster.

Using Namespace in Service - Example

Following is an example of a sample file for using namespace in service.

apiVersion: v1

kind: Service

metadata:

name: elasticsearch

namespace: elk

labels:

component: elasticsearch

spec:

type: LoadBalancer

selector:

component: elasticsearch

ports:

- name: http

port: 9200

protocol: TCP

- name: transport

port: 9300

protocol: TCP

**Kubernetes - Node**

A node is a working machine in Kubernetes cluster which is also known as a minion. They are working units which can be physical, VM, or a cloud instance.

Each node has all the required configuration required to run a pod on it such as the proxy service and kubelet service along with the Docker, which is used to run the Docker containers on the pod created on the node.

They are not created by Kubernetes but they are created externally either by the cloud service provider or the Kubernetes cluster manager on physical or VM machines.

The key component of Kubernetes to handle multiple nodes is the controller manager, which runs multiple kind of controllers to manage nodes. To manage nodes, Kubernetes creates an object of kind node which will validate that the object which is created is a valid node.

Service with Selector

apiVersion: v1

kind: node

metadata:

name: < ip address of the node>

labels:

name: <lable name>

In JSON format the actual object is created which looks as follows −

{

Kind: node

apiVersion: v1

"metadata":

{

"name": "10.01.1.10",

"labels"

{

"name": "cluster 1 node"

}

}

}

**Node Controller**

They are the collection of services which run in the Kubernetes master and continuously monitor the node in the cluster on the basis of metadata.name. If all the required services are running, then the node is validated and a newly created pod will be assigned to that node by the controller. If it is not valid, then the master will not assign any pod to it and will wait until it becomes valid.

Kubernetes master registers the node automatically, if –register-node flag is true.

–register-node = true

However, if the cluster administrator wants to manage it manually then it could be done by turning the flat of −

–register-node = false

**Kubernetes - Service**

A service can be defined as a logical set of pods. It can be defined as an abstraction on the top of the pod which provides a single IP address and DNS name by which pods can be accessed. With Service, it is very easy to manage load balancing configuration. It helps pods to scale very easily.

A service is a REST object in Kubernetes whose definition can be posted to Kubernetes apiServer on the Kubernetes master to create a new instance.

Service without Selector

apiVersion: v1

kind: Service

metadata:

name: Tutorial\_point\_service

spec:

ports:

- port: 8080

targetPort: 31999

The above configuration will create a service with the name Tutorial\_point\_service.

Service Config File with Selector

apiVersion: v1

kind: Service

metadata:

name: Tutorial\_point\_service

spec:

selector:

application: "My Application" -------------------> (Selector)

ports:

- port: 8080

targetPort: 31999

In this example, we have a selector; so in order to transfer traffic, we need to create an endpoint manually.

apiVersion: v1

kind: Endpoints

metadata:

name: Tutorial\_point\_service

subnets:

address:

"ip": "192.168.168.40" -------------------> (Selector)

ports:

- port: 8080

In the above code, we have created an endpoint which will route the traffic to the endpoint defined as “192.168.168.40:8080”.

Multi-Port Service Creation

apiVersion: v1

kind: Service

metadata:

name: Tutorial\_point\_service

spec:

selector:

application: “My Application” -------------------> (Selector)

ClusterIP: 10.3.0.12

ports:

-name: http

protocol: TCP

port: 80

targetPort: 31999

-name:https

Protocol: TCP

Port: 443

targetPort: 31998

**Types of Services**

**ClusterIP** − This helps in restricting the service within the cluster. It exposes the service within the defined Kubernetes cluster.

spec:

type: NodePort

ports:

- port: 8080

nodePort: 31999

name: NodeportService

**NodePort** − It will expose the service on a static port on the deployed node. A ClusterIP service, to which NodePort service will route, is automatically created. The service can be accessed from outside the cluster using the NodeIP:nodePort.

spec:

ports:

- port: 8080

nodePort: 31999

name: NodeportService

clusterIP: 10.20.30.40

**Load Balancer** − It uses cloud providers’ load balancer. NodePort and ClusterIP services are created automatically to which the external load balancer will route.

A full service yaml file with service type as Node Port. Try to create one yourself.

apiVersion: v1

kind: Service

metadata:

name: appname

labels:

k8s-app: appname

spec:

type: NodePort

ports:

- port: 8080

nodePort: 31999

name: omninginx

selector:

k8s-app: appname

component: nginx

env: env\_name

**Kubernetes - Pod**

A pod is a collection of containers and its storage inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it. For example, keeping a database container and data container in the same pod.

Types of Pod

There are two types of Pods −

**Single container pod**

**Multi container pod**

**1)Single Container Pod**

They can be simply created with the kubctl run command, where you have a defined image on the Docker registry which we will pull while creating a pod.

$ kubectl run <name of pod> --image=<name of the image from registry>

Example − We will create a pod with a tomcat image which is available on the Docker hub.

$ kubectl run tomcat --image = tomcat:8.0

This can also be done by creating the yaml file and then running the kubectl create command.

apiVersion: v1

kind: Pod

metadata:

name: Tomcat

spec:

containers:

- name: Tomcat

image: tomcat: 8.0

ports:

containerPort: 7500

imagePullPolicy: Always

Once the above yaml file is created, we will save the file with the name of tomcat.yml and run the create command to run the document.

$ kubectl create –f tomcat.yml

It will create a pod with the name of tomcat. We can use the describe command along with kubectl to describe the pod.

**2)Multi Container Pod**

Multi container pods are created using yaml mail with the definition of the containers.

apiVersion: v1

kind: Pod

metadata:

name: Tomcat

spec:

containers:

- name: Tomcat

image: tomcat: 8.0

ports:

containerPort: 7500

imagePullPolicy: Always

-name: Database

Image: mongoDB

Ports:

containerPort: 7501

imagePullPolicy: Always

**Kubernetes - Deployments**

Deployments are upgraded and higher version of replication controller. They manage the deployment of replica sets which is also an upgraded version of the replication controller. They have the capability to update the replica set and are also capable of rolling back to the previous version.

They provide many updated features of matchLabels and selectors. We have got a new controller in the Kubernetes master called the deployment controller which makes it happen. It has the capability to change the deployment midway.

Changing the Deployment

Updating − The user can update the ongoing deployment before it is completed. In this, the existing deployment will be settled and new deployment will be created.

Deleting − The user can pause/cancel the deployment by deleting it before it is completed. Recreating the same deployment will resume it.

Rollback − We can roll back the deployment or the deployment in progress. The user can create or update the deployment by using DeploymentSpec.PodTemplateSpec = oldRC.PodTemplateSpec.

Deployment Strategies

Deployment strategies help in defining how the new RC should replace the existing RC.

Recreate − This feature will kill all the existing RC and then bring up the new ones. This results in quick deployment however it will result in downtime when the old pods are down and the new pods have not come up.

Rolling Update − This feature gradually brings down the old RC and brings up the new one. This results in slow deployment, however there is no deployment. At all times, few old pods and few new pods are available in this process.

The configuration file of Deployment looks like this.

apiVersion: extensions/v1beta1 --------------------->1

kind: Deployment --------------------------> 2

metadata:

name: Tomcat-ReplicaSet

spec:

replicas: 3

template:

metadata:

lables:

app: Tomcat-ReplicaSet

tier: Backend

spec:

containers:

- name: Tomcatimage:

tomcat: 8.0

ports:

- containerPort: 7474

In the above code, the only thing which is different from the replica set is we have defined the kind as deployment.

Create Deployment

$ kubectl create –f Deployment.yaml -–record

deployment "Deployment" created Successfully.

Fetch the Deployment

$ kubectl get deployments

NAME DESIRED CURRENT UP-TO-DATE AVILABLE AGE

Deployment 3 3 3 3 20s

Check the Status of Deployment

$ kubectl rollout status deployment/Deployment

Updating the Deployment

$ kubectl set image deployment/Deployment tomcat=tomcat:6.0

Rolling Back to Previous Deployment

$ kubectl rollout undo deployment/Deployment –to-revision=2

**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.

Persistent Volume and Persistent Volume Claim

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.

Creating Persistent Volume

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

In the above code, we have defined −

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

Checking PV

$ kubectl get pv

NAME CAPACITY ACCESSMODES STATUS CLAIM REASON AGE

pv0001 10Gi RWO Available 14s

Describing PV

$ kubectl describe pv pv0001

Creating Persistent Volume Claim

kind: PersistentVolumeClaim --------------> 1

apiVersion: v1

metadata:

name: myclaim-1 --------------------> 2

spec:

accessModes:

- ReadWriteOnce ------------------------> 3

resources:

requests:

storage: 3Gi ---------------------> 4

In the above code, we have defined −

kind: PersistentVolumeClaim → It instructs the underlying infrastructure that we are trying to claim a specified amount of space.

name: myclaim-1 → Name of the claim that we are trying to create.

ReadWriteOnce → This specifies the mode of the claim that we are trying to create.

storage: 3Gi → This will tell kubernetes about the amount of space we are trying to claim.

Creating PVC

$ kubectl create –f myclaim-1

persistentvolumeclaim "myclaim-1" created

Getting Details About PVC

$ kubectl get pvc

NAME STATUS VOLUME CAPACITY ACCESSMODES AGE

myclaim-1 Bound pv0001 10Gi RWO 7s

Describe PVC

$ kubectl describe pv pv0001

Using PV and PVC with POD

kind: Pod

apiVersion: v1

metadata:

name: mypod

labels:

name: frontendhttp

spec:

containers:

- name: myfrontend

image: nginx

ports:

- containerPort: 80

name: "http-server"

volumeMounts: ----------------------------> 1

- mountPath: "/usr/share/tomcat/html"

name: mypd

volumes: -----------------------> 2

- name: mypd

persistentVolumeClaim: ------------------------->3

claimName: myclaim-1

In the above code, we have defined −

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.