**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.

**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”.

**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

**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

**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.

**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

In the above code, we have created one pod with two containers inside it, one for tomcat and the other for MongoDB.

**Kubernetes - Replication Controller**

Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time.

It is used in time when one wants to make sure that the specified number of pod or at least one pod is running. It has the capability to bring up or down the specified no of pod.

It is a best practice to use the replication controller to manage the pod life cycle rather than creating a pod again and again.

apiVersion: v1

kind: ReplicationController --------------------------> 1

metadata:

name: Tomcat-ReplicationController --------------------------> 2

spec:

replicas: 3 ------------------------> 3

template:

metadata:

name: Tomcat-ReplicationController

labels:

app: App

component: neo4j

spec:

containers:

- name: Tomcat- -----------------------> 4

image: tomcat: 8.0

ports:

- containerPort: 7474 ------------------------> 5

**Setup Details**

**Kind: ReplicationController** → In the above code, we have defined the kind as replication controller which tells the kubectl that the yaml file is going to be used for creating the replication controller.

**name: Tomcat-ReplicationController** → This helps in identifying the name with which the replication controller will be created. If we run the kubctl, get rc < Tomcat-ReplicationController > it will show the replication controller details.

**replicas: 3** → This helps the replication controller to understand that it needs to maintain three replicas of a pod at any point of time in the pod lifecycle.

**name: Tomcat** → In the spec section, we have defined the name as tomcat which will tell the replication controller that the container present inside the pods is tomcat.

**containerPort: 7474** → It helps in making sure that all the nodes in the cluster where the pod is running the container inside the pod will be exposed on the same port 7474.

**Replica Set**

Replica Set ensures how many replica of pod should be running. It can be considered as a replacement of replication controller. The key difference between the replica set and the replication controller is, the replication controller only supports equality-based selector whereas the replica set supports set-based selector.

**Setup Details**

**apiVersion**: extensions/v1beta1 → In the above code, the API version is the advanced beta version of Kubernetes which supports the concept of replica set.

**kind: ReplicaSet** → We have defined the kind as the replica set which helps kubectl to understand that the file is used to create a replica set.

**tier: Backend** → We have defined the label tier as backend which creates a matching selector.

**{key: tier, operation: In, values: [Backend]}** → This will help matchExpression to understand the matching condition we have defined and in the operation which is used by matchlabel to find details.