## Kubernetes Cluster

For this project, a Kubernetes cluster had to be set up which consists of a master and at least two worker nodes. For learning better about the cluster, I decided to deploy my web application that consists of several components like: web server that serves static files, login server that takes care of authenticating users, and two other backend servers (compile, and general) that interacts with database and performs different app logic.

Before deploying my app to the cloud, I thought about doing it in my local environment. For that I had to install a hypervisor called *hyperkit*. Once that was installed, I installed *minikube* which also installs *kubectl* (a command line tool that can talk to API Server of the Master process or node; it can interact with both minikube cluster or actual production ready cloud cluster) as a dependency. Minikube can run master and all worker processes in a single computer instead of running many virtual machines just for testing purposes.

# Starting minikube cluster using hyperkit as the hypervisor

minikube start --vm-driver=hyperkit

Now that kubectl is configured to use minikube cluster let’s get started and create the cluster.

# Viewing the master node

pawanbhatta@pawans-MBP ~ % kubectl get nodes

NAME STATUS ROLES AGE VERSION

minikube Ready control-plane,master 16m v1.20.2

# Doing the same thing using minikube

pawanbhatta@pawans-MBP ~ % minikube status

minikube

type: Control Plane

host: Running

kubelet: Running

apiserver: Running

kubeconfig: Configured

timeToStop: Nonexistent

Now to create the worker nodes, we can proceed to use kubectl commands which works with minikube behind the hood.

#Creating my first deployment using kubectl

pawanbhatta@pawans-MBP ~ % kubectl create deployment nginx-deployment --image=nginx

deployment.apps/nginx-deployment created

Here the latest nginx image will be downloaded from docker hub and the image will be used by pod(s) to run nginx container(s). Here by default only one replica will be created.

#Viewing deployment status

pawanbhatta@pawans-MBP ~ % kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 1/1 1 1 4m17s

#Viewing pods

pawanbhatta@pawans-MBP ~ % kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-deployment-84cd76b964-vldjg 1/1 Running 0 5m28s

#Viewing replicaset

pawanbhatta@pawans-MBP ~ % kubectl get replicaset

NAME DESIRED CURRENT READY AGE

nginx-deployment-84cd76b964 1 1 1 7m29s

In short, deployment manages replicaset which manages all the replica of pods. Pod is an abstraction of container. Pod can have one or more docker container running within it. However, in Kubernetes we work mostly with deployments.

#Editing the deployment.

pawanbhatta@pawans-MBP ~ % kubectl edit deployment nginx-deployment

deployment.apps/nginx-deployment edited

Here I edited the deployment configuration. Instead of default single replica, I bumped the replica number to four. All additional 3 replicas were automatically created for me as soon as the deployment configuration file was changed.

#Viewing all four replicas created

pawanbhatta@pawans-MBP ~ % kubectl get pod

NAME READY STATUS RESTARTS AGE

nginx-deployment-84cd76b964-dbczn 1/1 Running 0 4s

nginx-deployment-84cd76b964-hv86f 1/1 Running 0 4s

nginx-deployment-84cd76b964-t5jgf 1/1 Running 0 4s

nginx-deployment-84cd76b964-vldjg 1/1 Running 0 21m

However, there is better way to create/edit a deployment. It is by creating a yaml deployment configuration file. Since the configuration file is crucial in Kubernetes let’s try to break it down.

Each configuration file has three parts:

1. Metadata: It contains name of the component. This has to be unique if we want to run a new component of the same kind.
2. Specification: All the information about the functionality or settings of a component is specified here. Attributes that can be specified here may vary according to the kind of components we want to run. For instance: if we were creating a service instead of deployment, we would have different sets of attributes being applied in specification field. One of the attributes within specification is *template* which specifies the configuration of individual pods. It contains information like which image(s) should the pod use to run the container within itself, or which port(s) should the pod be listening to.
3. Status: This is automatically generated by Kubernetes with the help of etcd (also known as the brain of Kubernetes cluster). It contains information about the current state of the deployment. It compares the current status with the desired configuration specified in the specification part of the yaml file. If they are not the same, Kubernetes will perform necessary task to make current state equal to desired state.

Below is the sample of the file which when applied should run 3 replicas of nginx pods.

# Deployment configuration yaml file

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment2

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

#Creating yet another deployment using new yaml file

pawanbhatta@pawans-MBP CloudComputing % kubectl apply -f nginx-deployment.yaml

deployment.apps/nginx-deployment2 created

#Viewing all deployments created so far

pawanbhatta@pawans-MBP CloudComputing % kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 4/4 4 4 12h

nginx-deployment2 3/3 3 3 42s

#Service configuration yaml file

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx

ports:

- protocol: TCP

port: 80

targetPort: 8080

#Creating service that forwards requests to all the related pods

pawanbhatta@pawans-MBP CloudComputing % kubectl apply -f nginx-service.yaml

service/nginx-service created

#Viwing all running srrvices

pawanbhatta@pawans-MBP CloudComputing % kubectl get service

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 13h

nginx-service ClusterIP 10.97.69.24 <none> 80/TCP 2m54s

#Checking if service is forwarding requests to all the related pod instances

pawanbhatta@pawans-MBP CloudComputing % kubectl describe service nginx-service

Name: nginx-service

Namespace: default

Labels: <none>

Annotations: <none>

Selector: app=nginx

Type: ClusterIP

IP: 10.97.69.24

Port: <unset> 80/TCP

TargetPort: 8080/TCP

Endpoints: 172.17.0.7:8080,172.17.0.8:8080,172.17.0.9:8080

Session Affinity: None

Events: <none>

#Verifying the IP address of the pods

pawanbhatta@pawans-MBP CloudComputing % kubectl get pod -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

nginx-deployment2-66b6c48dd5-6q5fk 1/1 Running 0 59m 172.17.0.8 minikube <none> <none>

nginx-deployment2-66b6c48dd5-hjbc6 1/1 Running 0 59m 172.17.0.9 minikube <none> <none>

nginx-deployment2-66b6c48dd5-wsxh6 1/1 Running 0 59m 172.17.0.7 minikube <none> <none>

Endpoints attribute of the nginx-service contains three private IP addresses where each of the pods that we created are listening to which verifies that we have successfully connected our service to all related pods.

Deleting deployments and services that we created is easy with the help of kubectl commands. It is done as following:

#Deleting deployment and service

pawanbhatta@pawans-MBP CloudComputing % kubectl delete -f nginx-deployment.yaml

deployment.apps "nginx-deployment2" deleted

pawanbhatta@pawans-MBP CloudComputing % kubectl delete -f nginx-service.yaml

service "nginx-service" deleted

pawanbhatta@pawans-MBP CloudComputing % kubectl delete deployment nginx-deployment

deployment.apps "nginx-deployment" deleted