**KUBERNETES**

[**https://medium.com/stakater/k8s-deployments-vs-statefulsets-vs-daemonsets-60582f0c62d4**](https://medium.com/stakater/k8s-deployments-vs-statefulsets-vs-daemonsets-60582f0c62d4)

[**https://www.redhat.com/en/topics/containers/kubernetes-architecture**](https://www.redhat.com/en/topics/containers/kubernetes-architecture)

Important links :

<https://www.kubermatic.com/blog/exposing-apps-with-services/>

<https://www.kubermatic.com/blog/keeping-the-state-of-apps-5/>

<https://www.kubermatic.com/blog/introduction-to-pods/>

<https://www.kubermatic.com/blog/introduction-to-kubernetes-replicasets/>

<https://www.kubermatic.com/blog/introduction-to-kubernetes-deployment/>

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-update-deployment> - need to look into

<https://www.kubermatic.com/blog/keeping-the-state-of-apps-5-introduction-to-storage-classes/> - need to look into

<https://www.kubermatic.com/blog/keeping-the-state-of-apps-4-persistentvolumes-and-persistentvolum/>

<https://www.kubermatic.com/blog/keeping-the-state-of-apps-5/> - need to look into

<https://www.kubermatic.com/blog/keeping-the-state-of-apps-1-introduction-to-volume-and-volumemounts/>

**VERSIONS THAT WE USED :**

1.17

1.18

1.19

PROVIDER :

provider "kubernetes" {

host = data.aws\_eks\_cluster.core.endpoint

cluster\_ca\_certificate = base64decode(data.aws\_eks\_cluster.core.certificate\_authority[0].data)

token = data.aws\_eks\_cluster\_auth.core\_auth.token

load\_config\_file = false

version = "1.13.1"

}

provider "helm" {

version = "1.2.3"

}

**What is k8s ?**

Kubernetes is **an open-source container orchestration platform.** It manages complete life cycle of a container.

**K8s architecture:**

Graphical user interface, application

Description automatically generated

**Control plane:**

This is the center of the k8s cluster where we have all the components that controls the working of the cluster.

**Kube api server:**

This is like the front end of the k8s control plane. For any operation/action to be performed we will call these api’s. and the kube api server will authenticate the request and processes it.

**Kube scheduler:**

when a new container/pod is added then where it should be added? This is handled by kube scheduler.

**Kube controller manager:**

Controller manager has the controller functions.

**Etcd:**

This is the key value store that maintains the current state info of the cluster.

**Nodes:**

This is the compute capacity of the cluster. Pods are scheduled on nodes.

**Pod:**

*Pods* are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod is a group of one or more [containers](https://kubernetes.io/docs/concepts/containers/), with shared storage and network resources, and a specification for how to run the containers.

**Container run time engine:**

To run containers in a pod we need a CRE on nodes. It is docker in this case

**Kubelet:**

This makes sure that the pods are running on a node and also this communicates with the control plane for any actions.

**Kube-proxy:**

This handles network communication in k8s

**Namespace:**

This is the additional layer of separation. When we need isolation or grouping of few resources we can use different namespaces for different groups or types of resources.

We can even restrict access of users based on namespaces.

Switch between namespaces:

kubectl config set-context --current --namespace=testing

**How is k8s different from Docker ?**

Docker is open-source technology—and a container file format—for automating the deployment of applications as portable, self-sufficient containers that can run in the cloud or on-premises.

Docker Creates containerized applications.

K8s manages the orchestration of those containers.

Kubernetes lets you orchestrate a cluster of virtual machines and schedule containers to run on those virtual machines based on their available compute resources and the resource requirements of each container. Containers are grouped into pods, the basic operational unit for Kubernetes. These containers and pods can be scaled to your desired state and you are able to manage their lifecycle to keep your apps up and running.

**EKS CLUSTER CREATION USING EKSCTL**

EKSCTL is a command line tool which is used to provision, update, upgrade, manage amazon eks cluster.

For Creation of EKS using AWS CLI :

Prerequisites:

AWS CLI

Vpc

Public and private subnets

Nat gateway

Iam roles to attach to cluster

For Creation of EKS cluster using EKSCTL :

Kubectl

Eksctl

Iam permissions

AWS cli

Command to create :

**eksctl create cluster \**

**--name** *my-cluster* **\**

**--region** *us-west-2* **\**

**--nodegroup-name mymanagednodes \**

**--node-type t3.small \**

**--managed**

It creates – vpc, subnets in all the availability zones, sgs, route tables, policies, nat gateway, internet gateway, EC2(worker nodes)

Creation of node groups using eksctl

eksctl create nodegroup --cluster=mycluster –name=mynodegroup

**EKS CLUSTER CREATION USING TERRAFORM**

VPC public subnet

public subnet

route tables associations -🡪 1 for each subnet

internet gateway -🡪 attach to vpc

eks cluster

cluster iam role – iam policy – 2 policies – amazon eks cluster policy, amazon eks service policy

eks node group

node iam role – iam policy – 3 policies – EC2 container registry read only, amazon eks worker node policy, amazon eks cni policy

security group

path for code : <https://github.com/hashicorp/terraform-provider-aws/blob/v2.51.0/examples/eks-getting-started/eks-cluster.tf>

**STEPS TO CREATE EKS CLUSTER USING TERRAFORM:**

**Networking part :**

Vpc

Subnets – both public / 1 public 1 private

(public can be used to place external lb, private for internal lb)

**IAM roles and policies:**

**Cluster role –**

arn:aws:iam::aws:policy/AmazonEKSClusterPolicy

arn:aws:iam::aws:policy/AmazonEKSServicePolicy

**Node role –**

arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy

arn:aws:iam::aws:policy/AmazonEKS\_CNI\_Policy

arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly

**Security groups :**

**Cluster sg**

**Node sg**

**Git url :** [**https://github.com/Laharivegiraju22/Kubernetes-DevOps/blob/main/EKS\_Creation/code/module/eks\_setup/node\_sg.tf**](https://github.com/Laharivegiraju22/Kubernetes-DevOps/blob/main/EKS_Creation/code/module/eks_setup/node_sg.tf)

**Create cluster**

**Create node :**

|  |
| --- |
| apiVersion:v1 |
|  | kind: ConfigMap |
|  | metadata: |
|  | name: aws-auth |
|  | namespace: kube-system |
|  | data: |
|  | mapRoles: | |
|  | - rolearn: ${aws\_iam\_role.eks-node.arn} |
|  | username: system:node:{{EC2PrivateDNSName}} |
|  | groups: |
|  | - system:bootstrappers |
|  | - system:nodes |
|  | mapUsers: | |
|  | - userarn: arn:aws:iam::228699574855:root |
|  | username: root\_user |
|  | groups: |
|  | - system:masters |
|  | - userarn: arn:aws:iam::228699574855:user/Shyam |
|  | username: shyam |
|  | groups: |
|  | - system:masters |
|  | - userarn: arn:aws:iam::228699574855:user/L-jenkinsuser |
|  | username: L-jenkinsuser |
|  | groups: |
|  | - system:masters |

**NOTE: This role attachment is very important for node to attach to cluster :**

|  |
| --- |
| mapRoles: | |
| - rolearn: ${aws\_iam\_role.eks-node.arn} |
| username: system:node:{{EC2PrivateDNSName}} |
| groups: |
| - system:bootstrappers |
| - system:nodes |

**Connect to cluster from terminal**

1. Aws configure with the account credentials with which the cluster is created
2. aws eks --region region update-kubeconfig --name cluster\_name

**RBAC**

**Difference between IAM and RBAC:**

IAM is for authentication – identifies who(aws operations)

RBAC is for authorization – identifies wat operations can be performed in cluster by the user (inside k8s cluster wat all user can do)

Roles & Cluster Role

Cluster Role : available through out cluster

Role : available in specific namespace

Example :

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: readOnlyRole

rules:

- apiGroups: [""] #indicates the core API group

resources: ["configmaps", "nodes", "namespaces"]

verbs: ["list"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name : readOnlyRole

subjects: # You can specify more than one "subject"

- kind: User ## kind can be user or group

name: lahari\_k8suser ##This name must be used for binding IAM ROle with aws-auth configmap

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: readOnlyRole

apiGroup: rbac.authorization.k8s.io

aws\_auth :

apiVersion: v1

kind: ConfigMap

metadata:

name: aws-auth

namespace: kube-system

data:

mapRoles: |

- rolearn: ${aws\_iam\_role.eks-node.arn}

username: system:node:{{EC2PrivateDNSName}}

groups:

- system:bootstrappers

- system:nodes

mapUsers: |

- userarn: arn:aws:iam::228699574855:root

username: root\_user

groups:

- system:masters

# user that is mapped in RBAC

- userarn: arn:aws:iam::228699574855:user/lahari\_k8suser

username: lahari\_k8suser

- userarn: arn:aws:iam::228699574855:user/L-jenkinsuser

username: L-jenkinsuser

groups:

- system:masters

**Service account:**

Service account identifies a pod. i.e., if a pod wants to communicate with k8s API server then It needs service account to authenticate itself.

By default whenever a namespace is created a default service account is also created. When we don’t mention any service account for a pod it takes this default service account and communicates with k8s apiserver.

Serviceaccount.yaml :

apiVersion: v1

kind: ServiceAccount

metadata:

name: mysvcaccount

namespace: mysvc

kubectl apply -f Serviceaccount.yaml

After creation with -o yaml we can see the service account as follows:

A2ML44759M:yamls m1055938$ kubectl get serviceaccounts mysvcaccount -n mysvc -o yaml

apiVersion: v1

kind: ServiceAccount

metadata:

annotations:

kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"v1","kind":"ServiceAccount","metadata":{"annotations":{},"name":"mysvcaccount","namespace":"mysvc"}}

creationTimestamp: "2021-11-30T05:46:14Z"

managedFields:

- apiVersion: v1

fieldsType: FieldsV1

fieldsV1:

f:secrets:

.: {}

k:{"name":"mysvcaccount-token-5gvqp"}:

.: {}

f:name: {}

manager: kube-controller-manager

operation: Update

time: "2021-11-30T05:46:14Z"

- apiVersion: v1

fieldsType: FieldsV1

fieldsV1:

f:metadata:

f:annotations:

.: {}

f:kubectl.kubernetes.io/last-applied-configuration: {}

manager: kubectl-client-side-apply

operation: Update

time: "2021-11-30T05:46:14Z"

name: mysvcaccount

namespace: mysvc

resourceVersion: "2802"

selfLink: /api/v1/namespaces/mysvc/serviceaccounts/mysvcaccount

uid: 09de72a0-b4ec-4451-901f-b2f95b8dea70

secrets:

- name: mysvcaccount-token-5gvqp

Secret token gets created automatically, If we view the secret it will be as follows:

A2ML44759M:yamls m1055938$ kubectl get secrets mysvcaccount-token-5gvqp -n mysvc -o yaml

apiVersion: v1

data:

ca.crt: 

namespace: bXlzdmM=

token: 

kind: Secret

metadata:

annotations:

kubernetes.io/service-account.name: mysvcaccount

kubernetes.io/service-account.uid: 09de72a0-b4ec-4451-901f-b2f95b8dea70

creationTimestamp: "2021-11-30T05:46:14Z"

managedFields:

- apiVersion: v1

fieldsType: FieldsV1

fieldsV1:

f:data:

.: {}

f:ca.crt: {}

f:namespace: {}

f:token: {}

f:metadata:

f:annotations:

.: {}

f:kubernetes.io/service-account.name: {}

f:kubernetes.io/service-account.uid: {}

f:type: {}

manager: kube-controller-manager

operation: Update

time: "2021-11-30T05:46:14Z"

name: mysvcaccount-token-5gvqp

namespace: mysvc

resourceVersion: "2801"

selfLink: /api/v1/namespaces/mysvc/secrets/mysvcaccount-token-5gvqp

uid: 47fef850-3667-4cc9-b9df-6789d213b2dd

type: kubernetes.io/service-account-token

**Command to check your access in cluster:**

kubectl auth can-I create pod -n namespace –as system:serviceaccount:namespace:name\_of\_svcaccount

**Example:**

kubectl auth can-i create pod -n mysvc --as system:serviceaccount:mysvc:mysvcaccount

**K8S NETWORKING**

Container image 🡪 deploy 🡪 k8s cluster

**CRI** : container run time interface “**pulls the image”** and then “**runs the container”** on ECS or EKS (eg: docker etc)

**CNI** : To access the created pod we need ip address. CRI only creates the container, CNI is the interface between CRI and network.

Once the container is up CRI calls the CNI to add the container into network. Then CNI attaches an ip address to the new container and adds it to the n/w.

Each pod has an ip and containers within the pod shares the ip of the pod. If multiple containers are there in 1 pod then the conatiners use same ip but get exposed in different ports.

**FOR EKS :**

When eks cluster is created by default Amazon VPC Container Network Interface (CNI)

is installed.

**K8s concepts**

**RUNNING DOCKER CONTAINERS ON K8S :**

kubectl run myimage -it –image busybox – sh

Image gets downloaded into the worker node on which this pod is being scheduled.

To delete u can just run

kubectl delete deployment myimage

**Kubectl port forwarding :**

**Kubectl port-forward** allows you to access and interact with internal Kubernetes cluster processes from your localhost.

**Syntax :**

kubectl port-forward podname LOCAL\_PORT:REMOTE\_PORT

kubectl port-forward nginxpodname 8080:80

similarly we can use following,

kubectl port-forward deployment/redis-master 6379:6379

kubectl port-forward rs/redis-master 6379:6379

kubectl port-forward svc/redis-master 6379:6379

we can scale the deployment after creation,

kubectl scale deploy nginx –replicas 2

creating service using command – kubectl expose deployment nginx –type NodePort –port 80

Creating yaml file from existing deployment ,

kubectl get deployment nginx -o yaml > nginx.yaml

Creating deployment with file,

kubectl create -f pathoffile

**UDEMY COURSE EKS**

**SERVICE :**

A pod if it has to connect with another pod connecting via IP address is not an ideal way as pod or node might go down and new ones gets created with new ip address.

So for a pod to connect with another pod we use “service”.

Service exposes pods for it to connect with another pod. It identifies the pod by using “label selector”

Any pod that gets created with the label specified in service will be exposed to other pods or internet via this service.

It acts like a load balancer.

Moto of all service types is to discover and distribute traffic across underlying pods

**TYPES OF SERVICES:**

Loadbalancer

ClusterIP

NodePort

Clusterip is the default type, it is accessible from only within the cluster

NodePort is accessible from outside cluster

Range for nodeport: 30000-32767

We mention 3 ports in yaml file for service type NodePort :

nodePort : 32000 🡪 this is the port of node that is exposed to the external world

port : 80 🡪 service port with which node port communicates when external request is made (nodeport service is actually running on port 80)

targetPort : 80 🡪 this is port of pod where actual application is running (container port)

DIFFERENCE BETWEEN SELF-MANAGED AND AMAZON MANAGED MASTER NODE:

**Self managed:**

Need to make control plane high available

Scale control plane when needed

Keep etcd up and running

We have to manage the security patches, replace failed ec2 instances etc all these has to be done manually.

**AWS managed:**

EKS is nothing but aws manages the k8s control plane

Master node is maintained by Amazon

**NODE GROUPS(WORKER NODES):**

**Self-managed node groups:**

You maintain worker ec2s

You orchestrate version upgrade, security patching, AMI rehydration, keeping pods up during upgrade

Can use custom AMI

**Managed node groups:**

AWS manages worker EC2S

Aws provides AMI with security patches, version upgrade

AWS manages pod disruption during upgrade

Doesn’t work with custom AMI

**AWS fargate:**

No worker EC2

You define and deploy pods

Container + serverless

Node sizes that are currently used in dyson: number of pods depends on node type

small = "t3.2xlarge"

medium = "t3.2xlarge"

large = "r5.4xlarge"

**HPA :**

Horizontal pod autoscaler scales the pods. If the cpu utilization of pod increases over 30% it scales up the pod. We can manage the cpu limit and min, max pods in a manifest file of hpa.

**CLUSTER AUTOSCALER:**

Cluster autoscaler is at cluster level it scales up the number of nodes if cpu utilization increases by 50%.

**VPA:**

Vertical scaling means increase the size of pod not number of pods.

For eg: increase in cpu,memory etc

This restarts the pod and brings it up with new size. This should not be used in prod as it restarts the pod.

**INGRESS:**

[Ingress](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/#ingress-v1-networking-k8s-io) exposes HTTP and HTTPS routes from outside the cluster to [services](https://kubernetes.io/docs/concepts/services-networking/service/) within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

In order for the Ingress resource to work, the cluster must have an ingress controller running.

We can have multiple ingress controllers in a cluster and ingress resource can choose which one to use

url for creating ingress controller:

<https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/>

First step is to create a ingress controller it gets deployed as a new deployment(pod). It doesn’t create any loadbalancer in aws it gets only created when ingress resource is deployed.

**HELM**

It is a package manager for k8s.

Link for helm basic tutorial : <https://www.youtube.com/watch?v=-ykwb1d0DXU>