**What is Kubernetes?**

Kubernetes is an open-source container orchestration tool or system that is used to automate tasks such as the management, monitoring, scaling, and deployment of containerized applications. It is used to easily manage several containers (since it can handle grouping of containers), which provides for logical units that can be discovered and managed.

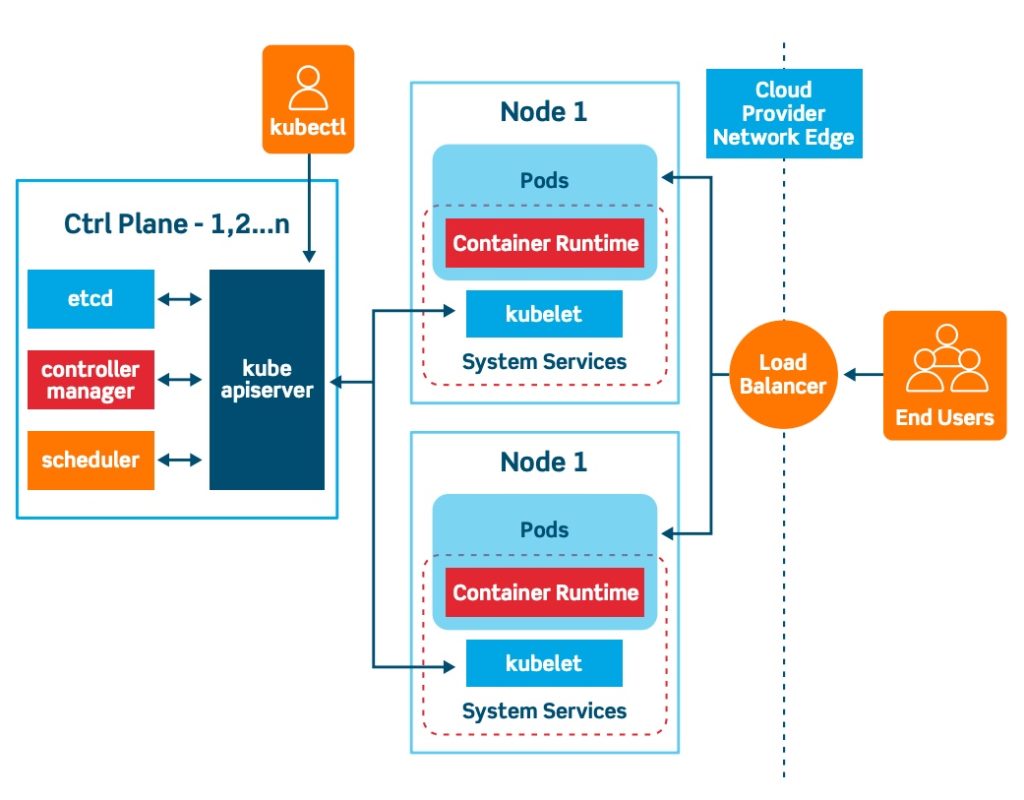
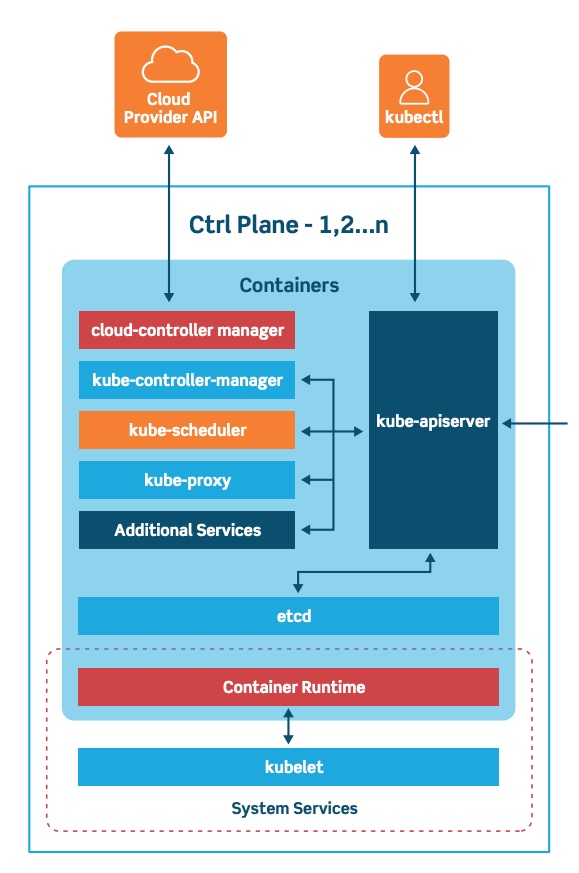
**Q. Kubernetes Features**

* Automated Scheduling: K8 provides advanced scheduler to launch container on based on their resource requirements and other constraints.
* Self-Healing Capabilities: Rescheduling, Replacing and restarting the containers which are died.
* Automated rollouts and rollback: supports rollouts and rollbacks for the desired state of containerized application.
* Horizontal scaling and load balancing: k8 can scale up and scale down the application as per the requirements.
* Service Discovery & Load Balancing: K8 will automatically assign IP addresses to containers and a single DNs name for a set of containers that can load-balance traffic inside the cluster.
* Storage Orchestration: local storage, public cloud provider such as aws, nfs, etc.

**Q. How Container orchestration is beneficial?**

* Provisioning and deployment of containers
* Upscaling or removing containers to divide application load evenly all across host infrastructure
* Redundancy and availability of containers
* Moving of containers from one host to another in case there is a shortage of resources in a host (or when the host dies)
* Allocation resources across containers
* Health monitoring of containers and hosts
* Externally exposing services running in a container to the outside world
* Load balancing of service discovery across containers
* Configuring an application relative to the containers running it.

# Q. Explain K8 Architecture?

**Kubectl**

* Kubectl is a CLI (command-line interface) that is used to run commands against Kubernetes clusters.
* As such, it controls the Kubernetes cluster manager through different create and manage commands on the Kubernetes component

**Master Node:**

* API
* scheduler
* controller-manager
* etch

1. Master Nodes:

* The master Node is responsible for the management of k8 cluster, it is mainly entry point for all administrative tasks and it handles the orchestration of the worker nodes. There can be more than one master node in the cluster for fault tolerance.

1. API Server:

* Kube API server interacts with API, it’s a frontend of the k8 control plane.
* Communication center for developers, sysadmin and other kubernetes components.

1. Scheduler

* Scheduler watches the pods and assigns the pods to run on specific hosts.

1. Controller-manager
2. Etcd

* Distributed key value data store (database) of k8.
* K8 cluster state (info).some of the dada maintained in etcd is nodes, deployment, services and nodes.

**Worker Nodes:**

* Pod1 ,Pod2 ,Pod3
* Docker
* Kubelet
* kube-proxy

K8s cluster Setup

Local K8 cluster /Single node clusters -- (for poc)

* minikube
* kind

Self-Managed / bare Metal K8's clusters

* Kubeadm -- Using kubeadm we can setup multi node cluster.
* kuberspray -- Ansible way of configuring the clusters(gain it is using kubeadm internal and playbooks internal)

Managed K8's clusters

* EKL -- Elastic K8 service (AWS Cloud)
* AKS -- Azure kubernetes service (Microsoft Azure cloud)
* GKE -- Google Kubernetes Engine (Google Cloud Platform)
* Rancher -- Using Rancher we can create k8s cluster in any kind of infrastructure on prem or cloud env.
* KOPS -- Kops can be used to setup production ready highly available k8s clusters in AWS as of now .It's supporting Azure cloud using beta version.
* KOPS internally create Auto scaling Group & its launch Configurations.

1. One ASG & its L.C for Master’s

2. One ASG & its L.C for Worker's

**k8's Resources/Objects**

1. Name spaces
2. POD's

* Pod life cycle

1. Service

* cluster IP
* Node Port
* Headless
* Load balancer

1. Controller

* Replication Controller
* Replica set
* Daemon set (Agent Based – Global Mode)
* Deployment (Stateless Application)
* Recreate
* Rolling Update
* StatefulSet

1. Network Policies

* Ingress
* Egress

1. Auto Scalar

* Pod Auto scalar
* HPA
* VPA
* AWS Auto scalar

1. Volumes

* Empty Dir
* Host path
* Nfs
* EFS

1. Configuration as volumes

* ConfigMaps
* Secrets

1. Persistent volume
2. Persistent claims
3. Access Modes & RECLAIM Polices
4. Scheduling

* Node selector
* Node Affinity
* Pod Affinity
* Pod Anti Affinity

1. Taints and Tolerations

* No scheduled
* No execute
* Prefer Node scheduled

1. Node Maintain

* Cordon
* Uncord on
* Drain

1. Resources Quota and Limits Ranges
2. Ingress Rules
3. Ingress Control

* HA Proxy
* Nginx ingress

1. RBAC

* Role
* Role Binding
* Cluster
* Cluster Binding

1. Helm

* Helm charts
* Helm repo

**Control plane**

Protocol Direction Port Range Purpose Used By

TCP Inbound 6443 Kubernetes API server All

TCP Inbound 2379-2380 etcd server client API kube-apiserver, etcd

TCP Inbound 10250 Kubelet API Self, Control plane

TCP Inbound 10259 kube-scheduler Self

TCP Inbound 10257 kube-controller-manager Self

**Worker node(s)**

Protocol Direction Port Range Purpose Used By

TCP Inbound 10250 Kubelet API Self, Control plane

TCP Inbound 30000-32767 Node Port Services† All

**Kubernetes Objects and Explanation**

1. **Name Spaces**

* Namespaces are used for dividing cluster resources between multiple users. They are meant for environments where there are many users spread across projects or teams and provide a scope of resources.
* In Kubernetes, *namespaces* provides a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects *(e.g. Deployments, Services, etc.)* and not for cluster-wide objects *(e.g. StorageClass, Nodes, Persistent Volumes, etc.)*

**Kubernetes starts with four initial namespaces**:

* default
* kube-node-lease
* Kube-public
* kube-system
* Default: The default namespace for Kubernetes objects that don't belong to other namespaces.
* kube-system: A namespace for objects created by the Kubernetes system.
* kube-public: A namespace readable by all users and mostly reserved for cases when some resources need to be visible and readable publicly throughout the cluster.

**To set the namespace for a current request, use the --namespace flag.**

For example:

* kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
* kubectl get pods --namespace=<insert-namespace-name-here>

Create a new YAML file called my-namespace.yaml with the contents:

apiVersion: v1

kind: Namespace

metadata:

name: <insert-namespace-name-here>

Alternatively, you can

Create namespace using below command:

* Kubectl create namespace <insert-namespace-name-here>

Deleting namespace

* kubectl delete namespaces <insert-some-namespace-name>

Get namespace

* kubectl get namespaces

Q. Describe what is namespace in Kubernetes and why is it used?

* It's like an isolation process. E.g. If you namespaces dev and prod, you can have pods named foo in both namespaces and there is no conflict. (In the same cluster)
* In Kubernetes, you can have the dev team their own namespace and prod can have its own namespace.

Q. What namespace does kube-scheduler reside in?

* kube-system

Q. On a fresh cluster, which namespace holds the system resources like kube-dns?

* Kube-system (Here the interviewer is probing for your hands on experience. If you work on Kubernetes regularly, you will know this)

Q. kubectl command to show labels of all pods in default namespace:

* kubectl get pods --show-labels

Q. Whenever you run a kubectl command, it runs in the default namespace. How do you make in run in a different namespace?

* Use -n namespacename (to whatever kubectl command you are running.)

Q. Command to create a namespace:

* kubectl create ns foobar # create a namespace

Q. Kubectl command to list all the pods in foo namespace:

* kubectl get pods -n foo

Q. Command to see a list of running pods in the custom namespace:

* kubectl get pods –n <Namespace>

Q. Command to describe a role:

* kubectl describe role foo -n foo\_namespace

Q. How does Kubernetes do DNS internally?

* Kube-system namespace has pod (pods) that DNS servers.

You can see them by running:

# kubectl get po -A | grep dns

Q. How to see what network policies you have in default namespace?

* kubectl get netpol

Q. How do you list out all pods running in the namespace foo?

* kubectl get pods -namespace=foo

Q. What namespace does kube-scheduler reside in?

* kube-system

Q. How to delete all pods in kubernetes namespaces?

Delete all the pods in a single namespace:

* kubectl delete --all pods --namespace=foo

1. **PODS**

* A pod always runs on a node.
* 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, with shared storage and network resources, and a specification for how to run the containers
* Containers in the same pod share a local network and the same resources, allowing them to easily communicate with other containers.
* A pod is a group of one or more containers which will be running on some node.
* Each pod has its unique IP address within the cluster.

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

To create the Pod shown above, run the following command:

* kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml

Example: Pod Syntax

apiVersion: v1

kind: Pod

metadata:

name: <Podname>

labels:

<Key>: <value>

namespace: <nsName>

spec:

containers:

- name: <name of container>

image: <image>

ports:

- containerPort: <containerPort>

volumeMounts:

- name: <volname>

mountpath: <containerDirPath>

resoure:

requets:

cpu: 200m

memory: 256Mi

limits:

cpu: 500m

memory: 512Mi

volumes:

- name: <volName>

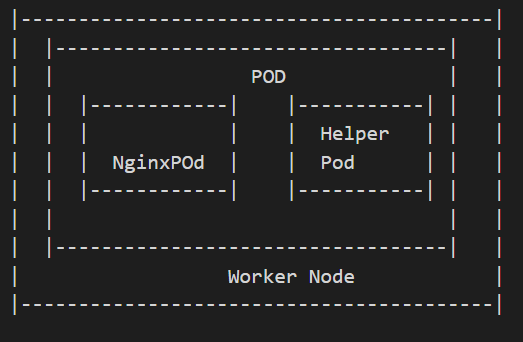
hostPath:

path: <Hostfolder>

* kubectl get pods -o wide
* kubectl describe pod <pod name>
* kubectl describe pod nginx

**Multi Container Pods**

* We can have multiple containers in a single POD, provided they are not same kind.

****

**Helper Container (Side car)**

* Data puller: pulls data required by main container.
* Data pusher : push data by collected by main container(logs)
* Proxies: writes static data to html files using helper container and reads using main container.

**Communication**

* + The two containers can easily communicate with each other easily as they are in same network space.
  + They can also easily share same storage space.

**Pod Life Cycle**

* Make a pod request to API server using a local pod definition file.
* The API server saves the info for the POD in ETCD.
* The scheduler finds the unscheduled pod and schedules it to node.
* Kubelet running on the node, sees the pods scheduled a fires up pod.
* PODs runs the container.

FDQN (Fully Qualified Domain Name)

If we want to communicate one pod in service A to another Pod in service B, with the help of FDQN, we can communicate

* <service Name>.<name Space>.svc.cluster.local
* curl -v <mavenwebappsvc>.<test-ns>.svc.sluster.locl:8080/maven-web-application/
* curl -v mavenwebappsvc.test-ns.svc.sluster.locl:8080/maven-web-application/

**Static pods**

* Static pods are managed directly by the kubelet and API server does not have any control over these pods. The kubelet is responsible to watch each static pod and restart it if it crashes.
* Static pods does not have any associated replication controllers, kubelet service itself watches it when it crashes and there is no health check for static pods.
* The main use for static pods is to run a self-hosted control plane: in other words, using the kubelet to supervise the individual control plane components.
* Static pods are always bound to one kubelet on a specific node.

**Note**

* Control planes can't controls it.

All default pods are static pods which controlled by kubelet

* etcd
* coredns
* kube-api
* kube-controller-manager
* kube-proxy
* kube-scheduler
* weave

Q. How service are identifies the pod. Which pods are having endpoints?

Ans: using Labels and selectors

labels:

<key>:<value>

* While creating service, we are using those labels as selectors.

Taints: pods can't deployed or scheduled in master node.

What is difference b/w kubectl create and kubectl apply?

Create will Create an Object if it's not already created. Apply will perform create if object is not created earlier. If it's already

Created it will update.

* kubectl apply (create & update)
* kubectl create -f <fileName.yml>
* kubectl update -f <fileName.yml

#### Q. If a container keeps crashing, how do you troubleshoot?

Answer: You can use --previous option with logs command to see the logs of a crashed container.

(kubectl logs --previous)

#### Q. What happens to containers if they use too much cpu or memory?

#### If they use too much memory, they are evicted.

#### If they use too much cpu, they are throttled.

1. **Networking Resources**

**Service**

* Service is responsible for making our pods discoverable inside the network or exposing them to the internet.
* A service identifies pods by its LabelSelector.

Types of service available.

* cluster IP
* NodePort
* Headless
* Loadbalancer

Q. What are Kubernetes Services?

Answer: A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called cluster IP). This address is tied to the lifespan of the Service, and will not change while the Service is alive.

* Syntax

apiVersion: v1

kind: service

metadata:

  name: <servicname>

  namespace: <namespaceName>

  labels:

    <key>:<value>

spec:

  type: <cluserIP/NodePort/LoadBalancer>

  selector:   #pods lables will used as selectors

    <key>: <value>

  ports:

   - port: <servicePort>

     targetPort: <containerPort>

Types of service available.

* Cluster IP: Used for communication between applications inside the k8s cluster.

Ex: Frontend app accessing backend application

* NodePort : Used for accessing application outside of the k8s cluster using wokernode ports

Ex: Accessing frontend application on browser

* Loadbalancer: Primarily for cloud providers to intergrate with their load balancer services.

Ex: Aws elastic load balancer

* Ingress : Ingress is an advanced load balancer which provide context path based routing, SSLRedirect and many more

Ex: AWS ALB

* External Name: To access externally hosted apps in k8 cluster.

Ex: Access aws RDB database endpoint by application present inside k8 cluster.

**3. a. ClusterIP**

* This helps in restricting the service within the cluster. It exposes the service within the defined Kubernetes cluster.
* Example : syntax

spec:

ports:

- port: 8080

nodePort: 31999

name: NodeportService

clusterIP: 10.20.30.40

**3. b. 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.
* Port range 30000-32767
* Example : syntax

spec:

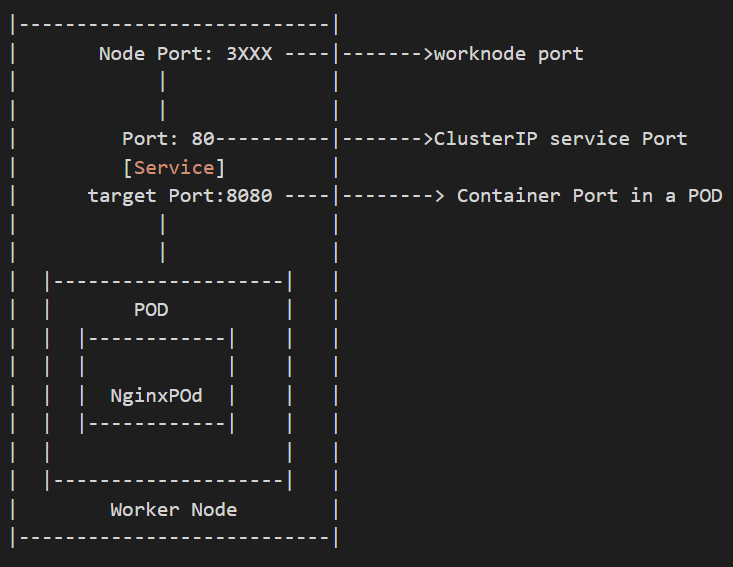
type:

ports:

- port: 8080

nodePort: 31999

name: portService



**Ports**

* 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.

**3. c. 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

* Loadbalancer: Primarily for cloud providers to intergrate with their load balancer services.

Ex: Aws elastic load balancer

* Ingress: Ingress is an advanced load balancer which provide context path based routing, SSL, SSLRedirect and many more

Ex: AWS ALB

* External Name: To access externally hosted apps in k8 cluster.

Ex: Access aws RDB database endpoint by application present inside k8 cluster.

**Service without Selector**

apiVersion: v1

kind: Service

metadata:

name: qa\_demo\_service # Service Name

spec:

ports:

- port: 8080

targetPort: 8001

**Service Config File with Selector**

apiVersion: v1

kind: Service

metadata:

name: qa\_demo\_service

spec:

selector:

application: "My Application" #Selector)

ports:

- port: 8080

targetPort: 8001

**COMMAND**

Get Service Info

* kubectl get service
* kubectl get svc

Get Public IP of Worker Nodes

* kubectl get nodes -o wide

Access the Application using Public IP

* http://<node1-public-ip>:<Node-Port>

Interact with a Pod

Verify Pod Logs

Get Pod Name

* kubectl get po

Dump Pod logs

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

Stream pod logs with -f option and access application to see logs

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

Connect to a Container in POD and execute commands

Connect to Nginx Container in a POD

* kubectl exec -it <pod-name> -- /bin/bash
* kubectl exec -it my-first-pod -- /bin/bash

Running individual commands in a Container

* kubectl exec -it <pod-name> env

Sample Commands

* kubectl exec -it my-first-pod env
* kubectl exec -it my-first-pod ls
* kubectl exec -it my-first-pod cat /usr/share/nginx/html/index.html

**Get YAML Output of Pod & Service**

Get pod definition YAML output (Get YAML Output)

* kubectl get pod my-first-pod -o yaml

Get service definition YAML output

* kubectl get service my-first-service -o yaml

**Clean-Up**

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

1. **Controller**

* Replication Controller
* Replica set
* Daemon set

Recommended for Agent based Applications (Global Mode same as in Docker swarm, each pod in each node)

* Deployment

Recommended for stateless Applications

* StatefulSet

Recommended for state full Applications

## Q. What is the difference between a replica set and a replication controller?

Both replica set and replication controller ensure that the given number of pod replicas are running at a given time. But the only point of difference between them is, replica leverages set-based selectors, while the replication controller uses equity-based controllers.

**Selector-based Selectors:**

* It filters the keys according to a set of values. The selector based selector locks for pods whose label is mentioned in the set.

**Equity-Based Selectors:**

* It filters by both label keys and values. The equity-based selector looks for the pods that have the exact phrase as mentioned in the label
* kubectl get rc
* kubectl get rc -n <namespace> 🡪 to see pods
* kubectl get all
* kubectl scale rc <rc Name> --replicas <no Of Replicas>
* kubectl describe rc <rc Name>
* kubectl delete rc <rc Name>

**4. b. Replica set**

What is difference b/w replica set and replication controller?

* Its next generation of replication controller. Both manages the pod replicas. But only difference as now is
* Selector support.

RC -- Supports only equality based selectors.

Key == value (Equal Condition)

selector:

app: javawebapp

RS -- Supports equality based selectors and also set based selectors.

key == value (Equal Condition)

\* Set Based

key in (value1, value2, value3)

key noting (value1)

selector:

matchLabels: # Equality Based

key: value

matchExpressions: # Set Based

- key: app

operator: IN

values:

- javawebpp

- javawebapplication

**COMMANDS**

* kubectl get rs
* kubectl get rs -n <namespace>
* kubectl get all
* kubectl apply -f mavenwebapprs.yml ---> deploy
* kubectl get rs -n test-ns ----> to see pods
* kubectl scale rs <rsName> --replicas <noOfReplicas>
* kubectl describe rs <rsName>
* kubectl delete rs <rsName>
* SYNTAX

apiVersion: v1

kind: ReplicaSet

metadata:

name: <replicationControllerName>

namespace: <nameSpaceName>

spec:

replicas: <no Of Replicas>

selector: # Pod labels will be used here as selector in rc

<label key>: <value>

**4.c. Daemon set**

DaemonSet

* Recommended for Agent based Applications(Global Mode same as in Docker swarm , each pod in each node)

**Syntax:**

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: <dSName>

spec:

selector: # To Match POD Labels.

matchLabels: # Equality Based Selector

<key>: <value>

matchExpressions: # Set Based Selector

- key: <key>

operator: <in/not in>

values:

- <value1>

- <value2>

template:

metadata:

name: <PODName>

labels:

<key>: <value>

spec:

- containers:

- name: <nameOfTheContainer>

image: <imageName>

ports:

- containerPort: <containerPort>

**4.d. Deployment**

Deployment

* Recommended for stateless Applications
  + Deployment will deploy a Replica set internally.
  + Updates pods(PodTemplateSpec)
  + Rollback to older Deployment version
  + pause and resume the deployment
  + Scale Deployment up and down.
  + Use the status of the deployment to determine state of replicas.
  + Clean up older RS that you dont need any more.

Deployment has two starages:-

* Recreate
* Rolling Update

**4.d.1. Deployment with Re-Create**

* In this type of very simple deployment, all of the pods are killed all at once and get replaced all at once with new one.

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: <dSName>

spec:

replicas: 2

selector: # To Match POD Labels.

matchLabels: # Equality Based Selector

<key>: <value>

strategy:

type: Recreate

template:

metadata:

name: <PODName>

labels:

<key>: <value>

spec:

- containers:

- name: <nameOfTheContainer>

image: <imageName>

ports:

- containerPort: <containerPort>

* kubectl get deployment
* kubectl get pods
* kubectl rollout status deployment <deployment Name>
* kubectl rollout history deployment <deployment Name>
* kubectl rollout history deployment <deployment Name> --revision 1
* kubectl scale deployment <deployment Name> --replicas <noOfReplicas>

We can update deployment using yml or using command

Update Deployment Image using command

* kubectl set image deployment <deployment Name> <container Name>=<imageNameWithVersion> --record

Roll back to previous revision

* kubectl rollout undo deployment <deployment Name> --to-revision

**4.d.2. Deployment with Rolling Update**

* The rolling deployment is the standard default deployment to kubernetes, it works by slowly, one by one, replacing pods of the previous version of your application with pods of the new version without any down time.
* A rolling update waits for new pods to become ready before if starts scaling down the old one.
* If there is a problem, the rolling update or deployment can be aborted without bringing the whole cluster down.
* Rolling deployment is default strategy.

apiVersion: apps/v1

kind: Deployment

metadata:

name: <ds name >

spec:

replicas: 2

revisionHistoryLimit: 5

selector:

matchLabels: # Equality Based Selector

<key>: <value>

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 1

maxSurge: 1

minReadySeconds: 30

template:

metadata:

name: <PODName>

labels:

<key>: <value>

spec:

- containers:

- name: <nameOfTheContainer>

image: <imageName>

ports:

- containerPort: <containerPort>

* kubectl get deployment
* kubectl get pods
* kubectl rollout status deployment <deployment Name>
* kubectl rollout history deployment <deployment Name>
* kubectl rollout history deployment <deployment Name> --revision 1
* kubectl rollout undo deployment <deployment Name> --to-revision 1
* kubectl scale deployment <deployment Name> --replicas <noOfReplicas>

Update Deployment Image using command

* kubectl set image deployment <deployment Name> <container Name>=<imageNameWithVersion> --record

**Blue Green Deployment**

* In this Deployment we will create a parallel env
* Routing request to new env
* If we need to roll back up, traffic is routed to old env.

**Deployment Strategies**

Blue/ Green (or Red / Black) deployments

* In a blue/green deployment strategy (sometimes referred to as red/black) the old version of the application (green) and the new version (blue) get deployed at the same time.
* When both of these are deployed, users only have access to the green; whereas, the blue is available to your QA team for test automation on a separate service or via direct port forwarding.
* After the new version has been tested and is signed off for release, the service is switched to the blue version with the old green version being scaled down.

# **Network Policy**

A network policy specification consists of four elements:

* Pod Selector: the pods that will be subject to this policy (the policy target) – mandatory
* Policy Types: specifies which types of policies are included in this policy, ingress and/or egress - this is optional but I recommend to always specify it explicitly.
* ingress: allowed **inbound** traffic to the target pods – optional
* egress: allowed **outbound** traffic from the target pods – optional

**Example Network Policy might look like this:**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: test-network-policy

namespace: default

spec:

podSelector:

matchLabels:

role: db

policyTypes:

- Ingress

- Egress

ingress:

- from:

- ipBlock:

cidr: 172.17.0.0/16

except:

- 172.17.1.0/24

- namespaceSelector:

matchLabels:

project: myproject

- podSelector:

matchLabels:

role: frontend

ports:

- protocol: TCP

port: 6379

egress:

- to:

- ipBlock:

cidr: 10.0.0.0/24

ports:

- protocol: TCP

port: 5978

1. **Auto scaler**
2. **Volumes**

**Volumes**

* Empty Dir
* Host path
* Nfs
* EFS
* awsElasticBlockStore(EBS)
* googlePersistantdisk
* azureFile
* azuredisk

**7. a. Emptydir**

* An emptyDir volume is first created when a pod is assigned to a node, and exists as long as pod is running on that node .As the name says, the emptyDir volume initially empty.

**7. b. Hostpath** (Node /worker node, the node where pods are running)

* A hostPath volumes mounts a file or directory for the host

Mongo db pod with volumes (HostPath)

containers:

- name: mongodbcontainer

image: mongo

ports:

- containerPort: 27017

- name: hostpathvol

mountPath: /data/db

volumes:

- name: hostpathvol

hostPath:

path: /tmp/mongodb

**7.c. Nfs**

* Configuration of NFS Server (Port 2049)

**Syntax:**

volumeMounts:

- name: mongodnfsvol

mountPath: /data/db

volumes:

- name: mongodnfsvol

nfs:

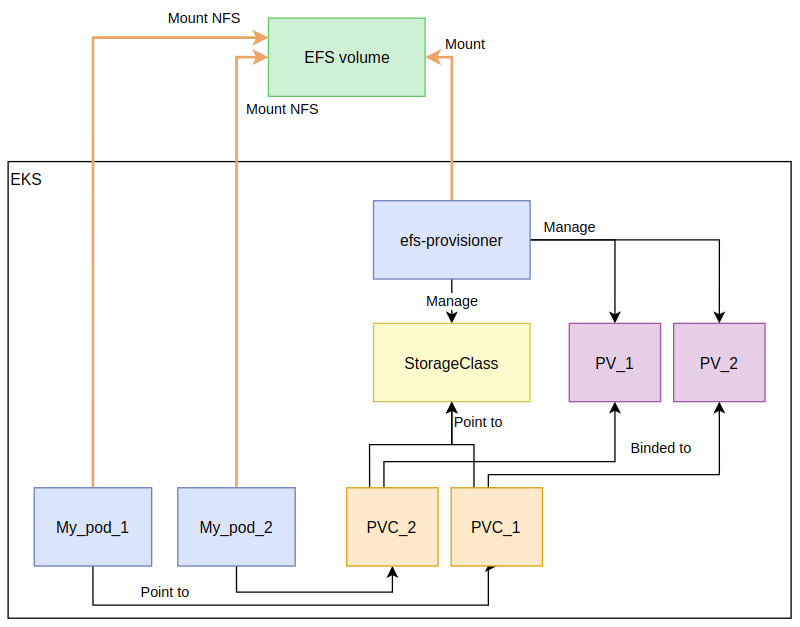
server: 172.31.38.98 # Update NFS Server IP and Path Based on your setup

path: /mnt/share

**7. d.EFS**

Here is the **architecture** that we will set up in the following sections:

URL: <https://www.padok.fr/en/blog/efs-provisioner-kubernetes> (for reference)



* Create a subdir in the EFS volume, dedicated to this PVC
* Create a PV with the URI of this subdir (Address of the EFS volume + subdir path) and related info that will enable pods to use this subdir as a storage location using NFS protocol
* Bind this PV to the PVC

**Note**

* Now when a pod is designed to use PVC, it will **use the PV’s info in order to connect** directly to the EFS volume and **use the subdir**.

### Use the EFS PVs

### Using your EFS-provisioner in order to **provide a pod with a PV** is quite simple. Just use the following yaml file:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-efs-volume

spec:

accessModes:

- ReadWriteMany

volumeMode: Filesystem

resources:

requests:

storage: 4Gi

storageClassName: aws-efs

You can then **create a pod** that uses this PVC with the following command:

apiVersion: v1

kind: Pod

metadata:

name: my-busybox

spec:

volumes:

- name: my-efs-volume

persistentVolumeClaim:

claimName: my-efs-volume

containers:

- name: my-container

image: busybox:latest

volumeMounts:

- name: efs-site

mountPath: /my-efs-volume

1. **Configuration as volumes**

* ConfigMaps
* Secrets

**ConfigMaps and Secrets**

A configMap is an API object used to store non-confidential data in key-value pair’s .Pods can consume configMaps as environment variables, command-line arguments or as configuration files in a volume.

We can create ConfigMap & Secretes in Cluster using command or also using yml.

ConfigMap Using Command

* kubectl create configmap springappconfig --from-literal=mongodbusername=devdb

Or using yml

apiVersion: v1

kind: ConfigMap

metadata: # we can define multiple key value pairs.

name: springappconfig

namespace: test-ns

data:

mongodbusername: devdb

---

Secret Using Command:

* kubectl create secret generic springappsecret --from-literal=mongodbpassword=devdb@123

OR Using Yml:

apiVersion: v1

kind: Secret

metadata:

name: springappsecret

namespace: test-ns

type: Opaque

stringData: # we can define multiple key value pairs.

mongodbpassword: devdb@123

---

echo "devdb@123" | base64

1. **Persistent Volume**

* It's a piece of storage (hostPath, nfs, EbS, azure file, azuredisk) in k8s cluster. PV exists independently from pod life cycle which is consuming.
* Persistent Volumes are provisioned in two ways, statically or dynamically.

1) Static Volumes (Manual Provisioning)

* As a k8's Administrator will create a PV manually so that PV’s can be available for PODS which requires.
  + Create a PVC so that PVC will be attached PV. We can use PVC with PODS to get an access to PV.

2) Dynamic Volumes (Dynamic Provisioning)

* It's possible to have k8's provision (Create) volumes (PV) as required. Provided we have configured storage class. So when we create PVC if PV is not available Storage Class will Create PV dynamically.
* They have life cycle independent of any individual pod that uses the pv.
* Note: we can’t use directly in the pod, for this we need to use the pvc.

apiVersion: v1

kind: PersistantVolume

metadata:

name: <PvName>

namespace: <nsname>

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

hostPath:

path: /mongo

---

nfs:

server: <server>

path: /mnt/nfs\_share

1. **PersistantVolumeClaim**

* If pod requires access to storage (PV), it will get an access using PVC. PVC will be attached to PV.
* A persistantVolumeClaim is request for storage by user.it is similar to a pod. Pods can request specific levels of resources (CPU and memory).
* Pods consume node resources and pvcs consume pv resource. Claims can request specific size and access modes (eg ReadWriteOnce,ReadOnlyMany,ReadWriteMany)

---

**Syntax:**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mongovol

namespace: test-ns

spec:

accessModes:

- ReadWriteOnce

Resources:

requests:

storage: 1Gi

hostPath:

path: /mongo

* PersistentVolume – the low level representation of a storage volume.
* PersistentVolumeClaim – the binding between a Pod and PersistentVolume.
* Pod – a running container that will consume a PersistentVolume.
* StorageClass – allows for dynamic provisioning of PersistentVolumes.

**11. Access Modes & RECLAIM Polices**

PV will have Access Modes

* **ReadWriteOnce** – the volume can be mounted as read-write by a single node
* **ReadOnlyMany** – the volume can be mounted read-only by many nodes
* **ReadWriteMany** – the volume can be mounted as read-write by many nodes

---

ReadWriteOnce:

* The volume can be mounted as read-write by single node. ReadWriteOnce access mode still can allow multiple pods to access the volume when the pods are running on the same node.

ReadOnlyMany:

* The volume can be mounted as read-only by many pods

ReadWriteMany

* The volume can be mounted as read-write by many nodes.

Read WriteOncePod

* The volume can be mounted as read-write by a single pod. Use ReadWriteOncePod access mode, if you want to ensure that only one pod across whole cluster can read that pvc or write to it.

In the CLI, the access modes are abbreviated to:

* RWO - ReadWriteOnce
* ROX - ReadOnlyMany
* RWX - ReadWriteMany

**11. a. CLAIM Policies**

* Retain (default)
* ReCycle
* Delete

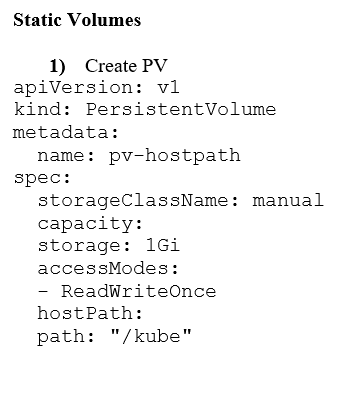
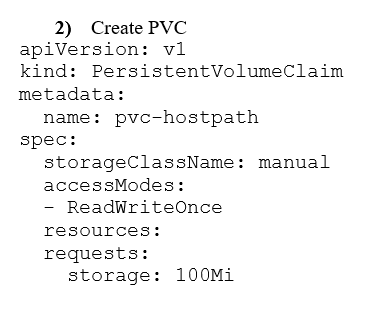
**A Persistent Volume can have several different claim policies associated with it including**

* Retain – When the claim is deleted, the volume remains.
* Recycle – When the claim is deleted the volume remains but in a state where the data can be manually recovered.
* Delete – The persistent volume is deleted when the claim is deleted.

The claim policy (associated at the PV and not the PVC) is responsible for what happens to the data on when the claim has been deleted.

**Commands**

* kubectl get pv
* kubectl get pvc
* kubectl get storageclass
* kubectl describe pvc <pvcName>
* kubectl describe pv <pvName>

* 1. **Scheduling**
* Node selector
* Node Affinity

- required During Scheduling Ignored during Execution (Hard Rule)

- preferred During Scheduling Ignored during Execution (Soft Rule)

- NodeAffinityWeight

* Pod Affinity
* Pod Anti Affinity

**12.a. Node Selector:**

Node selector is simple pod scheduling feature that allows scheduling a pod onto a node whose labels match the nodeSelector labels specified by the user.

* kubectl label nodes ip-172.10.43.76 name=WorkerOne

Syntax:

spec

nodeSelector:

Name: WorkerOne

**12.b. Node Affinity**

Nodes are scheduled using operator/conditions like IN/Not IN

* required During Scheduling Ignored during Execution (Hard Rule)

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: "node"

operator: In

values:

- workerone

* preferred During Scheduling Ignored during Execution (Soft Rule)

spec:

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 1

preference:

matchExpressions:

- key: name

operator: In

values:

- workerone

* NodeAffinityWeight (One or Two Node)

- weight: 1

preference:

matchExpressions:

- key: name

operator: In

values:

- workerone

- weight: 2

preference:

matchExpressions:

- key: name

operator: In

values:

- workerTwo

**12.c.PodAffinity** (InterPod Affinity) and # Pod-AntiAffinity

* PodAffinity and PodAnti Affinity works based on label present in pods, which are already in running in node.
* PodAffinity -- In certain scenarios, we might want to schedule certain pods together
* PodAntiAffinity -- In certain scenarios, we might want to schedule certain pods should never together.

**PodAffinity**

Syntax:

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- nginx

topologyKey: "kubernetes.io/hostname"

**12.d.Pod-AntiAffinity**

Syntax:

spec:

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- nginx

topologyKey: "kubernetes.io/hostname"

**13. Taints & Tolerations (Opposite to Node Selector and Node Affinity)**

**Taints & Tolerations**

Taint is a property of node that allows you to repel a set of pods unless those pods explicitly tolerates the said taint.

Example:

* kubectl taint nodes <node Name> <key>=<value>:<effect>
* kubectl taint nodes <node Name> node=HatesPods:NoSchedule
* kubectl taint nodes <node Name> node=HatesPods:PreferNodeSchedule
* kubectl taint nodes <node Name> node=HatesPods:NoExecute

**Taints have three Effects**

**13.a. No Schedule**

* + Doesn't schedule a pod without matching tolerations

**13.b. Prefer Node Schedule**

* + Prefers that the pod without matching toleration be not scheduled on the node.
  + It is a softer version of No Schedule effect.

**13.c. No Execute**

* + Evicts the pods that dont have matching tolerations.

**Syntax**

spec:

tolerations:

- key: node

operator: Equal

effect: NoSchedule

value: HatePods

* + kubectl taint nodes <node Name/IP> node=Evict:NoExecute
  + kubectl taint nodes <node Name/IP> node=Evict:NoExecute 🡪 To remove taint

Syntax

spec:

tolerations:

- operator: Exists

effect: NoSchedule

**14 .Node Maintence**

1. cordon
2. Drain ( Internally it will cordon & Evict pods from the node)
3. Uncord on

* kubectl describe node | grep "unschedulable" 🡪 To check unschedulable nodes
* kubectl cordon <NodeName/Ip> 🡪to Disable node to schedule
* kubectl drain <NodeName/Ip>
* kubectl uncordon <NodeName/Ip> 🡪 able node to schedule

**15. Resource Quota**

* Using Resource Quota we can limit aggregate (Total) resource consumption per namespace.
* Resource quotas are a tool for administrators to address this concern.
* A resource quota, defined by a Resource Quota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources(CPU, Memory) that may be consumed by resources in that namespace.

**Resources**

1. compute Resources
   * request.cpu
   * request.memory
   * limits.cpu
   * limits.memory
2. Storage Resource
   * storage.request.capacity
3. Quantity
   * no of objects(Pod/Deployment ,pvc, services) can be created in a node

Resource quotas work like this:

* The administrator creates one Resource Quota for each namespace.
* Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a Resource Quota.
* If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code 403 FORBIDDEN with a message explaining the constraint that would have been violated.
* If quota is enabled in a namespace for compute resources like CPU and memory, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation.

Hint: Use the Limit Range admission controller to force defaults for pods that make no compute resource requirements.

* kubectl get quota -n <namespace>
* kubectl get quota -A

Resource Quota

apiVersion: v1

kind: ResourceQuota

metadata:

name: testns-qs-quota

namespace: test-ns

spec:

hard:

requests.cpu: "0.5"

requests.memory: 512Mi

limits.cpu: "1"

limits.memory: 1Gi

pods: 2

Limit Range

apiVersion: v1

kind: LimitRange

metadata:

name: testns-limit-range

namespace: test-ns

spec:

limits:

- default:

cpu: 200m

memory: 512Mi

defaultRequest:

cpu: 100m

memory: 256Mi

type: Container

* kubectl get limitrange -n <namespace>

**16. Ingress Rules and Ingress Controller**

**There are two sorts of isolation for pod**

* Egress -- Incoming request
* Ingress -- outgoing (by default pod is non-isolated for ingress)

**How request goes.**

LoadBalancer:LoadBalancerPort

|

NodeIP: NodePort

|

ServiceIP (ClusterIP):ServicePort

|

POD IP: TargetPort

Note: TO avoid above request, ingressControl is used.

* IngressControl
* IngressRules

All the traffic comes from outside cluster will comes from Loadbalancer, ingressControl forwards the request to pods/service to application, and This IngressControl will use the ingressRules to allow the requests.

**Ingress**

* An API object that manages external access to services in a cluster, typically HTTP.

Client 🡪 Ingress-managed 🡪Ingresscontrol 🡪 Routing Rule 🡪 Service 🡪 Pods

(Load Balancer)

* Kubernetes ingress is a resource to add rules for routing traffic from external sources to the services in the kubernetes cluster.

**Kubernetes Ingress:**

Kubernetes ingress is a native kubernetes resource where you can have rules to route traffic from an external source to service endpoints residing inside the cluster. It requires an ingress controller for routing the rules specified in the ingress object.

**Kubernetes Ingress controller:**

Ingress controller is typically a proxy service deployed in the cluster, it is nothing but a kubernetes deployment exposed to a services.

* + nginx ingress controller
  + HAproxy
  + contour
  + traefik

**17. RBAC (Roll based Access control)**

1) Admin/Developers:

* Users that are responsible to do administrative or developmental tasks on the cluster .This includes operations like upgrading the cluster or creating the resources/workloads (PODs, Deployments/PV, Pvc. etc.) on the cluster.

2) End users:

* Users that access the applications deployed on our kubernetes cluster. Access restrictions for these users are managed by the applications themselves.
* For Example: a web application running on the kubernetes cluster, will have its own security mechanism in place, to prevent unauthorized access.

3) Application/Bots:

* These is possibility that other applications need access to kubernetes cluster or API, typically to talk to resources or workloads inside the cluster. Kubernetes facilitates this by using service accounts.

RBAC in k8's is based on three key concepts:

1. Verbs

* + This is a set of operations that can be executed on resources.
  + create, Read, Update or Delete (CRUD)

2. API Resources

* + The set of API objects available in the cluster are called resources.
  + Example: PODS, Deployments, Services, Nodes, PV etc.

3. Subjects

* -These are the objects allowed access to the API, based on verbs and Resources.
* Users, Groups, Processes (A service Account)

Note: keep in mind that once you'll create a binding you'll not able to edit its roleRef value-insted, you'll have to delete a Binding and recreate and again.

**Role:**

* Roles contains one or more rules that represent a set of permissions.
* Roles are namespaced, meaning roles work with in the constrains of a namespace.it would default to the default namespace if none was specified.
* After creating a Role, assign it to user or group of users by creating a Role Binding.

Example: An example Role in the default namespace that can be used to grant read access to pods.

kind: Role

apiVersion: ra.authorization.k8s.io./v1

metadata:

namespace: default

name: pod-reader

rules:

- apiGroup:[""]

resources: ["pods"] # we can add "services”, “configMaps"

verbs: ["get","watch","list"]

- apiGroup:["apps"] # for deployments

resources: ["deployments","daemonset","replicaset"]

verbs: ["get","watch","list","create","patch","delete"]

rules:

- apiGroup:["\*"]

resources: ["\*"]

verbs: ["\*"]

apiGroups: [""] -- set core API group

resources: ["pods"] -- which resources are allowed for access

["get","watch","list"] -- which actions are allowed over the resources above.

**Role Binding:**

* Role Binding is used for granting permission to subject
* Role Binding Hold a list of subjects (user, group or service accounts), and a reference to the role being granted.
* Role and Roles Binding are used in namespaces scoped.
* RoleBinding may reference any role in the same namespace.
* After you create a binding, you cannot change the role that it refers to. If you do want to change the roleRef for a binding, you need to remove the binding objects and create a again.

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io./v1

metadata:

name: read-pods

namespace: default

subject:

- kind: User

name: Balaji

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

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

**Cluster Role:**

* They are applied to the cluster as a whole.
* ClusterRole are not bound to a specific namespace. ClusterRole give access across more than one namespace or all namespace.
* After creating a clusterRole , you assign it to a user by creating a clusterRoleBinding.
* ClusterRole are cluster-scoped, you can use clusterRole to control access to different kinds of resources than you can with roles.

Cluster-scoped resources (eg: Nodes, pv, pvc, namespaces)

kind: ClusterRole

apiVersion: rabc.authorization.k8s.io./v1

metadata:

name: demo-clusterole

rules:

- apiGroup:[""]

resources: ["pods"] # we can add "services","configMaps"

verbs: ["get","watch","list"]

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

**ClusterRoleBinding:**

* ClusterRole and ClusterRoleBinding function like Role and Role binding, except they have wider scope.
* RoleBinding grants permissions with in a specific namespace, where as a clusterBinding grants access cluster-wide and to multiple namespaces.
* ClusterRole Binding is binding or associating a clusterRole with a subject (users, group or service accounts).

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io./v1

metadata:

name: demo-clusterRolebinding

subject:

- kind: User

name: Balaji

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: demo-clusterRole

apiGroup: rbac.authorization.k8s.io

-------

role vs clusterrole

\* A clusterrole look similar to role with the only difference that we have to set, its kind as clusterrole.

\* The difference is that role is used inside of namespace, while clusterrole is cluster-wide permission without a namespace boundariers for example.

- allow access to a cluster node.

- resources in all namespaces.

- allow access to endpoints like /health

Cluster Management

Display endpoint information about the master and services in the cluster

kubectl cluster-info

Display the Kubernetes version running on the client and server

kubectl version

Get the configuration of the cluster

kubectl config view

List the API resources that are available

kubectl api-resources

List the API versions that are available

kubectl api-versions

List everything

kubectl get all --all-namespaces

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

Daemonsets

Shortcode = ds

List one or more daemonsets

kubectl get daemonset

Edit and update the definition of one or more daemonset

kubectl edit daemonset <daemonset\_name>

Delete a daemonset

kubectl delete daemonset <daemonset\_name>

Create a new daemonset

kubectl create daemonset <daemonset\_name>

Manage the rollout of a daemonset

kubectl rollout daemonset

Display the detailed state of daemonsets within a namespace

kubectl describe ds <daemonset\_name> -n <namespace\_name>

---------

Deployments

Shortcode = deploy

List one or more deployments

kubectl get deployment

Display the detailed state of one or more deployments

kubectl describe deployment <deployment\_name>

Edit and update the definition of one or more deployment on the server

kubectl edit deployment <deployment\_name>

Create one a new deployment

kubectl create deployment <deployment\_name>

Delete deployments

kubectl delete deployment <deployment\_name>

See the rollout status of a deployment

kubectl rollout status deployment <deployment\_name

---

Events

Shortcode = ev

List recent events for all resources in the system

kubectl get events

List Warnings only

kubectl get events --field-selector type=Warning

List events but exclude Pod events

kubectl get events --field-selector involvedObject.kind!=Pod

Pull events for a single node with a specific name

kubectl get events --field-selector involvedObject.kind=Node, involvedObject.name=<node\_name>

Filter out normal events from a list of events

kubectl get events --field-selector type!=Normal

Logs

Print the logs for a pod

kubectl logs <pod\_name>

Print the logs for the last hour for a pod

kubectl logs --since=1h <pod\_name>

Get the most recent 20 lines of logs

kubectl logs --tail=20 <pod\_name>

Get logs from a service and optionally select which container

kubectl logs -f <service\_name> [-c <$container>]

Print the logs for a pod and follow new logs

kubectl logs -f <pod\_name>

Print the logs for a container in a pod

kubectl logs -c <container\_name> <pod\_name>

Output the logs for a pod into a file named ‘pod.log’

kubectl logs <pod\_name> pod.log

View the logs for a previously failed pod

kubectl logs --previous <pod\_name>

For logs we also recommend using a tool developed by Johan Haleby called Kubetail. This is a bash script that will allow you to get logs from multiple pods simultaneously. You can learn more about it at its Github repository. Here are some sample commands using Kubetail.

Get logs for all pods named with pod\_prefix

kubetail <pod\_prefix>

Include the most recent 5 minutes of logs

kubetail <pod\_prefix> -s 5m