**KUBERNETES**

* K8S is a Production-Grade **Container Orchestration Platform**
* K8S is open source software (OSS)
* K8S is used to manage containers of our application
* K8S will take care of **container deployment, scaling, de-scaling and containers load balancing**
* K8S is **not replacement for Docker**
* K8S is **replacement for "Docker Swarm"**
* K8S developed by **Google and donated to CNCF in 2014**
* CNCF means Cloud Native Computing Foundation
* K8S s/w **developed by using GO Lang**
* K8S **v1.0 released** to market in the year of **2015**

**K8S Official Website: https://kubernetes.io/**

**K8S Features**

1) Automated Scheduling

2) Self Healing Capabilities

3) Automated Rollouts and Rollbacks

4) Load Balancing

5) Service Discovery

6) Storage Orchestration

7) Secret and configuration management

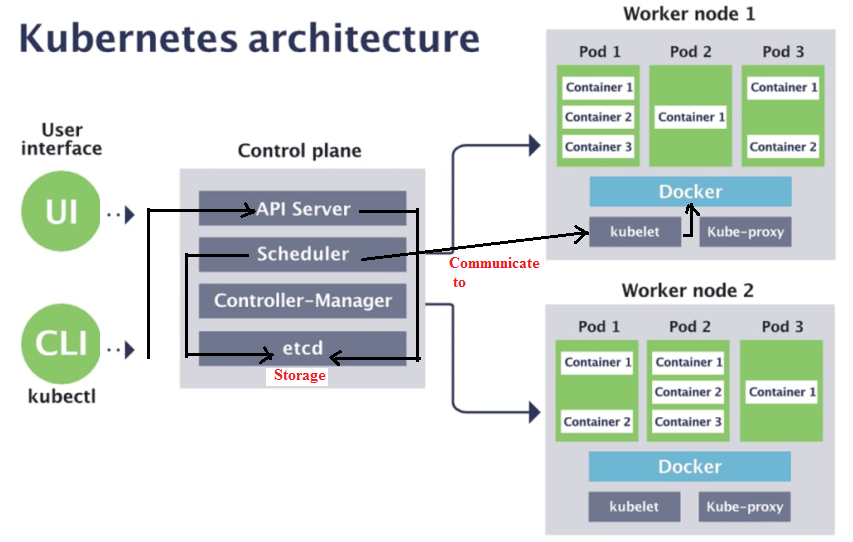
* K8S providing advanced Schedular concept to launch containers depends on our requirement
* Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
* If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.
* Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage

**Note: In Docker Swarm Load Balancing is manual process where as K8S supports Auto Scaling**

* No need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them
* Automatically mount the storage system of your choice, whether from local storage, a public cloud provider such as GCP or AWS, or a network storage system such as NFS, iSCSI, Gluster, Ceph, Cinder, or Flocker.
* Deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

**Kubernetes Architecture**

* K8S works on cluster model
* In K8S cluster we will have master node(s) and worker nodes



To communicate with Kubernetes Cluster we have 2 options

**1) UI (User Interface)**

**2) Kubectl (CLI s/w)**

**Master Node** manages worker nodes in the Cluster. It will **assign tasks to worker nodes** for execution.

Worker Nodes will run the tasks which are assigned by Master Node.

**What is API Server?**

* In K8S cluster we have several services/objects
* PODS
* ReplicationController
* ReplicationSet
* DeamonSet
* Deployment
* Volumes
* Services
* All the above K8S services implemented using GO lang. To use K8S servies we no need to learn GO language. **To use K8S servies K8S provided API server**.
* When we execute a command API server will interact with K8S s/w and it will perform required operation.
* API server will acts as communication channel between Developers / DevOps Engineers and K8S components

**What is ETCD?**

* It is a key-value pair Data Store in K8S
* It acts as database for kubernetes (How many pods, how many nodes, how many containers etc....)
* **When we ask K8S to run our application then API server will recieve that request and it will store into ETCD.**

**What is Schedular**

* It will **schedule PODS for executions** which are un-scheduled based on ETCD
* Schedular will schedule PODS on the nodes with the help of Kubelet
* Kubelet is a worker node component
* **Schedular will talk to kubelet** to to check the resources to our own application

**What is Kubelet?**

* Kubelet will **act as Node Agent**
* Kubelet will ensure that Containers are running healthy in the POD
* Kubelet will **interact with Docker Runtime** to create a container in the POD

**Note: Here we will use Docker Runtime to create our containers**

**What is POD?**

* A POD is the **smallest execution unit** in Kubernetes
* A POD encapsulates one or more applications
* Containers will be grouped as one POD in order to increase the intelligence of resources sharing
* **POD can run single container as well as can multiple container**

**What is Kube-Proxy?**

* Kube-Proxy acts as network proxy
* Kube-Proxy will **maintain network rules** on PODS
* **The network rules allow network communication to your PODS from inside or outside of your cluster**

**What is Controller Manager?**

* Controller Manager **runs controllers** in the background
* It is **responsible to run tasks in K8S cluster**
* **It performs cluster level operations**
* We have several Controllers in K8S
* NodeController
* ReplicationController
* EndpointController
* DeploymentController

**Kubernetes Cluster Setup**

* There are multiple ways to setup kubernetes cluster

**a) Self Managed K8S cluster**

**b) Provider Managed K8S cluster**

* Self Managed Cluster means we have to setup the K8S cluster on our own (Lot of commands to install)
* To create Self Managed Cluster we have 2 options

**1) Mini Kube (Single Node K8S Cluster )**

**2) Kubeadm (Multi Node K8S Cluster )**

* Provider Managed Cluster means we will use K8S cluster which is configured by someone
* EKS : Elastic Kubernetes Service ( AWS )
* AKS : Azure Kubernetes Service ( Microsoft Azure )
* GKE : Google Kubernetes Engine ( Google Cloud Platform )
* IKE : IBM Kubernetes Engine ( IBM Cloud )

**Kubernetes Core Components**

Kubernets Resources / Objects / Workloads

* Container
* POD
* Namespaces
* Service
* Deployment
* ReplicationController
* ReplicationSet
* DaemonSets
* PersistentVolumes
* StatefulSets
* Role
* Secret Config Maps
* We are using Docker to create Containers for our application
* Docker will be used as runtime engine in kubernetes cluster
* **Kubernetes is used to manage our Docker Containers**
* K8S will manage our containers but no directley (**It will use POD to manage containers**)
* **POD is a smallest building block which we can deploy in K8S cluster**
* **Containers will be wrapped under one unit called POD (Logical Grouping)**

**Note: In Docker, container is a smalletst part that we can deploy where as in K8S POD is smallet part we can deploy**

**Note:** To get clarify on PODS, we need to understand Namespaces first in K8S

**What is Namespace?**

Namespace represents a **cluster inside another cluster**

**Kubernetes components will be grouped logically using namespace**

Note: We can consider namespace as a package in java (dao pkg, service pkg, util pkg, controller pkg)

We can have multiple namespaces in k8s cluster

We can get all namespaces using below command

* **$ kubectl get namespaces**

or

* **$ kubectl get ns**

**Note:** When we setup our k8s cluster we will get below 3 namespaces

1) **default:** It will be used by default when we don't specify our namespace

2) **kube-system:** It contains k8s control plan pods

3) **kube-public:** It is reserved for kubernetes system usage

**Note:** It is not recommended to run our pods using default namespaces. We have to **create our own namespace to run our PODS**

Create our own namespace

* **$ kubectl create namespace <namespace-name>**

Ex:

* **$ kubectl create namespace sbi-customer-app**
* **$ kubectl create namespace sbi-agent-app**
* **$ kubectl create namespace sbi-report-app**

We will run our POD using custom namespace

How to get pods belongs to a namespace

* **$ kubectl get pods -n <name-space>**

Get the pods of all namespaces

* **$ kubectl get pods --all-namespaces**

Getting all pods of default namespace

* **$ kubectl get pods**

**Note:** If we delete a namespace, all the objects / resources / components also gets deleted

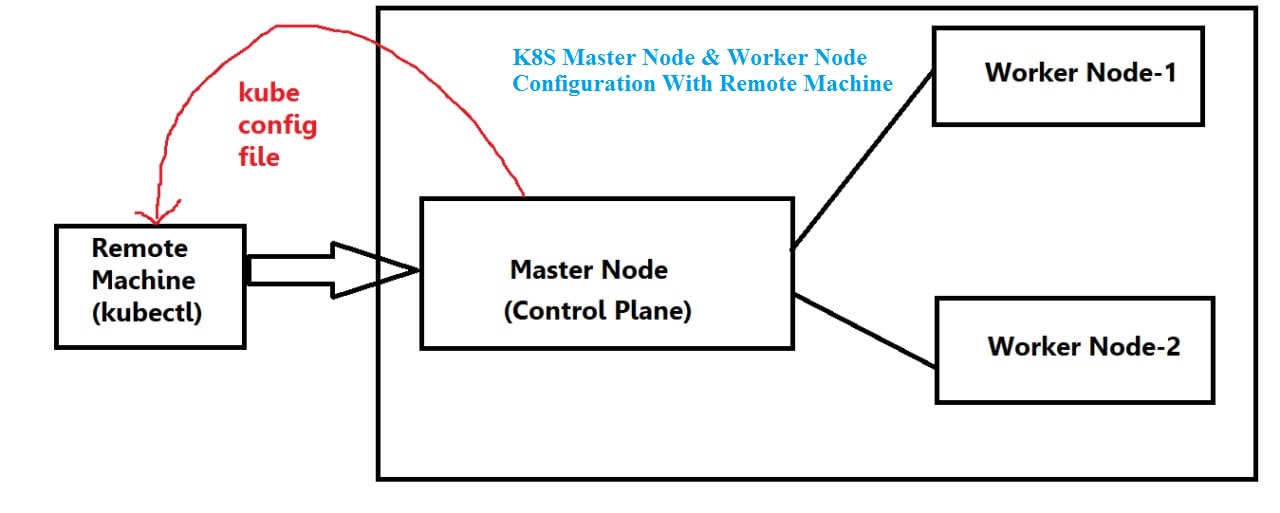
**Kubernetes Cluster Setup**

Create One Security group with Protocol as "All Traffic" Port Range as 0-65535

Create 3 Ubuntu Servers using above created security group

**1 - Master Node (t2.medium instance)**

**2 - Worker Nodes (t2.micro instances )**



**Master & Worker Node Common Commands Execution Start**

Upgrade apt packages

* **$ sudo apt-get update**

Create configuration file for containerd:

* **$ cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf overlay br\_netfilter**

**EOF**

Load modules:

* **$ sudo modprobe overlay**
* **$ sudo modprobe br\_netfilter**

Set system configurations for Kubernetes networking:

* **$ cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf**

**net.bridge.bridge-nf-call-iptables = 1**

**net.ipv4.ip\_forward = 1**

**net.bridge.bridge-nf-call-ip6tables = 1**

**EOF**

Apply new settings:

* **$ sudo sysctl --system**

Install containerd:

* **$ sudo apt-get update && sudo apt-get install -y containerd**

Create default configuration file for containerd:

* **$ sudo mkdir -p /etc/containerd**

Generate default containerd configuration and save to the newly created default file:

* **$ sudo containerd config default | sudo tee /etc/containerd/config.toml**

Restart containerd to ensure new configuration file usage:

* **$ sudo systemctl restart containerd**

Verify that containerd is running.

* **$ sudo systemctl status containerd**

Disable swap:

* **$ sudo swapoff -a**

Disable swap on startup in /etc/fstab:

* **$ sudo sed -i '/ swap / s/^\(.\*\)$/#\1/g' /etc/fstab**

Install dependency packages:

* **$ sudo apt-get update && sudo apt-get install -y apt-transport-https curl**

Download and add GPG key:

* **$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -**

Add Kubernetes to repository list:

* **$ cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list**

**deb https://apt.kubernetes.io/ kubernetes-xenial main**

**EOF**

Update package listings:

* **$ sudo apt-get update**

Install Kubernetes packages (Note: If you get a dpkg lock message, just wait a minute or two before trying the command again):

* **$ sudo apt-get install -y kubelet kubeadm kubectl kubernetes-cni nfs-common**

Turn off automatic updates:

* **$ sudo apt-mark hold kubelet kubeadm kubectl kubernetes-cni nfs-common**

**+++++++++++ Master & Worker Common Commands Execution End ++++++++++++**

**Only Master Node Commands Execution Start**

Initialize the Cluster-

Initialize the Kubernetes cluster on the control plane node using kubeadm

(Note: This is only performed on the Control Plane Node):

* **$ sudo kubeadm init**

Note: if we will get an error as "[ERROR NumCPU]: the number of available CPUs 1 is less than the required 2"

Kubeadm runs a series of pre-flight checks to validate the system state before making changes.

This error means the host don't have minimum requirement of 2 CPU.

You can ignore the error if you still want to go ahead and install kubernetes on this host.

sudo kubeadm init --ignore-preflight-errors=NumCPU

Set kubectl access:

* **mkdir -p $HOME/.kube**
* **sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config**
* **sudo chown $(id -u):$(id -g) $HOME/.kube/config**

Test access to cluster:

* **$ kubectl get nodes**

Install the Calico Network Add-On -

On the Control Plane Node, install Calico Networking:

* **$ kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml**
* **$ kubectl get nodes**

Join the Worker Nodes to the Cluster

In the Control Plane Node, create the token and copy the kubeadm join command (NOTE:The join command can also be found in the output from kubeadm init command):

* **$ kubeadm token create --print-join-command**

Note : In both Worker Nodes, paste the kubeadm join command to join the cluster. Use sudo to run it as root:

**sudo** kubeadm join ...

In the Control Plane Node, view cluster status (Note: You may have to wait a few moments to allow all nodes to become ready):

**What is POD?**

* POD is a smallest build block what we can execute inside K8S cluster
* POD will execute in a node
* One Node can execute multiple PODS
* POD can have one container & more than one container
* POD represents running process
* Containers inside the POD will share a unique network ip, storage and other specifications

**How to run our application in K8S?**

To run our docker image we need to create a pod then k8s will execute that pod in a node

**Note:** If we have pod then we can send request to K8S to schedule that POD execution.

**We can create POD in 2 ways**

1) Interactive

Interactive approach means using commands we can create a pod

Ex: kubectl run --name javawebapppod --image=ashokit/javawebapp

2) Declarative

Declarative approach means using manifest file (YML) we can create a pod

|  |  |
| --- | --- |
| **apiVersion:** | apiVersion represents version of our api like v1, v2, v3.... |
| **kind:** | kind represents what is the purpose of this manifest file |
| **metadata:** | metadata represents data about the (labels) |
| **spec:** | spec represents specification (what you want to use for this manifest) |

* **$ vi javawebapppod.yml**

---

apiVersion: v1

kind: Pod

metadata:

name: javawebapppod

labels:

app: javawebapp

spec:

containers:

- name: javawebappcontainer

image: ashokit/javawebapp

ports:

- containerPort: 8080

...

Get all pods

* **$ kubectl get pods**

Create POD using manifest file

* **$ kubectl apply -f javawebapppod.yml**

Describe the pod using below command

* **$ kubectl describe pod javawebapppod**

Check where the pod is running

* **$ kubectl get pods -o wide**

Note: we can access the POD across the cluster using POD IP.

* **$ curl pod-ip:8080**

**Note: We can't access POD using POD IP outside of the cluster** (this is default behaviour)

**POD Lifecycle**

* Make a request to API server using manifest file (YML) to create a POD
* API server will save the POD info in ETCD
* Schedular find un-scheduled POD info and schedule that POD for execution in NODE
* Kubelet will see that POD Execution schedule and it will trigger DOCKER Runtime
* Docker Runtime will run that container inside the POD.

**Note:** POD is ephemeral (**lives for short period of time**)

* When POD is **re-created then POD IP will change**
* **It is not recommended to access the POD using POD ID**
* We will use **"Kubernetes Service"** component to execute the PODs
* K8S service will make **POD accessible** / discoverable **inside the cluster and outside the cluster also**
* When we create a service we will get one Virtual IP (cluster IP).
* Cluster IP will be registered in K8S DNS with its name.

**What is K8S Service?**

Service is responsible to make our PODS discoverable / accessible inside and outside of the cluster

**Service will identify the POD using** **POD label / selector**

**We have 3 types of services**

1) ClusterIP

2) NodePort

3) Load Balancer

---

apiVersion: v1

kind: Service

metadata:

name: javawebappsvc

spec:

type: ClusterIP

selector:

app: javawebapp

ports:

- port: 80

targetPort: 8080

...

To get all services

* **$ kubectl get svc**

Schedule a service using manifest

* **$ kubectl apply -f javawebappsvc.yml**
* **$ kubectl get svc**

Note: In CluterIP one VIRTUAL IP will be assigned for our service. **Using that ClusterIP** **we can access service with in the cluster.**

**If we want to expose our service outside cluster we need to use NodePort Service**

---

apiVersion: v1

kind: Service

metadata:

name: javawebappsvc

spec:

type: NodePort

selector:

app: javawebapp

ports:

- port: 80

targetPort: 8080

#nodePort: 32611

...

For NodePort service kubernetes will assign random port number if we don't specify nodePort in manifest file

We can access our service outside cluster using **any cluster machine public IP** with node port

**Note:** Enable node port in security group.

URL access to app : http://ec2-vm-ip:nodeport/context-path

(http://13.233.63.130:32645/java-web-app/)

Q) What is the range of Node PORT in k8s cluster?

**Ans) 30000 – 32767**

**In the above scenario we have created the POD manually (it is not recommended)**

**If we create the POD then K8S will not provide high availability**

lets test it by deleting our pod

* **$ kubectl delete pod <pod-name>**

**Note: once pod got delete, k8s not creating another pod and application went down (not accessible)**

If we want to achieve high availability then we should not create pods manually

We need to use K8S components to create PODS then k8s will provide high availability for our application

Note: High Availability means always our application should be accessible

* **ReplicationController**
* **ReplicationSet**
* **DaemonSet**
* **Deployment**
* **StatefulSets**

**What is Replication Controller?**

* It is one of the key features in k8s
* It is responsible to manage POD lifecycle
* It will make sure given no. of POD replicas are running at any point of time.

**Note: if any POD got crashed/deleted/dead then Replication Controller will replace it.**

* Replication Controller is providing facility to create multiple PODS and it will make sure PODS always exists to run our application.
* **Using Replication controller we can achieve High Availability**
* Replication Controller and PODS are associated with **Labels and Selectors.**

---

# pod manifest configuration

apiVersion: v1

kind: ReplicationController

metadata:

name: javawebapprc

spec:

replicas: 1

selector:

app: javawebapp

template:

metadata:

name: javawebapppod

labels:

app: javawebapp

spec:

containers:

- name: javawebappcontainer

image: ashokit/javawebapp

ports:

- containerPort: 8080

---

# node-port service manifest

apiVersion: v1

kind: Service

metadata:

name: javawebappsvc

spec:

type: NodePort

selector:

app: javawebapp

ports:

- port: 80

targetPort: 8080

...

**What is Replica Set?**

* **It is next generation of Replication Controller**
* It is also used to manage POD life cycle
* We can scale up and scale down PODS using Replica Set also
* **The only** **difference** between Replication Controller and Replication Set is **'Selector support'**

We have 2 types of Selectors

**1) Equality Selector**

Ex: selector:

app: javawebappp

**2) Set based Selector**

selector:

matchExpressions:

- key : app

operator : in

values:

- javapp

- javaweb

- javawebapp

---

# pod manifest configuration

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: javawebapprc

spec:

replicas: 1

selector:

matchLabels:

app: javawebapp

template:

metadata:

name: javawebapppod

labels:

app: javawebapp

spec:

containers:

- name: javawebappcontainer

image: ashokit/javawebapp

ports:

- containerPort: 8080

---

# node-port service manifest

apiVersion: v1

kind: Service

metadata:

name: javawebappsvc

spec:

type: NodePort

selector:

app: javawebapp

ports:

- port: 80

targetPort: 8080

...

**What is DaemonSet? (For every worker it create PODS)**

* A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected.
* Deleting a DaemonSet will clean up the Pods it created.
* Some typical uses of a DaemonSet are:

1) Running a **cluster storage** daemon on every node

2) Running a **logs collection** daemon on every node

3) Running a **node monitoring** daemon on every node

**Note: Replicas not applicable for DaemonSet**

---

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: logging

spec:

selector:

matchLabels:

app: httpd-logging

template:

metadata:

labels:

app: httpd-logging

spec:

containers:

- name: webserver

image: httpd

ports:

- containerPort: 80

...

* **Manually POD Created ( Not recommended )**
* **POD creation using ReplicationController**
* **POD creation using ReplicaSet**
* **POD creation using DaemonSet**
* In above concepts **Auto-Scaling feature not available** (Manually we need to scale our pods)
* There is **no option to rollback** our pods creation.
* **To overcome these problems We have "Deployment" concept**

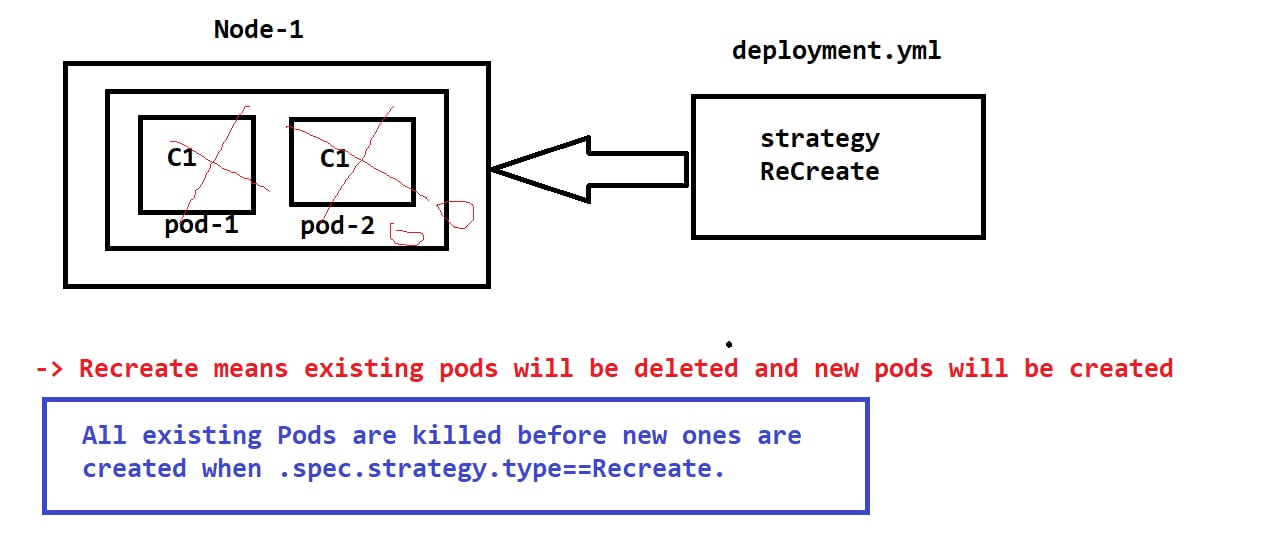
**What is Deployment?**

* Deploymet is used to tell Kubernetes how to create or modify instances of the pods
* **By using Deployment we can rollout and rollback our application deployment (if required)**
* **We can achieve Auto-Scaling by Deployment**
* **Deployment Strategy**

1) ReCreate

2) Rolling Update

3) Blue / Green (Approach)



**K8S deployment manifest file ( POD Manifest + Service Manifest )**

---

**# POD Deployment Manifest**

apiVersion: apps/v1

kind: **Deployment**

metadata:

name: javawebappdeployment

labels:

app: javawebapp

spec:

replicas: 1

strategy:

type: **Recreate**

selector:

matchLabels:

app: javawebapp

template:

metadata:

name: javawebapppod

labels:

app: javawebapp

spec:

containers:

- name: javawebappcontainer

image: ashokit/javawebapp

ports:

- containerPort: 8080

---

---

**# Service Manifest**

apiVersion: v1

kind: Service

metadata:

name: javawebappsvc

spec:

type: NodePort

selector:

app: javawebapp

ports:

- port: 80

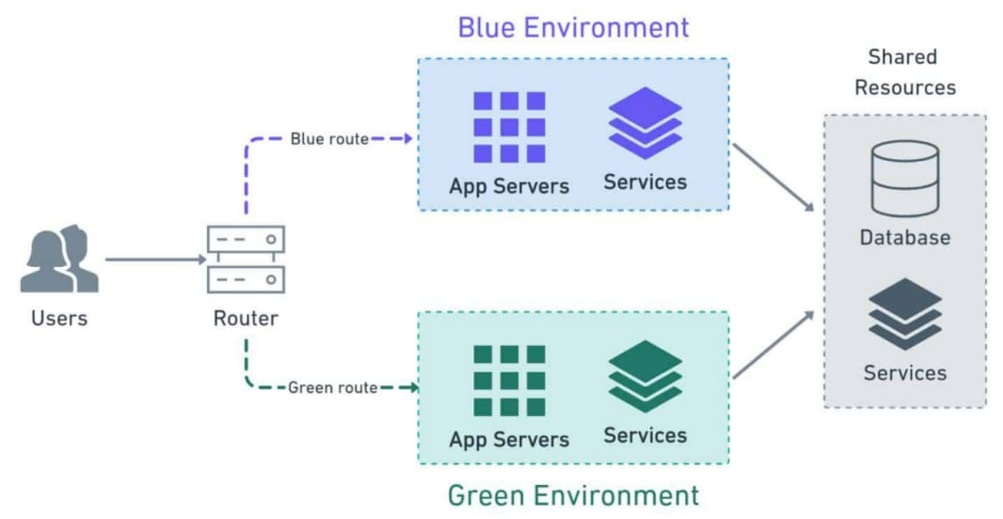
targetPort: 8080

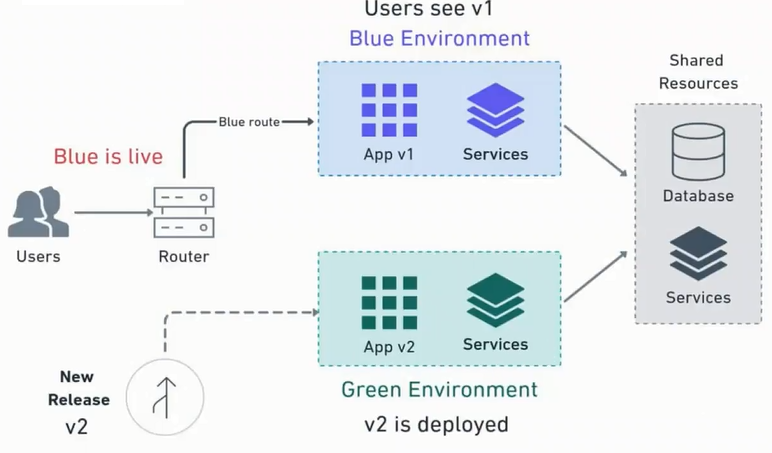
...

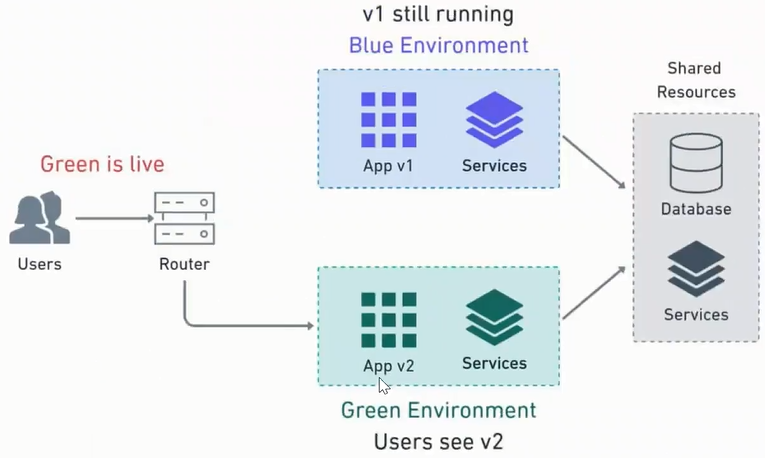
* **$ kubectl get pods**
* **$ kubectl get svc**
* **$ kubectl delete all --all**
* **$ kubectl apply -f deployment.yml**
* **$ kubectl get pods**
* **$ kubectl get svc**
* **$ kubectl get deployment**
* **$ kubectl delete deployment <deployment-name>**

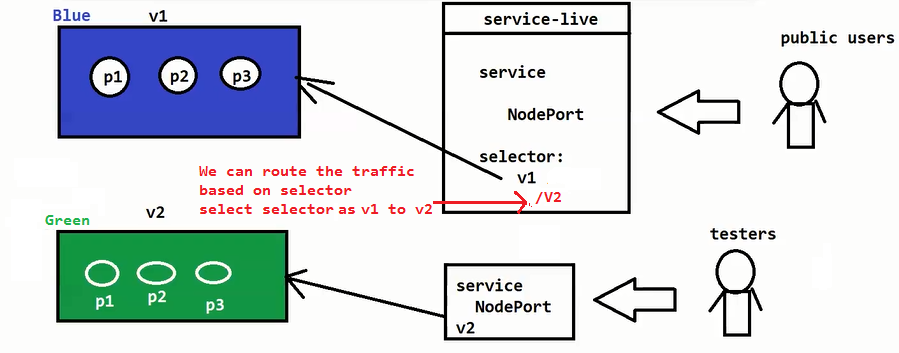
**Blue / Green Deployment (Approach)**

* It is an application release model.
* It reduces risk and minimizes downtime.
* It uses two production environment known as Blue and Green.
* Rapid releasing, Simple rollback, Zero downtime

****



****



**Autoscaling**

It is the process of increasing / decreasing infrastructure based on demand

Autoscaling can be done in 2 ways

1) Horizontal Scaling

2) Vertical Scaling

Horizontal Scaling means **increasing number of instances/systems**

Vertical Scaling means **increasing capacity** of single system

Note: For production we will use Horizontal Scaling

**HPA: Horizontal POD Autoscaling**

**VPA: Vertical POD Autoscaling (we don't use this)**

HPA: Horizontal POD Autoscaler which will **scale up/down number of pod** replicas of deployment, ReplicaSet or Replication Controller dynamically **based on the observed Metrics** (CPU or Memory Utilization).

HPA will interact with Metric Server to identify CPU/Memory utilization of POD.

To get node metrics

* **$ kubectl top nodes**

To get pod metrics

* **$ kubectl top pods**

Note: By default metrics service is not available

Metrics server is an application that collects metrics from objects such as pods, nodes according to the state of CPU, RAM and keeps them in time.

Metric-Server can be installed in the system as an addon. You can take and install it directley from the repo.

1) Clone git repo

* **$ git clone https://github.com/ashokitschool/k8s\_metrics\_server**

2) Check the cloned repo

* **$ cd k8s\_metrics\_server**
* **$ ls deploy/1.8+/**

3) Apply manifest files from manifest-server directlry

* **$ kubectl apply -f deploy/1.8+/**

Note: it will create service account, role, role binding all the stuff

We can see metric server running in kube-system ns

* **$ kubectl get all -n kube-system**

Check the top nodes using metric server

* **$ kubectl top nodes**

Check the top pods using metric server

* **$ kubectl top pods**

Note: When we install Metric Server, it is installed under the kubernetes system namespaces.

* **$ kubectl delete all -all**

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: hpadeployment

labels:

name: hpadeployment

spec:

replicas: 2

selector:

matchLabels:

name: hpapod

template:

metadata:

labels:

name: hpapod

spec:

containers:

- name: hpacontainer

image: k8s.gcr.io/hpa-example

ports:

- name: http

containerPort: 80

resources:

requests:

cpu: "100m"

memory: "64Mi"

limits:

cpu: "100m"

memory: "256Mi"

---

apiVersion: v1

kind: Service

metadata:

name: hpaclusterservice

labels:

name: hpaservice

spec:

ports:

- port: 80

targetPort: 80

selector:

name: hpapod

type: ClusterIP

---

apiVersion: autoscaling/v2beta1

kind: HorizontalPodAutoscaler

metadata:

name: hpadeploymentautoscaler

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: hpadeployment

minReplicas: 2

maxReplicas: 5

metrics:

- resource:

name: cpu

targetAverageUtilization: 50

type: Resource

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

Resources & requests

In cluster if none of the pods have this min resources availabile it will not schedue

Min resource and Memory we are configuring to schedule pods using HPA

Note: take hpademo.yml

* **$ kubectl get pods**
* **$ kubectl apply -f hpa.yml**

Note: as of now there is no load on application

Now we need to simulate the load

We can simulate load using busybox

* **$ kubectl run -it --rm loadgenerator --image=busybox**

Note: witht this command we are inside the pod

* **$ wget -q -O- http://hpaclusterservice**

Note: we got response

* **$ while true; do wget -q -O- http://hpaclusterservice; done**

Note: connect to control-pane and check pods

* **$ kubectl top pods**
* **$ kubectl get hpa**

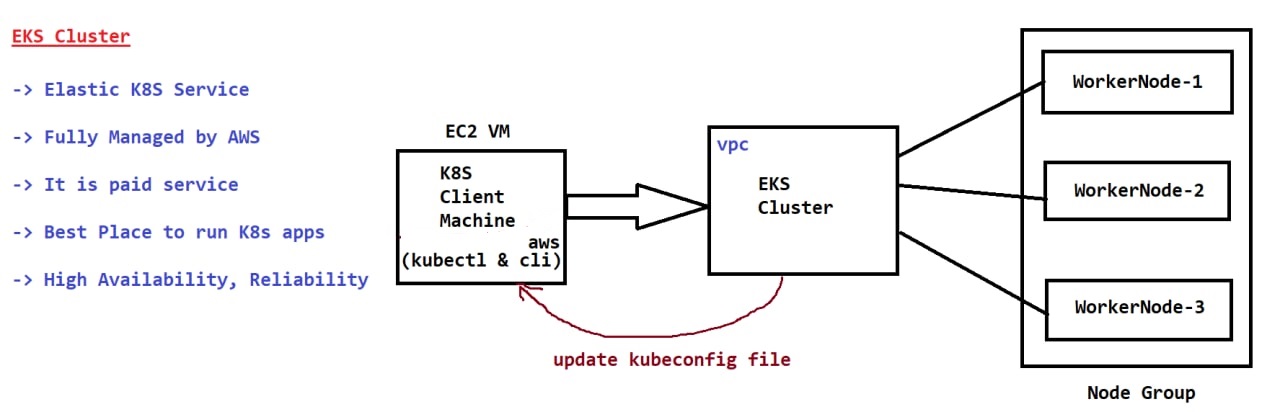
**AWS-EKS (Elastic Kubernetes Service)**

* EKS stands for “Elastic Kubernetes Service"
* EKS is a fully managed K8S service
* EKS is the best place to run K8S applications because of its security, reliability and scalability
* **EKS can be integrated with other AWS services such as ELB, CloudWatch, AutoScaling, IAM and VPC**
* EKS makes it easy to run K8S on AWS without needing to install, operate and maintain your own k8s control plane.
* Amazon EKS runs the K8S control Plane across three availability zones in order to ensure high availability and it automatically detects and replaces unhealthy masters.
* AWS will have complete control over Control Plane. We don't have control on Control Plane.
* We need to create Worker Nodes and attach to Control Plane.

Note: We will create Worker Nodes Group using ASG Group

* Control Plane Charges + Worker Node Charges (Based on Instance Type & No.of Instances)

**Note: $0.10 per hour**



**Pre-Requisites**

AWS account with admin priviliges

Instance to manage/access EKS cluster using Kubectl

AWS CLI access to use kubectl utility

**Steps to Create EKS Cluster in AWS**

**Step-1) Create VPC using Cloud Formation (with below S3 URL)**

**URL:** https://s3.us-west-2.amazonaws.com/amazon-eks/cloudformation/2020-10-29/amazon-eks-vpc-private-subnets.yaml

Stack name: EKSVPCCloudFormation

**Step-2) Create IAM role in AWS**

Entity Type: AWS Service

Select Usecase as 'EKS' ==> EKS Cluster

Role Name: EKSClusterRole (you can give any name for the role)

**Step-3) Create EKS Cluster using Created VPC and IAM Role**

Cluster endpoint access: Public & Private

**Step-4) Create RedHat ec2 Instance (K8S\_Client\_Machine)**

Connect to K8S\_Client\_Machine using Mobaxterm

**#################### Install Kubectl with below commands ######################**

* $ curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
* $ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
* $ kubectl version --client

**######### Install AWS ClI in K8S\_Client\_Machine with below commands ###########**

* $ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"
* $ sudo yum install unzip
* $ unzip awscliv2.zip
* $ sudo ./aws/install

**###################### Configure AWS Cli with Credentials ####################**

Access Key ID: AKIA4MGQ5UW7R76

Secret Access Key: ZoZZW+063Km49zi19FbPC3Ijo15auV

$ aws configure

Note: We can use root user accesskey and secret key access

**##########################################################################**

$ aws eks list-clusters

$ ls ~/.

### Update kubeconfig file in remote machine from cluster using below command ########

$ aws eks update-kubeconfig --name <cluster-name> --region ap-south-1

**###########################################################################**

**Step-5) Create IAM role for EKS worker nodes (usecase as EC2) with below policies**

a) AmazonEKSWorkerNodePolicy

b) AmazonEKS\_CNI\_Policy

c) AmazonEC2ContainerRegistryReadOnly

**Step-6) Create Worker Node Group**

* Go to cluster -> Compute -> Node Group
* Select the Role we have created for WorkerNodes
* Use t2.large
* Min 2 and Max 5

**Step-7) Once Node Group added then check nodes in K8s\_client\_machine**

$ kubectl get nodes

$ kubectl get pods --all-namespaces

**Step-8) Create POD and Expose the POD using NodePort service**

Note: Enable NODE PORT in security Group to access that in our browser

**Kubernetes Ingress**

* Deploy two application Into K8S using Service using Cluster IP
* $ kubectl apply -f javawebapp.yml
* $ kubectl apply -f mavenwebapp.yml
* Now we have 2 services running in K8S cluster with **Cluster IP service**. **We can't access them outside the cluster.**
* **We will use Ingress to provide routing for these two services from** **external traffic**
* K8S ingress is a resource to add rules for routing traffic from external sources to the services in the k8s cluster
* K8S ingress is a native k8s 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
* Ingress controller is typically a proxy service deployed in the cluster. It is nothing but a Kubernetes deployment exposed to a service.

**Ingress Setup**

# git clone k8s-ingress

$ git clone https://github.com/ashokitschool/kubernetes\_ingress.git

$ cd kubernetes-ingress

# Create namespace and service-account

$ kubectl apply -f common/ns-and-sa.yaml

# create RBAC and configMap

$ kubectl apply -f common/

# Deploy Ingress controller

-> We have 2 options to deploy ingress controller

1) Deployment

2) DaemonSet

$ kubectl apply -f daemon-set/nginx-ingress.yaml

# Get ingress pods using namespace

$ kubectl get all -n nginx-ingress

# create LBR service

$ kubectl apply -f service/loadbalancer-aws-elb.yaml

Note: It will generate LBR DNS

-> Map LBR dns to route 53 domain

-> Create Ingress kind with rules

============================

Path Based Routing

$ vi ingress-rules2-routes.yml

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: ingress-resource-2

spec:

ingressClassName: nginx

rules:

- host: ashokit.org

http:

paths:

- pathType: Prefix

path: "/java-web-app"

backend:

service:

name: javawebappsvc

port:

number: 80

- pathType: Prefix

path: "/maven-web-app"

backend:

service:

name: mavenwebappsvc

port:

number: 80

...

**K8S HELM**

* We deployed our apps in Kubernetes cluster using Manifest files
* Manifest files we can write in 2 ways

1) JSON

2) YML (more demand)

* It is difficult to write manifest files for our applications
* **Helm is a package manager for k8s applications**
* **Helm allows you to install or deploy applications on kubernetes cluster in a similar manner to yum/apt for linux distributions.**
* Helm lets you fetch, deploy and manage the lifecycle of applications both 3rd party apps and your own applications

Ex: promethus, graphana, nginx-ingress are third party apps

* Helm introduces several familiar concepts such as Helm Chart (**package contains k8s manifests - templates**)
* **Helm Repositories which holds helm charts/packages**
* A CLI with install/upgrade/remove commands

**Why to use Helm?**

Deploying application on K8S cluster is little difficult

As part of app deployment we need to create below k8s objects

Deployment

Service

ConfigMaps/Secrets

Volumes

Ingress Rules

HPA

Helm greatly simplifies the process of creating, deploying and managing applications on k8s cluster

Helm also maintains a versioned history of very chart (application) installation. If something goes wrong, you can simply call **'helm rollback'.**

Setting up a single application can involve creating multiple independent k8s resources and each resource requires a manifest file.

**Purpose of Helm**

* Create new charts from scratch
* Package charts into chart archive (tgz) files
* Interact with chart repositories where charts are stored
* Install and uninstall charts into an existing Kubernetes cluster
* Manage the release cycle of charts that have been installed with Helm

**What is Helm Chart?**

HELM chart is a basically just a **collection of manifest files** organized in a specific directory structure that describe a related K8S resource.

There are two main components in HELM chart

1) Template

2) Value

Templates and values renders a manifest which can understand by k8s

**Helm uses charts to pack all the required k8s components (manifests) for an application to deploy, run and scale.**

Charts are very similar to RPM and DEB packages for Linux.

Ex: yum install git

Note: it will interact with repo and it will download git

**HELM Concepts**

Helm packages are called charts, and they consist of a few YML configuration files and some templates that are rendered into K8S manifest files. Here is the basic directory structure of a chart.

charts : dependent charts will be added here

templates: contains all template files

values : It contains values which are required for templates

**HELM Architecture**

what-the-helm

├── Chart.yaml

├── charts

├── templates

│ ├── NOTES.txt

│ ├── \_helpers.tpl

│ ├── deployment.yaml

│ ├── ingress.yaml

│ ├── service.yaml

│ └── tests

│ └── test-connection.yaml

└── values.yaml

**Helm Installation**

$ curl -fsSl -o get\_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3

$ chmod 700 get\_helm.sh

$ ./get\_helm.sh

$ helm

check do we have metrics server on the cluster

$ kubectl top pods

$ kubectl top nodes

# check helm repos

$ helm repo ls

# Before you can install the chart you will need to add the metrics-server repo to

$ helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/

# Install the chart

$ helm upgrade --install metrics-server metrics-server/metrics-server

$ helm list

$ helm delete <release-name>