==========
Kubernetes Running Notes
==========

1) What is Kubernetes

- Orchestration Platform
- To manage containers
- Developed by Google using Go language
- Google donated K8S to CNCF
- K8S first version released in 2015
- It is free & Open source

Note: Kubernetes will use Docker internally. Using K8S we will manage our Docker containers.

2) Docker Swarm Vs K8S

- Docker Swarm doesn't have Auto Scaling (Scaling is manual process)
- K8S supports Auto Scaling
- For Production deployments K8S is highly recommended
- Kubernetes is replacement for Docker Swarm

Auto Scaling : Increasing and Decreasnig containers count based on incoming requests for our app.

3) What is Cluster

- Group Of Servers
- Master Node(s)
- Worker Node(s)
- DevOps Enginner / Developer will give the task to K8S Master Node
- Master Node will manage worker nodes
- Master Node will schedule tasks to worker nodes
- Our containers will be created in Worker Nodes
- Using Cluster we can achieve High Availability

4) Kubernetes Architecture

- Control Plane / Master Node / Manager Node
    - Api Server
    - Schedular
    - Control Manager
    - ETCD

- Worker Node (s)
    - Pods
    - Containers
    - Kubelet
    - Kube Proxy
    - Docker Runtime

5) How to communicate with K8S control plane ?

- Kubectl (CLI tool)

- Web UI Dashboard

==========
Kubernetes Architecture Components
==========

Control Plane : Control Plane is called as Master Node (Responsible to handle k8s related work)

Worker Nodes: Responsible to execute our application as PODS.

=> API Server : It is responsible to handle incoming requests of Control Plane (to deploy our applications)

=> Etcd : It is an internal database in K8S cluster, API Server will store requests / tasks info in ETCD

=> Schedular : It is responsible to schedule pending tasks available in ETCD. It will decide in which worker node our task should execute. Schedular will decide that by communicating with Kubelet.

=> Kubelet : It is a worker node agent. It will maintain all the information related to Worker Node.

=> Conroller-Manager : After scheduling completed, Controller-Manager will manage our task execution in worker node

=> Kube-Proxy : It will provide network for K8S cluster communication
(Master Node <---> Worker Nodes)

=> Docker Engine : To run our containers Docker Engine is required. Containers will be created in Worker Nodes.

=> Container : It is run time instance of our application

=> POD : It is a smallest building block that we will create in k8s to run our containers.

Note: Docker Containers will run inside POD.

==========
Kubernetes Cluster Setup
==========

1) Self Managed Cluster (We will create our own cluster)

    a) Mini Kube (Single Node Cluster)

    b) Kubeadm (Multi Node Cluster)

2) Provider Managed Cluster (Cloud Provide will give read made cluster) ---> Charges applies

    a) AWS EKS

    b) Azure AKS

    c) GCP GKE

==========
Minikube Cluster setup on windows OS
==========

1) Download and install Docker Desktop software in Windows
    (URL : https://docs.docker.com/desktop/install/windows-install/)

2) Download and install Minikube (URL : https://minikube.sigs.k8s.io/docs/start/)

3) Download and install Kubectl (URL : https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/)

==========
AWS EKS Cluster Setup
==========

Git Hub Repo : https://github.com/ashokitschool/DevOps-Documents/blob/main/EKS-Setup.md

==========
Kubernetes Components
==========

1) POD
2) Services
3) Namespaces
4) Replication Controller
5) Replication Set
6) DaemonSet
7) Deployment
8) StatefulSet
9) ConfigMap & Secrets
10) IngressController
11) K8S Volumes
12) HELM Charts
13) Grafana & Promethues
14) ELK Setup (Log Aggregation)
15) RBAC
16) K8S Web Dashboard

==========
What is POD ?
==========

=> POD is a smallest building block in k8s cluster

=> In K8S, every container will be created inside POD only

=> POD always runs inside Node

=> POD represents running process

=> POD means group of containers running on a Node

=> We can create multiple PODS on single node

=> Every POD will have unique IP address

=> We can create PODS in 2 ways

1) Interactive Approach

$ kubectl run --name <pod-name> image=<image-name> --generate=pod/v1

2) Declarative Approach (K8S Manifest YML)

==========
K8S Manifest YML
==========

```
---
apiVersion:
kind:
metadata:
spec:
...
```

==========
Kubernetes POD Manifest YML
==========

```
---
apiVersion: v1
kind: Pod
metadata:
  name: javawebapppod
  labels:
    app: javawebapp
spec:
   containers:
   - name: javawebappcontainer
     image: ashokit/javawebapp
     ports:
      - containerPort: 8080
...
```

```
# Display all pods which are created
$ kubectl get pods

# Create PODS using pod-manifest
$ kubectl apply -f <pod-yml>
```

# Descrie pod (get more info about pod)
$ kubectl describe pod <pod-name>

# Get pod logs
$ kubectl logs <pod-name>

# Get POD running on which worker node
$ kubectl get pods -o wide

################ Note: PODS we can't access outside. ##############

=> We need to expose PODS for outside access using Kubernetes Service concept.

============
K8S Service
============

=> Kubernetes service is used to expose PODS outside cluster

=> We have 3 types of K8S Services

1) Cluster IP
2) Node Port
3) Load Balancer

=> We need to write service manifest yml to expose PODS

```
---
apiVersion: v1
kind: Service
metadata:
 name: javawebappsvc
spec:
  type: NodePort
  selector:
    app: javawebapp # POD lable
  ports:
  - port: 80
    targetPort: 8080
...
```

Display existing services

$ kubectl get svc

# Create k8s service
$ kubectl apply -f <svc-maniest-yml>

# Display existing services
$ kubectl get svc

Note: We can see service information and Node Port Number assigned by K8S.

Note: Enable Node Port number in security group of Worker node in which our POD is running.

#Access application in browser

URL : http://node-public-ip:node-port-num/java-web-app/

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

==========
Working Procedure
==========

1) Write POD Manifest YML

2) Create POD using kubectl apply command

3) Check POD details and POD logs & In which worker node it is running

4) Write Service Manifest YML

5) Create Service to expose PODS

6) Check Service Details and Node PORT number assigned by K8S

7) Enable Node PORT number in Worker Node Security Group

8) Access application using Worker Node IP

==========

Cluster IP
==========

-> When we create PODS, every pod will get unique IP address

-> We can access POD inside cluster using its IP address

Note: PODS are short lived objects, When POD is recreated its ip will be changed so we can't depend on POD IP to access.

-> To expose POD access with in the cluster we can use ClusterIP service.

-> ClusterIP will generate one IP address to access our PODS with in cluster.

Note: ClusterIP will not change when PODS are re-created.

```
---
apiVersion: v1
kind: Service
metadata:
 name: javawebappsvc
spec:
  type: ClusterIP
  selector:
    app: javawebapp # POD lable
  ports:
  - port: 80
    targetPort: 8080
...
```

==========
Node Port
==========

=> NodePort service is used to expose pods outside cluster also

=> When we use NodePort service we can specify Node Port number in service manifest file

=> If we don't specify Node Port number then k8s will assign random node port number for our service.

Node Port Range : 30000 - 32767

```
========================================
Combined Manifest file for POD & Service
========================================

---
apiVersion: v1
kind: Pod
metadata:
  name: javawebapppod
  labels:
    app: javawebapp #very imp
spec:
  containers:
  - name: javaweappcontainer
    image: ashokit/javawebapp
    ports:
      - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
  type: NodePort
  selector:
    app: javawebapp  #POD label
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30785
...


# Delete all k8s components we have created
$ kubectl delete all --all

# Apply manifest
$ kubectl apply -f <manifest-yml>

# Get Pods
$ kubectl get pods
```

# Get Service
$ kubectl get svc

# Check POD running in which Node
$ kubectl get pods -o wide

Access Application in browser (make sure node port number enabled in Security Group)

URL : http://node-public-ip:node-port-num/java-web-app/

==========

==========
Load Balancer
==========

-> It is one type of K8s Service

-> It is used to expose our PODS outside cluster using Load Balancer

-> When we use Load Balancer as service type then one Load Balancer will be created in AWS Cloud.

-> Using Load Balancer URL we can access our application

-> Load Balancer will distribute the traffic to multiple worker nodes in Round Robbin fashion.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: javawebapppod
  labels:
    app: javawebapp #very imp
spec:
  containers:
  - name: javaweappcontainer
    image: ashokit/javawebapp
    ports:
      - containerPort: 8080
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: javawebappsvc
spec:
  type: LoadBalancer
  selector:
    app: javawebapp  #POD label
  ports:
    - port: 80
      targetPort: 8080
...
```

```
$ kubectl delete all --all
```

```
$ kubectl apply -f <manifest-yml>
```

```
$ kubectl get pods
```

```
$ kubectl get svc
```

Note: In service, we can load balancer URL as External IP. Using that URL we can access our application.

URL : load-balancer-url/java-web-app

Note: After practise is completed delete kubernetes pods& services.

```
$ kubectl delete all --all
```

==========
POD Life Cycle
==========

-> POD is a smallest building block that we can run in k8s cluster

-> We are using kubectl to send request to control plane to create POD

-> API Server will recieve our request and will store request details in ETCD.

-> Schedular will find un-scheduled PODS and it will schedule in worker nodes

-> Node Agent (kubelet) will see POD schedule and it will fire Docker Engine

-> Docker will engine will run our container

Note: PODS are ephemeral (Short Lived Objects)

==========
K8S Namespaces
==========

-> Namespaces are equal to packages in the Java.

-> Namespaces are used to group our k8s components logically.

app pods ----> ashokit-app-ns

db pods ----> ashokit-db-ns

-> We can create multiple namespaces in k8s cluster

ex: ashokit-app-ns, ashokit-db-ns etc.....

-> Namespaces are logically isolated with each other

Note: When we delete namespace all the components created under that namespace will be deleted.

-> We can get k8s namespaces using below command

$ kubectl get ns (or) $ kubectl get namespace

-> In K8s we will have below namespaces by default

1) default
2) kube-public
3) kube-system
4) kube-node-lease

Note: When we create k8s component without using namespace then k8s will consider 'default' namespace for that.

Note: kube-public, kube-system and kube-node-lease namespaces will be used by k8s cluster. We should not create our k8s components in these 3 namespaces.

# Command to get k8s components
$ kubectl get all

# Get k8s components of given namespace
$ kubectl get all -n <namespace>

Ex : kubectl get all -n kube-system

It is highly recommended to create our k8s components under custom namespace

# create namespace
$ kubectl create ns ashokitns

We can create namespace using manifest yml also

```
---
apiVersion: v1
kind: Namespace
metadata:
        name: ashokitdbns
...
```

$ kubectl apply -f <manifest-yml>

==========
Creating POD & Service under Custom Namespace
==========

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: ashokitns
---
apiVersion: v1
kind: Pod
metadata:
 name: javawebapppod
 labels:
   app: javawebapp #very imp
```

```
  namespace: ashokitns
spec:
  containers:
  - name: javaweappcontainer
    image: ashokit/javawebapp
    ports:
      - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
  namespace: ashokitns
spec:
  type: LoadBalancer
  selector:
    app: javawebapp  #POD label
  ports:
    - port: 80
      targetPort: 8080
...
```

```
$ kubectl apply -f <maniefest-yml>

$ kubectl get all -n ashokitns

$ kubectl get pods -n ashokitns

$ kubectl get svc -n ashokitns

$ kubectl logs <pod-name> -n ashokitns

$ kubectl delete ns ashokitns
```

==========

1) PODS
2) Services (ClusterIP, NodePort and LoadBalancer)
3) Namespaces

==========

-> As of now we have created PODS manually using POD manifest yml

-> If we delete our POD, K8S not re-creating the POD hence application will be down.

# Delete pod
$ kubectl delete pod <pod-name>

# check pods
$ kubectl get pods

-> POD is not recreated because we have created POD manually

############ It is not all recommended to create PODS manually ################

=> Always we need to create the PODS using below K8S components/resources

1) ReplicationController
2) ReplicaSet
3) Deployment
4) StatefulSet
5) DaemonSet

==========
ReplicationController (RC)
==========

=> It is a k8s component which is used to create PODS

=> It will make sure always given no.of pods are running for our application.

Note: It will take care of POD lifecycle

Note: When POD is crashed / damaged / deleted then RC will create new pod.

=> We can scale up and scale down PODS count using Replication Controller

```
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: javawebapprc
spec:
  replicas: 3
  selector:
```

```
    app: javawebapp
  template:
    metadata:
      name: javawebapppod
      labels:
        app: javawebapp
    spec:
      containers:
      - name: javawebappcontainer
        image: ashokit/javawebapp
        ports:
          - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
  type: NodePort
  selector:
    app: javawebapp
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30785
...

$ kubectl delete all --all

$ kubectl apply -f rc.yml

$ kubectl get pods -o wide

$ kubectl get svc

$ kubectl get rc

$ kubectl scale rc javawebapprc --replicas 5

$ kubectl scale rc javawebapprc --replicas 3

$ kubectl delete rc javawebapprc
```

==========
ReplicaSet
==========

-> ReplicaSet is replacement for ReplicationController (It is nextgen component in k8s)

-> ReplicaSet is also used to manage POD Lifecycle

-> ReplicaSet also will maintain given no.of pods always

-> We can scale up and we can scale down our PODS using ReplicaSet also

The only difference between ReplicationController and ReplicaSet is 'selector'

=> ReplicationController supports Equality Based Selector

ex:

```
        selector:
            app: javawebapp
```

=> ReplicaSet supports Set Based Selector

ex:

```
        selector:
            matchLabels:
                app: javawebapp
                version: v1
                type: backend
```

```
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: javawebapprs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: javawebapp
  template:
    metadata:
```

```
    name: javawebapppod
    labels:
      app: javawebapp
  spec:
    containers:
    - name: javawebappcontainer
      image: ashokit/javawebapp
      ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
  type: NodePort
  selector:
    app: javawebapp
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30785
...



$ kubectl delete all --all

$ kubectl apply -f rs.yml

$ kubectl get pods -o wide

$ kubectl get svc

$ kubectl get rs

$ kubectl scale rs javawebapprs --replicas 5

$ kubectl scale rs javawebapprs --replicas 3

$ kubectl delete rs javawebapprs


===========
```

Deployment
==========

=> Deployment is one of the K8S resource / component

=> Deployment is the most recommended approach to deploy our applications in k8s cluster.

=> Using Deployment we can scale up and we can scaledown our pods

=> Deployment supports Roll Out and Roll Back.

Note: To deploy latest code we have to delete RC & RS then PODS will be deleted and application will be down.

=> We can deploy latest code with Zero Downtime using "Deployment"

1) Zero Downtime Deployment

2) Rollout and Rollback

3) AutoScaling

==========
Deployment Strategies
==========

=> We have below Deployment stratagies

1) Re Create
2) Rolling Update

=> ReCreate means it will delete all the existing pods and it will create new pods
(downtime will be there)

=> RollingUpdate strategy means it will delete the pod creates new pod one by one.

If we don't specify Deployment Strategy, then by default it will consider as RollingUpdate

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: javawebappdeployment
spec:
  replicas: 2
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: javawebapp
  template:
    metadata:
      name: javawebapppod
      labels:
        app: javawebapp
    spec:
      containers:
      - name: javawebappcontainer
        image: ashokit/javawebapp
        ports:
          - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: javaweappsvc
spec:
  type: NodePort
  selector:
    app: javawebapp
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30785
...

$ kubectl delete all --all

$ kubectl apply -f java-app-deployment.yml

$ kubectl get pods -o wide

$ kubectl get svc
```

$ kubectl get deployment

$ kubectl scale deployment javawebappdeployment --replicas 5

$ kubectl scale deployment javawebappdeployment --replicas 3

$ kubectl delete deployment javawebappdeployment

==========
Blue - Green Deployment
==========

-> It is one of the approach to deploy application

1) less risk
2) zero downtime
3) easy rollbacks
4) semless user experience

=> We need to create below manifest ymls for blue-green deployment

1) blue-deployment.yml

2) live-service.yml

3) green-deployment.yml

4) pre-prod-service.yml

Step-1) Create blue-deployment

Step-2) Expose Blue PODS using Live-Service

Step-3) Access application using Live Service (Blue PODS will run)

Step-4) Clone git repo (https://github.com/ashokitschool/java_web_docker_app)

Step-5) Modify Code + Build Project using Maven + Create Docker Image + Push Image to Docker Hub

Step-6) Create Green-Deployment with latest image

Step-7) Expose Green PODS using Pre-Prod-Service

Step-8) Access application using pre-prod Service (green PODS will run)

Note: Notedown Live Service URL and Pre-Prod-Service URL

Step-9) Access both URLS and see the difference

Live Service URL : http://3.109.202.11:30785/java-web-app/

Pre-Prod-Service URL : http://3.109.202.11:31785/java-web-app/

Step-10) Modify the selector in live-service from v1 to v2 and apply that using kubectl

Step-11) Access live service url (latest code should be accessible)

==========
Config Map & Secrets
==========

=> For every application multiple environments will be available for testing purpose

a) DEV
b) SIT
c) UAT
d) PILOT

=> Once appilcation testing is completed in all above environments then it will be deployed into Production environment (Live Environment).

Note: For every envionment, application properties will be different

a) Database Properties
b) Kafka properties

c) SMTP properties
d) Redis properties etc...

We shouldn't hardcode application properties

=> Config Map & Secret concepts are used to avoid hard coded properties in the application

=> Config Map is used to store data in key-value (non-confidential)

=> Config Map allows us to de-couple application properties from Docker images so that our application can be deployed into any environment without making any changes for our Docker image.

=> Secret is also one of the k8s resource

=> Secret is used to store confidential data in key - value format (ex: pwd)

=> Secret is used to store confidential data in encoded format.

URL To encode : https://www.base64encode.org/

Note: ConfigMap & Secrets will make docker images as portable.

==========
ConfigMap Manifest
==========

=> Below is the configmap manifest yml

```
---
apiVersion: v1
kind: ConfigMap
metadata:
 name: weshopify-db-config-map
 labels:
   storage: weshopify-db-storage
data:
 DB_DRIVER_NAME_VALUE: com.mysql.cj.jdbc.Driver
 DB_SERVICE_NAME_VALUE: weshopify-app-db-service
 DB_SCHEMA_VALUE: weshopify-app
 DB_PORT_VALUE: "3306"
...
```

```
$ kubectl get configmap

$ kubectl apply -f <configmap-yml>

$ kubectl get configmap
```

==========
Secret Manifest
==========

=> Below is the secret manifest yml

```
---
apiVersion: v1
kind: Secret
metadata:
  name: weshopify-db-config-secret
  labels:
        secret: weshopify-db-config-secret
data:
  DB_USER_NAME_VALUE: cm9vdA==
  DB_PASSWORD_VALUE: cm9vdA==
type: Opaque
...

$ kubectl get secret

$ kubectl apply -f <secret-yml>

$ kubectl get secret
```

==========
Reading Data From ConfigMap
==========

```
- name : DB_DRIVER_CLASS
                valueFrom:
                        configMapKeyRef:
                                name: weshopify-db-config-map
                                key: DB_DRIVER_NAME_VALUE
```

==========

Reading Data From Secret
==========

```
- name : DB_PASSWORD
                valueFrom:
                        secretKeyRef:
                                name: weshopify-db-config-secret
                                key: DB_PASSWORD_VALUE
```

==========
Hard Coded Properties
==========

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://mysqldb:3306/sbms
    username: root
    password: root
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

==========
Application Properties with Env Variables
==========

```
spring:
  datasource:
    driver-class-name: ${DB_DRIVER:com.mysql.cj.jdbc.Driver}
    url: ${DB_URL:jdbc:mysql://mysqldb:3306/sbms}
    username: ${DB_USERNAME:root}
    password: ${DB_PASSWORD:root}
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

==========
Spring Boot with MySQL DB Deployment using ConfigMap + Secret
==========

Git Hub Repo : https://github.com/ashokitschool/kubernetes_manifest_yml_files.git

1) Create Config Map

$ kubectl apply -f <yml>

2) Create Secret

$ kubectl apply -f <yml>

3) Create PV

$ kubectl apply -f <yml>

4) Create PVC

$ kubectl apply -f <yml>

5) Create DB Deployment

$ kubectl apply -f <yml>

$ kubectl get pods

$ kubectl get svc

6) Create Application Deployment

$ kubectl apply -f <yml>

$ kubectl get pods

$ kubectl get svc

7) Check logs of application

$ kubectl logs <app-pod-name>

$ kubectl get pods -o wide

8) Enable Application Service Node Port in Security Group of Node in which pod is running

9) Access application using URL

URL : http://node-public-ip:node-port/

10) Insert records in the application

12) Check in MySQL DB records are inserted or not

```
$ kubectl exec -it db-pod-name bash
$ mysql -h localhost -u root -p
$ show databases;
$ use <db-name>
$ show tables;
$ select * from book;
```

13) Exit from DB client & from DB container

```
$ exit
$ exit
```

==========
Stateless application vs Stateful Application
==========

=> Stateless applications can't store the data permanentley. For every request they will get new data and it will process that data.

Every request will be treated as new request.

Ex: Node JS app, Nginx app, Boot app etc....

=> Stateful applications means they will store the data permanentley and they will keep track of data.

Ex: MySQL, Oracle, Postgres etc....

=> To run stateful containers in k8s cluster we will use StatefulSet concept.

=> StatefulSet will assign a sticky identity for each pod starting from zero (0)

=> In Stateful set primary and secondary pods will be created.

=> Once primary pod is created then by copying primary pod data secondary pod will be created

Note: New POD will be created by copying previous pod data. If previous pod is pending state then new pod will not be created.

=> In StatefulSet, pods will be deleted in reverse order (last to first)

==========
What is the difference between Deployment and StatefulSet ?
==========

-> Deployment will create the PODS with random ids
-> Deployment will scale down pods in random order
-> Deployment pods are stateless pods

-> StatefulSet will create the PODS with sticky identity
-> StatefulSet will scale down pods in reverse order
-> StatefulSet pods are statefull

Note:

-> For application deployment we will use 'DEPLOYMENT'
-> For database deployment we will use 'STATEFULSET'

Stateful Set Manifest YML Updated in Repo :
https://github.com/ashokitschool/kubernetes_statefulset_ymls.git ####

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

==========
DaemonSet
==========

-> It is k8s resource to create PODS

-> DaemonSet will create one pod in each worker node

-> If new node is created new DaemonSet is created auto

-> When we add node to cluster then POD will be created, if we remove node from cluster then POD will be deleted..

When to use DaemonSet
---

1) Logs Collector

2) Node Monitoing

==========
HELM Charts
==========

-> HELM is a package manager for K8S Cluster

-> HELM is used to install required softwares in k8s cluster

-> HELM will maintain Charts in chart repository

-> Chart means collection of configuration files organized in a directory

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

# Install the chart
$ helm upgrade --install metrics-server metrics-server/metrics-server

$ kubectl top pods

$ kubectl top nodes

$ helm list

$ helm delete <release-name>
```

==========
Metric Server Unavailability issue fix
==========
URL : https://www.linuxsysadmins.com/service-unavailable-kubernetes-metrics/

```
$ kubectl edit deployments.apps -n kube-system metrics-server
```

=> Edit the below file and add new properties which are given below

```
---------- Existing File----------
 spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443

---------- New File----------
---
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-insecure-tls=true
        - --kubelet-preferred-address-types=InternalIP
```

########################
Kubernetes Monitoring
########################

=> We can monitor our k8s cluster and cluster components using below softwares

1) Prometheus
2) Grafana

==========
Prometheus
==========

-> Prometheus is an open-source systems monitoring and alerting toolkit

-> Prometheus collects and stores its metrics as time series data

-> It provides out-of-the-box monitoring capabilities for the k8s container orchestration platform.

==========
Grafana
==========

-> Grafana is a analysis and monitoring tool

-> Grafana is a multi-platform open source analytics and interactive visualization web application.

-> It provides charts, graphs, and alerts for the web when connected to supported data sources.

-> Grafana allows you to query, visualize, alert on and understand your metrics no matter where they are stored. Create, explore and share dashboards.

Note: Graphana will connect with Prometheus for data source.

##
How to deploy Grafana & Prometheus in K8S
##

-> Using HELM charts we can easily deploy Prometheus and Grafana

##
Install Prometheus & Grafana In K8S Cluster using HELM
##

# Add the latest helm repository in Kubernetes
$ helm repo add stable https://charts.helm.sh/stable

# Add prometheus repo to helm
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

# Update Helm Repo
$ helm repo update

# install prometheus
$ helm install stable prometheus-community/kube-prometheus-stack

# Get all pods
$ kubectl get pods

Node: You should see prometheus pods running

# Check the services
$ kubectl get svc

By default prometheus and grafana services are available within the cluster as ClusterIP, to access them outside lets change it to NodePort / LoadBalancer.

# Edit Prometheus Service & change service type to LoadBalancer then save and close that file
$ kubectl edit svc stable-kube-prometheus-sta-prometheus

# Now edit the grafana service & change service type to LoadBalancer then save and close that file
$ kubectl edit svc stable-grafana

# Verify the service if changed to LoadBalancer
$ kubectl get svc

=> Access Promethues server using LoadBalancer URL

URL : http://LBR-URL:9090/

=> Access Grafana server using LoadBalancer URL

URL : http://LBR-URL/

=> Use below credentials to login into grafana server

UserName: admin
Password: prom-operator

=> Once we login into Grafana then we can monitor our k8s cluster. Grafana will provide all the data in graphs format.

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

==========
EFK Setup
==========

# Clone Git Repo
$ git clone https://github.com/ashokitschool/springboot_thyemleaf_app.git

# Create K8S Deployment
$ kubectl apply -f Deployment.yml

# Get PODS
$ kubectl get pods -n ashokitns

# Get service
$ kubectl get svc -n ashokitns

Access Application using Load Balancer URL (insert few records)

# Clone Git Repo URL for EFK manifest files
$ https://github.com/ashokitschool/kubernetes_manifest_yml_files.git

Note: navigate to 04-EFK-Log directory

# Create EFK Deployment
$ kubectl apply -f .

# Get PODS
$ kubectl get pods -n efklog

# Get service
$ kubectl get svc -n efklog

Note: Enable 5601 in Kibana LBR security

=> Access Kibana URL in browser

=> Create Index Patterns by select * & then @timestamp

=> Access logs from kibana dashboard.

Note: After practise completed delete ashokitns and efklog namespace.

==========
Autoscaling
==========

-> It is the process of increasing / decreasing infrastructure or resources based on demand

-> Autoscaling can be done in 2 ways

1) Horizontal Scaling

2) Veriticle Scaling

-> Horizontal Scaling means increasing number of instances/servers

-> Veriticle Scaling means increasing capacity of single system

HPA : Horizontal POD Autoscaling

VPA : Vertical POD Auoscaling (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 collect 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.

==========
Step-1 : Install Metrics API
==========

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.

==========
Step-2 : Deploy Sample Application
==========

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: hpa-demo-deployment
spec:
 selector:
   matchLabels:
     run: hpa-demo-deployment
 replicas: 1
 template:
   metadata:
     labels:
       run: hpa-demo-deployment
   spec:
     containers:
     - name: hpa-demo-deployment
       image: k8s.gcr.io/hpa-example
       ports:
       - containerPort: 80
       resources:
         limits:
           cpu: 500m
         requests:
           cpu: 200m
...

$ kubectl apply -f <deployment.yml>
```

```
$ kubectl get deploy


==================================
Step-3 : Create Service
==================================

---
apiVersion: v1
kind: Service
metadata:
 name: hpa-demo-deployment
 labels:
   run: hpa-demo-deployment
spec:
 ports:
 - port: 80
 selector:
   run: hpa-demo-deployment
...

$ kubectl apply -f service.yaml

$ kubectl get svc


==================================
Step-4 : Create HPA
==================================

---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: hpa-demo-deployment
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: hpa-demo-deployment
 minReplicas: 1
 maxReplicas: 10
 targetCPUUtilizationPercentage: 50
```

...

$ kubectl apply -f hpa.yaml
$ kubectl get hpa
$ kubectl get deploy

==========
Step-5 : Increase the Load
==========

$ kubectl get hpa

$ kubectl run -i --tty load-generator --rm --image=busybox --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://hpa-demo-deployment; done"

Note: After executing load generator open Duplicate tab to monitor hpa events

$ kubectl get hpa -w

$ kubectl describe deploy hpa-demo-deployment

$ kubectl get hpa

$ kubectl get events

$ kubectl top pods

$ kubectl get hpa

##########
Kubernetes web dashboard
##########

=> We can communicate with Kubernetes cluster in 2 ways

1) Kubectl (CLI)

2) Web dashboard (UI Based)

=> Using kubectl we can execute commands to deploy our components in k8s cluster

=> Web Dashboard will provide user interface to perform our operations in k8s cluster

==========
K8S Web Dashboard Setup
==========

$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.5.0/aio/deploy/recommended.yaml

$ kubectl -n kubernetes-dashboard get pods -o wide

$ kubectl -n kubernetes-dashboard get svc

# Edit k8s dashboard service and change it to NodePort
$ kubectl -n kubernetes-dashboard edit svc kubernetes-dashboard

Note: Check kubernetes-dashboard node port and enable that Node PORT in security group

# Check in which node kubernete-dashboard POD is running
$ kubectl get pods -o wide -n kubernetes-dashboard

Access k8s web ui dashboard using below URL

URL : https://node-public-ip:node-port/

#create admin user with below yml

$ vi create-user.yml
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
...

$ kubectl apply -f create-user.yml

```
# create cluster role binding
$ vi cluster-role-binding.yml

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
...

$ kubectl apply -f cluster-role-binding.yml


# Get the bearer token
$ kubectl create token admin-user -n kubernetes-dashboard
```

Note: Copy the token and enter in kubernetes web dashboard login.

==========

Kubernetes Ingress

Kubernetes ingress is a collection of routing rules that govern how external users access services running in a Kubernetes cluster.

-> 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 Summary
==========

1) What is Orchestration ?

2) Orchestration Tools ?

3) What is Kubernetes ?

4) Kubernetes Architecture

5) Kubernetes Cluster Setup (AWS EKS)

6) What is Kubectl ?

7) What is POD ?

8) What is K8S Service ?

- ClusterIP
- NodePort
- LoadBalancer

9) What is Headless Service ?

10) What is NodePort range in K8S?

11) What is Namespace ?

12) What are the Resources available to create PODS in k8s ?

- ReplicationController
- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet

13) Blue - Green Deployment Model ?

14) What are labels and selectors in k8s ?

15) ConfigMaps and Secrets

16) Spring Boot App Deployment with MySQL DB using ConfigMap & Secret

17) HELM Charts

18) K8S Monitoring (Prometheus & Grafana)

19) Logging and Log Monitoring

20) Log Aggreggation using EFK

21) What is Metric Server ?

22) K8S HPA (Horizontal POD Autoscaling)

23) Ingress Controller

24) RBAC (Role Based Access Control)

==========

$ kubectl get nodes

$ kubectl get pods

$ kubectl get pods -o wide

$ kubectl describe pod <pod-name>

$ kubectl logs <pod-name>

$ kubectl exec -it <pod-name> /bin/bash

$ kubectl get pods -n <namespace>

$ kubectl delete all --all

$ kubectl apply -f <yml-file-name>

$ kubectl get svc

$ kubectl get svc -n <namespace>

$ kubectl edit svc <svc-name>

$ kubectl delete svc <svc-name>

$ kubectl get ns

$ kubectl create ns <namespace-name>

$ kubectl delete ns <namespace-name>

$ kubectl scale deployment <deployment-name> --replicas 5

$ kubectl get all

$ kubectl top nodes
$ kubectl top pods

$ kubectl get configmap

$ kubectl get secret

$ kubectl get rc

$ kubectl get rs

$ kubectl get deploy

$ kubectl get sts

$ kubectl get hpa

$ kubectl get hpa -w

$ kubectl get events

$ kubectl get pv

$ kubectl get pvc

==========