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

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

3) What is Cluster

- Group Of Servers
- Master Node
- 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

4) Kubernetes Architecture

- Control Plane / Master 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
==========

=> API Server : It is responsible to handle incoming requests of Control Plane

=> Etcd : It is 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 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.

==========
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

==========
Kubernetes Components
==========
1) Pods
2) Services
3) Namespaces
4) ReplicationConroller
5) ReplicaSet
6) DaemonSet
7) Deployments
8) StatefulSet
9) K8S Volumes
10) ConfigMap & Secrets
11) Ingress Controller
12) K8S Web Dashboard
13) RBAC (Role Based Access in K8S)
14) HELM Charts (Package Manager)
15) Grafana & Promethues (Monitoring Tools)
16) ELK Stack (Log Monitoring)
17) EKS (Provider Managed Cluster - Paid Service)

==========
PODS
==========

-> POD is a smallest building block in k8s cluster

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

-> POD always runs on a Node

-> POD represents a running process

-> POD is a group of one or more containers running on a Node

-> Each POD will have unique IP with in the cluster

-> We can create K8S pods in 2 ways

1) Interactive Mode (By using kubectl command directley)

Ex: $ kubectl run --name <pod-name> image=<image-name> --generator=run-pod/v1

2) Declarative Mode (K8S Manifest YML file)

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

-> Once K8S manifest yml is ready then we can execute that using below kubectl command

$ kubectl apply -f <file-name>

```
================================
Kubernates Sample POD Manifest YML
================================
---
apiVersion: v1
kind: Pod
metadata:
   name: javawebapppod
   labels :
      app: javawebapp
spec:
   containers:
    - name: javawebappcontainer
      image: ashokit/javawebapp
      ports:
         - containerPort: 8080
...
```

```
$ kubectl get pods

$ kubectl apply -f <pod-yml>

$ kubectl get pods

$ kubectl describe pod <pod-name>
```

***** Note: By default PODS are accessible only with in the Cluster, Outside of the Cluster We can't access PODS*******

=> To provide PODS access outside of the cluster we will use 'Kubernetes Service' concept

```
=============
K8S Service
==============
```

-> K8S service makes PODs accessible outside of the cluster also

-> In K8S we have 3 types of Services

1) Cluster IP
2) Node Port
3) Load Balancer (Will work only with Provider Managed Cluster - we will learn this in EKS)

-> We need to Create k8s service manifest to expose PODS outside the cluster

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

```
$ kubectl get svc

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

$ kubectl get svc
```

Note: NodePort service will map our pod to a random port Number (Ex: 30002)

-> Enable Node Port in Security Group Inbound Rules

```
$ kubectl describe pod <pod-name>
```

Note: Here we can see in which Node our POD is running

-> We can access our application using below URL

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

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

-> It will expose our k8s service on a cluster with one internal ip

-> Cluster IP type service is accessible only with in cluster using Cluster IP

-> When we access cluster ip, it will redirect the request to POD IP

Note: POD is very short lived object, when pod is re-created POD ip will change hence it is not at all recommended to access pods using pod ips. To expose PODS with in cluster we can use 'Cluster IP' service

Note: ClusterIP service is accessible only with in cluster (can't accessed outside the cluster)

-> To expose POD using service, we will use POD label as a Selector in Service Manifest file like below

```
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
   type: ClusterIP
   selector:
        app: javawebapp # this is pod label
   Ports:
        - port: 80
          targetPort: 8080
...

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

$ kubectl get svc


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

-> Node Port Service is used to expose our PODS outside the cluster also

-> When we use NodePort Service we can specify PORT Number, if we don't specify port number then k8s
will assign one random port number for our service.

Q) What is the range of NodePort service PORT Number in k8S ?

Ans)   30000 - 32767


---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
   type: NodePort
   selector:
        app: javawebapp
   Ports:
        - port: 80
          targetPort: 8080
          nodePort: 30002
...

$ kubectl get svc

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

$ kubectl get svc

$ kubectl delete service <service-name>


Note: Once we expose our POD using NodePort service then we can access our pod outside the cluster
also.


# Get POD IP and POD Running Node IP
$ kubectl get pod -o wide


#URL To access our application
```

http://pod-running-node-public-ip:nodeport/<context-path>/

```
=====================================================
Comibining Pod manifest and Service Manifest using single YML
=====================================================
---
apiVersion: v1
kind: Pod
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: 30002
...

$ kubectl apply -f <manifest-yml>



============
POD Lifecycle
============

-> When we make a request to create a POD then API Server will recieve our request

-> API Server will store our POD creation request in ETCD

-> Schedular will find un-scheduled pods and it will schedule them in Worker Nodes

-> The Node Agent (Kubelet) will see POD Schedule and it will fire Docker Engine

-> Docker Engine runs the Container

-> The entire POD lifecycle is store in ETCD.


Note :  POD is ephemeral ( very short lived object )


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

-> Namespace represents a cluster inside the cluster
```

-> Namespaces are used to group k8s components

-> We can create multiple namespaces in single k8s cluster

-> Namespaces are logically isolated with each other

Note: In java we have packages concept to group the classes in k8s we have namespaces to group k8s components

-> We can get k8s namespaces using below command

```
$ kubectl get ns  ( or )  $ kubectl get namespaces
```

-> K8S cluster providing below namespaces by default

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

Note: If we create any k8s component without giving namespace then k8s will consider 'default' namespace for that.

-> The remaining 3 namespaces will be used by k8s for cluster management.

-> It is not recommended to create our k8s components under kube-node-release, kube-public and kube-system.

Command to get all k8s components

```
$ kubectl get all
```

Command to get all k8s components of particular namespace

```
$ kubectl get all -n <namespace>
```

Ex : $ kubectl get all -n kube-system

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

Command to create custom namespace

```
$ kubectl create ns ashokitns
```

-> We can create namespace using declarative approach also (manifest yml)

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
...
```

```
$ kubectl apply -f <namespace-name>
```

Note: If we delete namespace all the components of that namespace also gets deleted.

Command to delete namespace

```
$ kubectl delete ns ashokitns


=> When we execute below command the components of 'default' namespace got deleted.

                $ kubectl delete all --all


==================================
POD & Service Under Custom Namespace
==================================

---
apiVersion: v1
kind: Pod
metadata:
   name: javawebapppod
   labels :
      app: javawebapp
   namespace: ashokitns
spec:
   containers:
    - name: javawebappcontainer
      image: ashokit/javawebapp
      ports:
         - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
  namespace: ashokitns
spec:
  type: NodePort
  selector:
    app: javawebapp
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30002
...


$ kubectl apply -f <manifest-yml>

$ kubectl get all -n ashokitns

$ kubectl get pods -n ashokitns

$ kubectl describe pod <pod-name> -n ashokitns


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

-> As of now we have created K8S POD manually using POD manifest file.

-> If we delete the POD then our application will be down, k8s not re-creating the POD

# command to delete the pod
$ kubectl delete pod <pod-name>

# check the pods
$ kubectl get pods

-> POD is not re-created by k8s because we have created POD manually.
```

-> It is not all recommended to create pods manully.

-> K8S provided below components/resources to create the PODS

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

-> If we create the PODS using above resources then K8S will take care of pods & Pod lifecycle.

==========
Replication Controller
==========

-> It is one of the key feature of K8S

-> It is responsible to manage POD life cycle

-> It is responsible to make sure given no.of pods are running for our application at any point of time

Note: If any pod is crashed then it will replace that pod with new pod.

-> Using Replication Controller we can scale up and scale down our PODS

-> Create below manifest file to work with '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: 30002
```

```
...


$ kubectl apply -f <rc.yml>

$ kubectl get rc

$ kubectl get pods

$ kubectl get svc

$ kubectl delete pod <pod-name>

$ kubectl scale rc <rc-name> --replicas <count>

Ex: $ kubectl scale rc javawebapprc --replicas 3

$ kubectl get pods -o wide

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

-> ReplicSet is the replacement for ReplicationController (It  is next gen component)

-> ReplicaSet also manages Pod Lifecycle

-> ReplicaSet also mantains given no.of pod replicas at any point of time

-> We can scale up and we can scale down our POD replicas using ReplicasSet also

-> The only difference between ReplicationController and ReplicaSet is in 'selector'

-> ReplicationController supports 'Equlity Based Selector'

-> ReplicaSet supports 'Set Based Selector'

---------------------------------
Equality Based Selector
---------------------------------
selector:
   app: javawebapp


--------------------------
Set Based Selector
--------------------------
selector:
   matchLabels:
           app: javawebapp
           version: v1
           type: backend

-> Create below manifest to work with ReplicaSet

---
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: 30002
  ...


  $ kubectl apply -f <rs.yml>

  $ kubectl get rs

  $ kubectl get pods -o wide

  $ kubectl delete pod <pod-name>

  $ kubectl scale rs <rs-name> --replicas <count>

  Ex: $ kubectl scale rs javawebapprs --replicas 3

  $ kubectl get pods -o wide



  ================
  K8S Deployments
  ================

  -> Deployment is the most recommended approach to deploy our application in k8s cluster

  -> Deployment is used to tell how to create pods on the k8s cluster

  -> Using Deployment we can scale up and we can scale down our POD Replicas

  -> Deployment supports Roll Out and Roll Back

  -> Below are the key benefits of k8s deployment

                  1) Deploy a RS

                  2) Update PODS

                  3) Rollback to older deployment

                  4) Scale up & scale down
```

Note: When we use ReplicationController or ReplicaSet latest images cant be updated directley. We have to delete RC or RS to deploy lastest code (when we delete RC or RS all pods gets deleted then application will be down)

-> When we use Deployment concept we can easily update latest code without deleting Deployment. We can achieve zero downtime.

-> K8S deployment having below deployment strategies

1) ReCreate
2) RollingUpdate

-> ReCreate strategy 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 and it will re-create the pod one by one.

Note: If we don't specify deployment strategy in manifest yml, then k8s will consider 'RollingUpdate' as default deployment strategy.

-> We can use below 'Deployment' manifest yml to create deployment of our java web application in K8S cluster

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: javawebappdeployment
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
---
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
  type: NodePort
  selector:
    app: javawebapp
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30002
...
```

```
$ kubectl get deployment

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

$ kubectl get deployment

$ kubectl get pods

$ kubectl get svc

$ kubectl get pods -o wide

$ kubectl scale deployment javawebappdeployment --replicas 3


======================
Blue & Green Deployment
======================

=> It is one of the application release model with zero downtime


-> Create 'blue deployment' using below manifest yml (Deployment manifest)

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-boot-demo-deployment-blue
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: k8s-boot-demo
      version: v1
      color: blue
  template:
    metadata:
      labels:
        app: k8s-boot-demo
        version: v1
        color: blue
    spec:
      containers:
        - name: k8s-boot-demo
          image: ashokit/javawebapp
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
...


=> Create a service for blue pods exposing  (service-live.yml)

---
apiVersion: v1
kind: Service
metadata:
  name: k8s-boot-demo-service
spec:
  type: NodePort
  selector:
    app: k8s-boot-demo
```

```
    version: v1
  ports:
    - name: app-port-mapping
      protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 30002
...

=> After creating the service access our application using below URL

                http: // node-ip : 30002 / java-web-app/



==============================
Deploying Latest Code as Green
==============================

=>  Create green pods using below deployment manifest

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-boot-demo-deployment-green
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: k8s-boot-demo
      version: v2
      color: green
  template:
    metadata:
      labels:
        app: k8s-boot-demo
        version: v2
        color: green
    spec:
      containers:
        - name: k8s-boot-demo
          image: ashokit/mavenwebapp
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
...

=> To test green pods we are creating Pre-Prod Service

---
apiVersion: v1
kind: Service
metadata:
  name: k8s-boot-demo-service-preprod
spec:
  type: NodePort
  selector:
    app: k8s-boot-demo
    version: v2
  ports:
    - name: app-port-mapping
```

```
      protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 30092
...
```

=> Access the application using pre-prod service

http://node-ip:30092/maven-web-app/

Note: Once pre-prod testing completed then v2 pods we need to make live

==========
How to make Green as Live ?
==========

-> Go to service-live.yml and change selector to 'v2' and apply

-> After applying live service with v2 then our live service will point to green pods (latest code)

URL : http://node-ip:30002/maven-web-app/

What is Orchestration
What is K8S
K8S Architecture
K8S Cluster Setup (Kubeadm)
Pods
Pod Creation
Pod Manifest YML
Services (ClusterIP, NodePort, LoadBalancer)
Selectors & Labels
Namespaces
Pod Lifecycle
ReplicationController
ReplicaSet
Deployment (Recreate , RollingUpdate)
Blue - Green Deployment

==========
DeamonSet
==========

-> It is also one of the k8s resource used to create PODS in k8s cluster

-> DeamonSet will create copy of the pod on each worker node

Some typical uses of a DaemonSet are:

running a cluster storage daemon on every node
running a logs collection daemon on every node
running a node monitoring daemon on every node

=> Create fluentd-elasticsearch pod using daemonset

```
$  kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml

=>  fluentd-elasticsearch will be created on 'kube-system' namespace

=> Get all the k8s components belongs to kube-system

$ kubectl get all -n kube-system

$ kubectl get pods -o wide -n kube-system



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

-> We shouldn't hardcode  properties in the application, because from environment to environment
application properties might change.

                        Ex: DEV DB and PROD DB properties will be different

-> ConfigMap & Secrets are used to avoid hard coding properties in the application

                Ex: database properties, smtp properties

-> ConfigMap is used to store the data in the form of key-value  (non-confidential)

-> A ConfigMap allows you to decouple environment-specific configuration from your container images,
so that your applications are easily portable.

-> If we use ConfigMap concept for application environment properties then we can deploy our
application in any environment without re-creating images.

-> Secrets is also one of the kubernetes resource

-> Secrets is used to store confendential data in key-value format

                Ex: username, password, token etc...

-> Using Secrets we will store confidentials data in encrypted format



====================
Working with ConfigMap
====================

-> Below is the example for configmap

-> create a manifest file like below

---
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_HOST_SERVICE_NAME_VALUE: weshopify-app-db-service
 DB_SCHEMA_VALUE: weshopify-app
 DB_PORT_VALUE: "3306"
...
```

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


-> To get/refer data from config-map, in our pod manifest we will use below tag

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


==================
Working with Secrets
==================

-> Below is the example for Secrets

-> Create secrets yml file with below content

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


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


=> We can get data from Secrets in our POD manifest using below tag


 - name: DB_PASSWORD
              valueFrom :
                secretKeyRef :
                  name : weshopify-db-config-secrete
                  key :  DB_PASSWORD_VALUE


==============================
Hardcoded Application Properties
==============================
spring:
  datasource:
    username: ashokit
    password: ashokit@123
    url: jdbc:mysql://localhost:3306/sbms
    driver-class-name: com.mysql.jdbc.Driver

========================================
Application Properties with Environment Variables
========================================

spring:
  datasource:
```

```
    username: ${DB_USERNAME:ashokit}
    password: ${DB_PASSWORD:ashokit@123}
    url: ${DB_URL:jdbc:mysql://localhost:3306/sbms}
    driver-class-name: ${DB_DRIVER: com.mysql.jdbc.Driver}



==================================
Deploying MySQL Database in K8S Cluster
==================================

****** Git Url for K8S Manifest Files : https://github.com/ashokitschool/kubernetes_manifes_ymls.git
********

# Create Config Map
$ kubectl apply -f <config-manifest-yml>

# Create Secret
$ kubectl apply -f <Secret-manifest-yml>

# Create PV
$ kubectl apply -f <PV-manifest-yml>

# Create PVC
$ kubectl apply -f <PVC-manifest-yml>

# Create Database Deployment
$ kubectl apply -f <Database-manifest-yml>

# Check pods which are created
$ kubectl get pods

Note: We should be able to get 'database' as a pod

****************************** With above steps our Database Deployment Completed
************************************

# Connecto database pod
$ kubectl exec -it <pod-name> bash

# Connect to database
$ mysql -h localhost -u root -p

Note: It will ask for password, enter password as root.

# Check databases available in mysql
$ show databases;

# Use the database
$ use weshopify-app;

# create a table
$ create table emp(emp_id int, varchar(50));

# display all tables
$ show tables;

# insert record
$ insert into emp values(101, 'Raju');
$ insert into emp values(102, 'Rani');

# Retrieve records from table
$ select * from emp;

# exit from mysql database
```

```
$ exit

# exit from the pod
$ exit

Note: Finally we are back to control-plane
```